**Office of Liquor and Gaming Regulation**

QCOM 3 Interface Specification

v3.0.3

*© The State of Queensland (Department of Justice and Attorney-General) 2016-2022. Copyright protects this publication. Enquiries should be addressed to* [*crown.copyright@qld.gov.au*](mailto:crown.copyright@qld.gov.au)

*The information contained herein is subject to change without notice. The copyright owner shall not be liable for technical or other errors or omissions contained herein. The reader/user accepts all risks and responsibility for losses, damages, costs and other consequences resulting directly or indirectly from using this information.*

*Copyright protects this publication. Your use of this publication is subject to the terms and conditions of the licence agreement entered into between you and the State of Queensland, acting through the Department of Justice and Attorney-General (‘the Department’). If you have not entered into a licence agreement with the Department, then you are not permitted to use this publication except to the extent permitted by the Copyright Act 1968 (Cth). If you would like to enter into a licence agreement with the Department to use this publication, please contact the Office of Liquor and Gaming Regulation on 13 QGOV (13 74 68) or visit https://www.business.qld.gov.au/industries/hospitality-tourism-sport/liquor-gaming*

**Table of Contents**

**TOC Brief**

[1 Introduction 17](#_Toc105778357)

[2 Glossary 29](#_Toc105778358)

[3 General 35](#_Toc105778359)

[4 QCOM machine commissioning 46](#_Toc105778360)

[5 QCOM Users 55](#_Toc105778361)

[6 Authorities 68](#_Toc105778362)

[7 Privileges 77](#_Toc105778363)

[8 Certificates 79](#_Toc105778364)

[9 Timekeeping 82](#_Toc105778365)

[10 The QCOM Lua Engine 84](#_Toc105778366)

[11 QCOM Lua API – Function Full Descriptions 128](#_Toc105778367)

[12 QCOM Internet Protocol API 225](#_Toc105778368)

[13 Events 246](#_Toc105778369)

[14 State Events 258](#_Toc105778370)

[15 Meters 268](#_Toc105778371)

[16 System Lockup 271](#_Toc105778372)

[17 ECT 275](#_Toc105778373)

[18 TITO 278](#_Toc105778374)

[19 Player Accessible Event Log 281](#_Toc105778375)

[20 Fault Conditions 286](#_Toc105778376)

[21 Peripheral Device Support 290](#_Toc105778377)

[22 Credit Redemption – Gaming Machines 294](#_Toc105778378)

[23 QCOM User Account Access (UAA) (SSL/TLS solution) 300](#_Toc105778379)

[24 QCOM Command Interpreter 305](#_Toc105778380)

[25 Progressive Prize Support 315](#_Toc105778381)

[26 Advanced Linked Progressive Prize Support 326](#_Toc105778382)

[27 Content Auditing 330](#_Toc105778383)

[28 Machine Audit Mode 342](#_Toc105778384)

[29 QCOM WWW Interface 346](#_Toc105778385)

[30 Machine Software Upgrades 351](#_Toc105778386)

[31 Electronic Seal Support 360](#_Toc105778387)

[32 Machine Discovery Protocol 361](#_Toc105778388)

[33 Implementation, Development, Testing and Simulation 363](#_Toc105778389)

[34 Possible Future Functionality 368](#_Toc105778390)

[Appendix A – QCOM 3 Modes of Operation 373](#_Toc105778391)

[Appendix B - Designing a QCOM 3 Operating Environment 376](#_Toc105778392)

[Appendix C – ANZ Gaming Machine National Standards (GMNS) and QCOM 379](#_Toc105778393)

[Revision History 380](#_Toc105778394)

[Machine Requirements Index (MRI) 390](#_Toc105778395)

**TOC Full**

[1 Introduction 17](#_Toc105778396)

[1.1 Disclaimer and legal 17](#_Toc105778397)

[1.2 Background 18](#_Toc105778398)

[1.3 Scope 18](#_Toc105778399)

[1.4 Objectives of the new version of QCOM 18](#_Toc105778400)

[1.5 High level design requirements 19](#_Toc105778401)

[1.6 Changes of significance 20](#_Toc105778402)

[1.7 Benefits of QCOM 3 21](#_Toc105778403)

[1.7.1 Design note: Meeting diverse operating requirements 24](#_Toc105778404)

[1.8 QCOM 3 Machine – Main software components 27](#_Toc105778405)

[2 Glossary 29](#_Toc105778406)

[2.1 Acronyms and abbreviations 33](#_Toc105778407)

[2.2 Interpretations 34](#_Toc105778408)

[3 General 35](#_Toc105778409)

[3.1 Machine hardware requirements 35](#_Toc105778410)

[3.2 Machine software requirements 38](#_Toc105778411)

[3.3 Other 38](#_Toc105778412)

[3.3.1 RNG 38](#_Toc105778413)

[3.3.2 Machine RAM errors 39](#_Toc105778414)

[3.4 Required knowledge 39](#_Toc105778415)

[3.5 The QCOM Local Area Network 40](#_Toc105778416)

[3.5.1 General principles 40](#_Toc105778417)

[3.5.2 Machine-to-machine communications 41](#_Toc105778418)

[3.5.3 Replacing a QCOM v1 machine with a QCOM 3 machine 41](#_Toc105778419)

[3.6 Standard sounds and alarms 42](#_Toc105778420)

[3.7 “i” button 43](#_Toc105778421)

[3.8 Secrets 45](#_Toc105778422)

[4 QCOM machine commissioning 46](#_Toc105778423)

[4.1 Date and time configuration 46](#_Toc105778424)

[4.2 Physical inspection 47](#_Toc105778425)

[4.3 Network configuration 48](#_Toc105778426)

[4.3.1 Machine discovery 48](#_Toc105778427)

[4.4 Logic Area Seal Confirmation Function 48](#_Toc105778428)

[4.4.1 Rationale 50](#_Toc105778429)

[4.5 QCOM Master Authority (QMA) 51](#_Toc105778430)

[4.6 User Introductions 51](#_Toc105778431)

[4.7 Application Level Configuration – Gaming Machines 52](#_Toc105778432)

[4.8 Comissioning disable condition 54](#_Toc105778433)

[5 QCOM Users 55](#_Toc105778434)

[5.1 Anon Users 56](#_Toc105778435)

[5.2 User Introductions / Account Creation 57](#_Toc105778436)

[5.2.1 Example 58](#_Toc105778437)

[5.3 User Account Storage 59](#_Toc105778438)

[5.3.1 Script file archives 60](#_Toc105778439)

[5.3.2 Text file encoding 61](#_Toc105778440)

[5.4 User Memory Quota 61](#_Toc105778441)

[5.5 User Lua Instruction Quota 62](#_Toc105778442)

[5.6 User CPU Quota 65](#_Toc105778443)

[5.7 User Quota Summary 66](#_Toc105778444)

[5.8 User Quarantine 66](#_Toc105778445)

[5.9 User Deletion 67](#_Toc105778446)

[5.10 Built-in QCOM users 67](#_Toc105778447)

[6 Authorities 68](#_Toc105778448)

[6.1 QCOM Master Authority (QMA) 68](#_Toc105778449)

[6.1.1 The QMA has first access 70](#_Toc105778450)

[6.1.2 Introductory Scripts 70](#_Toc105778451)

[6.1.3 Autonomous Users 71](#_Toc105778452)

[6.1.4 QMA Hijacking 72](#_Toc105778453)

[6.1.5 Summary: QMA, anon & regular QCOM users 74](#_Toc105778454)

[6.2 Software Upgrade Authorities (SUA) 75](#_Toc105778455)

[6.3 Script Approval Authority (SAA) 76](#_Toc105778456)

[7 Privileges 77](#_Toc105778457)

[7.1 Non-privileged QCOM API function calls 77](#_Toc105778458)

[7.2 Multiple User Environment considerations 78](#_Toc105778459)

[7.3 Default Permissions 78](#_Toc105778460)

[8 Certificates 79](#_Toc105778461)

[8.1 The Local Machine CA Certificate 79](#_Toc105778462)

[8.2 UAA Service Certificate 80](#_Toc105778463)

[8.3 WWW Service Certificate 81](#_Toc105778464)

[8.4 QCOM Authorities: QMA, SAA’s & SUA’s 81](#_Toc105778465)

[9 Timekeeping 82](#_Toc105778466)

[9.1 Timekeeping Events 83](#_Toc105778467)

[10 The QCOM Lua Engine 84](#_Toc105778468)

[10.1 Machine hosted Lua Interpreter - Core Requirements 85](#_Toc105778469)

[10.2 Implementation Considerations 88](#_Toc105778470)

[10.2.1 Memory Usage, Monitoring and Control 88](#_Toc105778471)

[10.2.2 Software Exceptions 91](#_Toc105778472)

[10.2.3 CPU Usage, Monitoring and Control 92](#_Toc105778473)

[10.2.4 Privacy and Trust Implications 93](#_Toc105778474)

[10.2.5 Global & QCOM User Environment Protection 95](#_Toc105778475)

[10.2.6 Sharing Data Between QCOM Users 95](#_Toc105778476)

[10.2.7 Data Persistence 96](#_Toc105778477)

[10.2.8 EGM “Meters” and Lua *number* types 96](#_Toc105778478)

[10.2.9 QCOM User Scripts That Fail to Return 96](#_Toc105778479)

[10.2.10 Standard Output (stdout) 96](#_Toc105778480)

[10.2.11 Runtime Error Handling 97](#_Toc105778481)

[10.2.12 Diagnostics 98](#_Toc105778482)

[10.2.13 New Versions of Lua 99](#_Toc105778483)

[10.2.14 Legal - Queensland 100](#_Toc105778484)

[10.3 Libraries 100](#_Toc105778485)

[10.3.1 Basic & Standard Libraries 101](#_Toc105778486)

[10.3.2 External Libraries 102](#_Toc105778487)

[10.4 QLE Initialisation Script 103](#_Toc105778488)

[10.5 QCOM API 103](#_Toc105778489)

[10.5.1 QCOM API Function Naming Convention 103](#_Toc105778490)

[10.5.2 Function Arguments 103](#_Toc105778491)

[10.5.3 Conversions 104](#_Toc105778492)

[10.5.4 Return Values 104](#_Toc105778493)

[10.5.5 Cooldowns 105](#_Toc105778494)

[10.6 Scripts - Life Cycle 105](#_Toc105778495)

[10.7 Script Execution 106](#_Toc105778496)

[10.7.1 Starting-up the QLE in a Machine 107](#_Toc105778497)

[10.7.2 QCOM user start-up script & module 109](#_Toc105778498)

[10.7.3 Executing Lua Scripts by Proxy 110](#_Toc105778499)

[10.8 Overloading QCOM API functions 112](#_Toc105778500)

[10.9 Script Development Considerations 114](#_Toc105778501)

[10.10 Persistent Variables 115](#_Toc105778502)

[10.10.1 Privacy 116](#_Toc105778503)

[10.10.2 Updates (Commits) of PVs 116](#_Toc105778504)

[10.10.3 PV API Definitions 117](#_Toc105778505)

[10.10.4 PV Application examples: Storing an Event buffer 117](#_Toc105778506)

[10.11 User Created Meters 119](#_Toc105778507)

[10.12 Applications and Examples 120](#_Toc105778508)

[10.12.1 General 120](#_Toc105778509)

[10.12.2 Random Number Generator 120](#_Toc105778510)

[10.12.3 Externally Triggered Jackpot Systems 120](#_Toc105778511)

[10.12.4 Offline Operation 122](#_Toc105778512)

[10.12.5 Player Information Display (PID) 122](#_Toc105778513)

[10.12.6 Player Accessible GUI / Information Displays 123](#_Toc105778514)

[10.12.7 Protocol Converter 125](#_Toc105778515)

[10.12.8 Health Monitoring and Sanity Checks 125](#_Toc105778516)

[10.12.9 Credit Meter Limiting 125](#_Toc105778517)

[10.12.10 Large Win Lockups 126](#_Toc105778518)

[10.12.11 Residual Credit Removal Feature (RCRF) 126](#_Toc105778519)

[11 QCOM Lua API – Function Full Descriptions 128](#_Toc105778520)

[11.1 \_ID 129](#_Toc105778521)

[11.1.1 qcom\_idInterfaceVersion 129](#_Toc105778522)

[11.1.2 qcom\_idDeviceType 129](#_Toc105778523)

[11.1.3 qcom\_idMachinePlatform 129](#_Toc105778524)

[11.1.4 qcom\_idMachineID 129](#_Toc105778525)

[11.1.5 qcom\_idSetMachineID 129](#_Toc105778526)

[11.1.6 qcom\_idLogicUID 129](#_Toc105778527)

[11.1.7 qcom\_idCommissionUID 129](#_Toc105778528)

[11.1.8 qcom\_idMfr 130](#_Toc105778529)

[11.1.9 qcom\_idMfr3 130](#_Toc105778530)

[11.1.10 qcom\_idOSversion 130](#_Toc105778531)

[11.2 \_Network Management 130](#_Toc105778532)

[11.2.1 qcom\_networkIP 130](#_Toc105778533)

[11.2.2 qcom\_networkEthAddr 130](#_Toc105778534)

[11.3 \_Location Management 131](#_Toc105778535)

[11.3.1 qcom\_locSetCountryCode 131](#_Toc105778536)

[11.3.2 qcom\_locCountryCode 131](#_Toc105778537)

[11.3.3 qcom\_locSetCurrencyCode 131](#_Toc105778538)

[11.3.4 qcom\_locCurrencyCode 131](#_Toc105778539)

[11.3.5 qcom\_locSetStateProv 131](#_Toc105778540)

[11.3.6 qcom\_locStateProv 131](#_Toc105778541)

[11.3.7 qcom\_locSetVenueID 131](#_Toc105778542)

[11.3.8 qcom\_locVenueID 131](#_Toc105778543)

[11.3.9 qcom\_locSetVenueName 131](#_Toc105778544)

[11.3.10 qcom\_locSetVenueType 131](#_Toc105778545)

[11.3.11 qcom\_locVenueType 131](#_Toc105778546)

[11.3.12 qcom\_locVenueName 131](#_Toc105778547)

[11.3.13 qcom\_locSetVenueAddress 131](#_Toc105778548)

[11.3.14 qcom\_locVenueAddress 131](#_Toc105778549)

[11.3.15 qcom\_locSetFloorLocation 131](#_Toc105778550)

[11.3.16 qcom\_locFloorLocation 131](#_Toc105778551)

[11.4 \_NTP 132](#_Toc105778552)

[11.4.1 qcom\_ntpSetp 132](#_Toc105778553)

[11.4.2 qcom\_ntpEnable 132](#_Toc105778554)

[11.4.3 qcom\_ntpStatus 132](#_Toc105778555)

[11.5 \_Time 133](#_Toc105778556)

[11.5.1 qcom\_timeSet 133](#_Toc105778557)

[11.5.2 qcom\_timeSetStrfmt 134](#_Toc105778558)

[11.5.3 qcom\_timeStrfmt 134](#_Toc105778559)

[11.5.4 qcom\_timeSetTimezone 134](#_Toc105778560)

[11.5.5 qcom\_timeTimezone 135](#_Toc105778561)

[11.5.6 qcom\_timeSetOSD 135](#_Toc105778562)

[11.5.7 qcom\_timeOSD 135](#_Toc105778563)

[11.6 \_Timer 135](#_Toc105778564)

[11.6.1 qcom\_timerCreate 135](#_Toc105778565)

[11.6.2 qcom\_timerSetp 135](#_Toc105778566)

[11.7 \_Machine 135](#_Toc105778567)

[11.7.1 qcom\_machineAttendantRequired 135](#_Toc105778568)

[11.7.2 qcom\_machineSetLanguage 135](#_Toc105778569)

[11.7.3 qcom\_machineLanguage 135](#_Toc105778570)

[11.7.4 qcom\_machineLanguagesSupported 135](#_Toc105778571)

[11.7.5 qcom\_machineSetMeterDenom 135](#_Toc105778572)

[11.7.6 qcom\_machineMeterDenom 136](#_Toc105778573)

[11.7.7 qcom\_machineSetNumberDisplayFormat 136](#_Toc105778574)

[11.7.8 qcom\_machineSetCurrencyDisplayFormat 136](#_Toc105778575)

[11.7.9 qcom\_machineQueueReboot 136](#_Toc105778576)

[11.7.10 qcom\_machineQueueRestart 136](#_Toc105778577)

[11.7.11 qcom\_machineQueueShutdown 137](#_Toc105778578)

[11.7.12 qcom\_machineQueueHibernate 137](#_Toc105778579)

[11.7.13 qcom\_machineShuttingDown 137](#_Toc105778580)

[11.7.14 qcom\_machineSetPowerUpTime 137](#_Toc105778581)

[11.7.15 qcom\_machinePowerUpTime 137](#_Toc105778582)

[11.7.16 qcom\_machinePowerSaveEnter 137](#_Toc105778583)

[11.7.17 qcom\_machinePowerSaveExit 137](#_Toc105778584)

[11.7.18 qcom\_machinePowerSaveActive 137](#_Toc105778585)

[11.7.19 qcom\_machineInactivityTimer 137](#_Toc105778586)

[11.7.20 qcom\_machineSetFastBoot 137](#_Toc105778587)

[11.7.21 qcom\_machineRAMclear 137](#_Toc105778588)

[11.7.22 qcom\_machineInDebugMode 137](#_Toc105778589)

[11.7.23 qcom\_machineTakeScreenshot 137](#_Toc105778590)

[11.7.24 qcom\_machineOperatingTime 138](#_Toc105778591)

[11.7.25 qcom\_machineRand 139](#_Toc105778592)

[11.7.26 qcom\_machineReady 139](#_Toc105778593)

[11.7.27 qcom\_machineUpgradeGetVerify 139](#_Toc105778594)

[11.7.28 qcom\_machineUpgradeQueue 139](#_Toc105778595)

[11.7.29 qcom\_machineUpgradeGetp 139](#_Toc105778596)

[11.7.30 qcom\_machineUpgradeSetp 139](#_Toc105778597)

[11.8 \_Peripheral Devices 139](#_Toc105778598)

[11.8.1 qcom\_peripheralSupported 139](#_Toc105778599)

[11.8.2 qcom\_peripheralConnected 140](#_Toc105778600)

[11.8.3 qcom\_peripheralStatus 140](#_Toc105778601)

[11.9 \_Security 140](#_Toc105778602)

[11.9.1 qcom\_secQMAcert 141](#_Toc105778603)

[11.9.2 qcom\_secAddSUAcert 141](#_Toc105778604)

[11.9.3 qcom\_secSUAcerts 141](#_Toc105778605)

[11.9.4 qcom\_secMachineCert 141](#_Toc105778606)

[11.9.5 qcom\_secSetUAAverifyCert 141](#_Toc105778607)

[11.9.6 qcom\_secUAAverifyCert 141](#_Toc105778608)

[11.9.7 qcom\_secLogicSealOk 141](#_Toc105778609)

[11.9.8 qcom\_eSealPublicKey 141](#_Toc105778610)

[11.9.9 qcom\_eSealVerify 141](#_Toc105778611)

[11.10 \_Digest 142](#_Toc105778612)

[11.10.1 qcom\_dgst 142](#_Toc105778613)

[11.10.2 qcom\_dgstCreate 142](#_Toc105778614)

[11.10.3 qcom\_dgstHMAC 142](#_Toc105778615)

[11.10.4 qcom\_dgstHMACcreate 142](#_Toc105778616)

[11.11 \_Encryption 142](#_Toc105778617)

[11.11.1 qcom\_encEncrypt 143](#_Toc105778618)

[11.11.2 qcom\_encDecrypt 143](#_Toc105778619)

[11.11.3 qcom\_encEncryptCreate 143](#_Toc105778620)

[11.11.4 qcom\_encDecryptCreate 143](#_Toc105778621)

[11.12 \_x509 143](#_Toc105778622)

[11.12.1 qcom\_X509decode 143](#_Toc105778623)

[11.13 \_Content Auditing 143](#_Toc105778624)

[11.13.1 qcom\_cAuditCommonStart 143](#_Toc105778625)

[11.13.2 qcom\_cAuditCommonResults 144](#_Toc105778626)

[11.13.3 qcom\_cAuditGameStart 144](#_Toc105778627)

[11.13.4 qcom\_cAuditGameResults 144](#_Toc105778628)

[11.14 \_WWW 144](#_Toc105778629)

[11.14.1 qcom\_wwwGenerateCert 145](#_Toc105778630)

[11.14.2 qcom\_wwwCert 145](#_Toc105778631)

[11.14.3 qcom\_wwwStop 145](#_Toc105778632)

[11.14.4 qcom\_wwwStart 145](#_Toc105778633)

[11.14.5 qcom\_wwwStatus 145](#_Toc105778634)

[11.15 \_Lua 145](#_Toc105778635)

[11.15.1 qcom\_luaUsedMem 145](#_Toc105778636)

[11.15.2 qcom\_luaErrors 146](#_Toc105778637)

[11.15.3 qcom\_luaPrintHistory 146](#_Toc105778638)

[11.15.4 qcom\_luaHookScript 146](#_Toc105778639)

[11.15.5 qcom\_luaQCIAddCommand 147](#_Toc105778640)

[11.15.6 qcom\_luaEventTime 147](#_Toc105778641)

[11.15.7 qcom\_luaEventData 147](#_Toc105778642)

[11.15.8 qcom\_luaSEID 147](#_Toc105778643)

[11.15.9 qcom\_luaPanic 147](#_Toc105778644)

[11.15.10 qcom\_luaPublish 147](#_Toc105778645)

[11.15.11 qcom\_luaPublishGetValue 149](#_Toc105778646)

[11.15.12 qcom\_luaOverload 149](#_Toc105778647)

[11.15.13 qcom\_luaOverloadClear 149](#_Toc105778648)

[11.15.14 qcom\_luaDoString 150](#_Toc105778649)

[11.15.15 qcom\_LuaBigNum 150](#_Toc105778650)

[11.16 \_Audit 151](#_Toc105778651)

[11.16.1 qcom\_auditReg 151](#_Toc105778652)

[11.16.2 qcom\_auditResp 152](#_Toc105778653)

[11.17 \_Persistent Variables 153](#_Toc105778654)

[11.17.1 qcom\_pvCommit 153](#_Toc105778655)

[11.17.2 qcom\_pvCommit\_string 154](#_Toc105778656)

[11.17.3 qcom\_pvRestore 154](#_Toc105778657)

[11.17.4 qcom\_pvRestore\_string 154](#_Toc105778658)

[11.17.5 qcom\_pvStats 154](#_Toc105778659)

[11.17.6 qcom\_pvSetp 154](#_Toc105778660)

[11.18 \_User Meters 154](#_Toc105778661)

[11.18.1 qcom\_userMeterCreate 155](#_Toc105778662)

[11.18.2 qcom\_userMeterUpdate 155](#_Toc105778663)

[11.18.3 qcom\_userMeterDestroy 155](#_Toc105778664)

[11.18.4 qcom\_userMeterRead 155](#_Toc105778665)

[11.18.5 qcom\_userMeterOwner 155](#_Toc105778666)

[11.18.6 qcom\_userMetersSetMax 155](#_Toc105778667)

[11.18.7 qcom\_userMetersAvail 155](#_Toc105778668)

[11.19 \_EGM Maintenance 155](#_Toc105778669)

[11.19.1 qcom\_egmSetMinMaxRTP 155](#_Toc105778670)

[11.19.2 qcom\_egmMinRTP 155](#_Toc105778671)

[11.19.3 qcom\_egmMaxRTP 155](#_Toc105778672)

[11.19.4 qcom\_egmSetMaxRTPDeviation 155](#_Toc105778673)

[11.19.5 qcom\_egmMaxRTPDeviation 155](#_Toc105778674)

[11.19.6 qcom\_egmSetCreditDenom 156](#_Toc105778675)

[11.19.7 qcom\_egmCreditDenom 156](#_Toc105778676)

[11.19.8 qcom\_egmCreditInputDisable 156](#_Toc105778677)

[11.19.9 qcom\_egmCreditInputEnable 156](#_Toc105778678)

[11.19.10 qcom\_egmCreditInputEnabled 156](#_Toc105778679)

[11.19.11 qcom\_egmSetMaxBet 156](#_Toc105778680)

[11.19.12 qcom\_egmMaxBet 157](#_Toc105778681)

[11.19.13 qcom\_egmCreditMeter 157](#_Toc105778682)

[11.19.14 qcom\_egmBetMeter 157](#_Toc105778683)

[11.19.15 qcom\_egmWinMeter 157](#_Toc105778684)

[11.19.16 qcom\_egmMeter 157](#_Toc105778685)

[11.19.17 qcom\_egmMeters 157](#_Toc105778686)

[11.19.18 qcom\_egmSetMeterDisplayFunction 158](#_Toc105778687)

[11.19.19 qcom\_egmDisplayMeters 159](#_Toc105778688)

[11.19.20 qcom\_egmState 160](#_Toc105778689)

[11.19.21 qcom\_egmDoorsClosed 161](#_Toc105778690)

[11.19.22 qcom\_egmDoorState 161](#_Toc105778691)

[11.19.23 qcom\_egmInFault 161](#_Toc105778692)

[11.19.24 qcom\_egmInQuietFault 161](#_Toc105778693)

[11.19.25 qcom\_egmInTestMode 161](#_Toc105778694)

[11.19.26 qcom\_egmInAuditMode 161](#_Toc105778695)

[11.19.27 qcom\_egmOK 161](#_Toc105778696)

[11.19.28 qcom\_egmKeySwitchDisable 161](#_Toc105778697)

[11.19.29 qcom\_egmKeySwitchEnable 161](#_Toc105778698)

[11.19.30 qcom\_egmKeySwitchStatus 161](#_Toc105778699)

[11.19.31 qcom\_egmSPAMA 161](#_Toc105778700)

[11.19.32 qcom\_egmSPAMB 161](#_Toc105778701)

[11.19.33 qcom\_egmSPAMC 161](#_Toc105778702)

[11.19.34 qcom\_egmGPM 161](#_Toc105778703)

[11.19.35 qcom\_egmSMS 161](#_Toc105778704)

[11.19.36 qcom\_egmGambleSetp 162](#_Toc105778705)

[11.19.37 qcom\_egmGambleGetp 162](#_Toc105778706)

[11.19.38 qcom\_egmReserve 162](#_Toc105778707)

[11.19.39 qcom\_egmSetReserve 162](#_Toc105778708)

[11.20 \_Game Maintenance 162](#_Toc105778709)

[11.20.1 qcom\_gameCurrent 163](#_Toc105778710)

[11.20.2 qcom\_gameNum 163](#_Toc105778711)

[11.20.3 qcom\_gameList 163](#_Toc105778712)

[11.20.4 qcom\_gameMaxEnabled 163](#_Toc105778713)

[11.20.5 qcom\_gameLanguagesSupported 163](#_Toc105778714)

[11.20.6 qcom\_gameUID 163](#_Toc105778715)

[11.20.7 qcom\_gameGetp 163](#_Toc105778716)

[11.20.8 qcom\_gameName 166](#_Toc105778717)

[11.20.9 qcom\_gameGEF 166](#_Toc105778718)

[11.20.10 qcom\_gameDisable 166](#_Toc105778719)

[11.20.11 qcom\_gameEnable 166](#_Toc105778720)

[11.20.12 qcom\_gamePCmeters 166](#_Toc105778721)

[11.20.13 qcom\_gameVarNum 166](#_Toc105778722)

[11.20.14 qcom\_gameVarCurrent 166](#_Toc105778723)

[11.20.15 qcom\_gameSetVar 166](#_Toc105778724)

[11.20.16 qcom\_gameVarList 168](#_Toc105778725)

[11.20.17 qcom\_gameVarGetp 168](#_Toc105778726)

[11.20.18 qcom\_gameVarMeters 170](#_Toc105778727)

[11.21 \_Progressive Prize Support 170](#_Toc105778728)

[11.21.1 qcom\_progrNum 171](#_Toc105778729)

[11.21.2 qcom\_progrGamesList 171](#_Toc105778730)

[11.21.3 qcom\_progrMeters 171](#_Toc105778731)

[11.21.4 qcom\_progrTurnoverMeter 171](#_Toc105778732)

[11.21.5 qcom\_progrLastHit 171](#_Toc105778733)

[11.21.6 qcom\_progrList 171](#_Toc105778734)

[11.21.7 qcom\_progrGetp 171](#_Toc105778735)

[11.21.8 qcom\_progrSetp 171](#_Toc105778736)

[11.21.9 qcom\_progrSetPrize 171](#_Toc105778737)

[11.21.10 qcom\_progrResetLockup 171](#_Toc105778738)

[11.21.11 qcom\_progrPosAdj 171](#_Toc105778739)

[11.21.12 qcom\_progrNegAdj 171](#_Toc105778740)

[11.21.13 qcom\_progrModeChange 171](#_Toc105778741)

[11.22 \_Events 171](#_Toc105778742)

[11.22.1 qcom\_eventsGetIterator 172](#_Toc105778743)

[11.22.2 qcom\_eventsGetLatest 172](#_Toc105778744)

[11.22.3 qcom\_eventsGetLast 172](#_Toc105778745)

[11.22.4 qcom\_eventsStatus 172](#_Toc105778746)

[11.22.5 qcom\_eventsAddUserEvent 172](#_Toc105778747)

[11.22.6 qcom\_eventsSetTTL 172](#_Toc105778748)

[11.22.7 qcom\_eventsSetSLx 172](#_Toc105778749)

[11.22.8 qcom\_eventsGetSLx 172](#_Toc105778750)

[11.23 \_ECT 173](#_Toc105778751)

[11.23.1 qcom\_ectIsEnabled 173](#_Toc105778752)

[11.23.2 qcom\_ectDisable 173](#_Toc105778753)

[11.23.3 qcom\_ectEnable 173](#_Toc105778754)

[11.23.4 qcom\_ectMaxECT 173](#_Toc105778755)

[11.23.5 qcom\_ectSetMaxECT 173](#_Toc105778756)

[11.23.6 qcom\_ectAddCredit 173](#_Toc105778757)

[11.23.7 qcom\_ectSubtractCreditAuthorised 175](#_Toc105778758)

[11.23.8 qcom\_ectSubtractCredit 176](#_Toc105778759)

[11.23.9 qcom\_ectSubtractCreditDeclined 178](#_Toc105778760)

[11.23.10 qcom\_ectTicketInAddCredit 178](#_Toc105778761)

[11.23.11 qcom\_ectTicketOutSubtractCredit 180](#_Toc105778762)

[11.24 \_CancelCredit 181](#_Toc105778763)

[11.24.1 qcom\_cancelCredit 181](#_Toc105778764)

[11.25 \_System Lockup 183](#_Toc105778765)

[11.25.1 qcom\_slRequest 183](#_Toc105778766)

[11.25.2 qcom\_slReset 188](#_Toc105778767)

[11.25.3 qcom\_slStatus 189](#_Toc105778768)

[11.26 \_Play Control 190](#_Toc105778769)

[11.26.1 qcom\_playSetPEF 190](#_Toc105778770)

[11.26.2 qcom\_playPEF 191](#_Toc105778771)

[11.26.3 qcom\_playOKex 191](#_Toc105778772)

[11.27 \_PID 193](#_Toc105778773)

[11.27.1 qcom\_pidEnable 193](#_Toc105778774)

[11.27.2 qcom\_pidEnabled 193](#_Toc105778775)

[11.27.3 qcom\_pidList 193](#_Toc105778776)

[11.28 \_Custom User Interface 193](#_Toc105778777)

[11.28.1 qcom\_infoAddMenuItem 193](#_Toc105778778)

[11.28.2 qcom\_infoUIclrscr 194](#_Toc105778779)

[11.28.3 qcom\_infoUIgetnum 194](#_Toc105778780)

[11.28.4 qcom\_infoUIgetstring 194](#_Toc105778781)

[11.28.5 qcom\_infoUIhotspot 194](#_Toc105778782)

[11.28.6 qcom\_infoUIprint 195](#_Toc105778783)

[11.29 \_Health 195](#_Toc105778784)

[11.29.1 qcom\_healthFreeMem 195](#_Toc105778785)

[11.29.2 qcom\_healthTemperatures 195](#_Toc105778786)

[11.29.3 qcom\_healthVoltages 195](#_Toc105778787)

[11.29.4 qcom\_healthFanSpeeds 195](#_Toc105778788)

[11.29.5 qcom\_healthDrives 195](#_Toc105778789)

[11.30 \_User Maintenance 195](#_Toc105778790)

[11.30.1 qcom\_userCreate 195](#_Toc105778791)

[11.30.2 qcom\_userDelete 195](#_Toc105778792)

[11.30.3 qcom\_userDeleteSelf 196](#_Toc105778793)

[11.30.4 qcom\_userQuarantine 196](#_Toc105778794)

[11.30.5 qcom\_userIsQuarantined 196](#_Toc105778795)

[11.30.6 qcom\_userSetUAApublicKey 196](#_Toc105778796)

[11.30.7 qcom\_userSetMyUAApublicKey 196](#_Toc105778797)

[11.30.8 qcom\_userUAApublicKey 196](#_Toc105778798)

[11.30.9 qcom\_userEnable 196](#_Toc105778799)

[11.30.10 qcom\_userDisable 196](#_Toc105778800)

[11.30.11 qcom\_userEnabled 196](#_Toc105778801)

[11.30.12 qcom\_userSetDiskQuota 196](#_Toc105778802)

[11.30.13 qcom\_userDiskStats 197](#_Toc105778803)

[11.30.14 qcom\_userSetMemoryQuota 197](#_Toc105778804)

[11.30.15 qcom\_userMemoryStats 197](#_Toc105778805)

[11.30.16 qcom\_userSetFullName 198](#_Toc105778806)

[11.30.17 qcom\_userFullName 198](#_Toc105778807)

[11.30.18 qcom\_userSetPrivilege 198](#_Toc105778808)

[11.30.19 qcom\_userPrivileges 199](#_Toc105778809)

[11.30.20 qcom\_userSetReadyState 199](#_Toc105778810)

[11.30.21 qcom\_userWhoAmI 199](#_Toc105778811)

[11.30.22 qcom\_userWho 199](#_Toc105778812)

[11.30.23 qcom\_userLoggedOn 199](#_Toc105778813)

[11.30.24 qcom\_userList 199](#_Toc105778814)

[11.30.25 qcom\_userSetSAAcert 199](#_Toc105778815)

[11.30.26 qcom\_userSAAcert 199](#_Toc105778816)

[11.30.27 qcom\_userLoadScripts 199](#_Toc105778817)

[11.30.28 qcom\_userSetScripts 199](#_Toc105778818)

[11.30.29 qcom\_userScriptHashes 199](#_Toc105778819)

[11.30.30 qcom\_userRestart 199](#_Toc105778820)

[11.30.31 qcom\_userShutdown 200](#_Toc105778821)

[11.30.32 qcom\_userIsShutdown 200](#_Toc105778822)

[11.30.33 qcom\_userAnonSetPass 200](#_Toc105778823)

[11.30.34 qcom\_userAnonPass 201](#_Toc105778824)

[11.30.35 qcom\_userCPUstats 201](#_Toc105778825)

[11.30.36 qcom\_userSetCPUquota 201](#_Toc105778826)

[11.30.37 qcom\_userInstructionStats 202](#_Toc105778827)

[11.30.38 qcom\_userSetInstructionQuota 202](#_Toc105778828)

[11.31 \_RemoteControl 204](#_Toc105778829)

[11.31.1 qcom\_rcSetAutoPlay 204](#_Toc105778830)

[11.31.2 qcom\_rcAutoPlay 204](#_Toc105778831)

[11.31.3 qcom\_rcButtonState 204](#_Toc105778832)

[11.31.4 qcom\_rcButtonPress 204](#_Toc105778833)

[11.31.5 qcom\_rcPlay 204](#_Toc105778834)

[11.31.6 qcom\_rcRandomResponse 204](#_Toc105778835)

[11.31.7 qcom\_rcResetFault 204](#_Toc105778836)

[11.31.8 qcom\_rcResetLockup 204](#_Toc105778837)

[11.31.9 qcom\_rcResetKey 204](#_Toc105778838)

[11.31.10 qcom\_rcGoFault 204](#_Toc105778839)

[11.31.11 qcom\_rcCollectPress 204](#_Toc105778840)

[11.32 \_Banknote Acceptor Maintenance 204](#_Toc105778841)

[11.32.1 qcom\_bnaGetp 205](#_Toc105778842)

[11.32.2 qcom\_bnaSetEnableFlag 205](#_Toc105778843)

[11.32.3 qcom\_bnaFirmwareID 205](#_Toc105778844)

[11.32.4 qcom\_bnaFirmwareUpgrade 205](#_Toc105778845)

[11.32.5 qcom\_bnaNoteStatus 205](#_Toc105778846)

[11.32.6 qcom\_bnaSetNoteStatus 205](#_Toc105778847)

[11.32.7 qcom\_bnaMetersByDenom 205](#_Toc105778848)

[11.32.8 qcom\_bnaRejectTicket 205](#_Toc105778849)

[11.33 \_Ticket Printer Maintenance 205](#_Toc105778850)

[11.33.1 qcom\_tpGetp 205](#_Toc105778851)

[11.33.2 qcom\_tpFirmwareID 205](#_Toc105778852)

[11.33.3 qcom\_tpFirmwareIDUpgrade 205](#_Toc105778853)

[11.33.4 qcom\_tpMeters 205](#_Toc105778854)

[11.34 \_Coin Acceptor Maintenance 205](#_Toc105778855)

[11.34.1 qcom\_caGetp 206](#_Toc105778856)

[11.34.2 qcom\_caSetEnableFlag 206](#_Toc105778857)

[11.34.3 qcom\_caFirmwareID 206](#_Toc105778858)

[11.34.4 qcom\_caFirmwareIDUpgrade 206](#_Toc105778859)

[11.34.5 qcom\_caSetDiverterAutoFlag 206](#_Toc105778860)

[11.34.6 qcom\_caDenom 206](#_Toc105778861)

[11.34.7 qcom\_caSetDenom 206](#_Toc105778862)

[11.35 \_Hopper Maintenance 206](#_Toc105778863)

[11.35.1 qcom\_hopperGetp 206](#_Toc105778864)

[11.35.2 qcom\_hopperSetDefaultRefillAmount 206](#_Toc105778865)

[11.35.3 qcom\_hopperRecordRefill 206](#_Toc105778866)

[11.35.4 qcom\_hopperDenom 206](#_Toc105778867)

[11.35.5 qcom\_hopperSetDenom 206](#_Toc105778868)

[11.35.6 qcom\_hopperPayout 206](#_Toc105778869)

[11.36 \_uart 207](#_Toc105778870)

[11.36.1 qcom\_uartSetp 211](#_Toc105778871)

[11.36.2 qcom\_uartGetp 212](#_Toc105778872)

[11.36.3 qcom\_uartOpen 213](#_Toc105778873)

[11.36.4 UART object member functions 214](#_Toc105778874)

[11.37 \_udp 220](#_Toc105778875)

[11.37.1 qcom\_udp 220](#_Toc105778876)

[11.37.2 qcom\_udpSetp 220](#_Toc105778877)

[11.37.3 qcom\_udpGetp 221](#_Toc105778878)

[11.37.4 qcom\_udpGetStats 221](#_Toc105778879)

[11.38 \_tcpClient 221](#_Toc105778880)

[11.38.1 qcom\_tcpClient 221](#_Toc105778881)

[11.38.2 qcom\_tcpClientSetp 222](#_Toc105778882)

[11.38.3 qcom\_tcpClientGetp 223](#_Toc105778883)

[11.38.4 qcom\_tcpClientGetStats 224](#_Toc105778884)

[12 QCOM Internet Protocol API 225](#_Toc105778885)

[12.1 Connection Management 227](#_Toc105778886)

[12.2 TCP Client Socket API 227](#_Toc105778887)

[12.2.1 socket() 227](#_Toc105778888)

[12.2.2 connect() 228](#_Toc105778889)

[12.2.3 setConnectCallback() 228](#_Toc105778890)

[12.2.4 setRxCallback() 229](#_Toc105778891)

[12.2.5 setDisconnectCallback() 230](#_Toc105778892)

[12.2.6 setErrorCallback() 231](#_Toc105778893)

[12.2.7 setCanWriteCallback() 232](#_Toc105778894)

[12.2.8 send() 232](#_Toc105778895)

[12.2.9 close() 233](#_Toc105778896)

[12.2.10 free() 234](#_Toc105778897)

[12.3 Secure Client Socket API 235](#_Toc105778898)

[12.3.1 addTrustedCert() 235](#_Toc105778899)

[12.3.2 useCert() 237](#_Toc105778900)

[12.3.3 getPeerCert() 237](#_Toc105778901)

[12.3.4 verifyCert() 238](#_Toc105778902)

[12.3.5 getCiphers() 239](#_Toc105778903)

[12.3.6 setCiphers() 239](#_Toc105778904)

[12.4 UDP API 240](#_Toc105778905)

[12.4.1 socket() 241](#_Toc105778906)

[12.4.2 bind() 241](#_Toc105778907)

[12.4.3 connect() 242](#_Toc105778908)

[12.4.4 setBroadcast() 242](#_Toc105778909)

[12.4.5 setRxCallback() 243](#_Toc105778910)

[12.4.6 setErrorCallback() 243](#_Toc105778911)

[12.4.7 setCanWriteCallback() 243](#_Toc105778912)

[12.4.8 sendto() 243](#_Toc105778913)

[12.4.9 send() 244](#_Toc105778914)

[12.4.10 shutdown() 244](#_Toc105778915)

[12.4.11 close() 244](#_Toc105778916)

[12.4.12 free() 244](#_Toc105778917)

[13 Events 246](#_Toc105778918)

[13.1 The QCOM Event Buffer 247](#_Toc105778919)

[13.2 Event Retention 248](#_Toc105778920)

[13.2.1 Smart event retention using TTL 250](#_Toc105778921)

[13.2.2 Closing remarks on event retention 251](#_Toc105778922)

[13.3 Event Buffer Full Condition 252](#_Toc105778923)

[13.4 Event Buffer QCOM API 252](#_Toc105778924)

[13.5 QCOM event schema 252](#_Toc105778925)

[13.6 QCOM Event Definitions 253](#_Toc105778926)

[13.6.1 The meaning re ‘log the’. 253](#_Toc105778927)

[13.7 Implementation Notes 254](#_Toc105778928)

[13.8 Event Forwarding 255](#_Toc105778929)

[13.9 Event SPAM Reduction 255](#_Toc105778930)

[14 State Events 258](#_Toc105778931)

[14.1 State Event Definitions 259](#_Toc105778932)

[14.2 State Event Transitions 259](#_Toc105778933)

[14.2.1 Illegal State Transitions 260](#_Toc105778934)

[14.3 State Event Timing 261](#_Toc105778935)

[14.4 Sync events 263](#_Toc105778936)

[14.5 QCOM Event Dispatcher 265](#_Toc105778937)

[14.6 Special Types of State Events 266](#_Toc105778938)

[14.6.1 Call-back Functions 266](#_Toc105778939)

[14.6.2 Internal-Use Only State Events 266](#_Toc105778940)

[14.7 State event buffer full 266](#_Toc105778941)

[15 Meters 268](#_Toc105778942)

[15.1 Meter List and Definitions 268](#_Toc105778943)

[15.2 Meter Representation 268](#_Toc105778944)

[15.3 Meter Rollover 269](#_Toc105778945)

[15.4 Self Audit 269](#_Toc105778946)

[15.5 Audit Mode Master Meter Display 269](#_Toc105778947)

[16 System Lockup 271](#_Toc105778948)

[16.1 Multi-user support 272](#_Toc105778949)

[16.2 Orphaned System Lockups 273](#_Toc105778950)

[16.3 System Lockup vs PEF Disable 273](#_Toc105778951)

[17 ECT 275](#_Toc105778952)

[18 TITO 278](#_Toc105778953)

[18.1 Summary of methods 278](#_Toc105778954)

[18.2 Ticket-Out Sequence 279](#_Toc105778955)

[18.2.1 Cash Ticket Layout 279](#_Toc105778956)

[18.3 Ticket-In Sequence 279](#_Toc105778957)

[18.4 Ticket Details / Layout 280](#_Toc105778958)

[19 Player Accessible Event Log 281](#_Toc105778959)

[19.1 Accessing 282](#_Toc105778960)

[19.2 Appearance 282](#_Toc105778961)

[19.3 Posting events 283](#_Toc105778962)

[19.4 Persistence 284](#_Toc105778963)

[19.5 Configuration 284](#_Toc105778964)

[19.6 Other supporting QCOM API functions: 284](#_Toc105778965)

[20 Fault Conditions 286](#_Toc105778966)

[20.1 Fault Condition Backgrounding / Quieting 286](#_Toc105778967)

[20.2 Door Events 288](#_Toc105778968)

[20.2.1 Logic door access 288](#_Toc105778969)

[21 Peripheral Device Support 290](#_Toc105778970)

[21.1 Coin / Token Hoppers 291](#_Toc105778971)

[21.2 Peripheral Firmware Upgrades 291](#_Toc105778972)

[22 Credit Redemption – Gaming Machines 294](#_Toc105778973)

[22.1 Support for Multiple Service Providers 295](#_Toc105778974)

[22.1.1 Credit redemption process for an account based gaming service provider 297](#_Toc105778975)

[22.1.2 Credit redemption process for a TITO system 298](#_Toc105778976)

[22.1.3 Credit redemption process for other systems 298](#_Toc105778977)

[22.2 Attendant Paging System Support 298](#_Toc105778978)

[22.3 Residual Credit Removal Feature 299](#_Toc105778979)

[23 QCOM User Account Access (UAA) (SSL/TLS solution) 300](#_Toc105778980)

[23.1 Client Authentication Methodology 300](#_Toc105778981)

[23.2 The QCOM UAA Service Configuration 302](#_Toc105778982)

[23.3 Summary of machine’ UAA service. 303](#_Toc105778983)

[24 QCOM Command Interpreter 305](#_Toc105778984)

[24.1 Command Privileges 306](#_Toc105778985)

[24.2 Commands 306](#_Toc105778986)

[24.2.1 qmaloadcert 307](#_Toc105778987)

[24.2.2 qmacertfingerprint 307](#_Toc105778988)

[24.2.3 qmaexecscript 308](#_Toc105778989)

[24.2.4 userloadscripts 309](#_Toc105778990)

[24.2.5 restartuser 310](#_Toc105778991)

[24.2.6 shutdownuser 310](#_Toc105778992)

[24.2.7 dostring 311](#_Toc105778993)

[24.2.8 lua 311](#_Toc105778994)

[24.2.9 docmd 312](#_Toc105778995)

[24.2.10 help 312](#_Toc105778996)

[24.2.11 exit 312](#_Toc105778997)

[24.3 Custom Commands 312](#_Toc105778998)

[25 Progressive Prize Support 315](#_Toc105778999)

[25.1 Terminology 315](#_Toc105779000)

[25.2 Configuration and Control 315](#_Toc105779001)

[25.2.1 Progressive Level Properties 316](#_Toc105779002)

[25.2.2 SAP Current Amount Adjustments 321](#_Toc105779003)

[25.3 Progressive Prize Meters 322](#_Toc105779004)

[25.4 QCOM API Progressive Function summary 322](#_Toc105779005)

[25.5 LP Prize Support and Operation 323](#_Toc105779006)

[25.5.1 LP Prize Contributions and Totalisation 323](#_Toc105779007)

[25.5.2 Prize Control 323](#_Toc105779008)

[25.5.3 Awarding LP Jackpots 324](#_Toc105779009)

[25.6 Specific arrangements 326](#_Toc105779010)

[25.6.1 Multi-variation progressive games 326](#_Toc105779011)

[25.6.2 Hardcoded vs non-hardcoded progressive level properties 326](#_Toc105779012)

[26 Advanced Linked Progressive Prize Support 326](#_Toc105779013)

[27 Content Auditing 330](#_Toc105779014)

[27.1 Common Content Auditing 332](#_Toc105779015)

[27.2 Game Content Auditing 337](#_Toc105779016)

[27.3 Peripheral Content Auditing 338](#_Toc105779017)

[27.4 Related Conventions 340](#_Toc105779018)

[27.5 Other Notes Concerning Content Auditing 340](#_Toc105779019)

[28 Machine Audit Mode 342](#_Toc105779020)

[28.1 Possible Machine Audit Mode Future Functionality 343](#_Toc105779021)

[29 QCOM WWW Interface 346](#_Toc105779022)

[29.1 Default page 347](#_Toc105779023)

[29.2 Machine Audit Mode Access 347](#_Toc105779024)

[29.3 /qcom 347](#_Toc105779025)

[29.4 /qcom/users 348](#_Toc105779026)

[29.5 /qcom/users/<username> 348](#_Toc105779027)

[29.6 /qcom/certs 349](#_Toc105779028)

[29.7 /qcom/lua 349](#_Toc105779029)

[29.8 /qcom/audit 349](#_Toc105779030)

[29.9 Event Log Access 349](#_Toc105779031)

[29.10 Other 349](#_Toc105779032)

[30 Machine Software Upgrades 351](#_Toc105779033)

[30.1 General 351](#_Toc105779034)

[30.2 Upgrade Packages 352](#_Toc105779035)

[30.3 Version and Compatibility Control 354](#_Toc105779036)

[30.4 Ethernet Based Upgrades Requirements 355](#_Toc105779037)

[30.5 USB Attached Storage Based Upgrades Requirements 356](#_Toc105779038)

[30.6 Commissioning Caveats 357](#_Toc105779039)

[30.7 Upgrade process short summary 358](#_Toc105779040)

[30.7.1 Via Ethernet 358](#_Toc105779041)

[30.7.2 Via USB 359](#_Toc105779042)

[31 Electronic Seal Support 360](#_Toc105779043)

[32 Machine Discovery Protocol 361](#_Toc105779044)

[32.1 Features & requirements 361](#_Toc105779045)

[32.2 Broadcast Message Format 362](#_Toc105779046)

[32.3 Audit mode 362](#_Toc105779047)

[32.4 QCOM 3 API Support 362](#_Toc105779048)

[33 Implementation, Development, Testing and Simulation 363](#_Toc105779049)

[33.1 Machine Implementation 363](#_Toc105779050)

[33.1.1 Getting started 364](#_Toc105779051)

[33.1.2 Summary of QCOM 3 NV data 365](#_Toc105779052)

[33.2 Machine Implementation Testing 366](#_Toc105779053)

[33.3 Development tools for System Developers 366](#_Toc105779054)

[33.4 Software Development Kit (SDK) 366](#_Toc105779055)

[34 Possible Future Functionality 368](#_Toc105779056)

[34.1 Wi-Fi Interface 368](#_Toc105779057)

[34.2 Built-in Player ID Card Readers 368](#_Toc105779058)

[34.3 Screen Capture 370](#_Toc105779059)

[34.4 Video Capture 370](#_Toc105779060)

[34.5 Streaming Video onto Machine Displays 371](#_Toc105779061)

[Appendix A – QCOM 3 Modes of Operation 373](#_Toc105779062)

[Appendix B - Designing a QCOM 3 Operating Environment 376](#_Toc105779063)

[Appendix C – ANZ Gaming Machine National Standards (GMNS) and QCOM 379](#_Toc105779064)

[Revision History 380](#_Toc105779065)

[Machine Requirements Index (MRI) 390](#_Toc105779066)

# Introduction

QCOM 3 is the successor to the QCOM v1.6 gaming machine protocol. QCOM 3 is an interface specification that facilitates the monitoring and control of gaming machines.

QCOM 3 is a highly dynamic and versatile interface specification with a broad range of potential functionality and operating modes. This will help ensure that QCOM 3 will be an attractive standard to adopt in as many jurisdictions as possible. However, it must be noted that QCOM 3's support in any given area of functionality should not be taken as an indication of any pending or future changes to the State of Queensland’s regulatory operating environments for gaming machines.

OLGR appreciates feedback to our published requirement documents and technical standards; feedback may be submitted at any time to:

[olgrnotify@justice.qld.gov.au](mailto:olgrnotify@justice.qld.gov.au)

This document is published on the internet. The latest version may be downloaded via the OLGR website:

<https://www.business.qld.gov.au/industries/hospitality-tourism-sport/liquor-gaming>

## Disclaimer and legal

The QCOM specification is the intellectual property of The State of Queensland. In order to implement the QCOM specification or subsequent versions, the necessary licensing arrangements will be required to be entered into.

OLGR provides no guarantee that implementing a product or service to the requirements or standards in this document and associated attachments will result in a product or service that does not infringe the intellectual property rights of any third parties. Organisations implementing and operating products and services conforming to OLGR requirements must perform their own due diligence checks to ensure they have obtained all the necessary licences, approvals and consents.

Please contact OLGR immediately in the event it is believed that any OLGR requirements or standards are likely to result in an implementation that infringes the intellectual property rights of any third parties.

It is not the intent of any OLGR requirement or standard to cause conforming products or standards to infringe the intellectual property rights of third parties. Any OLGR requirement or standard that is likely to result in a conforming product or service that infringes the intellectual property rights of any other parties may be used as the basis for a dispensation request against the concerned requirement or standard.

Whilst every effort has been made to ensure that the information contained in this document is accurate as at the date of issue, OLGR makes no warranty or representation whatsoever with regards to the information in this document. OLGR assumes no responsibility or liability for any consequences (financial or otherwise) suffered directly or indirectly by persons who have entered into commercial activities upon reliance on any information in this document.

The information in this document is subject to change from time to time to adapt to the continual development and evolution of the industry. OLGR reserves the right to change its policies and to amend, modify or supplement any information in this document.

Also refer to the copyright notice on the second page.

## Background

QCOM 3 is the successor to the QCOM v1.6 gaming machine network protocol specification. QCOM was created in the mid 1990’s to facilitate Queensland’s introduction of Licensed Monitoring Operators. QCOM v1 has been in use in Queensland gaming machines since the late 1990’s and has since been adopted by a number of jurisdictions across Australia and New Zealand comprising of over 80,000 gaming machines.

## Scope

This document, coupled with the QCOM 3 Summary Spreadsheet, is intended as a reference manual for QCOM 3. It also represents a set of minimum requirements for a ‘machine’ (i.e. an EGM – refer glossary).

Note that it is not possible to implement QCOM 3 in a machine without the QCOM 3 SDK, which contains the platform independent QCOM Lua Engine software driver and additional information essential to the implementation of QCOM 3.

This document does not cover system or service provider implementations of the QCOM interface specification. This will be dealt with in a separate publication.

Interpretations

This document is written for and directed at QCOM 3 machine manufacturers by default e.g. a requirement with no clear object such as *“this must be done*” should be interpreted as “*this must be done by a QCOM 3 machine*”.

While this document defines interfaces that 3rd parties use, this document contains no requirements for those 3rd parties. There are some recommendations directed at 3rd parties, but these should not be considered requirements. Requirements for 3rd parties wishing to interface to QCOM 3 machines will be contained in separate OLGR publications.

## Objectives of the new version of QCOM

The primary driving force behind the development of the new version of QCOM is the need to move the standard from the old RS-232 physical networks to higher bandwidth network technologies, make use of latest associated protocols, security and to support the latest evolving gaming technologies being developed by machine manufacturers.

It is also an opportunity to make the QCOM standard more attractive to a broader user base which will facilitate its commercial viability. The intent is to make QCOM 3 the gaming machine interface specification of choice regardless of jurisdiction, operating or regulatory environment.

## High level design requirements

The high level design requirements / mandates for QCOM 3 are:

* Make QCOM 3 the machine interface of choice irrespective of whatever regulatory or operating environment a machine is destined for. The specification must be **attractive** and useful to a diverse range of jurisdictions and regulatory environments, as well as new markets (even markets with gaming machines but currently lacking a gaming network or infrastructure).
* In support of the above, QCOM must support an extensive and diverse range of possible **network** **modes of operation** (see bottom), **regulatory environments** including varying levels of trust with respect to roles within those environments. See below for examples of modes to be supported.
* The specification must be highly **versatile, extensible and dynamic** both in design and operation (e.g. Protocols implemented by a QCOM machine must be upgradeable even after deployment).
* The specification requires an **access control system** with the ability to implement arbitrary arrangements of access control concerning machine-related service providers as required.
* The specification must be able to be **secure**, **robust** and **reliable** given the level of operating risk.
* **Multi-user/service provider support**. (Where multiple service providers can interact with the machine independently of each other.) *This feature alone is capable of increasing workload and complexity significantly. In order to mitigate the cost of delivering these demands, QCOM will attempt to utilise compatible existing technology where possible (e.g. from the machine’s OS, existing protocols, standards, methods and technology).*
* **Minimise resources** needed to implement and host QCOM 3 for QCOM machine manufacturers. *This is achieved by keeping the working set of functions to be implemented by a QCOM machine to their lowest common output functionality. This also has the benefit of minimising the amount of functionality hard-coded into a QCOM machine (and therefore difficult to change later on). The choice of scripting language used by QCOM is another factor here.*
* **Maximum functionality**. If a QCOM stakeholder/client wants specific functionality they should be able to have the functionality and not necessarily be impeded by a third party in achieving it (regulatory framework permitting). QCOM is designed such that functionality demanders would by default bear the cost of implementing their desired functionality.
* **Minimise orphan functionality, creep and bloat**. A specification that minimises the amount of functionality that has to be implemented by a machine manufacturer that won’t be applicable to ALL jurisdictions and yet still allows all desired functionality by a given stakeholder.
* Support for **independence of all services** in that the delivery of one service by a provider is not dependent upon another provider (unless intended e.g. by the regulatory framework). *This is a very significant mandate and has a profound impact on the design of the new QCOM specification.*
* The base specification must **minimise dependencies**, but allow these to be customised for each jurisdictions’ needs.
* Support QCOM machines based on either **Windows or Linux Operating Systems** i.e. only API’s, protocols and standards that are readily available to both operating systems will be utilised.
* Network protocols utilised will be based on **Ethernet & TCP/IP based network protocols**.
* **Utilise existing**, **free**, **open** and established protocols, API’s, libraries and concepts whenever possible as an alternative to developing a new proprietary methodology for any given task.
* The ability to speak **multiple protocols** and the ability to change those protocols without having to upgrade machine firmware. The only protocols that should be hardcoded into the machine are those that are already well established standards and have a proven industry track record. As most existing and expected QCOM machines at the present time are not upgradeable over a network, the choice of protocols hard-coded into the machine is a crucial decision from a security and integrity perspective.
* Support for strong **authentication and encryption**.
* When a machine manufacturer has implemented QCOM 3, the machine can be shipped to any jurisdiction in the world without the machine manufacturer requiring a version of machine software for each jurisdiction and that any **localisation** can be fully implemented via QCOM. *(This is a long term goal.)*
* The primary machine applications of a QCOM machine must be able to remain closed source. *This affects the choice of API’s and libraries used by the specification.*
* The hardcoded factory default network interfaces should be able to be driven by either a human using freely available tools or via networked computer system application.

Network modes of operation to be supported:

Stand-alone (i.e. intermittent or no network)

A QCOM 3 machine at commissioning time must be able to be configured to operate without any subsequent or ongoing access to a monitoring and control system or network. This isolated mode of operation must have intelligence. For example, a QCOM 3 machine may be able be set up to automatically purge events and enable/disable according to licensed gaming hours.

Networked Environments

Support networks with high available bandwidth, or networks with limited available or costly bandwidth (e.g. support machines networked on a LAN or a WAN).

Support Multi-user / service provider / role environment with configurable roles and privileges (e.g. multiple service providers can send credit to a QCOM machine).

Support operating environments with multiple operators / service providers.

Refer section 1.7 for more information on supported operating modes.

## Changes of significance

Significant changes in QCOM 3 with respect to the previous version of QCOM are:

**Physical network:**

While QCOM v1 was an RS-232 based network protocol, QCOM 3’s utilised network protocols are primarily based on existing established Ethernet and TCP/IP-based protocols.

While QCOM v1 was designed to be very efficient due to the limited available bandwidth at the time (19.2k baud rate), there is less concern about bandwidth usage with respect to QCOM 3.

**Protocols**

QCOM v1 was, excluding the physical layer, a proprietary network protocol, where QCOM 3 primarily uses open standard and hardened network protocols whenever possible (that is, when a protocol is mandated).

**Multi-master support:**

Aka support for multiple ‘service providers’[[1]](#footnote-1) interacting each QCOM 3 machine.

Where QCOM v1 was essentially a single host protocol (in that all services were designed to be delivered by a single entity), QCOM 3 is designed to cater for multiple service providers interacting with the machine at the same time. QCOM 3 also gives special consideration to ensure independence of services with the optional ability to create dependencies if desired.

**Type of specification:**

QCOM 3 also has one very significant change when compared with the former version of QCOM. While QCOM v1 was a network protocol specification, **QCOM 3 is not a network protocol specification**. QCOM 3 at its core is predominantly a specification for an Application Programming Interface (API)\*. QCOM 3 could also be described as delivering a “protocol development framework”. Accordingly QCOM 3 labels itself as an “Interface Specification” rather than a “protocol”.

*\*QCOM 3 essentially defines a machine hosted API and scripting engine hosted within the machine. The benefits and opportunities as a result of this change are very significant and are discussed in more detail in the following section.*

From the protocol perspective, a QCOM 3 machine is multilingual and will be capable of implementing a wide range of (IP based) protocols.

One notable reason for making QCOM 3 an API based interface specification was because everything external to the machine (e.g. the network, the regulatory environment, demanded services and functionality) was considered highly volatile, impossible to predict and highly subject to change over time. Thus development of a machine hard-coded application level network protocol would likely only be attractive to a small number of jurisdictions.

## Benefits of QCOM 3

QCOM 3 incorporates all the high level benefits of existing application level network protocols (such as QCOM v1). For example:

* Machine manufacturers and system/service developers only have to implement a single specification for interconnectivity.
* A precisely defined standard for communication with gaming machines.

New benefits with QCOM 3 are:

* Multi-master aka multi-user (multi-service provider) support;
* Strong encryption and authentication as required;
* A huge range of possible network and regulatory operating modes;
* Significantly lower cost and ability to be able to make major changes to the operating mode (when compared with hard-coded protocol solutions);
* Independence of services (as desired);
* Execution of signed scripts by proxy. (QCOM 3 Users can implement functionality by proxy via other users loading and executing scripts on their behalf. This is primarily how new users are introduced to a QCOM machine and has many applications);
* A QCOM 3 machine is multilingual from a network protocol perspective; a QCOM 3 machine will basically be able to implement whatever (IP based) protocol/s are programmed into it and that program may be changed on-the-fly.
  + Service providers interfacing to a QCOM machine can utilise the network protocols most suited to their business.
  + QCOM 3 machine can be configured to speak legacy (IP based) protocols in order to integrate into older systems.
* Machine manufacturers are not burdened by the large amount of functionality and modes of operation a QCOM 3 machine can support. All the machine manufacturer has to implement is the QCOM Lua Engine (s10) and the QCOM API (s10.5). The remainder of QCOM 3 related / network functionality is provided by those demanding it as QCOM users (s5);
* QCOM 3 is able to implement additional functionality on-the-fly as demanded by stakeholders;

* A QCOM 3 machine is able to move between major modes of operation (discussed in more detail in the following sections) without upgrading the machine firmware or software or having to perform a full machine factory reset;
* Once a service provider has implemented a service as a QCOM user, their service can be downloaded into a QCOM 3 machine and operate without reprogramming the QCOM 3 machine or affecting other users and their services. The service can be upgraded on demand to address issues or add new functionality;
* QCOM 3’s design is intended to promote a market for third party implemented products and services that operate on QCOM 3 machines via the *QCOM 3 scripting engine*;
* Former application level network protocols (like QCOM v1 for example) were problematic in that they hard-code application / network level behaviour into the machines. This often gets challenged and broken by new ideas and concepts, or by network topology changes over time and is typically very costly to upgrade. QCOM 3 minimises the amount of functionality that will end up being hard-coded into machine firmware and thus is cheaper to implement compared to network protocols of similar functionality;
* Even with next generation machines, which should allow remote upgrading of their software, service providers, operators and regulators are still totally dependent on the machine manufacturer in order to implement new or changed network protocol functionality. However with QCOM 3, application level functionality is primarily delivered on-the-fly by the organisations who are also demanding the functionality and thus there is no impediment to change as demanded by those organisations;
* As will be explained in later sections, a proportion of machine monitoring and control related functionality in a QCOM 3 machine comes from scripts downloaded to the machines, if changes are required, it will be inexpensive to implement (*as* *compared to the cost implementing changes in a hard-coded machine protocol / interface).*
* The primary interface in QCOM 3, the QCOM API, is not directly exposed to the network. Protocols that are exposed on the network will be implemented by the QCOM API and are easily and securely able to modified remotely;
* New monitoring and control related services and functionality can actually be implemented within QCOM 3 machines and could potentially save on having to install additional hardware at gaming venues. A common scenario seen here is the installation of third party interface boards in gaming machines, which is always a costly solution. QCOM 3 has the potential to eliminate the need to install interfacing boards in gaming machines as the services provided by the interface boards can be uploaded and executed by the gaming machine via the QCOM 3 scripting engine;

Examples of some applications and services which will benefit from being run within the EGM utilising the QCOM API and scripting support may be found in section 10.12;

* QCOM 3’s major change from being a network protocol specification to being primarily an API utilised by scripts downloaded to the machine and executed in a jail within a script engine virtual machine, provides for the greatest range of operating modes possible in a monitoring and control specification. A QCOM 3 machine should be able to function aptly in any known regulatory or operating environment;

Also refer to section 1.7.1 for additional benefits.

QCOM 3 operating modes (also referred to as *modes of operation*), including benefits:

* Standalone (no ongoing / persistent network connection)

This mode of operation is supported by the ability to setup a QCOM 3 machine with a high degree of independence via a set of once-only downloaded control scripts that execute autonomously on the machine (these scripts can either be setup at commissioning or out of the factory).

Network protocols can subsequently be added to the machine without having to upgrade the machine’s firmware.

This feature makes QCOM 3 machines attractive to new gaming markets currently without a network infrastructure or monitoring system. If or when the market does adopt a network or monitoring infrastructure, the machines will be ready to integrate into them.

It could also potentially be used to implement a disaster recovery solution e.g. in the event of an extended network outage the machines could be reconfigured to be able to be switched back into a standalone mode in order to continue operation.

* Variable available bandwidth networks (e.g. WANs)

A QCOM machine can operate in network environments which have intermittent, limited, or costly bandwidth. This is because network protocols are not hardcoded into a QCOM 3 machine, arbitrary protocols can be implemented by each service provider to meet their business needs and operating environment, thus more network efficient methods can be utilised in bandwidth limited environments.

QCOM 3 is also able to function well in networks with ongoing severe availability issues. As QCOM 3 machines are protocol multilingual, the most suitable protocol can always be used with respect to the available bandwidth of the network and operating environment, as well as being able to download monitoring and control related functionality and services into the local machine thus making that functionality immune to network outages.

* Support for multiple service providers for different services

QCOM 3 is designed so that service providers can operate independently from each other without interference if desired. Optionally, arbitrary methods of access control and dependencies can be created if need be.

Refer to the appendices for more examples and case studies concerning QCOM Modes of Operation.

### Design note: Meeting diverse operating requirements

QCOM 3 is designed to be adopted across a very wide range of jurisdictions demanding a very diverse range of operating modes and requirements. In design, this is very difficult to accommodate since only a small number of jurisdiction’s specific needs and requirements are known to OLGR. Requirements can also be contentious or not complementary with each other. Accordingly, trying to satisfy every possible jurisdictional or stakeholder requirement would be impossible in a conventional hard-coded protocol specification, or result in a very bloated protocol specification at best.

The solution to this problem adopted by QCOM 3 is to make the specification an API based interface. This is achieved via the addition of a machine hosted scripting engine (a process virtual machine), meaning a significant amount of monitoring and control related functionality in a QCOM 3 machine is only implemented when it is actually demanded and is implemented by the demanding party to meet their exact requirements.

Basically, a QCOM 3 machine contains a number of programmable virtual environments, each able to implement arbitrary network protocols and monitoring and control related functionality as demanded. Specific benefits of this are:

* Machine manufacturers do not spend resources implementing a very large functionality set to cater for multiple jurisdictions or implementing functionality that is seldom or sparsely used.
* Jurisdictions are never locked into any specific behaviour or functionality set.
* Each jurisdiction can have any functionality it requires from the QCOM interface specification.
* The solution makes service providers needing to interface to QCOM machines much more in control of the functionality they demand.
* No locked in functionality; most issues can be easily addressed and new functionality added or removed on demand.
* Service providers needing to interface to QCOM machines are not necessarily dependent on a third party for new functionality.
* Minimises the risk of orphaned functionality being created and functionality bloat over time.
* Future-proofs the specification against major infrastructure changes.
* Systems providers have direct control over how a QCOM machine communicates with their system.
* Scripted and brute force testing is far more easily achievable with an API centred interface specification than compared to a protocol.

Traditionally in an EGM protocol like QCOM v1, there may be a control parameter where the end result is that the machine would disable or enable. This control parameter may not be required in some jurisdictions and other jurisdictions there may also be additional special requirements or functionality to be implemented. In light of the final outcome (or common functionality) being to disable or enable the EGM, QCOM 3 makes the original control parameter out-of-scope and simply caters for only the lowest common needed functionality (the ability to enable/disable the machine).

For example, QCOM 3 does not provide a communications timeout function. In QCOM 3 a communications timeout function is implemented as a script running inside the EGM operating according to each jurisdictions preferred specifications. In this way the EGM manufacturer is not burdened with implementation of a feature, or multiple possible variations there-of that may not be required in all cases.

Generally speaking, if a range of potential QCOM control parameters would result in a common task, then typically under QCOM 3 only the common task is required to be implemented by the EGM manufacturer under this specification and all possible functionality surrounding that task is implemented by the party demanding the functionality via scripts executed within a VM.

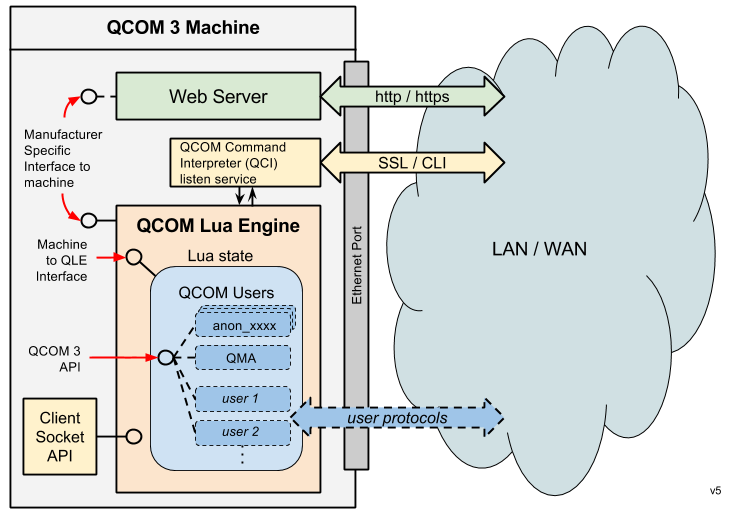
Another example relates to the QCOM API function which allows a permitted user to display an arbitrary text message on the machine’s display. Now different jurisdictions may have very competing and different requirements as to the possible use of this function e.g.:

* Single / multi-service provider function access to the function, and/or
* Text message content being controlled or uncontrolled.

With QCOM 3, it is possible to add / extend / overload the existing QCOM API with a new interface that can provide multi-user support, or controlled or varying behaviour, even restricting access to the former / raw API function. In this way the machine manufacturer is not burdened with trying to implement functionality which caters for every possible required variation and each jurisdiction gets exactly the behaviour they desire.

To summarise; in QCOM 3 because a wide range of high level functionality is not hard-coded into the machine, this functionality can be changed on demand and can always be implemented to the demanding party’s requirements, even in rapidly changing operating environments.

## QCOM 3 Machine – Main software components



Notes:

* The Web Server is an optional component. Refer section 29.
* The QCOM Command Interpreter (QCI) listen service relates to section sections 23 and 24. The machine must provide connection management and message forwarding to the QCOM Lua Engine (QLE). The QLE contains a built-in command interpreter as a part of the QCOM 3 SDK Lua source code which must be used to process the command line messages.
* The QLE hosts the QCOM Lua state the core component of QCOM 3. The machine must provide this hosting process / thread and code. The QCOM 3 SDK comes with full demo “C” source code, for reference purposes and the QLE Lua software driver. Refer to the QCOM 3 SDK for more information.
* The Lua state within the QLE must be implemented by the mandatory Lua source code driver modules from the QCOM 3 SDK. cp:[[2]](#endnote-1) This Lua component is by far the bulk of the code involved in implementing QCOM 3 in a machine and is provided like a driver in source form (Lua) as a part of the QCOM 3 SDK. Refer to the QCOM 3 SDK for more information.
* Finally, as a part of the QLE the host machine must integrate an SSL capable client socket API into the QCOM Lua engine. It is used indirectly by QCOM users to implement any desired protocols. Refer section 12 for more information.
* As of QCOM 3 draft 3, the QLE no longer mandates any special watchdog component. The need for a watchdog has been made redundant by the Lua instruction Quota (s5.5). The QLE Lua State also emits an acknowledgement message in response to a one sec timer dispatch message sent to it which indicates the Lua state and QLE is still functioning.

# Glossary

**Ante-bet Game/s**

In this document the term *ante-bet game* refers to games in gaming machines which have one or more game features that are only unlocked if the player bets above a minimum threshold amount which is above the game’s minimum bet.

For example, a game whereby it is only possible to win the free game feature or a specific prize or jackpot if the player bets max.

**Audit Mode**

Refer to the latest version of the Gaming Machine National Standards for EGMs.

**btable**

A way of implementing **sets** in Lua. A *btable* type denotes a Lua variable of base type *table* where all the table values are of type *boolean* and equal to *true*. The table keys indirectly represent the boolean value by whether or not they actually exist in the table.

Lua Example:

-- define a ‘btable’

IsPresent = {DeviceA = true, DeviceC = true}

-- usage:

if (IsPresent[“DeviceA”]) then print(“yes”) else print(“no”) end – prints “yes”

if (IsPresent[“DeviceOther”]) then print(“yes”) else print(“no”) end – prints “no”

-- alternatively:

if (IsPresent.DeviceC) then print(“yes”) else print(“no”) end – prints “yes”

*Refer PiL section 11.5 (www.lua.org/pil/11.5.html)*

**Cancel Credit**

Means a manual payment or ‘hand pay’.

**Chunk**

Refer Lua reference manual.

**Design Requirement**

‘Design Requirements’ appearing through the document are for specification development purposes only. They will be ultimately deleted from the final specification or moved to a stand-alone document.

They represent a checklist for the designer used to ensure that what is a complex interlinked specification ultimately functions and meets all intended specifications.

**Device**

Refers to any embedded computer system such as a gaming machine.

**Event**

This term when used in the context of ‘logging an event’ or similar, means an event re section 13. However if the text reads ‘state event’, then this is always a reference to a state event as defined in section 14.

**External Jackpot/s**

This refers to any type of jackpot or prize that can be won on an EGM for which the EGM does not trigger as a part of any of its games i.e. the jackpot is triggered by a device ‘external’ to the EGM.

**Game**

1. To play; refer “Play” definition below. 2. Refer GMNS.

**Global Type** (table of)

Refers to the QCOM Summary Spreadsheet – “Global Types” sheet where all the definitions of QCOM 3 global types are located.

**Hardcoded**

In this document the definition of hardcoded is not as per its general definition with respect to software. In this document hardcoded currently refers to software that is only able to be changed via a secure software upgrade as per the requirements in section 30.

**Host machine**

Refers to the machine that hosts the QLE. In other words, the machine that implements QCOM 3. Example: a QCOM 3 gaming machine (or EGM) is the host machine referred to in this document.

Host machine state (hms)

A Lua serialised table as a string, built by the QCOM 3 machine on each boot up and sent into the QLE LSD to initialise the machine’s state within the QLE LSD just before the QLE starts proper. The hms table defines the machine’s state at the time that is of interest to the QLE. Before accepting the data, the QLE LSD applies a schema to the table defined in QCOM 3 SDK *lua\hms\_schema.lua*.

**“i” button**

A player / attendant accessible “information” button available on a machine. Refer 3.7 for more information.

**Idle mode**

Refer QCOM v1’s definition of idle mode and also the ‘General Status Response’. In QCOM 3 the following API functions are related:

egmState()

egmDoorsClosed()

egmDoorState()

egmInFault()

egmInAuditMode()

egmInTestMode()

egmOK()

**Jackpot**

The term ‘jackpot’ refers to any amount won or ‘prize’ in machine gaming or jackpot systems. Not necessarily a special, labelled as such, or top prize. In this document the term ‘jackpot’ also is synonymous with the term ‘prize’.

**Lua**

Lua is a scripting language. Refer <http://www.lua.org/>

**Machine**

QCOM 3 is an interface specification intended to be implemented by a ‘machine’. Accordingly, most of requirements in this document apply to a ‘machine’. Typically, ‘machines’ are gaming machines. However, in the longer term the scope of the QCOM specification may be made broader to a range of gaming devices, or computer / embedded systems in general. Thus, the generic term ‘machine’ is used in the document unless it is specifically intended to refer to a gaming machine.

**Meter**

*Without other context ‘Meter’ in this document refers to the NV meters used in modern Gaming Machines as defined in GMNS. They are often referred to as ‘software meters’ to distinguish them from older pulsed-based electro-mechanical meters used in older gaming machines and devices.*

**NV Memory**

Non-Volatile Memory. Memory in a machine or device whose state is preserved over machine restarts and power interruptions.

**Persistent Variable (PV)**

Refer section 10.10.

**Play**

A play refers to the sequence of events on an EGM initiated by a player through an irreversible wagering of credit and ends when the play show is over, the results of the play revealed to the player and either the wager has been lost or winnings become payable (i.e. winnings are able to be transferred to the EGMs total win meter). Free games, free spins, bonus features and double up/gamble, are considered to be a part of a single play. A play is considered finished upon return to idle mode.

**Prize**

The term ‘prize’ refers to any amount won in machine gaming or jackpot systems. In this document ‘prize’ also is synonymous with the term ‘jackpot’.

**QCOM Lua Engine**

Refers to the process/thread and any supporting threads that host the QLE Lua software driver. Refer section 10 for more information.

**QCOM User/s**

A term used throughout this document to refer to any party that wishes to interact with a QCOM 3 machine over a network. For example any gaming machine related service provider (see *Service Providers* below). They are termed ‘*users’* because they essentially have a login / account in the machine, a bit like a user in an operating system but very strictly jailed-in. QCOM users are introduced in section 5.

**QLE Lua software driver**

This refers to the Lua source code component of the overall QCOM Lua Engine. The QLE Lua software driver (that the machine must use\*) is provided via the QCOM 3 SDK. \*It is a core requirement that the machine must utilise the QLE Lua software driver modules; refer s10.1.)

**QoS**

Quality of Service

**RAM Clear**

The process of resetting a machine to factory defaults.

In this document, the term ‘RAM clear’ is equivalent to the term ‘factory reset’ and infers the machine is being reset to factory defaults. Two exceptions are the QCOM API function **qcom.machineRAMclear()** and any RAM clear that may be performed by the machine as a result of a machine software upgrade (s30) which must preserve a few things. Refer to the function description for **qcom.machineRAMclear()** to see what must be preserved.

**Return To Player**

Unless specified otherwise, RTP is represented as a percentage in this protocol (0%...100%). RTP refers to the theoretical return to player of a game or component of a game e.g. an 85% game would mean that on average for every dollar bet in the game the average prize (including prizes of zero) would be 85 cents.

**Service Providers**

**Services**

In support of the definition of ‘service provider’ below are some examples of ‘**services**’ related to gaming and deliverable by ‘Service Providers’:

* Auditing (AUDIT)
* Monitoring (MON) e.g. Queensland LMOs
* Venue Performance and Analysis (PERF)
* Configuration and Control (CONF)
* Player Loyalty Systems (PLS)
* Card Based Gaming (CBG)
* Ticket-In / Ticket-Out (TITO)
* Attendant Paging Systems (APS)
* Jackpot Display Systems (JDS)
* Jackpot Totalisation
* Jackpot Operator
* Jackpot Broadcaster
* Regulatory Enforcer
* Service consolidator
* Service router
* Interface provider
* Credit Redemption Manager (CRM) Refer section 22.

**State event**

Refer section 14.

**Test Mode**

Refer to the latest version of the Gaming Machine National Standards for EGMs.

**Turnover**

Means a total or partial total of bets made previously on an EGM.

**UAA Service**

Refer section 23

**User**

Refer section 5

**Variation**

Refers to a specific percentage return to player of a game in an EGM. In QCOM, each game in an EGM may have one or more game variations. Each variation in a game typically has a different RTP.

**Virtual Machine**

Means a ‘process virtual machine’. (Not to be confused with a ‘system virtual machine’.) Refer <http://en.wikipedia.org/wiki/Virtual_machine>

e.g. a Java virtual machine or Lua virtual machine.

## Acronyms and abbreviations

ANZ Australia and New Zealand

API Application Programming Interface

AU Australia

CEO Refers to a delegated authority within OLGR

CPU Central Processing Unit

DS Digital Signature

DSA Digital Signature Algorithm

DST Daylight Savings Time

ECT Electronic Credit Transfer

EGM Electronic Gaming Machine (Poker Machine)

GMNS A/NZ Gaming Machine National Standards Document for EGMs

GUI Graphical User Interface

ID Identification

LAN Local Area Network

LP Linked Progressive

LSD Lua Software Driver (refer glossary entry: QLE Lua software driver)

LSR Line Status Register (UART related)

MAC Message Authentication Code

NA Not Applicable

NTP Network Time Protocol

NV Non-Volatile

OLGR Office of Liquor and Gaming Regulation (Queensland, AU)

PID Player Information Display (Refer GMNS for more information)

PiL Programming In Lua, 1st edition (free eBook)

PK Private Key

PRNG Pseudo Random Number Generator

PSD Program Storage Device

QCOM Queensland Local Area EGM Communications Protocol

QLE QCOM Lua Engine

QMA QCOM Master Authority

RAM Random Access Memory

RCR Residual Credit Removal

RCRF Residual Credit Removal Feature

RNG Random Number Generator

RTP Return To Player

RSA Rivest-Shamir-Adelman Encryption

RTC Real Time Clock

RUGM Remotely Upgradeable Gaming Machine

SAA Script Approval Authority (refer section 6.3)

SAP Stand Alone Progressive

SDK Software Development Kit

SFTP SSH File Transfer Protocol (This is not FTP tunnelling over SSH)

SUA Software Upgrade Authority

TBA To Be Announced

TITO Cash Ticket In/Out

UART Universal Asynchronous Receiver-Transmitter

UDP User Datagram Protocol

UI User Interface

URL Uniform Resource Locator

VM Process Virtual Machine (refer glossary)

VPN Virtual Private Network

XOR Exclusive OR

## Interpretations

The word ‘must’ appearing in this document means the encompassing requirement or standard is mandatory in order to obtain compliance under the QCOM standard and approval under relevant legislation.

The word ‘should’ means ‘recommended’ or ‘preferred’ depending on context but is ultimately optional

The word ‘printable’ when used in relation to ASCII characters means as per the C isprint() function unless stated otherwise i.e. ASCII codes 32 …126.

The term ‘valid Lua identifier’ means any string comprised of only letters, digits and underscores that does not begin with a digit.

Please report all instances of potential subjective standards and requirements contained within this document to OLGR. The CEO OLGR reserves the right to arbitrate an interpretation of any subjective requirements contained in this specification in relation to any approvals under related Queensland legislation.

Endnotes throughout this document of the format: cp:x are used to designate QCOM 3 requirements that machines must implement.

In this document, “state events” (refer section 14) will always be referred to as “state events”; any other mention of an event in the context of a buffered or logged event is a reference to a machine event under section 13.

# General

## Machine hardware requirements

cp:[[3]](#endnote-2)

* One Ethernet port (IEEE 802.3), minimum speed of 100Mbps. An ESD protected port is preferred but is not mandatory.
* Two (min) RS-232 serial (uart) ports. Related: s11.36.
* Machine hardware must support software upgrades without seal-break technology according to the requirements in this document. Refer section 30 for more information.
  + At a minimum, the machine will require sufficient persistent storage space to store a full replacement software upgrade (in the background) in addition to its current working software set.
  + One specific possible\* hardware requirement here is that for a machine at factory defaults, all resident software which may be executed up until the machine’s first software upgrade must be stored within removable PSDs[[4]](#footnote-2). It is recommended that the applicable PSDs here are also able to be verified using widely available and cheap readers.
    - In simpler terms; where possible, ensure the machine’s BIOS (or the equivalent) is socketed along with all other PSDs in the machine.
    - \*This requirement may be made mandatory for machines in high risk (prize) scenerios in the opinion of the applicable regulator e.g. trigger jackpots >$1m.
* Power off detection of processor door access (all other doors are optional) as per QCOM v1.x requirements.
* Support for OLGR Electronic Seal Minimum Requirements is **optional** unless specifically mandated for a given range of machines by a regulatory authority. (Most EGM’s level of operating risk is not high enough to justify making an Electronic Seal mandatory for all EGMs)
  + Electronic Seal support is the ability for the machine to detect access to the machine’s logic area even when the machine does not have mains power for a time, including the ability for the machine to immediately erase a small NV data store (holding a secret key) whenever access is detected. For more information refer to OLGR Electronic Seal Minimum Requirements publication and section 31 of this document.
* 1 x Non-Volatile Real-Time Clock for persistent timekeeping across power disruptions.
  + The RTC battery or equivalent must be able to provide power for at least **3** months while the machine is disconnected from mains power and have a minimum operational life of **7** years.
* Power management functions:
  + Shutdown – desirable[[5]](#footnote-3). (Related: QCOM API function machineQueueShutdown())
  + Reboot (Related: QCOM API function machineQueueReboot() )

Memory and storage minimum requirements:

***Please note that memory and storage requirements values are subject to change until the later stages of the QCOM 3 project implementation phase.***

* Refer section5.4 for **RAM** requirements relating to QCOM users.
* Refer section 5.3 for **storage** requirements pertaining to QCOM user account storage space. NB Write cycle limited memory (e.g. flash memory) is acceptable for user account storage (i.e. script storage).
* Refer QCOM 3 SDK /lua sub-directory for the schema related for storing general QCOM user data (per QCOM user) in **NV memory**.
* Refer section 10.10 for **NV** **memory** requirements for QCOM Persistent Variable storage. Fast r/w NV memory is recommended here as updates to PV’s may occur most frequently.
* Refer section 10.11 - User Created Meters for **NV** **memory** requirements for QCOM user Created Meters. (This feature is currently now a concept only – do not implement)
* Refer section 13.1 for **NV memory** requirements pertaining to QCOM event storage.
* Expected **RAM** requirements pertaining to the QCOM Lua Engine is expected to approximately in the order of a few megabytes.
* Web server disk/flash memory requirements estimate is TBA.

Related: section 33.1.2 - Summary of QCOM 3 NV data

If a machine does intend to utilise flash-based memory device in a r/w configuration for QCOM 3 storage, then the type used must be suitable for embedded system use i.e.:

* it must be fault tolerant to unexpected power interruptions (especially during any write operations)e.g. some older generation SSD devices were at risk of “bricking” or a total loss of all data if interrupted during a write operation.
* power interruptions during a write operation must be automatically & gracefully managed. After a restart, any data that was being written must not result in it being either fragmented or corrupted.
* automatic management of bad sectors.
* the overall MTBF and typical write cycles before failure are suiltable wrt its intended use in the machine.
* near end-of-life detection must be implemented if there is a reasonable risk that this could occur during the life of the machine e.g. if its is used for QCOM 3 Persistent Variable storage (s10.11) would be the most likely to cause this.
* Un-cached writes, or otherwise the ability to force the write cache to be written on demand.

Possible / under consideration:

* Provision for one IEEE 802.11 Wi-Fi based network adaptor (optional). Refer section 34.1

Recommended (but optional) hardware to consider:

* Additional internal power ports. Individually mains sourced and protected. Such as:
  + 1x spare mains power port.
  + Multiple spare USB power only ports.
* A hardware based true RNG.

## Machine software requirements

Operating System

A multi-tasking Operating System will be required that can implement the range of protocols and services listed below such as Windows based or Linux based Operating Systems.

Other operating systems that have similar functionality to the above should also be acceptable provided the OS supports protected mode multi-tasking and all the software requirements listed in this section.

Required Network Protocols

* Ethernet (IEEE 802.3)
* IPv4
* DHCP
* IPv6
* SLAAC (RFC 4862)
* HTTP
* TCP
* SSL
* UDP
* ICMP echo (*Ping*) cp:[[6]](#endnote-3)

Services

* Web Server (refer section 29 - optional)
* UAA listen service (refer section 23 - mandatory)

Other

* OpenSSL (QCOM’s secure communications API is based on openSSL)
* Lua interpreter (aka the QCOM Lua Engine)

## Other

### RNG

The machine must utilise an industry recognised cryptographically secure RNG for use with QCOM 3 required random numbers and entropy. cp:[[7]](#endnote-4)

QCOM 3 required random numbers pertain to:

* Key generation (certificates, SSL/TLS connections)
* QCOM API function: qcom\_machineRand()

Recommended RNG properties:

* The RNG does not trust a single source of entropy. A combination of a strong PRNG and entropy gathered from multiple hardware based sources is considered good practice.
* There must be detection of entropy starvation; any call to the RNG must block/wait if there is insufficient entropy. (The PRNG component must be excluded from these entropy estimates.)
* The PRNG component must be seeded from hardware based entropy from multiple sources.
* The RNG should not have to save its state to machine NV memory.
* If the machine has a high demand for entropy (such as a gaming machine), then a solid state hardware based RNG device must be incorporated into the machine in order to help prevent entropy starvation.

For Linux based QCOM 3 machines, /dev/random (and API’s using it) is acceptable provided the distro/kernel version is fairly current.

For Windows based machines, the Windows Crypto API’s RNG and secure socket API is acceptable provided the version of windows used is still supported.

Tip: Both Linux and Windows OS’s understand that applications running on it may have other sources of entropy and thus allow applications to inject entropy into the OS for which the OS will hash into its entropy pool. For example, in a gaming machines there are a number of I/O devices such as buttons, hoppers, banknote acceptors, touch screens etc, for which at least event time based entropy can be gathered by the EGM’s application and injected into the OS.

### Machine RAM errors

(Same requirements as per QCOM v1)

1. EGMs operating under this specification must display the local date & time and source of a RAM error condition on the machine’s primary display when a RAM error occurs in the machine e.g. “Self Audit Failure”, “Meter Corruption”, “Event Queue Corruption”, “RNG State Corrupted”, etc. cp:[[8]](#endnote-5)
2. EGM meters in audit mode must be able to be displayed on the EGM in a RAM error condition. cp:[[9]](#endnote-6)
3. Upon the occurrence of any RAM corruption, program mismatch or fault condition requiring a RAM clear to reset, the machine must immediately stop the QLE and all associated communications. This is to prevent the machine from transmitting any corrupt information while it resides in any of these error conditions. cp:[[10]](#endnote-7)

## Required knowledge

#Prerequisite knowledge

This section lists the required knowledge to be able to fully understand, implement or test QCOM 3:

* Secure hashing algorithms
* Public key cryptography principles
* X509 certificates
* SSL / TLS
* TCP / IP
* PEM file format
* [OpenSSL command line tool]
* Linux or Windows Operating System
* Lua
  + - Suggested reading:
      * Lua reference manual
      * ‘Programming In Lua’ book (PiL)[[11]](#footnote-4)
* Gaming machine operation, principles and development
* QCOM Protocol v1.x
* JSON file format
* Also refer the above hardware and software requirements lists

## The QCOM Local Area Network

The QCOM 3 machine local area network is also referred to in this document as the gaming machine LAN.

This section is specific to Queensland Clubs and Hotel market and will eventually become a separate standard.

### General principles

The principles in this section are specific to the Queensland Clubs and Hotel market.

In the Queensland Clubs and Hotel market, licensed gaming venues must **own** their gaming machine LAN infrastructure, such as cabling, switches and router hardware. This QCOM 3 LAN infrastructure must be off-the-shelf, widely commercially available hardware intended for use generic office business LAN use. In other words, the gaming venue’s LAN hardware and firmware infrastructure must not have to change if the venue changes their LMO for example.

It is acceptable that WAN modems or similar hardware located at the gaming venue be proprietary devices and also not necessarily owned by the gaming venue. The QCOM 3 LAN must remain intact and functional if the WAN modem or similar hardware is disconnected.

OLGR does not consider the gaming machine LAN infrastructure (see above) as regulated gaming equipment. However, once the LAN is connected to gaming machines, then accessing the LAN in a criminal or malicious way or with intent thereof could be seen as *unlawful interference with gaming equipment* (s291) under the Gaming Machine Act 1991, or as general computer related hacking or similar criminal behaviour under general wider IT / computer related legislation.

OLGR may declare specific QCOM LAN equipment as regulated gaming equipment if OLGR sees the equipment is significant to the security, integrity or availability of gaming.

OLGR is open to considering proposals for the use of secured wireless networks (wifi) for the QCOM 3 LAN, especially in situations where this can save on costs.

In relation to copper-based networks such as Ethernet, gaming venues should be made aware that this could result in their gaming machines being more vulnerable to localised lightning strikes and to take any precautions they deem necessary in order to protect their investment in gaming-related products. *(In comparison, the old QCOM v1 LANs were fibre optic based and this arguably has provided over the years a degree of additional protection to gaming machines with respect to surges and spikes caused for example by lightning strikes.)*

Queensland LMOs may offer value added services such as QCOM 3 LAN quality of service and intrusion monitoring.

The physical QCOM 3 LAN must not be easily accessible by the general public i.e. a member of the public must have to enter an area they are not permitted to enter in order to gain access to any physical part of a QCOM LAN. The level of physical security here is up to the licensed gaming venue. (As QCOM 3 supports strong encryption, the risk here mostly pertains to availability of gaming.)

Once local physical access to the gaming machine LAN has been obtained there must be no further impediment in connecting to gaming machines on the LAN from any point on the LAN. (Contact OLGR if the gaming machine LAN is so large that it is considered necessary to break it into subnets or related proposals.)

In the Queensland clubs and hotel market, it is acceptable for QCOM 3 machines to share the LAN with other services and equipment as long as they are gaming related. For example in Queensland clubs and hotels non-LMO based player loyalty system are permissible. Gaming venues must only have to install and maintain a single physical gaming machine LAN.

If the gaming venue or LMO requires power over LAN, this must be to an accepted industry open standard.

### Machine-to-machine communications

In Queensland gaming venues, OLGR has no objection to QCOM gaming machines communicating with other local machines in order to deliver new gaming features (one example here is synchronised animations and game features). However when doing this, machine manufacturers must meet all applicable legislation and obtain necessary approvals.

In order to facilitate this, QCOM 3 machines located in physical proximity\* to each other must be visible\*\* to each other on the local network.

\* Such as in the same room but as venue wide as convenient.

\*\*i.e. ping reachable & local LAN broadcasts by one machine are visible by all.

### Replacing a QCOM v1 machine with a QCOM 3 machine

When replacing a QCOM v1 with a QCOM 3 machine, the gaming venue should retain the old fibre optic interface card from the QCOM v1 machine (if the new machine does not have one) and install it inside, or in close proximity to the QCOM 3 machine (e.g., in the cashbox) if the venue needs to keep the old QCOM v1 fibre LAN operative for any remaining QCOM v1 machines at the venue. Also, if the QCOM 3 machine if it is going to be running the qcom16 app over serial as an interim measure, then it will still need a fibre optic interface card.

Gaming venues should check with their monitoring operator for advice here.

## Standard sounds and alarms

**Concept only; do not implement this section at this time.**

This section is a proposal relating to the standardisation of certain QCOM mandated sounds.

QCOM 3 (as per QCOM v1) requires the machine to emit a sound upon specific QCOM specific operations. Namely:

* [ECT to Credit Meter (e.g. “ka-ching” like sound with a rising terminal)]
* [ECT from Credit Meter (e.g. “ka-ching” like sound with a falling terminal)]
* System Lockup – *alertsound* flag (e.g. “ding”)
* System Lockup – *fanfaresound* flag (e.g. “tada”)
* System Lockup – *cashoutsound* flag (e.g. “ka-ching”)
* SPAM - qcom\_egmSPAMA() & qcom\_egmSPAMB()
* GPM - qcom\_egmGPM()
* Key-switch Disabled (refer QCOM API)

It is proposed in the long term to make the above listed QCOM mandated sounds standardised. (Excluding [] bracketed sounds.) This uniformity will make it easier for a player/user to understand what is happening across all makes and models of QCOM machines.

Feedback on this concept may be provided at any time.

*Related: There is also the possibility of the ability to being able to download custom sounds to the machine in future versions of QCOM as well as the addition of a qcom\_playsound() QCOM API function.*

## “i” button

The majority of ANZ gaming machines currently incorporate an information button, or “i” button on its GUI. This button when activated currently provides the player / attendant with access to a sub-menu which gives access to information such as on-screen game rules, and is also used for the ANZ jurisdictionally dependent Player Information Display (PID).

QCOM 3 requires and extends the functionality of the “i” button for the purpose of delivering some of the functionality in this specification such as QCOM PAEL (refer section 19).

Accordingly, QCOM 3 EGMs must have a player / attendant accessible “i” button implemented to the requirements in this section. cp:[[12]](#endnote-8) (These requirements are intended to complement current “i” button implementations)

The “i” must be visible and able to be activated only during:

* EGM idle mode. cp:[[13]](#endnote-9)
* PEF “play disable” (refer to the qcom\_playSetPEF() QCOM API function). cp:[[14]](#endnote-10)

Provided there are no faults on the machine and all doors are closed.

The “i” button (an example shown left), when visible, must be represented by a lowercase letter “i”, in a font similar to the one depicted in the example; preferably encased in a circle and be blue in colour. However, the “i” button shape and colour may be different from the example shown, to allow the button to fit-in more aesthetically with other buttons and colours on the UI of the machine. cp:[[15]](#endnote-11)

When activated, the user must be presented with a menu offering selection of:

* [“Reserve”] (If enabled. Ref: qcom\_egmSetReserve() )
* “Game Rules” (Always present for gaming machines)
* [“Player Information Display”] (If implemented / enabled; refer section 10.12.5)
* “Messages” (Refer QCOM PAEL section 19)feature on hold
* “<Custom1>” (See “<CustomX>” paragraph below) feature on hold
* “<Custom2>”
* “<Custom3>”
* …

The menu item order must be as shown above. cp:[[16]](#endnote-12)

Pressing the “i” button again once in this menu must cause the EGM to exit the menu and return to the previous mode.

A timeout of **60** seconds must apply to any consecutive period of inactivity during any “i” button functionality. Upon a time-out the machine must exit the “i” button functionality and return to the mode immediately before it was pressed. cp:[[17]](#endnote-13)

**Do not implement “<CustomX>” menu items at this time.**

“<CustomX>” menu items shown above denote optional new menu options added by authorised QCOM users. If present and selected by the user/player, the machine must launch an embedded GUI in the control of the respective QCOM user via call-back functions. Refer 10.12.6 for more information and possible future applications.

## Secrets

To implement QCOM 3, machines must store a range of secrets in NV memory / persistent storage. Refer to the term “secret” throughout this document and especially the QCOM 3 summary spreadsheet to find all mentions of QCOM 3 specific secrets.

There must be no interface or facility in production machines to extract or examine these secrets. Secrets must not be able to be displayed or revealed by the machine at any time except during early software develpoment.

Secrets must be stored by the machine in NV memory memory / persistent storage in encrypted form.

The machine’s secrets, or its decryption key (or passphrase) must be destroyed upon:

* a RAM clear operation
* In the case an Electronic Seal is mandated, then refer to section 31

The secret decryption key itself must be kept a secret and stored securely by the machine.

A decryption key should only be able to be recovered in a decommissioned machine (i.e after a tamper seal break), by the respective machine manufacturer, regulator, or regulator approved testing facility; using their intimate knowledge of the both the machine’s hardware, software and access to the source code.

An example acceptable implementation of the above could be:

* An encryption key is stored in NV RAM in plain text. This key was created by the machine’s RNG upon ‘logic seal confirmation’ taking place.
* An encryption key is hardcoded into source code in plain text. This key was created during s/w development and may be changed periodically over time the source code is worked on. (Also note possible variations below)
* Then upon every boot, the above keys are combined (XOR’d, hashed together, or similar) by the machine to form the actual encryption key used during machine operation which is stored in volatile memory in the machine and recreated each reboot.

Variations, or extra steps re the above points are encouraged so that each make-and-model machine does it differently and therefore requires knowledge of how the source code works, such as:

* Make another encryption key to hash/XOR in by hashing a commonly changed binary in the machine’s base code.
* Hash in a value from the machine’s motherboard hardware; such as CPUID and/or Ethernet address or similar.
* Hash in Version ID strings from components of the machine motherboard firmware, where available.

# QCOM machine commissioning

This section describes the typical process of a full commissioning of a RAM-cleared (factory default) and physically unsealed QCOM 3 machine. (For the highest possible security, a machine should enter this procedure in a powered-down state and be physically disconnected from any network in a RAM-cleared state.)

Requirements are also embedded in this section.

Commissioning process summary:

* Set / confirm date, time and time zone.
* Set / confirm the network adaptor setting used for QCOM 3.
* Logic seal confirmation.
  + *Refer section* 4.4 *for a list of post confirmation actions the machine must perform.*
* Uploading QMA certificate (*allows QCOM user introductions*)
* QCOM user creation / introductions.
* Application level configuration.
  + One-time items
  + *Machine indicates “ready”*
  + Customise or go

## Date and time configuration

The machine’s **UTC** and **local** date and time and **time zone** bias must be set or confirmed on the machine as early as possible in the machine commissioning process. This is required for some initial event logging and certificate generation. Certain QCOM 3 related services cannot start until this is done.

Accordingly, on every power up, until once the machine’s logic seal has been “confirmed” via the ”Logic Area Seal Confirmation Function” (see section 4.4), the machine must automatically[[18]](#footnote-5) (from the user via a UI on its built-in display) request confirmation or edit of the machine’s current:

* Local date and time. Labelled “***Local date and time***”
* The machine’s current total time zone bias relative to GMT; (i.e. where east of GMT is positive. The machine must assume this value includes any daylight saving offset.) Labelled “***Time zone bias****”.* The machine may Optionally include the words “***(include the 1 hour offset for DST if DST is currently in effect on local time)***” or equivalent wording. Entry units must be +/- hh:mm.

cp:[[19]](#endnote-14)

The machine must use the above information to set is value for UTC and epoch times.

Conversions:

UTC = (local date and time) – (time zone bias)

Once the machine’s logic seal has been “confirmed” via the Logic Area Seal Confirmation Function (4.4), there must be no way to change the machine’s time again via a UI on the machine’s display devices unless the machine is reset to factory defaults. cp:[[20]](#endnote-15) (NB the date and time may still be subsequently changed anytime by the other methods listed in section 9.)

A machine must not reset its date or time settings entered above if it is RAM-cleared. cp:[[21]](#endnote-16) Alternatively, whatever value is currently in the machine’s NV RTC and memory must be used/carried over as the new default provided the data appear valid. If the date and time or time zone is not valid, or before the default value shown below, then the machine must assume the **default** values of:

* 1/1/2017 00:00:00
* GMT time zone (0)

cp:[[22]](#endnote-17)

Related:

* timekeeping – section 9.
* Machine Audit Mode – section 28.

## Physical inspection

Theoretically, at machine commissioning, an authorised agent (depending on the level of operating risk) should first complete a manufacturer specific checklist which details how to verify, for example:

* Firmware and software PSDs. (Related: section 3.1.)
* The integrity of the logic area cabinet and its door detection circuit and sensor.
* The machine’s components inside the processor cabinet to be sealed are as expected.

(What items are actually checked depend on the specific machine, its expected level of operating risk and the resulting desired level of machine operating integrity.)

If successful, the agent would then power up the machine, confirm the date and time (see below) and then activate the machine’s Logic Area Seal Confirmation Function (section 4.4). Refer following sections for detailed requirements.

## Network configuration

Network configuration must be possible from within machine audit/service mode provided that the machine main door is also open.

A QCOM 3 machine must support the following IP configuration options: cp:[[23]](#endnote-18)

* IPv4: DHCP, manual configuration.
* IPv6: SLAAC (RFC 4862), manual configuration.

The machine RAM clear/factory default must be both version IP protocols enabled with DHCP and SLAAC.

The machine must store last entered configuration/values in NV memory. cp:[[24]](#endnote-19)

It is acceptable for the machine to have to restart after any changes.

There must also be a checkbox (or equivalent) which, if checked, the machine to display a network disconnected icon[[25]](#footnote-6) on the machine’s primary display whenever the machine is not physically connected to a wired network e.g. network cable is unplugged. cp:[[26]](#endnote-20) The factory default for this checkbox must be checked. cp:[[27]](#endnote-21) For EGMs, it is only mandatory to display the network disconnected icon during modes intended for use by players, however there must be at least one test mode display that also displays the network connection status. cp:[[28]](#endnote-22)

FAQs:

* A QCOM 3 machine may be still playable when it loses a network connection.
* It’s up to specially authorised QCOM users as to whether the machine is playable when it loses a remote network connection.
* There are currently no QCOM 3 events relating to Ethernet physical connection disconnected / connected events.

Related:

* Machine to machine communications; section 3.5.2.

### Machine discovery

Refer:

* Chapter 32 – Machine Discovery Protocol.

## Logic Area Seal Confirmation Function

As a part of initial machine commissioning, a QCOM machine must have a special UI function able to be accessed from within audit/test mode of the machine by a local human operator. cp:[[29]](#endnote-23) The function must be labelled: cp:[[30]](#endnote-24)

**“Logic Area Seal Confirmation”**

The purpose of this function is to:

* allow a human operator to display the state and test the operation of the machine’s logic area door open detection circuit;
* (once the human operator is satisfied with the integrity of the machine’s logic area and door open detection circuit) it allows the operator to inform the machine that a physical seal has been placed on the machine’s logic area;
* support the use of Electronic Seals.

Once in the machine’s Logic Area Seal Confirmation function UI within machine audit mode, the machine must display to the operator (for the duration of the function) the following information: cp:[[31]](#endnote-25)

* The message “**This display allows the user to inform the machine that its logic area has been physically tamper sealed**”;
* The current state of the machine logic area door/s in real time;
* The current local date and time in real time;
* The message “**Before sealing the logic area, confirm the machine’s logic area door open circuitry is as expected and working**” **[[32]](#footnote-7)**;
* The message “**Do not proceed if the date and time is not correct.**”[[33]](#footnote-8)

If the logic seal has already been confirmed, then the machine must also display the message “**Logic area sealed**” plus the following information: cp:[[34]](#endnote-26)

* The date and time stamp that seal confirmation occurred.
* The return value with repsect to the QCOM API function qcom\_idCommissionUID().

If the logic seal has not already been confirmed, then while the logic / processor door is in the closed state and the machine’s date and time has been configured and the logic seal has not been confirmed already, then a button (or equivalent UI) must enable which displays “**Confirm**”. cp:[[35]](#endnote-27) If the button is activated then the machine must ask the operator the following question or equivalent:

“**Are you sure you wish to confirm that the logic area has been physically tamper sealed?**” cp:[[36]](#endnote-28)

To confirm, the operator must instigate a new action, unable to be easily performed accidentally. cp:[[37]](#endnote-29)

Any other input should be considered a “no” response and declining must return the operator to the previous state. cp:[[38]](#endnote-30)

Confirmation of a physical seal must only be possible on the machine once per RAM clear. cp:[[39]](#endnote-31)

It must be possible to exit the seal confirmation function at any time without confirming the seal. cp:[[40]](#endnote-32)

When the operator confirms the machine is physically tamper sealed, the machine must then undertake a number of operations in order as follows: cp:[[41]](#endnote-33)

* Seed (or reseed or otherwise randomise) the machine’s RNG/s state;
* Create the encryption key for storing secrets. Refer section 3.8.
* Generate a machine private/public key pair (used to make a machine self-signed certificate (refer section 8));
* Generate a local machine CA certificate and any required derived certificates. Refer section 8.
* Initialise what will be the return value of any call to the QCOM API function qcom\_idCommissionUID(). Section 11.1.7;
* Initialise and start the QCOM Lua Engine (refer section 10.7.1). Related: *qle\_ready* STH mesage. Allow logging of events to the QCOM event buffer. Refer section 13;
* Log the **MACHINE\_SEAL\_CONFIRMED** event (*contains the commissioning UID*). This event must be the first event logged by the machine to the QCOM event buffer;
* Start the QCOM User Account Access (UAA) service. Refer section 23.
* Start the QCOM 3 Machine Discovery Protocol service. Refer section 32.

The above operations may take some time and the machine must display a suitable progress indicator while it is undertaking the above operations. cp:[[42]](#endnote-34)

The above operations will require a lot of entropy. As the machine has only just seeded its RNG, in order to speed up the process, the machine if desired may direct the operator to aid in entropy generation in some manner (for example by requesting the operator to spam a UI device, such as a touch screen interface or input buttons).

If there is any interruption of the above listed operations the machine must abort and return to the logic seal unconfirmed state. cp:[[43]](#endnote-35)

Related:

* QCOM API function qcom\_secLogicSealOk().
* Logic door access section 20.2.1

### Rationale

Physically speaking there is no protection against an unauthorised person declaring the seal was in place. However, the person/s responsible for the correct sealing are going to know that they didn’t perform the task and their records would show this. The machine would appear on the network with a new commissioning ID and certificate which can then be used by anyone to trigger a cross check against the authorised commissioner’s records to ensure that only intended authorised persons commissioned the machine.

This arrangement allows for scalable levels of security at commissioning as the desired level of security is setup by externally designed commissioning roles and procedures and not hardcoded into the machine. Many gaming machines are low risk and do not require a very onerous level of due diligence during commissioning; whereas other machines may have million dollar plus prizes and would have a commensurate level of commissioning procedures. For example commissioners can even go so far as to have multiple people witness commissioning & then sign the machine generated certificate that the machine generates shortly after sealing is confirmed; and conversely, in a low security operating environment, it may be sufficient just to allow licensed contractors to commission machines whereby they just record the commissioning ID number the machine creates after sealing is confirmed.

Finally, this approach also has the advantage of not complicating machine setup & operation during test and development.

## QCOM Master Authority (QMA)

Once the machine’s logic area tamper seal has been confirmed (see previous section) and the machine’s UAA service is started (refer s23), it will now be possible to upload to the machine the self-signed certificate pertaining to the QMA. The QMA is the mandatory authority under QCOM and performs a critical role. Refer to section 6.1 for a full explanation of the QCOM Master Authority (QMA).

The only way to setup the QMA certificate in the machine is via the QCOM Command Interpreter via an *anon* user login. cp:[[44]](#endnote-36) Refer to section 24.2 for the relevant command.

Once the machine sanity checks and accepts the QMA certificate, the machine must publish the certificate on its web server[[45]](#footnote-9) in PEM format and the certificate’s SHA256 fingerprint and other details in machine audit mode. cp:[[46]](#endnote-37)

For example:

SHA256 Fingerprint=E0:1D:27:FD:B4:15:84:B6:A0:DF:29:78:31:96:39:30:9E:54:3E:94: B4:15:84:B6:A0:DF:29:78:31:96:39:30

*The output above is the output of an openssl command line call equivalent to:*

**openssl x509 –sha256 -in QMACertificate.pem -noout –fingerprint**

or

**openssl x509 –outform der -in QMACertificate.pem | openssl dgst –sha256**

The intent is that once the QMA certificate is loaded, the person commissioning the EGM should confirm the QMA fingerprint is correct as expected before signing off on the machine.

## User Introductions

Please refer to section 5 for an explanation of QCOM users in QCOM 3.

Once a QMA’s certificate has been setup, it is then possible to start creating / introducing QCOM users to the machine.

Refer to section 5.2 on ‘User Introductions’ for more information.

Once the desired number of QCOM users has been setup on the machine, all future configuration and control of the machine is at the discretion of the QCOM users with respect to their assigned privileges.

## Application Level Configuration – Gaming Machines

*Historical information: In a QCOM v1 gaming machine, configuration at this level consisted of a staged approach whereby configuration occurred in the following order:*

* *Machine Configuration (denomination, min/max RTP etc.)*
* *Game Configuration (Variation selection)*
* *[Progressive Configuration if the machine contained one or more progressive games]*

*The way in which QCOM v1 configuration was implemented resulted in some duplication of functionality in that there was a method to setup a parameter at initial configuration and then another method to subsequently alter it.*

The previous sections in this chapter must have been successfully completed before a QCOM 3 can be setup at the application level as follows:

1. Setup the following global values:

Before a machine will accept any credit (including any banknote or ticket into escrow) or permit any functionality that results in any amount being added to any meter on the machine in units of currency (coin / token in, banknote or ticket accepted into escrow, etc[[47]](#footnote-10).) the following machine global values below must be first setup on the machine via their respective QCOM API functions cp:[[48]](#endnote-38):

Write-once only items:

* machineID -- qcom\_idSetMachineID()
* countryCode -- qcom\_locSetCountryCode()
* currencyCode -- qcom\_locSetCurrencyCode()
* stateProv -- qcom\_locSetStateProv()
* meterDenom\* -- qcom\_machineSetMeterDenom()

Also:

* time zone -- qcom\_timeSetTimezone()\*\*
* local date and time -- qcom\_timeSet()\*\*

*\*meterDenom cannot be set until currencyCode has been set first. cp:[[49]](#endnote-39)*

*\*\*Time zone & local time are not write-once only but local time must be set/at least once via the QCOM API before the machine can be ‘ready’. cp:[[50]](#endnote-40) Rationale: Setting the time & time zone via the UI, being human based data entry, would be prone to error, or inaccurate. Therefore, it was made a requirement for the time & time zone to also be set via the QCOM API to help force a time set operation from possibly more accurate & secure source. This also ensures that if the time was set inaccurately initially via the UI, time changed events would be logged which is useful information.*

The QLE Lua software driver will message the host machine as each of the above items are set.

2. Machine Ready

Once all the values in the previous step have been successfully written, the machine must throw the **MACHINE\_READY** state event. cp:[[51]](#endnote-41)

From this point on (until next RAM clear), the return value from any call to the QCOM API function **qcom\_machineReady()** will indicate *ready* (i.e. return *true*).

3. Customise or Go

All remaining parameters in QCOM must have reasonable and useable factory default values. This means that if the QCOM user responsible for configuring the machine, didn’t care about the suitability of the factory default values (e.g. in a test / development environment), then all the QCOM user has do to enable a game for play on the machine at this stage, is to allow credit input (CIEF) and set the game enable flag/s (GEF) to *true*. cp:[[52]](#endnote-42)

Typically however, QCOM user/s will wish to customise various settings before enabling credit input, or any games.

Related:

* The set of QCOM mandated default values are located in the QCOM Summary Spreadsheet – “Global Types” sheet.

## Comissioning disable condition

Section 4.7 (1) defines a machine disable condition based on the requirement that the machine must not accept credit after a RAM clear, up until it has logged the **MACHINE\_READY** state event.

This section outlines example options as to how this may be visualised on gaming machines.

The intent is to allow the gaming machine to be as verbose as it likes as to the progression of all the items requiring commissioning in this chapter and use similar terms.

The following examples are provided:

It is acceptable if the EGM never displays a game screen until after it logged the **MACHINE\_READY** state event. This allows the EGM to display a full list of items on its display to check-off as commissioning is progressed. (Similar to a displayed OS boot sequence you might see in Linux). Something along the lines:

* *Date and time cfg via audit mode UI*
* *Logic seal confirmation*
* *QMA certificate installed*
  + **machineID**
  + **countryCode**
  + **currencyCode**
  + **stateProv**
  + **meterDenom**
  + **time zone**
  + **local date and time**

Alternatively, for example; if the machine prefers to jump straight into a game display, a system lockup like display is suggested, with just a few of prioritised significant checkpoint messages; each one replacing the last, such as:

* *Date and time required*
* *Logic seal confirmation required*
* *QMA certificate required*
* *Machine not ready*

Final example; the machine just states: “Machine not ready” and entry into audit mode provides the user with more verbose information along the lines of previous examples above.

# QCOM Users

*Note: almost all requirements in this section are implemented as a part of the QLE Lua software driver. Related: section 33.1.*

QCOM users or simply ‘users’, are a fundamental concept in QCOM concerning the provision of multi-master support, i.e. where multiple machine related service providers can monitor and control a QCOM machine simultaneously at their assigned level of privilege in order to implement their roles and services.

Any organisation, computer system, service provider or service, or individual with a need to interact with a QCOM machine via the QCOM API are referred to as a “QCOM user” in this document. Users on QCOM 3 machines can login to the machine remotely for maintenance purposes. Refer chapter 23 for more information regarding QCOM User Account Access (UAA).

QCOM users can download and execute scripts in the machine. QCOM users have limited, strictly controlled resources and privileges and are jailed inside a scripting engine virtual environment inside an application running in the machine. QCOM users do not have machine file-system access.

The number of expected QCOM users is small, see below. Typically the number of users will be around the same number of different gaming-related services the machine is associated with. However, a QCOM user may sometimes be created by an authority or service provider just to perform an important specific task. For example; a jackpot trigger, RCRF, implement a new meter, a special limit or other special behaviour.

The number of users a machine must support is **12 users[[53]](#footnote-11)**. cp:[[54]](#endnote-43)

A machine may optionally support more users if desired. The maximum user limit is implemented by the QLE Lua software driver.

All QCOM user resources and quotas are managed by the QLE Lua software driver on behalf of the host machine.

Uncreated QCOM users in a QCOM 3 machine must use no resources. In other words recompiling a machine with a max user limit of 120 should result in exactly the same resources left in the machine after compilation/build/boot-up as the machine with a max user limit of 1. cp:[[55]](#endnote-44)

Users in QCOM are typically created / deleted and their privileges maintained by the entity who is the QCOM Master Authority (QMA). However, the QMA can assign this privilege over to a user if desired. Refer to section 6.1 for more information on the QMA.

**To a QCOM 3 machine, all QCOM users are, is**:

* A single zip archive file of scripts per user for which the machine extracts and provides to the QLE Lua software driver on each restart of the machine or user.
* Various user specific settings the machine must store in its NV memory when the QLE Lua software driver tells it to; a list of which may be found in the QCOM 3 SDK.
* Something the QLE Lua software driver manages on behalf of the machine.

Typical examples of users in QCOM:

* Users by organisation;
* Users by role or service (e.g. credit redemption, RCRF, account based gaming, general monitoring, loyalty, jackpots\*, etc. *\*Jackpot/prize triggering is an example of a service will typically be a standalone user service due to the level of security and controls surrounding that type of service*).

It will be common for a single organisation to be granted multiple QCOM user accounts by the QMA (e.g. one for monitoring and one implementation for a jackpot solution). Giving a client multiple QCOM user accounts allows for separation of services when, for example, one service requires third party authentication of scripts and the other does not. It allows an organisation to update non-critical services without having to incur the costs of script approvals as would be the case when a non-critical service is mixed with a critical service such as jackpot/prize triggering (including implementation of a RCRF or similar feature). There are benefits to reliability and integrity when regulatory roles are separated from other services by utilising multiple QCOM users.

Who should not be a QCOM user?

There are no restrictions or security concerns here, as any given QCOM user will only be granted the minimum set of privileges required in order to perform their roles there will never be any risk to the integrity of the machine. Convenience, cost, trust and regulatory environment are factors though. *It is up to the QMA or delegated authority as to what users are created and for whom and what privileges a user receives. It is then up to each individual user as to how much data and functionality they want to share with other users.* However, a common situation for not granting a user account is sometimes due to efficiency; e.g. when multiple organisations are delivering a similar or compatible service which can potentially share a standard network interface, such as performance monitoring or jackpot display systems. In these cases, it may be more efficient to keep those parties outside the machine and utilise a standard protocol interface delivered by a commercially neutral third party (with respect to the delivery of gaming machine related services) implemented as an autonomous QCOM user in the QCOM Lua Engine.

For further reading, refer to the relevant appendix in this document on designing QCOM modes of operation or operating environments.

Related: “Summary: QMA, anon & regular QCOM users” section 6.1.5.

## Anon Users

An anon QCOM user (“anon” is short for anonymous) is a special **temporary** QCOM user, which the QLE Lua software driver will automatically create as a result of any successful anon user login via the UAA service (s23) via the ‘anon’ username and password.

As there may be more than one anon user logged in via the UAA service at any given time, the QLE Lua software driver will create **multiple** anon QCOM user accounts when this occurs. In the QLE Lua software driver, each anon user is implemented as a discrete QCOM user with a unique username and jailed environment.

Whenever an anon QCOM user UAA session ends, the QLE Lua software driver will automatically **delete** the respective anon QCOM user account that was created for that session.

An *anon* user has very **limited** default privileges and quotas which are typically not extended or changed. The quotas pertaining to anon QCOM users may be found in section 6.1.5.

An anon QCOM user has no account storage space and they cannot upload scripts packages like regular QCOM users can.

Like regular QCOM users, an *anon* user will be able to execute signed scripts of other users at their privilege level, for example and most importantly, the QMA. This feature specifically allows users to self-introduce and set up their login and privileges on a QCOM machine. A new user (with no existing user account on a QCOM machine) can log in as an *anon* user and execute their QMA signed introductory script which creates and initialises their user account and privileges. Once set up (a once per machine RAM clear operation), they logout of *anon* and can then log into their actual QCOM user account and start interacting with the machine at their granted level of privileges.

The QMA can also pass the privilege of creating new users to a QCOM user and/or create arbitrary dependencies by controlling what privileges it gives to any single user.

An *anon* user must always utilise a password for logon (all other QCOM users utilise key-based authentication as this permits the QCOM self-introduction feature). cp:[[56]](#endnote-45)

The QLE Lua software driver factory default *anon* user password is “*anonuser*”, this can be changed by the QMA or any QCOM user with rights to the QCOM API function qcom\_userAnonSetPass(), s11.30.33.cp:[[57]](#endnote-46)

An *anon* user must only be able to execute script code via the QCOM command interpreter (s24), *dostring* command (s24.2.7). An anon user cannot hook scripts onto State Events (i.e. via s11.15.4).

As a result of access to the QCI *dostring* and *lua* commands, it is possible for an anon user to quarantine themselves (s5.8). If a quarantine event occurs for a given anon user, the anon user session will be immediately disconnected and deleted as per a normal disconnect (as anon users are only temporary) by the QLE Lua software driver. Events logged will have recorded the IP address and reason for the quarantine.

## User Introductions / Account Creation

In order to start interacting with a QCOM 3 machine via the QCOM API, a QCOM user must have an account created on the machine. To do this, a script must be executed in the machine’s QCOM Lua Engine. Typically, user account creation is performed by the new user themselves by logging into the machine’s QCI as an *anon* user and executing a signed QMA introductory script on the machine. This is called a **self-introduction** and it allows new users to create an account on the machine without being dependent on any third party other than the QMA.

For more information about QCOM user introductory scripts refer to section 6.1.2.

***It is recommended that the QMA always assign all usernames to avoid contentions.***

The first user account on a RAM-cleared machine can only ever be set up by logging into the QCI as an *anon* user and executing the QCI command **qmaexecscript** (section 24.2.3). Subsequent users are typically set up the same way; however the QMA can delegate the required privileges to another existing user provided the security risks here are understood.

To ensure security in the account creation process, user introductory scripts contain the intended user’s public key which is used for subsequent QCOM user account access/login. This eliminates any need to include a password in the introductory scripts and also prevents a user who was delegated the role of user introductions (or even the QMA) from accessing another user’s account.

Once a QCOM user account has been created, the user may then login via the machine’s UAA service (s23) and upload their scripts and commence interacting with the machine at their assigned level of privileges.

### Example

This section outlines one possible typical approach to setting up QCOM users (aka machine related service providers) in a machine. An understanding of the following QCOM 3 facets is required before reading this example:

* QCOM 3 users
* QCOM 3 authorities
* UAA service

Setup

1. Prospective QCOM users aka machine related service providers (**SP**) have previously contacted the QMA to request authorisation to access machines as QCOM 3 users. The SP also provides the QMA with their public key as a part of this process, how many users they require and proposed privileges.
2. If the QMA approves the SP, the QMA provides the SP with a signed QMA introductory script. Refer to section 6.1.2.

This example commences after the machine’s logic seal has been “confirmed” via the “Logic Area Seal Confirmation Function” (refer section 4.4).

1. The QMA’s self-signed certificate is installed on the machine by a methodology decided by the QMA.

Typically, either a QMA nominated QCOM user will install it, otherwise the first QCOM user to connect to the machine can also install it.

Typical installation methodology: UAA anon access via the QCI command: **qmaloadcert** (s24.2.1)

Anyone with access to the machine’s audit mode display, or the network, can confirm the correct QMA certificate is installed on the machine. If the incorrect QMA certificate is installed, other QCOM users will not be able to install themselves, drawing further attention to the problem.

**Once the QMA public key is installed, the machine is** **in the proxy control of the QMA and will allow only signed QMA scripts to execute.**

1. A QCOM user takes their QMA signed introductory script, connects as an anon user to the intended machine/s and executes the script.

Methodology: UAA anon access via command: **qmaexecscript** (s24.2.3)

This will create the SP’s QCOM user/s in the machine and setup all privileges and also installs the user’s public key.

* A generic QMA introductory script for use as a template is provided with QSIM 3
* NB The QMA may enforce the use of a Script Approval authority (SAA). See s6.3 for more information

1. The SP logs out of anon and can now log back into the machine’s UAA service using their private key for authentication.

While logged in, the SP then uploads their intended scripts to the machine which typically implements any desired network protocols that will connect and talk to the SP’s systems.

Method: UAA anon access via command: **userloadscripts** (s24.2.4)

1. The SP logs out of the UAA and would be unlikely to require the UAA service again unless the machine is factory reset or they need to upgrade their scripts.

All further communications between the machine and the SP systems is typically via the SP installed scripts using the QCOM API which has support for the IP / TCP/UDP protocols as well as serial port access.

## User Account Storage

Machines must provide each user with a limited amount of storage space associated with their account. The purpose of the user account storage space is to **store QCOM user script file archives**. It should be noted that QCOM users never have any access to the machine’s file system.

The QLE Lua software driver limits the total amount of storage space for QCOM user script storage to **12MB**. The individual quota limit set per QCOM user **must** be applied by the machine during execution of related QCI and QCOM functions listed below.

***Note: These limits are subject to change until the later stages of the QCOM 3 project implementation phase.***

The per-user limits are adjustable via the QCOM API function qcom\_userSetDiskQuota()*.* *#disklimit #disk quota.* Disk quota limits must be applied by the machine per QCOM user only during the download of a user’s script file archive. Refer ***userLoadScripts*** sendToHost message required action in the QCOM 3 summary spreadsheet.

The types of storage devices that may be utilised by the machine for QCOM user accounts are hard disks, flash memory and all types of battery-backed memory devices.

The QLE Lua software driver ensures QCOM users never have access to the machine’s OS file system. cp:[[58]](#endnote-47) All user control and access is strictly controlled by the QLE Lua software driver via the QCOM API (refer section 10.5) and QCOM command interpreter (refer section 24).

User Account storage space provided to users on a machine, is intended for low frequency use, to allow the use of flash-based memory for the storage of QCOM user scripts & apps. The QLE Lua software driver limits QCOM user’s rate of possible file writes by enforcing cooldowns on associated QCI and API function calls.

Related:

* Section 5 – Maximum number of QCOM users.
* Section 23 on QCOM User Account Access.
* Section 24.2.3 - qmaexecscript
* Section 24.2.4 - userloadscripts
* QCOM API function: qcom\_userLoadScripts()

### Script file archives

Refers to QCOM user ZIP file archives (which may be signed or unsigned), typically comprised of one or more .lua script and .txt files and stored by the machine in their user account storage.

The machine must be able to handle ZIP files that conform to the ISO/IEC 21320-1:2015 standard cp:[[59]](#endnote-48) and gracefully reject all malformed zip files, including zip file bombs (ref: userLoadZIP Clua function).

The machine must ignore file extensions of the script file archives (refer QCOM 3 summary spreadsheet – userLoadScripts sth message – required actions). Instead: cp:[[60]](#endnote-49)

* If the QCOM user does not have a SAA certificate installed[[61]](#footnote-12), the machine must treat the file’s format as the **ZIP file format**.
* If the QCOM user has a SAA certificate installed for it, the machine must treat the file’s format as a SMIME v3.1 signed ZIP file, signed equivalent to the openssl command: cp:[[62]](#endnote-50)

**openssl** **cms** [-md sha256] **-sign** -in userscripts.zip -nodetach -out userscripts.zip.sig -binary -signer saa1\_cert.pem -inkey saa1\_privkey.pem

*\*Filenames shown above are examples only*

Malformed SMIME signed ZIP files must be gracefully rejected by the machine. (ref: smimeVerify Clua function).

The machine’s verification of the signed script archive file must be equivalent to the openssl command: cp:[[63]](#endnote-51)

**openssl** **cms** -**verify** -in userscripts.zip.sig -CAfile Q3SAA\_sscert.pem -out <username>.zip

*\*Filenames shown above are examples only*

The expected overall **size** of individual script archive is expected to be small on average e.g. ranging from **1kB** to **100kB**, meaning that the download and verification of the script archive can potentially be performed in ordinary machine RAM until being saved to a persistent storage option as listed in section 5.3 upon a successful verification.

The script archive must be thoroughly tested / sanity checkedcp:[[64]](#endnote-52), treated as possibly malformed until proven otherwise. This operation may occur either immediately after the download or upon next user restart.

The machine must only extract specific file types from the archive; namely Lua script files with the extension **.*lua*** and ***.txt***files. All other files types in the archive and any subdirectories must be ignored (not extracted) by the machine. cp:[[65]](#endnote-53)

The QLE Lua software driver will not allow precompiled lua files as they are a risk (refer section 10.7).

Related:

* The SAA & QCOM API function: **qcom\_userSetSAAcert()** (refer section 6.3)
* Installing QCOM user scripts in the machine:
  + QCI command: userloadscripts (s24.2.4)
  + QCOM API function qcom\_userLoadScripts(). Refer to the QCOM 3 summary spreadsheet, Lua-API worksheet.

### Text file encoding

This section relates to text file encoding of .lua and .txt files:

The machine is only required to be able to use and display (in audit mode) **UTF-8** encoded text files[[66]](#footnote-13).

The uploading and audit mode display of .lua and .txt files that are just random bytes (aka full of non-printable characters) will be tested. If machine for example crashed upon trying to display these files, or otherwise emitted behaviour here that appeared to relate to the integrity of the machine then this will not be acceptable. It is acceptable if the machine refuses to display any .lua or .txt file that contains any non-printable ASCII character. cp:[[67]](#endnote-54)

CR/LF issues.

The machine must be able to display text files whose lines are terminated with CRLF characters. (No empty lines should result.)

It is desirable if the machine whose lines are terminated with only a LF character can also be displayed properly. If the text is display all on a single line however, then this is currently acceptable provided all characters are still able to be seen.

Related: Machine Audit Mode s28

## User Memory Quota

The QLE Lua software driver implements a limit on the amount of machine Lua interpreter memory each QCOM user utilises during the execution of their scripts in the QCOM Lua Engine.

This limit may be adjusted by the following QCOM API function:

qcom\_userSetMemoryQuota() s11.30.14

The QLE Lua software driver ensures that the total memory granted across all QCOM users will not exceed **6MB**. This overall limit is enforced upon each call to the above function. Individual QCOM user memory limits set by the above function are applied in real-time during the execution of any QCOM user script. Refer s10.2.1 for more information.

Each user’s memory limit set via this function is also used for one additional purpose. Each user’s memory limit is also applied by the QLE a cumulative file size limit during .lua file extraction from the user’s script file archive. Refer ‘userloadscripts’; #zip file bomb. Tripping this .lua total file size limit will only generate an error result and will not quarantine the user like a user Lua memory limit breach would.

Related:

* Section 5 – Maximum number of QCOM users.
* Section 5.1 – Anon user memory limit.
* Section 6.1 – QMA memory limit.
* Section 10.2.1 – Memory Usage, Monitoring and Control

## User Lua Instruction Quota

**The method described in this section hard limits the amount CPU time available to each QCOM user.**

The QLE Lua software driver uses the Lua instruction counter built into the Lua interpreter to apply an effective CPU quota to each QCOM user. The Lua instruction quota applies a hard limit to the number of Lua instructions able to be executed per user script hooked on a QCOM State Event and automatically interrupts a user script that exceeds its instruction quota.

Lua simplifies the implementation of a Lua instruction quota, as the Lua interpreter already has an instruction counter built-in that the host machine can hook a quota onto. For more information refer Lua documentation for the debug library function sethook().

The Lua instruction quota is implemented by the QLE Lua software driver to the requirements in this section.

The Lua instruction quota is applied by the QLE Lua software driver in real-time during the execution of all QCOM user scripts. There is one quota per QCOM user.

Each time a QCOM user’s script returns control to the host machine[[68]](#footnote-14), the user’s instruction counter is reset by the QLE Lua software driver.

If a QCOM user script fails to return before their instruction Lua quota is reached, then the QLE Lua software driver will automatically interrupt the executing user script and execute a pre-set arbitrary function. This function will:

* Quarantine (s5.8) the responsible QCOM user.
* Abort the executing script via a call to either the Lua API **error()** function, or the Lua C API function **lua\_error()**.

**Supporting QCOM API methods:**

qcom\_userInstructionStats() -- Ref s11.30.37.

qcom\_userSetInstructionQuota(username) -- Ref s11.30.38.

The Lua API function that can set an interrupt after a given number of Lua instructions have been executed is: debug.sethook().

Refer to either section 6.1.5 or 5.7 for user defaults pertaining to QCOM user Lua instruction quotas.

**Implementation**

Just prior to calling any QCOM user script, the QLE Lua software driver will activate the Lua instruction quota interrupt via:

debug.sethook(abortscriptfunc, “”, userInstructionQuota)

where *abortscriptfunc* is basically (in Lua script):

function abortscriptfunc()

quarantine(currentUser)

error(“user exceeded their Lua instruction quota”)

end

and the argument *userInstructionQuota* is a count of Lua instructions after which the Lua interpreter will interrupt the running script with a call to *abortscriptfunc().* Refer to the QCOM 3 SDK for the full function listing.

Anytime a QCOM user script returns the QLE Lua software driver, it will disable/reset the hook via:

debug.sethook()

**The advantages of the Lua instruction quota:**

* The debug.sethook() function uses no extra CPU time when enabled as shown above. The function is built into the core Lua interpreter accessible via the Lua debug library.
* It can trap and gracefully break out of QCOM user scripts that fail to return (even the script: *while (1) do end*) in a fraction of a millisecond.
* Recovery from a hung QCOM user script no longer requires a QCOM Lua Engine restart (like the watchdog did) and therefore a hung QCOM user script does not affect other QCOM users.
* A watchdog process is no longer required to monitor the QCOM Lua Engine.
* It is easy to extend the Lua instruction quota feature to include either a global per user value, or a custom limit per user, per State Event.

**The Lua instruction quota vs CPU quota - FYI**

Summary of main differences (for reference purposes):

* The Lua instruction quota is **applied** in real-time during script execution.
* The CPU quota is **applied** each time a user script returns and just before returning from certain potentially CPU intensive non-Lua implemented QCOM API functions. Refer to the QCOM 3 SDK for the list of functions.
* The Lua instruction quota guarantees all scripts return and uses no additional CPU time in order to achieve this (thanks to Lua).
* The CPU quota can be cheaply **totalised** and reported on. Refer QCOM API function qcom\_userCPUstats()
* While the Lua instruction quota can also be used to easily totalise Lua instructions per QCOM user in software (QSIM 3 / the QLE Lua software driver can do this on demand in order to allow system developers and testers to benchmark QCOM user scripts), this feature is not recommended for production machine because it costs additional CPU time in proportion to the desired instruction counter granularity. This is why the QCOM API function qcom\_userInstructionStats() only reports the set Lua instruction quota limits per user.
* The Lua instruction quota if totalised per user, is directly **indicative of the amount of CPU time used by QCOM user scripts**; but note this does not include time spent executing Lua “C” functions[[69]](#footnote-15) called by the user.
* The totalised CPU quota is more **indicative of the amount of CPU time each QCOM user has used**, but may be slightly inflated by other processes executing in the host machine as a result of OS process multitasking.

Combined, these two mechanisms effectively monitor and control CPU usage of QCOM users using minimal resources in the host machine.

*Design note: If tighter control is desired, it is very easy to extend the Lua instruction quota to a limit per user, per hooked script.*

## User CPU Quota

The User CPU Quota is primarily for benchmarking purposes. The Refer to section 10.2.3 for more information.

## User Quota Summary

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Quota:** | **Units** | **Default**  **at user creation** | **Min**  **(applied by QLE LSD)** | **Max**  **(applied by QLE LSD)** | **Max**  **(Total for all Users)** | **Quarantine**  **(if breached)** |
| **Storage** | kB | 0 | 0 | 12,000 | 12,000 | NA |
| **Memory** | kB | 0 | 1 | 6,000 | 6,000 | Yes |
| **CPU** | msecs | 0 | 0 | NA | NA | Yes |
| **Instruction** | none | 0 | 50 | 100,000 | NA | Yes |

Notes:

* The above values are hardcoded into the QCOM 3 SDK Lua software driver. The SDK values override the above values if there are any discrepancies.
* CPU use totalisation in a machine may not always be perfectly indicative of actual QCOM user CPU use in every machine type depending on its architecture. On some machine types, CPU use may include occasional spikes from OS process switching and I/O. Refer s10.2.3 for more information.
* **It is the Lua instruction quota that is directly indicative of QCOM user CPU use and control.**
* +The machine manufacture can add a grace amount to the CPU limit on top to the value set via the QCOM API based on machine performance benchmarks taken during development of CPU/Memory speed and OS. Refer QCOM 3 SDK : machine.lua : *qapi\_cpulimitgrace* global. It is acceptable to be generous in order to be safe as QCOM user CPU usage is more accurately limited via the Lua instruction quota.
* The QCOM Lua Engine adds roughly **20** instructions to each called script as overhead.

## User Quarantine

There are certain events which will result in a QCOM user being automatically ‘quarantined’ by the machine. These events are:

* CPU quota exceeded (refer s10.2.3)
* Lua instruction quota exceeded (refer s5.5)
* Lua memory quota exceeded (refer s10.2.1)

Quarantine of a QCOM user means that the QLE Lua software driver will abort and suspend all further script execution for the quarantined QCOM user. In addition:

* It will no longer be possible for the QCOM user to login to the machine (refer s23) using their login credentials. The QLE Lua software driver will refuse any connection attempt for the user.
* Any “i” button menu items for the user must be removed (11.28);
* Any call-back functions the QCOM user has registered for will be cleared by the QLE Lua software driver. See: Content Auditing (chapter 27), the QCOM API function **qcom\_egmDisplayMeters()** and Custom UI (10.12.6) & QCOM API qcom\_info class of functions;
* Any System Lockup’s current or pending for the user must be auto-cleared by the machine (16.2);
* The user’s scripts inherently become locked and may become viewable by third parties for diagnostic purposes.

Once quarantined, the only way to restore the QCOM user is for an authorised QCOM user (typically this will be the QMA) to delete the user’s account on the machine. Once deleted, the account can then be re-created in the usual fashion. *NB a QCOM user with a self-introduction script (s6.1.2) won’t be able to fix a quarantine* *by simply re-running their introductory script again. Introductory scripts typically first check to see if the user already exists and abort if they do exist. Even if this check was inadvertently omitted from the script, QMA introductory scripts will rarely, if ever, ‘delete’ a QCOM user as a normal part of their programming.*

Note, the QMA user is also subject to quota limits; however, the QLE Lua software driver will never quarantine the QMA, but it will still abort any QMA script that exceeds any quota.

Related:

* QCOM API function: **qcom\_userDelete()**. Refer section 11.30.2.

## User Deletion

QCOM user deletion should be a rare event. Typically a QCOM user would last for the duration of the machine time operating at a particular gaming venue.

Typical reasons for QCOM user deletion:

* User quarantine. See previous sub-section.
* Changes to the operational environment. For example a change in machine related service provider.

Typically a QMA should not privilege out user deletion and deal with user deletion via the same method as QCOM user creation.

Typical approach to QCOM user deletion:

1. The QCOM User contacts the QMA and requests a deletion script be signed; must provide reasons and scope.
2. If satisfied the QMA signs a user deletion script. (Like introductory scripts, user deletion scripts have a fairly narrow scope and lifespan).
3. QCOM user takes the script and executes in the same fashion as an introductory script.

## Built-in QCOM users

A built-in QCOM user is QCOM user that is hardcoded into the machine and automatically created by the machine once per RAM clear under specific conditions. The conditions for activation for any QCOM 3 built-in user will be set by OLGR.

Built-in QCOM user support is **optional** for a machine to implement.

All built-in QCOM users must be approved by OLGR and will be designated with an OLGR assigned username.

Built-in users allow for certain functionality to be always available in a machine even before network access has been established.

A typical and common use-case for a built-in QCOM user is the **MDP** user (s32). As the MDP is a part of the QCOM 3 specification, no special approval is required before making/activating MDP a built-in QCOM user in a machine other than final script approvals. The username for a built-in user implementing the MDP comnponent of QCOM 3 must be “MDP”.

Another existing QCOM user that may be allowed to be built-in is the OLGR qcom16 user that implements the legacy QCOM v1.6.7 protocol.

All built-in QCOM users must only be able to be installed once per RAM clear as per their approved cinditions. Once installed they must appear as regular QCOM users, e.g. in that they must be able to be subsequently removed or replaced by the QMA, or a delegated user.

Built-in QCOM users still need a user introductory script (s5.2), but this script will also be built into the machine. This introductory script must be approved by OLGR. The only difference is that the introductory script of a built-in user will not be signed and bypasses the QMA. In some cases (such as MDP built-in user), the user may be able to be created and running before a QMA has been setup in the machine. Others may require special access (such as logic door open, or a passcode) in order to create.

Once a QMA is installed, it must no longer be possible to install built-in users.

Built-in users and RAM-clears via the QCOM API:

Whether a built-in user reappears after a RAM clear instaigated via the QCOM API is on a case-by-case basis at the discretion of the OLGR. Typically a built-in QCOM user will be allowed to automatically reappear if it was present before the QCOM API RAM clear and the version that was running still matches the built-in version via the QCOM user *srchash* property.

# Authorities

*Note: almost all requirements in this section are implemented as a part of the QLE Lua software driver. Related: section 33.1.*

There are a number of authorities in QCOM which are listed and whose roles are defined in this section.

Authorities are represented by x509 self-signed certificates. This is primarily for the metadata in the certificate.

## QCOM Master Authority (QMA)

A QCOM 3 machine must allow for a single QMA authority to be installed in it via the QCI command

qmaloadcert

Refer section 24.2.1 for more information on this command.

The QMA is a mandatory and critical role with respect to security and integrity of the machine. The role of the QMA should always remain with a competent, trusted and commercially neutral[[70]](#footnote-16) authority.

The QLE Lua software driver implements QMA functionality. The QMA has full privileges to the entire QCOM API.

The QMA typically operates by proxy; it gets QCOM users to execute signed QMA scripts in the machines.

The QMA’s typical functions are to:

* authorise QCOM users to interact with QCOM 3 machines.
* perform initial machine configuration for items it does not want to delegate,
* create new QCOM users in the machine and set privileges thereof, and
* perform important operations that cannot be delegated to QCOM users.

Only the QMA can introduce (create/setup) the first user on the machine (and typically introduces all of them). The machine will stay at RAM clear defaults until the QMA starts to configure the machine when it introduces the first user. The QMA always has first access to machine configuration.

Typically the QMA (via the ability to create users and assign privileges) basically creates the roles represented as QCOM users and functionality thereof, in terms of privilege to the QCOM API. The QMA can also create additional functionality or modify existing functionality and set up and create autonomous users, services and dependencies as required.

The QMA is also a special kind of a QCOM user within the machine and is fully implemented by the QLE Lua software driver. The term ‘special’ in the context of the last sentence means that the QMA:

* is not permitted to log into the machine via the QCOM Command Interpreter like other QCOM users can.
* is not able to hook scripts into state events (via s11.15.4).
* is only able to execute scripts by proxy i.e. by other users on the machine including the *anon* user. Refer section 10.7.3.
* like all QCOM users is hard quota limited in all respects but is never quarantined (s5.8). A QMA script that exceeds its quota will always just be aborted.
* Has persistent full access to the QCOM API however some QCOM API functions behave differently for the QMA as detailed in the QCOM API.

In QCOM, the QMA is denoted by a self-signed x509 certificate. The QMA certificate is uploaded to the machine via the QCOM Command Interpreter (refer section 24) once the UAA/QCI service becomes available at the final stage of machine comissioning.

### The QMA has first access

This section expands on the implications of the QMA having first access to machine configuration at its factory default settings.

As the first QCOM user in newly RAM-cleared machine can only be introduced to the machine by the QMA, **thus a QMA-signed script will always be the first external script to execute on any QCOM machine** (via a QCI *anon* user login). Accordingly, a QMA script at some stage will find the machine has not been configured (i.e. the machine is still at factory defaults). This means the QMA always has first access to machine configuration and can use this advantage to create any possible desired QCOM mode of operation from this point on. Whenever a QMA script finds an un-configured machine, it should set up the machine according to its needs. Every QMA user introduction script should also contain the ability to set up a freshly RAM-cleared machine in addition to any function it is performing (e.g. user introduction).

*Design Requirement: Ensure not even the QMA can access a QCOM user’s private data or hijack their credentials.*

### Introductory Scripts

QCOM user introductory scripts are created and signed by the QMA on a per QCOM user basis.

The primary purpose of QCOM user introductory scripts is to allow QCOM users to obtain proper access to QCOM 3 machines for the purpose of facilitating machine related services over a network. Typically, the introductory scripts do the following:

* Create new QCOM user accounts on QCOM machines. Related: QCOM API function: qcom.userCreate()
* Set up QCOM user account privileges. Related: QCOM API function: qcom. userSetPrivilege()
* Install the QCOM user’s public key which gives the user access to the machine via the UAA service. Related: QCOM API function: qcom.userSetUAApublicKey()
* Allows the QMA to detect and configure the machine with any defaults that it wants to control
* Allows the QMA to create and set up any desired QCOM Autonomous user functionality

This section outlines an example of a QCOM user introductory script.

When a QCOM user introductory script detects a newly RAM cleared machine, it will determine what QCOM parameters and QCOM API functions are going to be controlled by other users and which it will retain. For example in Queensland, a QMA user introduction script will set up and retain control over the following items:

* Country Code;
* Currency Code;
* State/Province;

A typical introductory script may, for example, follow the logic:

*Am I an obsolete script? (i.e. check for correct QCOM API version)*

*Yes: abort with error*

*Am I an out-of-date, expired or superseded script?*

*Yes: abort with error*

*[Am I sooner than a ‘not-before’ date?]*

*Yes: abort with error*

*Is the machine in debug mode?*

*Yes: abort with error*

*Is the machine in a good state? (i.e. [idle mode, disabled,] no errors)*

*Yes: abort with error*

*Is the machine configured as I expect?*

*No: { Does the configuration indicate the wrong jurisdiction or corruption?*

*Yes: abort with critical error*

*If the configuration indicates RAM-cleared machine then set up my defaults and continue*

*}*

*Does the user already exist?*

*Yes: abort with error*

*Should other users be introduced first or are there other prerequisites?*

*Yes: abort with error*

*Is the machine relevant/ready for the given user?*

*Yes: create and set up user with correct privileges, UAA public key and quotas*

*No: abort with error*

*Print success or fail (with reasons) upon any exit / abort*

**A QMA script template may be found in the QCOM 3 SDK.**

### Autonomous Users

The QMA (or delegated QCOM user) can create what is termed autonomous users[[71]](#footnote-17), whereby a QCOM user is created which will subsequently automatically perform any conceivable set of desired functions.

Autonomous users can be very useful for certain roles and operations. For example, allowing the QMA to retain control or implement functionality that the QMA may not be comfortable in delegating to a third party (e.g. when commercial neutrality is needed). Other examples could be for common services that would facilitate multiple service providers thus avoiding duplication, or simply when a service is easily automated in order to achieve network outage immunity or other benefits[[72]](#footnote-18).

Below are some examples of functionality that an autonomous user could provide:

* Broadcast protocols

The QMA sets up an autonomous user and scripts on the machine which implements a simple broadcast protocol that e.g. performs regular UDP broadcasts (~every second) concerning the machine current state. The information broadcast would contain information equivalent to QCOM v1’s “Default Response”) as well as broadcasting any desired range of events and machine meters. This would potentially allow a number of third parties (e.g. gaming performance systems, player loyalty systems, jackpot display systems, gaming venues and anyone with access to the local network), read-only access to all the information they require in order to implement their services. This eliminates the cost in setting up QCOM user accounts for them. This type of arrangement could be used to reliably and inexpensively disseminate any machine-related information as desired.

* Legacy protocols.

QCOM 3 machines can be made to implement arbitrary protocols, allows them to integrate with almost any TCP/IP based system. These would be set up once per RAM clear via an autonomous user.

* Roles and tasks a QMA does not want to delegate.

The QMA can set up autonomous users to automatically perform arbitrary roles, tasks, or enforce constraints it wants to retain control over.

* Multi-user support.

When a QCOM API function supporting only a single user is extended to support multiple users, this is best done by the QMA (or other commercially neutral party with respect to the delivery of gaming machine related services) via an autonomous user.

* QCOM user health monitoring

QCOM user quarantines should be extremely rare and are almost always indicative of a software bug. But what if a QCOM user performing a critical function was quarantined? Often the QMA will create an autonomous QCOM user/s to monitor, trap and gracefully finalise a quarantined QCOM user where the risk is appropriate. This type of autonomous user is very small and simple; basically all that is required is a hook on the QCOM user quarantine event and an EGM disable. Approximately three lines of code.

Final points of note concerning autonomous users

* Since the QMA always has first access (refer 6.1.1), functionality provided by QMA autonomous users cannot be easily circumvented. Autonomous user functionality can be incorporated into all user introductions scripts to ensure the desired functionality is always installed onto the machine no matter what users are created or the order they are created in.

### QMA Hijacking

This section deals with the risks associated with QMA hijacking.

A fully RAM-cleared QCOM 3 machine (factory defaults) will accept as the QMA, the first submitted QMA certificate it receives. This provides opportunity for any party with access to the machine’s communications port to install an alternative QMA.

QCOM 3 is designed like this to make testing and development on QCOM 3 machines very easy.

Risk mitigating factors:

* the QMA key fingerprint is able to be verified on the machine’s built-in display (s28). This fingerprint should be verified during machine commissioning. Machine owner/operators/commissioners could also be encouraged to participate here to ensure the integrity of their machine. **Person’s commissioning a QCOM 3 machine, should not leave the machine until they have verified the QMA key fingerprint in machine audit mode.**
* QCOM users can verify that the QMA on the machine is as expected via the QCOM API. Refer QCOM API function qcom\_secQMAcert(), s11.9.1.
* Installing an unexpected QMA draws attention from all other QCOM users associated with the machine. Signed QCOM user introductory scripts will all fail with invalid signature which will trigger an investigation by affected service providers.
* Any party with access to a machine’s communication port / LAN can also verify the QMA certificate installed in a machine (as well as the machine’s ssCert) via the QCOM Command Interpreter (24) *anon* user login.

* The QMA certificate, once set, cannot be changed\* without a physical seal break and a full factory reset procedure being performed on the machine. Note: A RAM clear via the QCOM API retains the last set QMA certificate.

\*Why a QMA change QCOM 3 is only possible via a full factory reset of the machine:

* A temporary QMA hijack (s6.1.4) prior re-commissioning could occur allowing:
  + non-SUA signed upgrades to be installed;
  + the machine to be left in an otherwise a bad or unpredictable state;
  + the creation of unauthorised QCOM users such as QCOM superusers;
  + the attempted imitation of other QCOM users via a hijack of their username.

### Summary: QMA, anon & regular QCOM users

|  |  |  |  |
| --- | --- | --- | --- |
| **QMA vs anon user vs regular QCOM users** | | | |
|  | **QMA** | **anon user** | **QCOM users** |
| **QCOM API Privileges** | Full Access | Very limited | Varies |
| **Jailed environment for script execution** | Yes | Yes | Yes |
| **Event Hooking** | No | No | Yes |
| **Script Execution via** | Signed scripts exec'd by other users | QCI only | Event handlers |
| **QCOM Command Interface (QCI) login** | No | Yes | Yes |
| **Persistence** | Always | Created on demand | Created on demand |
| **Can be quarantined** | No | Yes | Yes |
| **User Script Storage Quota userSetDiskQuota()** | Yes | NA | Yes |
| **User Memory Quota userSetMemoryQuota()** | 100k | 10k | Yes |
| **User Lua Instruction Quota**  **userSetInstructionQuota()** | 999 | 10 | Varies |

Related:

* User Account Storage (s5.3)
* User Memory Quota (s5.4)
* User Lua Instruction Quota (s5.5)
* User Quota Summary (s5.7)

## Software Upgrade Authorities (SUA)

Software Upgrade Authorities are applicable to machine software upgrades and patching which do not also require a physical seal break (e.g. upgrades over a network or via a removable storage device, such as a flash storage device or disk media).

Related: Machine Software Upgrades (section 30)

*SUAs in QCOM are synonymous with OLGR publication “Principals for Remotely Upgradeable Gaming Machines”.*

A SUA in QCOM is denoted by a self-signed x509 certificate. SUA certificates stored by the machine are used for the purpose of authenticating software upgrade packages (refer s30.2) received by the machine before installing. Refer section 30.

A QCOM 3 machine must require that each software upgrade package contains an authentic digital signature of each installed SUA certificate before installing the package. cp:[[73]](#endnote-55) The machine must reject (i.e. not install) any software upgrade package if any of the digital signatures do not successfully authenticate against a public key of a resident SUA or if not all SUA are accounted for (missing a signature). cp:[[74]](#endnote-56) Related: OLGR principles document publication concerning remotely upgradeable Gaming Machines.

A limited number of SUA certificates can be uploaded to the machine via the QCOM API function:

qcom\_secAddSUAcert()

There must always be one persistent, hardcoded[[75]](#footnote-19) SUA for which the machine trusts implicitly and this is the SUA certificate of the machine manufacturer. This SUA must be persistent across all types of machine RAM clears. cp:[[76]](#endnote-57) This SUA can only be change by the machine manufacturer via a software upgrade operation.

The key strength of SUA certificate keys must be sufficient to cover the expected operating life of the machine software for which it is used. Refer section 8 for minimum requirements.

A QCOM machine must accept additional Software Upgrade Authorities via the QCOM API function qcom\_secAddSUAcert() and add them to its trusted software upgrade certificate store.

Once an SUA certificate is uploaded, it must persist until the machine is subsequently RAM-cleared.

* Downloaded SUA certificates can be replaced following a **qcom.machineRAMclear()** operation.
* The machine manufacturer’s built-in SUA certificate can be replaced as a part of a software upgrade operation.

The minimum number of SUA certificates’s the machine must cater for is **4** (including its own). cp:[[77]](#endnote-58)

*In Queensland, QCOM 3 machines will be required to utilise two SUA certificates; the machine manufacturer’s and OLGR’s.*

Related:

* Machine Software Upgrades – section 30
* Refer to the QCOM API \_Security class of functions for relevant functions (section 11.9).
* QCOM API function class prefix machineUpgrade

## Script Approval Authority (SAA)

Each QCOM user may optionally have **one** **(1)** x509 certificate pertaining to a SAA which is installed via the QCOM API function:

qcom\_userSetSAAcert() (refer QCOM summary spreadsheet for more information)

The QCOM user creation default is no SAA and the QCOM user may upload their scripts at their discretion provided they have privileges to either of the following:

* The QCI command: userloadscripts (section 24.2.4)
* The QCOM API function: qcom\_userLoadScripts (section 11.30.27)

If a self-signed SAA certificate has been installed on a machine for a given QCOM user, then the machine must require that all subsequent scripts uploaded for that user by the machine be digitally signed by that certificate’s public key else the machine must delete the scripts without ever letting the QLE Lua software driver compile or execute them. cp:[[78]](#endnote-59)

QCOM users assigned a SAA have no ability to load and execute scripts on a machine unless the scripts are validly signed by the SAA.

The purpose of the SAA is to allow the machine to run applications in the QCOM Lua Engine requiring a high degree of integrity and security given another party is signing off on the scripts before use.

An SAA should keep a copy of all scripts it signs for a reasonable period. An SAA should not sign scripts that restrict copies of scripts being made for diagnostic purposes and investigations by relevant parties, or scripts with complex or obscure copyright terms. Relevant parties are for example: the machine manufacturer, a regulatory authority, the machine owner and the machine operator.

Example applications which may utilise an SAA are applications using an RNG where the outcome involves awarding prizes, such as jackpot triggering algorithms (refer section 10.12.3), or a residual credit removal gamble feature (section 10.12.11).

**In the first few years of the operation of QCOM 3 machines and until further notice, all QCOM users will be SAA controlled.**

Related:

* Privacy and Trust Implications (s10.2.4)

# Privileges

*Note: almost all requirements in this section are implemented as a part of the QLE Lua software driver. Related: section 33.1.*

The privilege system in QCOM 3 is very simple and is based on individual QCOM user access rights to specific functions in the QCOM API.

The privilege of being able to invoke a QCOM API function or QCOM Command Interpreter command (24) is controllable on a per function basis for each QCOM user via the two QCOM API privilege functions below:

qcom\_userSetPrivilege()

qcom\_userPrivileges() –- query function

At a machine RAM clear / factory defaults, only the QMA will have the rights to invoke the entire QCOM API. The QMA may elect to retain the role of setting privileges (typical) or delegate the role to another user.

Permissions are binary i.e. a given QCOM user either has permission to call the given QCOM Lua API function / QCOM Command Interpreter command, or they don’t.

## Non-privileged QCOM API function calls

QCOM users will always know what QCOM API functions they have privileges to beforehand and this can also be verified by the user at any time. Calls to QCOM API functions for which the QCOM user does not have privilege should therefore not occur in operation. However, since it’s physically possible, it’s necessary to define standard behaviour in this case.

When a call to a QCOM API function is made for which the invoking QCOM user is not privileged to access, the following options below were considered:

***The QLE Lua software driver implements option 2. Option 2 is inherent to the QCOM 3 SDK and does not require specific implementation by the machine manufacturer.***

Option 1 – Assert Friendly

If a user does not have the rights to call a given QCOM API function and they are able call it regardless, the call must have no effect (i.e. the function returns immediately) with the two return values: nil, “permission denied”*.* cp:[[79]](#endnote-60) *(This approach is Lua assert() friendly)*

For example if a user called the following QCOM API functions without the privilege to do so:

a, b = qcom\_idSetMachineID()

a, b = qcom\_ntpSetHost(“garbage”)

then regardless of any arguments provided or possible documented return values listed in the functions full description, the return values must, in all cases, be:

a = nil, b = “permission denied”

The following QCOM state event would also be thrown:

QCOM\_API\_NON\_PERMITTED\_CALL cp:[[80]](#endnote-61)

and keep a meter of the number of occurrences (per user and global) and report this information on any QCOM Lua Engine status display cp:[[81]](#endnote-62).

Option 2 – Runtime Error

If a non-privileged user attempts to invoke the function the QCOM Lua Engine (Lua) will generate a Lua runtime error of the form:

“: attempt to call global ‘funcName’ (a nil value)”

or,

“: attempt to call field ‘funcName’ (a nil value)”

This approach is inherent Lua interpreter behaviour and does not require any implementation. The Lua interpreter will gracefully abort the execution of the current script for the given user which would then be handled as a general error of a Lua script. Refer 10.2.11.

## Multiple User Environment considerations

Most QCOM API functions are inherently designed to allow multiple users the right to call the function without any contention issues. Examples are all the ‘Get’ or ‘Read-only’ functions. However in some cases, it should be noted that some QCOM API functions are only intended to be privileged to a single QCOM user. One example is the **qcom\_ectSubtractCredit()** function.

Refer to the QCOM API summary table ‘*nUsers*’ column for the status of each QCOM API function in the above regard.

Other QCOM API functions which can modify the EGM’s state or data are typically only assigned to a single user in order to avoid contention issues. Examples are the majority of the ‘Set’ API functions. However, in some rare cases, some functions which can modify the EGM’s state or data are specifically designed to allow multiple users to safely share rights to invoke the given function. One example of this is the qcom\_playSetPEF() function. In this case, the target data is stored on a per user basis in order to provide each user with rights to disable play on the EGM.

## Default Permissions

The QLE Lua software driver creates QCOM users (s11.30.1) with zero privileges.

Related:

* Section 10.2.7 Data Persistence.

# Certificates

QCOM machines utilise a number of certificates related to QCOM related authorities and services. Specifically:

* A Local Machine self-signed Certificate. (The local machine CA.)
* The QCOM User Account Access (UAA). Refer section 23.
* The QCOM WWW interface. Refer section 29.

Private keys held by the machine must be stored securely. cp:[[82]](#endnote-63) The machine must not have any ability to disclose its private keys by any reasonable means without leaving evidence behind (for example a broken security seal). In all cases accessing private keys in a machine should require specialised access, equipment and knowledge. cp:[[83]](#endnote-64)

The machine must display all held certificates (excluding private key data) and associated fingerprints (refer section 24.2.2 for fingerprint display format) in audit mode (s28) and allow this information to be downloaded via the QCOM web server on demand (if implemented). cp:[[84]](#endnote-65) The default format is PEM, however certificate displays must be supplemented with a human readable display of certificate metadata such as a text dump of each certificate as per the output of the openssl command: **openssl x509 -in cert.pem -noout –text**.cp:[[85]](#endnote-66)

The machine must not create any private keys prior to confirmation of the machine’s logic seal. Refer to Section 4.4.

Any certificate provided via the QCOM API or other interface in the machine must never contain the private key. cp:[[86]](#endnote-67)

The hashing algorithm that must be used, where not stated otherwise, is **SHA256**. cp:[[87]](#endnote-68)

The minimum RSA algorithm key length, where not stated otherwise, must be **2048** bits. cp:[[88]](#endnote-69)

The certificate lifetime/expiry, where not stated otherwise, must be **3650** days.

Unless stated otherwise, QCOM machines must **not** validate certificate not-before / not-after dates. cp:[[89]](#endnote-70)

Machines must be able to support all algorithms contained in the openssl release current at the time the machine software was programmed. This requirement is regardless of whether or not the machine uses openssl.

The following sub-sections describe all the QCOM 3 specific local machine certificates.

## The Local Machine CA Certificate

A local machine CA certificate (also referred to as the machine self-signed certificate) must be generated by the machine upon logic area seal confirmation. Refer section 4.4.

Required certificate parameters:

**Type**: x509 self-signed certificate

**Country** (/C) = <Machine manuf. registered business address country>

**Organisation** (/O) = <Return value (RV) of QCOM API function: qcom\_idMfr() (refer s11.1.8)>

**Common Name** (/CN) = CA <RV of qcom\_idMfr3()> <RV of qcom\_idDeviceType()> <RV of qcom\_idLogicUID()>

Example (openssl / qsim):

openssl req -x509 -subj "/C=AU/O=OLGR/CN=CA OGR egm aa:bb:cc:dd:ff:00" -nodes -SHA256 -days 3650 -newkey rsa:2048 -keyout qsim\_privkey.pem -out qsim\_sscert.pem

Gives:

Country (/C) = AU

Organisation (/O) = OLGR

Common Name (/CN) = CA OGR egm aa:bb:cc:dd:ff:00

Local Machine CA Certificate generation must complete before certain services are started in the machine. Refer section 4.4 for more information.

QCOM users can retrieve the local machine CA certificate via the QCOM API function qcom\_secMachineCert() s11.9.4.

This local machine self-signed certificate is used by the machine to create additional certificates for use with the machine’s:

* UAA service.
* WWW service.

## UAA Service Certificate

The UAA certificate is related to QCOM User Account Access to allow the machine to authenticate itself to UAA clients. Refer section 23. This certificate must be generated by the machine upon logic area seal confirmation (s4.4) using the local machine CA certificate as the root CA.

Required certificate parameters:

**Type**: x509 self-signed certificate

**Country** (/C) = <Machine manuf. registered business address country>

**Organisation** (/O) = <Return value (RV) of QCOM API function: qcom\_idMfr() (refer s11.1.8)>

**Common Name** (/CN) = UAA <RV of qcom\_idMfr3()> <RV of qcom\_idDeviceType()> <RV of qcom\_idLogicUID()>

Example (openssl / qsim):

openssl req -x509 -subj "/C=AU/O=OLGR/CN=UAA OGR egm aa:bb:cc:dd:ff:00" -nodes -SHA256 -days 3650 -newkey rsa:2048 -keyout qsim\_privkey.pem -out qsim\_sscert.pem

Gives:

Country (/C) = AU

Organisation (/O) = OLGR

Common Name (/CN) = UAA OGR egm aa:bb:cc:dd:ff:00

Related:

* QCOM User Account Access, section 23.

## WWW Service Certificate

This certificate must be generated by the machine upon logic area seal confirmation (s4.4) using the local machine CA certificate. It is basically a self-signed certificate but this certificate’s common name is slightly different to the certificate in s8.1.

Required certificate parameters:

**Type**: x509 self-signed certificate

**Country** (/C) = <Machine manuf. registered business address country>

**Organisation** (/O) = <Return value (RV) of QCOM API function: qcom\_idMfr() (refer s11.1.8)>

**Common Name** (/CN) = WWW <RV of qcom\_idMfr3()> <RV of qcom\_idDeviceType()> <RV of qcom\_idLogicUID()>

Related:

* By QCOM WWW Interface section 29.

## QCOM Authorities: QMA, SAA’s & SUA’s

Authority: QMA / SAA / SUA

Type: x509 self-signed certificate

Key length: may vary\*

Signature Algorithm: may vary\*

\*These certificates come from sources **external** to the machine and therefore their parameters may vary within the range of algorithms in which the machine’s crypto API supports. The machine should not be provided with the private keys in relation to these certificates.

# Timekeeping

QCOM 3 machines must incorporate a reliable, industry standard, hardware based NV Real-Time Clock (RTC) for persistent timekeeping across interruptions and power downs. cp:[[90]](#endnote-71)

The machine may maintain its NV RTC in either local time or UTC time.

In a QCOM 3 machine, the time may be set via the following methods:

* Post RAM clear, during machine power up via a machine built-in user interface but only until the machine’s logic seal is confirmed. Refer section 4.1.
* Via the QCOM API function qcom\_timeSet() - privilege permitting. Refer section 11.5 for QCOM API functions related to time keeping.
* Via a NTP client in the machine (if enabled) from a designatable NTP server. Related: section 11.4 for the API.

Any timers and delay functions used by the machine must not be affected by time changes. cp:[[91]](#endnote-72) A specific mention here to the QCOM 3 global known as the *machine operating time*.

The machine must only use the local time for event time stamping and display purposes and must not use any representation of world / local time in any calculation or other operation except for Certificate authentication (not-before, not-after validation) if required. cp:[[92]](#endnote-73)

Machines must **not** automatically adjust their clock for daylight savings or leap seconds. cp:[[93]](#endnote-74) QCOM 3 manages daylight savings through the QCOM 3 time zone bias value. Refer sections 4.1 and 11.5.

Conversions:

UTC = (local date and time) – (time zone bias where east of GMT is positive)

*Related: The QCOM v1 TZADJ parameter.*

## Timekeeping Events

The machine must log all accepted time and time zone changes performed by any possible method (i.e. machine UI & QCOM API) via QCOM state event (s14) and an event (s13). The state event must be logged first re section 14.3.

Time changed events must be thrown roughly within a second **after** any time / time zone change has been applied by the machine. QCOM 3 timekeeping state events (and the corresponding event) must not be generated more than the rate of once per second.

Exception: Small time adjustments of less than **5 seconds** must not be logged by the machine or throw a state event. (Same as QCOM v1) For example:

qcom.timeSet(os.date(“\*t”)) – setting time to current time => no events logged

*The above script would not cause any time changed events.*

The QCOM state event the machine must throw is the **TIME\_CHANGED** state event. cp:[[94]](#endnote-75)

The QCOM event logged by the machine must be the **MACHINE\_TIME\_CHANGED** event. The timestamp on the event must correspond to the new time. cp:[[95]](#endnote-76)

Refer to the respective event definition in the QCOM summary spreadsheet for more information.

Related:

* section 4.1 - Date and Time Configuration
* section 11.5 - Timekeeping QCOM API
* section 13.5 - QCOM Event Schema

# The QCOM Lua Engine

*Note: almost all requirements in this section are implemented as a part of the QLE Lua software driver. Related: section 33.1.*

*Design Requirements:*

* *Implement a machine application embedded QCOM Lua Engine that utilises a scripting engine that is compact, secure and jailed by default.*
* *Where possible, all QCOM functionality that was previously performed by the machine will now be performed by a script, minimising the amount of functionality that is hard-coded into the machine and thus maximising the machine’s ability to meet the needs of a diverse range of operating environments.*
* *The ability to restrict any QCOM user to the execution of scripts that have also been authenticated (i.e. digitally signed) by a third party. This makes it possible to implement critical operations and functionality that requires third party authentication before use. For example, gambles, prize and jackpot triggering and other high risk regulatory functions such as audit and verification and the implementation of regulatory policies.*

Why use a scripting engine in QCOM?

Using a scripting engine and defining an API achieves the following benefits:

* Allows fully on-line, partially on-line, or totally offline modes of operation.
* Offline operations can have programmed intelligence.
* Services can be made more immune to disruptions in communications.
* The implementation and delivery of new functionality inherently falls back onto the party demanding the functionality.
* Promotes independence of delivery of services.
* Makes services far more immune to network latencies and network issues.
* Services implemented in script will have zero latency and be immune to network issues.
* Arbitrary network protocols can be implemented by a QCOM 3 machine.
* Allows risks and issues to be easily addressed in otherwise predominantly hard-coded machines. This is significant given the underlying use of internet protocols in QCOM 3.
* Uniform behaviour across different makes and model machines is easier to achieve than compared with a protocol based interface. Refer section 33.1.

The scripting engine adopted by QCOM 3 is **Lua**. 

For more information on Lua refer to:

<http://www.lua.org/>

Lua was specifically chosen for QCOM 3 for the reasons summarised below:

* Lua is specifically designed to be embedded in an existing application.
* The Lua Interpreter is small and fast.
* Lua is well established, stable and supported.
* Lua is written in “C” and has minimal OS, software and hardware dependencies. For example even the most recent version of Lua successfully compiles with zero warnings on a turbo C compiler dated 1990. A C compiler is Lua’s only dependency.
* Lua has a small learning curve.
* Lua is resource minimal by default and provides the hosting application (the EGM) significant monitoring and control over the resources granted to any Lua scripts to be executed.
* Lua specifically has powerful sandboxing and jailing capabilities built into the language specification (refer Lua closures / sandboxes). This makes it perfect to meet all QCOM’s design requirements relating to diverse operating modes e.g. multi-user support, trusted/non-trusted users, creating dependencies, data sharing etc. *This was a major feature contributing to the specific choice of Lua for QCOM.*
* Lua is open source and free to use and modify. Specific advantages here are:
  + Issues can be addressed by third parties.
  + The source can be modified as required to meet special design requirements if any.
* Lua compiles onto a huge range of platforms. Specifically any platform that supports a C compiler.
* Being C based and open source, the longevity / lifespan of Lua is that of C. So long as C compilers exists, then so will Lua.
* The Lua documentation is excellent.
* Lua tables (Lua’s only complex structure type) are easily natively serialised which is beneficial for protocol development.
* Lua is effective in dealing with data of an unknown and extensible nature which is perfect for protocol applications.
* Lua is simple and makes the process of reviewing and controlling source code light on resources.
* Lua is small and it’s therefore more likely that the QCOM Lua Engine will be implemented as a part of the main machine application as intended.
* Good support for function interface overloading means a range of issues that could potentially occur in the hard-coded QCOM Lua Engine can be addressed without having to retrofit the machine’s hard-coded firmware.
* The Lua engine is a process virtual machine which, in QCOM sits in between the core machine and the network. This adds an additional layer of security with respect to the protocols implemented by QCOM users. Lua being a high level, memory managed language, is immune to a wide range of network protocol based attacks. (Arguably a good reason why the protocol analyser software Wireshark uses Lua.)

## Machine hosted Lua Interpreter - Core Requirements

*Design Requirements:*

* *The integrity and security of the host machine is never compromised.*
* *Critical: QCOM users must not have to implement different versions of scripts across different manufacturer machines, makes and models (given the same version of QCOM, Lua and QCOM API has been implemented). The same QCOM user script executed on different machine makes and models must not require any modification to work.*

The machine must incorporate a Lua language interpreter for use with QCOM. cp:[[96]](#endnote-77)

The machine must use **v5.3.4** of the Lua interpreter**.** cp:[[97]](#endnote-78)

(Referred to as simply “Lua v5.3” throughout this document.)

Refer section 10.2.13 regarding new releases of Lua cp:[[98]](#endnote-79)

**QCOM Lua Engine operating environment:**

The QCOM Lua interpreter must be implemented within its own process or thread[[99]](#footnote-20) in the machine. This process/thread is referred to as the **QCOM Lua Engine (QLE)** throughout this document.

The priority of the QCOM Lua Engine should be equivalent to normal or lower with respect to the machine’s operation system / environment. Depending on the machine.

Within this hosting process/thread, the machine must utilise a single Lua state instance in which all QCOM user scripts must execute via the QLE Lua software driver. cp:[[100]](#endnote-80)

Within this single Lua state, QCOM user’s scripts will be executed in a **Lua environment per user** (refer Lua reference manual: \_ENV) by the QLE Lua software driver. cp:[[101]](#endnote-81)

QCOM user scripts are executed as event handlers by the machine inside jailed environments within the QCOM Lua Engine. Refer section 14.cp:[[102]](#endnote-82)

**The QLE Lua software driver is designed so that the host machine’s state will appear frozen to QCOM users during execution of their scripts**:

In the QCOM Lua Engine, during the execution of a set of scripts hooked to a given state event, the state and meters of the machine’s main application as seen by QCOM users inside the QCOM Lua Engine will not change during the execution of those scripts. In other words, the host machine’s state appears **frozen in time** to a set of scripts executing inside the QCOM Lua Engine as a result of a thrown QCOM state event.[[103]](#footnote-21) From the perspective of QCOM users, the state of the machine’s main application will only change between successive state events.

The above core requirement ensures that all scripts attached to a given state event, when executed, will all see exactly the same machine state. Related section 14 on State Events.

The QLE Lua software driver ensures this ‘frozen’ behaviour (from the perspective of QCOM users) by requiring the host machine to use an event based interface/feeder between itself and the host machine’s main application. On start up, the machine must send its current full state to the QCOM Lua Engine (as an event) and from that point on it must keep the QLE up to date by sending further events over time. While the QLE Lua software driver is servicing one or more QCOM user scripts triggered by a state event, the host machine must buffer any new events that may occur during this time. The main application must not wait here either unless the state event is defined as a sync state event (refer s14.3). As a result, from the perspective of QCOM users who only see the state of the machine that is currently stored in the QCOM Lua Engine, the machine appears to be frozen in time during script execution. cp:[[104]](#endnote-83)

*Design Notes:*

*This approach for the hosting of the QCOM Lua Engine has the following design rationale / advantages:*

* *CPU and memory efficient.*
* *No single user can block or degrade the progression of the machine’s main application as the QCOM Lua interpreter is running in its own process[[105]](#footnote-22).*
* *As of QCOM 3 draft 3; a QCOM user can no longer:*
  + *block another users script by committing quarantine suicide, or*
  + *delay another users script for any longer than their instruction quota allows.*
* *Enables simple and easy sharing of functions, libraries and data between users. Refer section 10.2.6*
* *QCOM API and hosted scripts do not need to be thread-safe allowing for faster implementation.*
* *The machine’s application is protected by three layers, i.e. QCOM user’s scripts execute in a jail (\_ENV) within a virtual machine (being the Lua scripting engine) within an OS protected process.*

Deterministic behaviour; core requirements:

Given any QCOM user script, if the same script does not behave identically across all machines implementing the same version of QCOM (i.e. the script would require modification in order to perform the same functions or service), then (so long as this is not as a result of an ambiguous or subjective QCOM requirement), then the machine in question will be deemed non-conforming to QCOM.

The QCOM 3 SDK contains a full implementation of the QCOM Lua Engine written in Lua referred to in this document as the QLE Lua software driver. It is a core requirement that the machine must utilise the most recent version of the QLE Lua software driver (QLE LSD). cp:[[106]](#endnote-84) Refer <qsim>\lua\readme.txt for a list of what modules makeup the QLE LSD. Contact OLGR if any modifications to the QLE Lua software driver source code modules appears necessary.

## Implementation Considerations

The QCOM 3 SDK contains a fully implemented QCOM Lua Engine which adheres to these requirements. If the SDK is utilised, then a large proportion of following requirements would not have to be implemented from scratch by the machine manufacturer.

An advantage in using a scripting language interface as part of QCOM 3 is that the Lua script code made available in the QCOM 3 SDK will be inherently drop-in usable by all machine manufacturers. It will also help result in unified behaviour with respect to the QCOM interface across all manufacturers’ machines. Finally the interface enables fully automated machine interface implementation testing as it can be fully scripted.

### Memory Usage, Monitoring and Control

A QCOM machine must be able to hold in resident memory the QCOM Lua interpreter and QCOM user environments. cp:[[107]](#endnote-85) Refer section User Memory Quota (s5.4).

The QCOM Lua software driver is able to exactly monitor each individual QCOM users memory use in real-time, and immediately abort a script that exceeds its memory quota limit before the memory is actually allocated.

In production machines, a QCOM user will be assigned a memory limit via the QCOM API function:

qcom\_userSetMemoryQuota() -- s11.30.14

The instant a QCOM user tries to exceed this limit they will be automatically shutdown and quarantined by the QCOM Lua software driver.

There are two specific facets of Lua to note:

Memory management: The Lua garbage collector

Memory leaks in Lua are not applicable. In Lua, the task of freeing up memory is performed by the Lua garbage collector. The Lua garbage collector is typically automatic but may also be manually controlled as much as desired. Even though garbage collection may be manually controlled, when the Lua garbage collector actually decides to free unused memory blocks is highly unpredictable.

Memory management: The Lua memory allocator

The Lua interpreter allows for the host program to supply a custom memory allocation / deallocation function for which it calls for all Lua memory use.

In order to monitor QCOM Lua Engine memory usage on a per user basis, the machine must supply a custom *lua\_Alloc[[108]](#footnote-23)* function for the QLE, which must be based on[[109]](#footnote-24) the QSIM function qlua\_alloc()[[110]](#footnote-25) in the QCOM 3 SDK. cp:[[111]](#endnote-86) Refer the QCOM 3 SDK qlua.cpp module for the source code to this function.

Once the required lua\_Alloc function has been set during initialisation[[112]](#footnote-26), the Lua interpreter instance used for the QCOM Lua Engine will direct all memory allocations / deallocations through this function making it possible to totalise memory usage per QCOM user as required for the application QCOM user memory quotas. The QCOM SDK supplied *lua\_Alloc* function also makes it possible to validate all memory allocation against the user’s quota and to deny unreasonable memory requests before any allocation attempt takes place.

When the machine supplied *lua\_Alloc* function detects that a user’s memory quota will be exceeded during the machine supplied memory allocation function*,* it will refuse to allocate the memory and return NULL. Returning NULL will cause the Lua interpreter to automatically gracefully abort the running script with the error “not enough memory”. On return, the QCOM Lua software driver will log the **USER\_QUARANTINED** message with details and quarantine the QCOM user (5.8). The execution of all other user scripts won’t be affected by this event.

Individual QCOM user memory usage is achieved by qlua\_alloc() by tagging each individual memory block allocated with a pointer to a small structure that identifies the owning QCOM user and the details of their current memory use totals. An extra 4-8 extra bytes of memory (i.e. **sizeof(\*void)** ) is allocated at the end of each memory block to hold a pointer to the struct. When there is no owner (e.g. when QLE Lua software driver code is running), the extra bytes are filled with zeros by the qlua\_alloc() *lua\_Alloc* function.

On the next page is the typical QCOM SDK qlua\_alloc() *lua\_Alloc* function a machine would use for QCOM user memory monirting and control. (Machines should use the latest version from the QCOM 3 SDK in all cases.) This function is the crux of QCOM user memory use monitoring and in general represents a solution for Lua for the question:

How much memory does function f() use?

QCOM SDK qlua\_alloc() *lua\_Alloc* function: (Always use the latest code from the QCOM 3 SDK module : qlua.cpp)

static void **\***qlua\_alloc\_no\_qsim**(**void **\***ud**,** void **\***ptr**,** size\_t osize**,** size\_t nsize**)**

**{**

void **\***newptr**;**

qlua\_qusermem\_t **\***r\_memstatsp**;** // the object type that memory may be tagged with a ptr of.

QLuaType **\***lp **=** **(**QLuaType **\*)**ud**;** // access to the Lua state qlua struct. It must never be null.

int increment **=** nsize **-** **(**ptr **?** osize **:** 0**);** // Lua v5.3 specific

**if** **(**ptr**)** // then it's a free or realloc; getthe existing qlua\_qusermem\_t pointer (may be NULL)

memcpy**(&**r\_memstatsp**,** **(**void **\*)((**char **\*)**ptr **+** osize**),** **sizeof(**void **\*));**

**else** // it's an alloc

r\_memstatsp **=** **NULL;**

**if** **(**nsize **==** 0**)** **{**

**if** **(**ptr**)** **{** // then its a free operation

free**(**ptr**);**

**if** **(**r\_memstatsp**)** **{**

r\_memstatsp**->**memused **+=** increment**;**

**}**

**}**

**return** **NULL;**

**}**

**else** **{** // its either a alloc or relloc

// If a user script is running will the user's memory quota has be exceeded?

**if** **(**lp**->**allocUserRunning **&&** **!**lp**->**memCriticalSection**)** **{**

**if** **(**ptr**)** **{** // if (ptr) then its an realloc

**if** **(**r\_memstatsp **&&** **(**r\_memstatsp **==** lp**->**usermemstats**)** **&&** **((**r\_memstatsp**->**memused **+** increment**)** **>** r\_memstatsp**->**memlimit**))** **{**

r\_memstatsp**->**memlimitbreach **=** true**;** // Memory quota breach! Quarantine!

**return** **NULL;** // generate a Lua runtime error "not enough memory".

**}** // NB Lua will try twice to get the memory before giving up.

**}**

**else** **{** // else its a alloc

**if** **((**lp**->**usermemstats**->**memused **+** increment**)** **>** lp**->**usermemstats**->**memlimit**)** **{**

lp**->**usermemstats**->**memlimitbreach **=** true**;** // Memory quota breach!!!

**return** **NULL;** // force the Lua runtime error "not enough memory".

**}** // NB Lua will try twice to get the memory before giving up.

**}**

**}**

newptr **=** realloc**(**ptr**,** nsize **+** **sizeof(**void **\*));**

**if** **(!**newptr**)** **return** **NULL;**

**if** **(**r\_memstatsp**)** **{**

r\_memstatsp**->**memused **+=** increment**;**

memcpy**((**char **\*)**newptr **+** nsize**,** **&**r\_memstatsp**,** **sizeof(**void **\*));**

**}**

**else** memset**((**char **\*)**newptr **+** nsize**,** 0**,** **sizeof(**void **\*));** // zero it otherwise

// if (a user script is running && it was a new memory alloc) then update their mem stats

**if** **(**lp**->**allocUserRunning **&&** **!**ptr**)** **{**

// copy a pointer to the QCOM user's memstats struct to the memory just alloc'd

memcpy**((**char **\*)**newptr **+** nsize**,** **&**lp**->**usermemstats**,** **sizeof(**void **\*));**

lp**->**usermemstats**->**memused **+=** increment**;**

**}**

**return** newptr**;**

**}**

**}**

// where qlua\_qusermem\_t is defined as:

**typedef** struct **{**

char tag**[**QLUA\_USERDATA\_TAG\_LEN**];** // failsafe in the event the wrong userdata is passed

char username**[**QLUA\_UNAME\_MAX\_LEN**];** // populated but not used

int memlimit**;** // set just prior each call to a user script (in case its been changed)

int memused**;** // the user's current memory usage

unsigned int nobjs**;** // total number of memory alloc'd objects for this user

bool memlimitbreach**;** // i.e. memory quarantine will result

**}** qlua\_qusermem\_t**;**

### Software Exceptions

The QLE Lua software driver performs all the work in keeping QCOM user scripts inside jailed environments, but there are a couple of exceptions for which the host machine must specifically deal with. These exceptions are dealt with by this section.

Stack Overflow

There are two types of stack overflow conditions of interest that can occur in a QCOM machine.

The first type concerns a stack overflow of the QCOM Lua Engine host process. Stack overflows here will not occur unless there is a bug in the machine software or Lua interpreter[[113]](#footnote-27). Whatever the machine is currently doing with its other applications with respect to stack overflows is also acceptable for the QCOM Lua Engine process/thread provided it meets any other regulatory requirements, if any.

The second type of overflow concerns an attempted stack overflow of the QCOM Lua virtual machine stack. For example the Lua script: function f() f() end f();. In Lua v5.3 and above, this and similar code does not cause a host process/thread stack overflow, but is caught by Lua interpreter which generates a “stack overflow” runtime error and gracefully exits the script as per any script runtime error. In a QCOM machine however, before this will occur, the user’s memory quota (s5.4) or instruction quota (s5.5) will trip first (depending which is set the lowest) and the issue dealt with as per the requirements in those sections which would result in the user being quarantined.

cp:[[114]](#endnote-87)

Divide by Zero

In many programs that host a Lua interpreter, the following script:

a = 1/0 print(a)

will output ‘+INF’.

This result is also the mandatory behaviour for the QCOM Lua Engine in a QCOM 3 machine. cp:[[115]](#endnote-88)

**This behaviour above is not inherent to the Lua interpreter and must be setup by the host program.**

The required behaviour is setup by disabling floating point program exceptions in the process/thread hosting the QCOM Lua Engine. This can only be done by the machine. How to do this is highly operating system / platform / development environment dependent, but there is an example in the QCOM 3 SDK applicable to Windows OS based machines. For other platforms, there are plenty of code examples available on the internet for disabling floating point program exceptions.

If implemented correctly, floating point number exceptions should only be masked out in the QCOM Lua Engine hosting process/thread and the wider machine should still behave as desired. cp:[[116]](#endnote-89)

QLE Lua software driver runtime error

This scenario should never occur outside of test but must still be handled by production machines as a failsafe.

Detection: A protected mode call[[117]](#footnote-28) into the QLE’s Lua state returns an error value.

Testing: QCOM API function **qcom.luaPanic()**.

The machine must set a flag that prevents its QLE code from calling any function/script within the QLE Lua state until next machine restart. cp:[[118]](#endnote-90)

The machine must shutdown its UAA listen service and drop all UAA sessions until next machine restart cp:[[119]](#endnote-91)

The machine must also log the **QCOM\_ENGINE\_EXCEPTION** event cp:[[120]](#endnote-92) and lock up in a fault condition. cp:[[121]](#endnote-93)

The machine’s main application must be suspended as much as possible while the QLE is in this exception state’s fault condition. cp:[[122]](#endnote-94)

This exception’s fault condition must only be able to be reset after a machine restart. cp:[[123]](#endnote-95)

Other *never should happen* type exceptions

A QCOM 3 machine receives data from the QLE Lua software driver that it may perform sanity checks of at its discretion. This data is in the following forms:

* QLE LSD (*sendToHost*)messages and data
* Parameters to QLE C Lua functions

The machine’s handling of these exceptions must be as per QLE Lua software driver runtime errors above.

### CPU Usage, Monitoring and Control

Note:

* The method of CPU Usage, Monitoring and Control described in this section is typically not indicative of QCOM user CPU use; as on some machines it may include occasional spikes from Operating System (OS) process switching, interrupts and I/O.
* This section defines what should be considered as primarily a benchmarking tool.
* QCOM user CPU usage is actually controlled by the *User Lua Instruction* Quota (section 5.5).

The QLE Lua software driver will totalise CPU usage per QCOM user and make current usage statistics available via the QCOM API.

The timing precision must be microseconds or smaller. In the process of timing, it is accepted that a QCOM user script may be temporarily interrupted during execution (e.g. an operating systems task switch) and the time relating to the interruption might be added onto the respective QCOM user’s CPU quota. Rationale:

* A perfect level of accuracy is not required and is typically CPU costly to achieve.
* Makes QCOM user CPU totalisation highly efficient and compatible with the way many machine operating systems work.
* Because the QCOM Lua Engine spends so little time actually executing, interruptions should be infrequent.
* Any spikes in timing go against QCOM users and don’t work for them.
* Any spikes in timing are useful information during machine evaluations.
* Most Operating Systems do not have high precision timing of threads/process. Operating System API’s in this area are often fairly granular in timing. For example, in the MS Windows version of QSIM 3, the Windows API function GetThreadTime() typically returns a zero increment for the QCOM Lua Engine thread as it uses so little CPU.

The QMA or delegated authority should take possible CPU spikes into account when setting QCOM user CPU quotas.

Related: QCOM API functions:

qcom\_userSetCPUquota()

qcom\_userCPUstats()

The QLE Lua software driver will check a QCOM user has not exceeded their CPU quota each time a QCOM user script returns and just before returning from certain potentially CPU intensive non-Lua implemented QCOM API functions. Refer to the QCOM 3 SDK for the list of functions.

A QCOM user that exceed their CPU quota will be quarantined by the QLE Lua software driver. Refer section 5.8.

*Tech info: It should also be noted that with respect to the execution model for the QCOM Lua Engine in this document (i.e. event based – time frozen), that it makes no sense for a QCOM user to invoke most QCOM API functions more than once in a given event handler. This property may be used in subsequent releases of this standard to prevent QCOM API spam. It also means that the machine can cache return values for specific QCOM API function calls whose return value cannot change during a single set of scripts executing for a given state event. This may eventually lead to some QCOM API functions being replaced outright with simple global read-only tables shared across authorised QCOM user’s environments (jails).*

### Privacy and Trust Implications

**Privacy of QCOM user scripts**

There is no privacy pertaining to QCOM user scripts in a QCOM 3 machine. QCOM user scripts once download into an EGM QCOM user script are viewable via EGM audit mode (s28). Accordingly QCOM users must not hardcode secrets into their scripts.

If an SAA is present in the machine for a given QCOM user (refer section 6.3), the SAA will require copies of the script to sign and will also keep copies for possible diagnostic purposes and investigations.

The QMA will always have read access to all QCOM user scripts uploaded to a QCOM 3 machine.

QCOM user scripts will be provided to the machine manufacturer and the regulator on demand for diagnostic purposes and investigations. There must be no restrictions or approvals from script copyright owners required in order to do this. An SAA will be unlikely to sign scripts with restrictions here, or scripts with complex or obscure copyright terms. Refer section 6.3.

QCOM user script copyright must not restrict copies of their scripts from being made for the purpose of diagnostics or investigations by relevant parties. Relevant parties are for example: the machine manufacturer, a regulatory authority, the machine owner and the machine operator.

In the State of Queensland (where the SAA and QMA will be OLGR), QCOM users who are also licensees under gaming legislation may incur additional evaluation fees if their script’s copyright licenses are custom made, or long or complicated (not plain English), or generally not well known.

**Privacy of QCOM user variables**

Any QCOM user created regular variables created in their QLE “jail” will remain private in commissioned, production QCOM 3 machines, i.e. machines not running in development or debug mode.

Note however if the user passes variable data to a QCOM API function or some QLE library functions, then it may be possible under some circumstances for the QMA to access it. Related: “QCOM API function overloading” section 10.8.

During normal operation of a QCOM 3 machine, the only other methods for a QCOM user to share variable data publically, is via the QCOM API function qcom\_luaPublish()*,* or print() it, or transmit it over the network.

Related:

* “QCOM API function overloading” section 10.8.
* qcom\_machineInDebugMode() section 11.7.22

**Persistent Variables**

Refer Section 10.10. Privacy and Trust Implications here is as per variables (above).

**Privacy of functions**

As per variables above.

**Trust - General**

Any output resulting from a Lua script running on a QCOM machine can only be trusted as far as the recipient of the output or script owner trusts the hosting machine and the script’s authenticity, whichever is lower. This trust level is dependent on the implemented QCOM mode of operation, which is typically established at machine commissioning and also depends somewhat on the specific commissioning procedure used. If for example, the machine has been compromised (e.g. logic area access) then theoretically speaking the validity of any further data provided from either the machine or any Lua scripts running on it, must be put into question.

### Global & QCOM User Environment Protection

The QLE Lua software driver will restrict all access by QCOM users to the QCOM Lua Engine global environment. Lua makes this requirement easy as Lua has built-in support for virtual environments or “jails” (also referred to as a “sandbox”). When a QCOM user is jailed[[124]](#footnote-29), Lua will automatically protect the Lua global environment from view or changes by any QCOM user.

### Sharing Data Between QCOM Users

It may be convenient for QCOM users to be able to share data among themselves.

A method in which a QCOM user can share specific types of data with other QCOM users is via the **qcom\_luaPublish()** QCOM API function (refer section 11.15.10). This function allows a QCOM user to share arbitrary datawith one or more QCOM users.

The QCOM API permits read-only access to shared data via the above QCOM API function.

QCOM users must note the following risks associated with sharing functions in the QLE:

* QCOM users must never call a shared **function** of another user unless they implicitly trust the user. This is because all usage during shared function execution (w.r.t. QCOM user quotas) counts towards the calling user.
* A QCOM user sharing a **function** must be careful with respect to the shared function return values. A shared function must be careful to return newly created strings and tables, or copies of strings[[125]](#footnote-30) and tables and not references.

**QCOM users should not share functions with other QCOM users unless the user (i.e. programmer) has a thorough understanding of Lua and Lua environments / jails and associated risks as implemented in the QLE.**

Refer to the QCOM SDK for more information and examples.

*Design note: Allow UDP localhost sockets? Could be another safe way for users to share data. Comments welcome.*

### Data Persistence

This section summarises QCOM 3 related persistent data, or NV data for short.

In this section the term ‘persist’ refers to data that must be stored in machine NV memory.

Regular QCOM users created via the QCOM API function qcom\_userCreate() must persist across machine restarts until specifically deleted via the QCOM API function qcom\_userDelete() or the machine is again RAM cleared. cp:[[126]](#endnote-96)

For a complete list of QCOM 3 required persistent data that the machine must load into the QLE Lua state on start-up refer to the QCOM 3 SDK modules:

* hms\_schema.lua
* qusersNVdata\_schema.lua

During operation, The QLE Lua software driver sends a message to the host machine whenever there is a change in state or data that the machine must store in NV memory. Related: The *qapi\_sendToHost()* method (see QSOM 3 SDK) and also refer to QCOM 3 summary spreadsheet: *sendToHost* worksheet for a list of all QLE to machine messages.

### EGM “Meters” and Lua *number* types

The Lua v5.3, the *number* type now supports both 64 bit signed integers (in addition to ongoing support for double-precision 64 bit floats (related s10.2.8)). Accordingly in QCOM 3, 64 bit signed integers are used for meters.

Refer:

* Working with Lua v5.3 *number* data types and possible precision issues; section 15.2.
* Meter rollover requirements; section 15.3.

### QCOM User Scripts That Fail to Return

The Lua interpreter has a built-in, CPU efficient method that handles this scenario extremely well. For more information refer to the section titled User Lua Instruction Quota (s5.5) for more information.

The QLE Lua software driver efficiently manages any QCOM User scripts that fail to return on behalf of the machine.

### Standard Output (stdout)

In QCOM, *stdout* refers to the QCOM Lua Engine’s *stdout* (i.e. the *stdout* specifically fed out from the Lua API and library – not the hosting applications *stdout*) and no other *stdout* from any other application on the machine.

The ability to access the Lua engine *stdout* by the machine is required for development, diagnostics and operation. There are specific requirements where the host machine must capture and display QLE print output. See term *stdout* throughout this document to see all related requirements here.

In implementation, Lua *stdout* is not an easy thing to hook onto when the interpreter is embedded into a host program. Do not try to hook onto *stdout*, instead it is much easier to ignore the Lua interpreter *stdout* (this will occur by default) and just intercept (i.e. overload) all function calls to the relevant Lua function/s that output to *stdout* instead.

Accordingly the machine must overload the Lua print() function with a Lua C function which allows it to see all QLE prints (aka *stdout*) for the purposes of displaying certain QLE stdout on the host machine as required by this specification. cp:[[127]](#endnote-97)

Refer to the QCOM 3 SDK for an example host machine Lua print function (in “C”) needed when the host machine needs to process QLE prints.

The machine provided C Lua print function must have protection against being invoked with arguments resulting in long strings. There must be no possible way for a QCOM user to cause a stack overflow or other software exception via the print function. The machine provided Lua print function must limit by truncation, any single call’s output via the print function to **160** characters.

*In the longer term there may be additional requirements concerning stdout e.g. ability for the user to send arbitrary text to the web server for immediate publishing which can be downloaded or viewed in the user’s section on the web server.*

### Runtime Error Handling

This section concerns QCOM Lua Engine runtime errors.

It is possible that Lua errors can occur while the machine is compiling or executing any Lua script[[128]](#footnote-31). Errors in machine sourced Lua scripts should be extremely rare and would warrant a patch of the machine software to fix. Pretty much all errors during machine operation will occur in QCOM user’s scripts and most of these will occur during development.

To manage the worst case scenario of potentially many (runtime) errors, there is no QCOM Lua error log in the machine in the traditional sense (as it could grow too large). Alternatively, a QCOM Lua error table exists in the QLE Lua software driver (per QCOM user), where each error is indexed by the Lua returned error message itself. *The rationale here is that while there may be many errors over a period, there will only ever be in comparison, a very small number of possible locations in source code in which they occur.*

Accordingly, all QCOM Lua errors are stored by the QLE software driver Lua state in a dedicated **QCOM Lua Engine error table per QCOM user**. This error table is able to be parsed or downloaded from the machine on demand by any permitted QCOM user via the QCOM API function qcom\_luaErrors() (s11.15.2).

Each time a Lua error occurs, the QLE Lua software driver searches for it in the QCOM Lua Engine error table. If a match is not found, it adds the error as a new entry to the table with a “first occurrence” timestamp. If the error already exists in the table, the QLE Lua software driver simply increments an error counter and “last occurrence” timestamp fields for the given error entry in the table.

Each entry in the error table has the following schema:

|  |  |  |
| --- | --- | --- |
| **QCOM Lua Engine – Lua error table schema** | | |
| Each table (there is one Lua error table per user) is an associative indexed table where the table key is the actual Lua error message string (as returned by the Lua pcall() function). | | |
| **Value** | **Value Type** | **Description** |
| firsttime | table | First occurrence date and time (=os.date(fmtstr)) |
| lasttime | table | Last occurrence data and time (=os.date(fmtstr)) |
| count | integer | Number of occurrences of this error. |

where fmtstr = “%Y-%b-%d %H:%M:%S”

The QCOM Lua Engine error table is automatically **purged** by the QLE Lua software driver upon each restart of the machine and for a given user, upon user deletion and each time the user successfully loads a new archive of scripts into the machine.

For each Lua error generated the QLE software driver will throw the ***LUA\_ERROR*** QCOM state event (14).

After an error in a script is triggered (Lua will abort execution of that script), the QLE Lua software driver will still execute the same script again if triggered.

### Diagnostics

This section discusses the topic of diagnostics with respect to QCOM 3 related issues in machines and QCOM user scripts, and outlines the diagnostic features inherent to all QCOM 3 machines.

Given that QCOM 3 is fundamentally a script driven API, QCOM 3 machines inherently have a built in remote debugger which each QCOM user can be granted access either permanently, or on demand. Refer s24.2. Also, as these scripts can be remotely upgraded, it is easy to add diagnostic functions to scripts as required in production QCOM 3 machines.

Given encryption is now the standard operating mode for network protocols, it will become impossible for a 3rd party to record a network conversation in order to diagnose in-field application layer protocol issues. Issues with respect to encrypted network protocols can only be diagnosed by a party privy to the encrypted conversation. This would be a major issue for any traditional protocol specification framework where the two peers are separate organisations. Fortunately in QCOM 3, the remote peer always has debug level access to both sides of the network protocol implementation as well as the QCOM 3 API and all events used to feed the QLE state. As a result the ability to diagnose and debug issues in QCOM 3 machine network protocols far exceed any traditionally implemented network protocols to date.

Any machine issues discovered by QCOM 3 users can be confirmed by other QCOM users meaning that 3rd parties can help resolve any issue disputes. This is a huge advantage for QCOM 3 operating environments and something that would be missing in traditional standard protocol operating environments when the protocol is encrypted (which is now the norm).

With respect to the QCOM 3 network protocols, it is envisaged that there will also be fewer issues with protocols implemented by QCOM users in QCOM 3 machines because of the following:

* Typically in long term QCOM 3 machine operating environments, the same party implements both sides of the network protocol conversation. This always results in fewer issues than a protocol where each side is implemented by a different party.
* It takes less resources to test an API (aka QCOM 3) than a network protocol to the same level of confidence and functionality. Not to mention that the QCOM 3 API is implemented only once no matter how many machine brands use it.
* QCOM 3 test scripts are written in a platform independent high level scripting language which are easy to share and build upon by cooperating testers.

Issues in general are further reduced because:

* More than 60% of the QCOM 3 related code in a QCOM 3 machine is Lua code[[129]](#footnote-32). Lua code, being platform independent, means it can be imported directly from the QCOM 3 SDK. It implements the critical, externally exposed QCOM 3 interface, including all the sanity checks for the entire QCOM API, as well as a wide range of sanity checks on the data it receives from the host machine.
* Protocols implemented by QCOM users in the machines are implemented in Lua. Compared to protocols that are implemented by C or CPP, the Lua interpreter in a QCOM 3 machine provides two additional layers of protection. The jailed user environment and the Lua interpreter itself. Lua being a high level, memory managed language, is immune to a wide range of network protocol based attacks. (Arguably a good reason why the protocol analyser software Wireshark uses Lua.)

Related:

* QCOM Lua Engine Error Table

Refer section 10.2.11

* Lua Standard Output Logging

Refer section 10.2.10

### New Versions of Lua

Minor releases of Lua may be adopted once stability and confidence in the release has been achieved in the opinion of OLGR. (A minor release is a release in which existing QCOM Lua scripts do not need updating in order to operate.)

When a major new version of Lua is released it will not be adopted by the QCOM specification until:

* It has a demonstrated track record of stability.
* There are no unacceptable bugs or issues outstanding.
* All required associated libraries have also been updated to support or be compatible with the new version and have achieved a similar level of stability.

### Legal - Queensland

OLGR reserves the right in the State of Queensland to disable without notice any QCOM 3 scripts or user accounts running on Queensland gaming machines. QCOM 3 scripts running on gaming machines in the opinion of OLGR, must not:

* Commit illegal acts with respect to any Australian State or Federal Laws.
* Contain offensive or objectionable material.
* Be unrelated to machine gaming.
* Be anti-competitive.
* Contain unacceptable advertising content.
* Present undue risk concerning machine gaming reliability, integrity or security, or any privacy related issues.
* Degrade machine performance to an unacceptable level.
* Utilise licences, approvals and consents in relation to the script which infringe the intellectual property rights of any other parties.
* Interfere with or attempt to modify or expose the host environment or the environment of another user.

## Libraries

This section lists what libraries (both internal and external libraries to Lua) are to be made available to QCOM users as well as any special requirements or modifications.

Libraries or modules made available to QCOM users as a part of the standard QLE Lua software driver, have been specifically vetted by OLGR (or delegated authority) to ensure they are suitable for use in the QLE. (Note, the primary check is that the library member functions are protected from user to user). There is no risk to the host machine because the QLE Lua software driver doesn’t need any special or external libraries other than those provided directly by the EGM manufacturer e.g. communications).

The number of libraries needed for QCOM 3 is minimal, and unless many QCOM users need to use the same library, it’s more efficient (machine memory wise) for QCOM users to bring their own libraries as required.

QCOM users can bring their own libraries. This is perfectly safe because the libraries reside along with the QCOM user in their jail and respective quota limits. Any issue with a QCOM user imported library can only affect that QCOM user.

Constraints for QCOM user imported libraries are:

* They must be written in pure Lua.
* The Lua language “require” command is not available to QCOM users.
* They will be scrutinised for suitability and efficiency and approved as per all SUA controlled QCOM user scripts, or will otherwise be hard constrained by the given user’s assigned resource quotas.

### Basic & Standard Libraries

Lua has a number of basic and standard built-in libraries for common operations in math, string, table, input/output and operating system functions.

This section defines changes and restrictions concerning the implementation of these libraries by the QLE Lua software driver.

The following Lua libraries must be loaded by the machine into the QLE Lua state prior to handing control to the QLE Lua software driver. cp:[[130]](#endnote-98) On start-up the QLE Lua software driver will delete any unwanted functions from memory and ensures that QCOM users are setup with the intended set of library functions in each QCOM user’s jail:

* basic library
  + The QLE LSD will prevent QCOM users from accessing these functions:
    - *\_G*
    - *collectgarbage()*
    - *dofile()*
    - *getfenv()*
    - *getmetatable()*
    - *load()*
    - *loadfile()*
    - *loadstring()*
    - *module()*
    - *pcall()*
    - *print() -- Overloaded. Refer section 10.2.10.*
    - *rawequal()*
    - *rawget()*
    - *rawlen()*
    - *rawset()*
    - *require*
    - *setmetatable()*
    - *xpcall()*
* string library [string]
  + Note:
    - RE: *find(), gmatch(), gsub(), match()* string library functions.The host machine must utilise a modified Lua Interpreter that limits/aborts CPU intensive string search patterns. cp:[[131]](#endnote-99) Refer to the QCOM 3 SDK (lualib subdirectory) for the required changes.
  + The QLE Lua software driver supplements the string library with some simple functions to work with hexstring types and base64 encoded data. These functions are documented in the QCOM 3 Summary spreadsheet; Lua-API worksheet.
* table library [table]
* mathematical library [math]
  + The QLE LSD will prevent QCOM users from accessing these functions:
    - *randomseed(); this is called by the QLE LSD*
* Operating system library [os]
  + Only the following functions from this library will be made available to QCOM users by the QLE LSD:
    - time() The host machine must ensure that the Lua os.time function is returning POSIX compliant Epoch time[[132]](#footnote-33)[[133]](#footnote-34). cp:[[134]](#endnote-100)
    - clock()
    - date()[[135]](#footnote-35) Notes for QCOM users using this function:
      * QCOM users must not use the *isdst* field returned by this function if present.
      * QCOM users should not use this function to get a historical, or future **local** time (for a given epoch time argument) if the machine is in a locale that has daylight savings.
      * Calls and conversions from os.time() to a os.date() UTC value and vice versa must work correctly when the machine is using manually set time, or NTP controlled time. cp:[[136]](#endnote-101)
    - difftime()

Excluded functions are removed from the QCOM user’s Lua *environment* by the QLE Lua software driver upon QCOM user creation before passing control to any user script. For more information, refer Lua documentation regarding “Lua Sandboxing” [[137]](#footnote-36), section 10.1 and the required implementation in the QCOM 3 SDK.

The following Lua libraries are not made available to QCOM users by the QLE Lua software driver:

* Input / output library [**io**]
* Package library [**package**]
* Debug library [**debug**]
* Co-routine library [**coroutine**]
* UTF-8 support library [**utf8**] (Possible future inclusion)

### External Libraries

In addition to the Lua standard (built-in) libraries, The QLE Lua software driver will include the following external libraries and make these available to priviledged QCOM users.

**BigNum**

Pure Lua number library providing BigNum and fixed width unsigned integer types

<https://github.com/user-none/lua-nums>

Conversions only. Accessible via the QCOM API function **qcom.luaBigNum()**

## QLE Initialisation Script

Refer to the QLE Lua software driver in the QCOM 3 SDK for all required Lua source code here. The ‘main’ script here is the **qle.lua** script. For the full initialisation sequence see section 10.7.1.

## QCOM API

The QCOM API is the primary interface that QCOM 3 defines as seen by QCOM users (section 5). The QCOM API is summarised in the QCOM API summary spreadsheet. For a complete list of QCOM API function full descriptions, refer to section 11.

The QLE Lua software driver implements all applicable and mandatory functions in the QCOM API. Note that a number of QCOM API functions are optional and others have a ‘not before’ incept date. Refer to the QCOM API summary spreadsheet for more information.

### QCOM API Function Naming Convention

*This section is only design notes (i.e. no requirements in this section)*

The function name will be a *verb* if the function changes the machine state in any way.

The function name will be a *noun* if the function only returns a value.

Functions are in mixed case and the convention used is “CamelCase”; the only exception is when a word follows from another capital (such as an abbreviation), lowercase is used in this situation e.g.

“qcom\_secQMA**c**ert” is used instead of “qcom\_secQMA**C**ert”

A function which returns a “meter” value or similar (refer GMNS) will always have “Meter” as a part of its function name.

### Function Arguments

The requirements in this section apply only to QCOM 3 API function arguments. The QCOM 3 API function argument interface is implemented and sanity checked by the QLE Lua software driver. Refer to the QCOM 3 SDK for all possible QCOM 3 API error return values in relation to issues with function arguments.

Unless stated otherwise for a given QCOM API function definition, if a QCOM 3 API function argument fails any sanity check then the function will return without affecting the state of the machine.

Unless stated otherwise for a given QCOM API function definition, a QCOM API function will generally abort upon the first sanity check failure and return *nil, string* where the string identifies the offending key and a description of the issue.

Generic argument and table argument key-value sanity checks

All arguments (and table argument key-values) are checked by the QLE Lua software driver for the expected type as per the function’s definition.

Refer to the QCOM SDK for all applied sanity checks.

Number of Arguments

The QLE Lua software driver does not care if more arguments are passed to it via a QCOM API function than it expects. This is a standard feature of Lua and supports extensibility.

For QCOM API function calls which do not contain all defined arguments and where the function cannot simply retain the current value or use a default value, will be treated as an error and the function will return an error without affecting the state of the machine.

Refer to the QCOM SDK for all required sanity checks and associated error messages.

Tables as Arguments

Many QCOM API functions accept tables as arguments. Unless stated otherwise, arguments of type table are permitted to have unexpected/unknown entries which the QLE Lua software driver will ignore. This is for extensibility.

Table arguments which do not contain all defined keys and where the function cannot simply retain the current value (i.e., where none exists), or use a defined default value, the QLE Lua software driver will treat this as an error and abort without affecting the state of the machine.

### Conversions

Nothing to state.

### Return Values

*Design note: Lua: Any QCOM function that fails will return nil as the first return value. This makes the function friendly for use with the Lua assert() function.*

The Lua language supports functions with a variable number of return values. Possible return values and formats for the QCOM API are defined in the QCOM summary sheet and also in full in each function’s full description (refer section 11).

As QCOM is a multi-user environment, return values will not be shared objects (e.g. equivalent in the “C” language to pointer return value pointing to the same object). All return values in the QCOM API will be a dedicated or unique instance of the object across all QCOM user environments.

Return value/s for non-privileged QCOM API function calls

All QCOM API functions have a specific return value in the event the function is invoked by a QCOM user who hasn’t been granted privileges to the respective function. There is a dedicated section on this: Refer section 7.1.

When QCOM API functions return

All I/O functions in the QCOM API must be implemented by the machine as non-blocking.

### Cooldowns

QCOM API functions that also generate a message in the machine have cooldowns. This is to limit message spam.

If a function is called again before a cooldown period has expired then the function will have no effect and will return an error.

Refer to the QCOM 3 summary spreadsheet; ‘Lua-API’ worksheet for a list of functions with cooldowns.

## Scripts - Life Cycle

Categories of Lua scripts in a QCOM 3 machine:

* QLE Lua software driver scripts

**Source**: OLGR. Mandatory.

**Related**: Hardcoded in the machine by the machine manufacturer during QCOM 3 implementation. Ensures a consistent environment as seen by QCOM users across all brands of machines.

**Persistence**: Life of the machine.

* QMA scripts

**Source**: QMA. Machine cannot get out of RAM clear defaults until a QMA script is executed. QMA scripts have first and full access to the QCOM API and thus control all priviledges pertaining to QCOM 3 users in the machine.

**Related**: Instigated via QCI command: **qmaexecscript** (refer s24.2.3).

**Persistence**: The machine must not save scripts downloaded via qmaexecscript in NV memory; once the script has been executed by the QLE Lua software driver, it will delete the script from memory.

* QCOM user scripts

**Source**: QCOM users

**Related**:

Installed via:

QCI command: **userloadscripts** (s24.2.4), or

QCOM API function **qcom\_userLoadScripts()**

Execution: handlers hooked to state events (s14).

**Persistence:** If the download and hash verification of the script archive is successful, the machine must store the script archive file in NV memory, replacing the user’s existing script archive file with the newly downloaded script archive file. A user’s scripts are deleted by a machine at RAM clear, or if the QCOM user is deleted via the QCOM API function **qcom\_userDelete()**.

There are multiple approaches to QCOM user script management / maintenance in a machine. This is QMA controlled.

A QMA (introductory) script (aka **qmaexecscript**), can contain embedded QCOM user scripts (that the hosting QMA script will install via a call to **qcom.userLoadScripts()** ). Typically **qmaexecscript** scripts will only contain embedded scripts when the QMA wants to install a QCOM user’s script for a QCOM user that it creates and owns for its own purposes. The QMA can install any QCOM user’s script that way if it liked though. Typically however, the QMA will let the QCOM user install their own scripts, if script approval/authority control is needed, the QMA will assign a SAA to sign the script.

In summary of the above paragraph, the following two scenarios are functionally equivalent:

1. QMA script creates a QCOM user and installs a QCOM user’s script; all via an introductory script via **qmaexecscript**.
2. QMA script creates a QCOM user in the machine, assigns it an SAA via an introductory script via **qmaexecscript**. From here on, the SAA now signs acceptable QCOM user scripts for which the QCOM user downloads to the machine via userloadscripts.

The advantage of point 2 above is that the QMA can use a third party to evaluate and approve QCOM user scripts.

In conclusion, the reason for all this and other variability in QCOM 3 around QCOM users, is so that no matter what the machine’s host jurisdiction’s regulatory operating environment is, there exists an arrangement of QCOM 3 authorities and QCOM users to suit.

## Script Execution

*Design Requirements:*

* *QCOM users must be able to upload and execute scripts with respect to their assigned set of privileges.*
* *QCOM users must be able to load any number of Lua “chunks” with respect to their set memory/disk quota limits.*
* *There needs to be a start-up script or a configuration file which tells the machine what chunks exist for each user and the order in which to load them.*

Script execution is 100% event driven. Scripts cannot block, if they do the owning QCOM user will be quarantined by the QLE Lua software driver.

The QLE Lua software driver will never allow precompiled Lua scripts to be installed or executed from a source external to the machine. cp:[[138]](#endnote-102) Accordingly, the Lua dofile() function will never be used by the machine on scripts received from untrusted sources. The QLE Lua software driver will always use the Lua functions load() or loadfile() which will be set to automatically prohibit the loading of pre-compiled scripts via the *mode* argument. Refer to the QLE Lua software driver in the QCOM 3 SDK for more information.

Methods in which a QCOM user can install scripts in a machine:

* The QCI **userloadscripts** command (Refer 24.2)
* The QCOM API function **qcom\_userLoadScripts()**

Methods in which a QCOM user can execute scripts in a machine:

* By hooking scripts into the QCOM state events (refer section 14) which are then executed by the QCOM event dispatcher.
* All users must have a start-up script, which is executed by the machine in main application start-up and is used by the user for example to setup all the event call-backs (as the QCOM Lua Engine is event driven; refer next section).
* By hooking scripts into the QCOM Command Interpreter (s24) which may then be executed on demand via a remote QCOM UAA connection (s23).
* A QCOM machine in debug / development mode may make available a QCOM Lua Engine command prompt.
* Via signed QMA script execution by proxy via QCOM Command Interpreter (24).
* Arbitrary user script execution (signed and unsigned scripts) also via the QCOM Command Interpreter (24). Debug mode only; availability in production machines is not intended.

All QCOM user scripts executed by the QLE Lua software driver will be executed by the same dispatch method in order to ensure they are all monitored with respect to set QCOM user quotas.

### Starting-up the QLE in a Machine

*This section assumes that machine’s logic seal has been confirmed (s4.4). Prior logic seal confirmation, the machine must not start the QCOM Lua Engine (QLE).*

At every machine start-up, after the machine has successfully completed its integrity checks, but before the machine has started logged any machine events (e.g. power up, door state changes, fault conditions etc)[[139]](#footnote-37) cp:[[140]](#endnote-103) and moved from its boot/loading display to its main application display, the machine must start the QLE Lua software driver via a procedure which must be equivalent to the following ordered list: cp:[[141]](#endnote-104)

* Output the text “QCOM Lua Engine Starting.” cp:[[142]](#endnote-105)
* Create the QLE Lua state and supporting thread. cp:[[143]](#endnote-106) (Refer to the QCOM 3 SDK module: qlua.cpp : qlua\_new() function for example code and start up sequence.)
  1. Initialise any supporting buffers structs / classes and the QLE thread.
  2. Create the Lua state
  3. Bind all the required C Lua callbacks the Lua state. Refer to the QCOM 3 summary spreadsheet: **CLua** worksheet for specifications for all C Lua functions the machine must supply.
  4. Load & execute the main module ‘**qle.lua**’ of the QLe Lua software driver in Lua state e.g. equivalent to a Lua API call: *LuaL\_dofile(L, "qle.lua")*
* Initialise the QLE’s copy of the machine’s host machine state (hms) from EGM NV memory by logging a QLE\_DOSTRING state event where the string to execute is equivalent to the QCOM 3 SDK call: *qle\_dostring("hms="<s>" qleHMSreceived()")* where <s> is a string serialised Lua table declaration **denoting the host machine state at this time**. The machine can generate s by either printing to a string[[144]](#footnote-38), or by using one of the many available C/C++ struct/class to Lua conversion APIs[[145]](#footnote-39). cp:[[146]](#endnote-107) The QLE LSD uses the SDK module *hms\_schema.lua* to verify the received hms is the correct schema.
* If the above call to qleHMSreceived() is sucessful, the QLE Lua Software driver will log shortly after, the sendtoHost message “***qle\_ready***”.
  + If not successfull, the call to qle\_dostring will either return an immediate error, in other cases the QLE Lua Software driver will log the “panic” message. cp:[[147]](#endnote-108)
* In response to the “***qle\_ready***” *SendToHost* message, the machine perform the ‘required actions’ as described in the QCOM 3 summary spreadsheet for this message. cp:[[148]](#endnote-109)
* The machine may now start the UAA, MDP[[149]](#footnote-40) services and the optional www service.
* Aside: As a part of the **qleHMSreceived()** function, the QLE Lua software driver will for each QCOM user:

1. Output the following message:
   * 1. **“Starting <username>”**
2. Create the QCOM user’s jailed environment in the QCOM Lua Engine.
3. Restore any persistent variables for the user into the user’s jail. The QLE Lua software driver will try to retrieve this information from the machine by calling the machine supplied C Lua function **hqcom.usersLoadNVdata()**. cp:[[150]](#endnote-110)
4. Verify the integrity of the QCOM user’s script package (via the original package load SHA256 hash stored in machine NV memory)
   * 1. If ok, then extract all .lua script and .txt files[[151]](#footnote-41) from the package and continue.
     2. If not ok, output the error message **“Incorrect hash”** and skip to the next user (if any).
5. Then **for each extracted Lua script file**, starting with the user’s start-up script (see below for more information on this start-up script):
   * 1. Load and compile\* the .lua file for the QCOM user into the user’s jail. (\*e.g. loadfile(filename, “t”, \_userENV))
     2. If the above is successful then execute the script (Lua API pcall()) via the same dispatch method as used for any user scripts hooked to any machine state event. Any error will initially be dealt with as per section 10.2.11.
     3. If there were no errors, continue to the next script if any.
6. If there were any issues at all in the last step, then the QLE Lua software driver will:
   * 1. Print any Lua error messages as a result of the loading or execution. Refer s10.2.10.
     2. Shutdown the user (meaning its scripts will not be executed again until the next restart of the machine or the user).
7. Repeat for the next QCOM user if any.

* If all ok, then output the text “QCOM Lua Engine Ready” and continue booting. cp::[[152]](#endnote-111)

It is acceptable for the machine to hide all QLE print messages being directed to the machine’s boot/loading display if the machine alternatively provides a method to make them visible on demand via a human user physically present at the machine. A machine may prefer to display a less intimidating (aka non-technical / user friendly) aesthetic boot loading image instead and display a “more information”, or “i” button, or similar on the machine. If pressed, the machine then displays the more technical boot loading messages as required by QCOM and those inherent to the machine.

**FYI only:** How QSIM starts up (in the case you’re looking at the QSIM source code in the SDK):

**NB** because the QSIM start-up sequence uses **QLE\_DOSTRING (**instead ofthe Lua API function: luaL\_dostring() ), theQSIM’s startup sequence appears a bit more complicated (but it’s still equivalent to the above required startup sequence overall):

* Create Lua state
* Bind all the required callbacks from \*C-Lua\* sheet to that state
* Initialize \*hms\* (Initialise the QLE’s host machine state (hms) from EGM NV data by logging a QLE\_DOSTRING state event where the string to execute is equivalent to: qle\_dostring("hms="<s>" qleHMSreceived()") where <s> is a string serialised Lua table declaration denoting the host machine state, i.e. the state of the QCOM 3 machine at power up.)
* In QSIM logging the QLE\_DOSTRING state event wakes with the QLE thread for the first time. Before servicing QLE\_DOSTRING, on first time wake up the QLE thread :
  + Load & exec \*qle.lua\*
  + If ok then starts the state event service loop which only then services the QLE\_DOSTRING state event logged above.
* the QLE Lua software driver in response to a successful qleHMSreceived() will send the qle\_ready message. In response to the qle\_ready message the machine must log the **QLE\_READY** state event to QLE driver. (NB If anything goes wrong with qleHMSreceived() the QLE lua software driver will send a panic message instead)

### QCOM user start-up script & module

Each QCOM user in the machine must have a start-up script module which will be the first script executed by the QLE Lua software driver for the user.

This script module filename must be called: ***qinit.lua***

If the user’s app is comprised of multiple modules, the start-up script module dictates the user’s remaining script module load order via a specially named global table in the module. This table must be named ***qcomscriptloadorder.*** It must have the following simple schema:

qcomscriptloadorder = {

filename1, -- string, denoting a .lua ext moudule filename

filename2,

filename3,

…

-- The above filenames must exclude the “.lua” extension

-- (the QLE will append the extension)

-- filename must be a valid Lua identifier

-- 1-32 characters in length

-- Max modules = 20

-- The user’s account storage quota must be applied at all times

-- The QCOM SDK provides a function to parse this table

}

The *qcomscriptloadorder* table may be *nil* (missing) if the QCOM user does not have any other modules other than ‘qinit.lua’.

On user start-up, once ‘qinit.lua’ is loaded & executed, the QLE Lua software driver will load and execute the user’s remaining modules in the order listed in the *qcomscriptloadorder* table.

The *qcomscriptloadorder* table is a replacement to Lua’s built-in ***require*** function which isn’t available to QCOM user’s because QCOM users inside a jailed environment. The absence of a require function means a QCOM user software developer must think about the module load order, it is recommended the utility modules go first, followed by supporting modules, then the primary module/s last. Program globals are best defined in qinit.lua.

To help compensate for not having a ‘*require’* function, if a user module when load/executed at start-up, returns a *table*[[153]](#footnote-42), the QLE Lua software driver will save this table in the user’s environment in a global table called ‘**libs**’ with a string key equal to module’s filename (minus the extension). the QLE Lua software driver will create the libs table if it doesn’t exist, so user’s should be sure not to use a global variable with the same name.

All functionality related to start-up scripts and script module loading is implemented by the QLE Lua software driver.

Initial script module execution, will typically perform the user’s initialisation routines, setup any global variables and libraries (within the user’s script jail / environment), and also setup the QCOM users desired set of state event call-back / hook functions for subsequent execution over time as the machine operates. Refer section 14 for more information regarding Lua call-back functions.

Start-up script modules will be executed by the QLE Lua software driver as per any other QCOM user hooked state event script handler, in that the QCOM user’s set CPU, memory and instruction quotas will be applied.

### Executing Lua Scripts by Proxy

Executing Lua Scripts via proxy allows one user to run another user’s script with the second user’s privileges. (*Similar to the Linux ‘sudo’ command*.) The purpose of this feature is to enable QCOM user self-introductions (refer 5.2 for more information on QCOM user introductions).

***Concept only. Not implemented by the QLE Lua software driver at this time.***

***This feature is currently only available in relation to QMA scripts (i.e. a QCOM user may execute a signed QMA script at the QMA’s privilege level). This feature is possible via the QCOM Command Interpreter. Refer section 24.2.3 – qmaexecscript QCI command.***

***At this time, the ability for User A to execute User B’s scripts at User B’s privilege level is not to be supported, however this may be added later if need be. This section is therefore just an overview of the concepts pertaining to QMA scripts.***

The following prerequisites apply before one QCOM user can run a QCOM script at another QCOM user’s privilege level (or the QMA’s privilege level):

* The script must be signed by the user or QMA (if it’s a QMA script) whose privilege level is intended to be used.
* Depending on the user, the script may also need a digital signature from the SAA (refer section 6.3)
* The target privilege level user must also have an account on the machine unless it is a QMA proxy script.
* The executing user does not necessarily need an account on the machine as a QCI *anon* user can be utilised (if the script allows it – see section following below on constraining proxy scripts).
* The target privilege level user must have previously uploaded a copy of their certificate onto the machine via the QCOM API unless it a QMA proxy script.

Before signing a script for proxy use, a user should consider the following very carefully:

* Should it be signed at all? A user should never sign a script unless it is intended to be run by another user.
* Have the script expire as soon as practicable (see below).

Constraining the scope of a proxy script

Within a script intended for proxy use, the script itself should constrain itself and narrow its scope where necessary in order to limit potential misuse; for example:

* Limit the script to work only for a specific venue, state, country, or gaming venue ID/s.
* Include not-before and not-after time constraints. Scripts should always enforce an expiry date.
* Should the script be run only by certain user/s or the QMA?
* Should the script be allowed to be executed from within an *anon* user?
* Should the script only be permitted to run once?

The QLE Lua software driver will not execute a Lua script at another user’s privilege unless:

* The first line of the script must include the following exact string:

-- WARNING: The presence of line allows execution of this script by proxy on a QCOM machine. This is a security risk. DELETE this line if unsure or if this not intended.

* The second line of the script must include the line:

-- Target Privilege: <username>

Where <*username*> is either the text “QMA” (without the quotes) or the *username* text of the user whose level of privilege must be used during the execution of the script.

* The user account of the target privilege exists on the machine (unless it is a QMA signed script).
* The script’s digital signature successfully verifies against the public key in the target user’s certificate or in the QMA certificate if the target privilege is the QMA.

If all the above requirements are satisfied then the machine will execute the script at the privilege level of the certificate holder.

*Design note: If executing scripts by proxy finds an application that utilises it heavily, the machine may be required to remember that a script has already been authenticated for reuse.*

## Overloading QCOM API functions

In Lua, overloading a function in the current environment is as simple as:

qcom\_idInterfaceVersion = myFunc

However, as the Lua global environment in a QCOM machine is protected (refer section 10.1), then a QCOM API function will be required which allows only permitted QCOM users (typically the QMA) to overload any of the QCOM API functions[[154]](#footnote-43) in the QCOM Lua Engine global environment.

The proposed supporting QCOM API functions are:

qcom\_luaOverload() -- refer section 11.15.12

qcom\_luaOverloadClear()

*Note that these functions are not to be implemented until further notice (pending all risks listed below are sufficiently addressed).*

Overloading a function in the QCOM Lua Engine affects not only the global environment but each QCOM user’s own environment. The feature supports persistent overloads as well as chain calling the original function.

Once a function has been overloaded, then whenever it is subsequently called it will call the overloaded function rather than the original function.

Risk Assessment

It will not be possible to overload any QLE Lua library functions made available to QCOM users, only QCOM API functions can be overloaded.

This feature is generally not a risk to machine integrity provided the overload functions remain privileged only to the QMA. However the overload feature does pose a serious privacy and integrity risk to QCOM users with respect to any data they may wish to keep protected from even the QMA. This issue will be addressed before the overload functions may be implemented.

Options considered to date:

* Limit the overload feature to machine debug mode (i.e. make the overload feature unavailable in production machines)
* Provide an overload inhibit QCOM API function intended to be generally privileged out to all QCOM users so they can inhibit overloads on all or specific QCOM API functions as they see fit.
* Constrain each function that overloads a QCOM API function into a bare bones jail (Lua environment) which has no access to the outside world or QCOM API or privileges in general.

Potential Benefits of Overloading

This feature allows for the following benefits (in addition to all the classical benefits of function overloading in a programming language) and applications:

Bug fixing and patching

As the QCOM Lua Engine, QCOM API and provided libraries will typically be hard-coded into the machine, overloading makes it possible to patch a range of possible issues in the QCOM Lua Engine and environment even though those functions are actually hard-coded into the machines. Missing sanity checks can be added or a function with a known vulnerability can possibly be fixed post production.

It may also possible to address issues that may occur should both the machine and QCOM user service running on the machine no longer be supported by the vendors.

Debugging

This allows for calls to any function in the QCOM Lua Engine global environment to be intercepted for debugging purposes; even in production machines.

Extensibility

The feature allows for the extension, improvement or altering of the behaviour of the existing hard-coded QCOM Lua Engine API and global environment.

This feature can allow opportunities for the behaviour of existing user scripts to be changed (within the scope of its interface to the QCOM Lua Engine, API and operating environment) for various reasons.

Overloading allows existing QCOM API functions to have additional controls or sanity checks implemented post production. It can satisfy a need to be able to give a specific QCOM user partial, crippled, changed, or controlled access to any QCOM API function.

Overloading can turn an existing QCOM API function from a single user only function into a function that support multi-users by adding appropriate contention control.

Monitoring

There may be a need to see how many times a function is being called (and with what parameters) for performance monitoring, logging and debugging purposes.

In this case the QCOM API function may be overloaded but its original behaviour is not changed. The overload will just enable the desired logging or monitoring.

For example say there becomes a requirement to log how many times a parameter such as *maxbet* is changed by authorised users. Currently there is no inherent function for this, however via an overload, the desired level of monitoring of the max bet function can be inserted without affecting any existing scripts utilising that function.

With overloading it is possible for new events to be created off the invocation of functions.

## Script Development Considerations

This section primarily concerns QCOM users.

QCOM users should keep in mind the following things when developing Lua scripts for QCOM machines.

QSIM 3

Recommended for QCOM users. QSIM 3 is a QCOM 3 machine simulator and QCOM 3 script IDE and debugger. Alternatively, QCOM users can also develop scripts using real QCOM 3 machines. The same debugging features are in both QSIM 3 and actual QCOM 3 machines. However, QSIM 3 is often more convenient for example, as machine behaviour can be scripted, and can also do things like script performance benchmarking.

Power volatility

Machines may lose power at any time. Lua scripts must implement their own state protection/recovery as necessary. A user can utilise the QCOM API relating to persistent variables (refer 10.10) to help achieve persistence across machine power disruptions and restarts.

Machine State Progression

Refer to the requirements in section 10.1. The ramifications to QCOM users regarding the requirements in the section, mean that the example script lines below if executed in a QCOM 3 machine, would cause the script to never return and the owning QCOM user would invariably be quarantined (s5.8) as a result of the User Lua Instruction Quota (s5.5).

while (egmState() == “idle”) do … end – wait to leave idle

while (qcom\_egmCreditMeter() == 0) do … end – wait for non-zero CM

QCOM users cannot loop within a script while waiting for a machine state change because the state change will never happen. **From the perspective of a QCOM user’s script in a machine, the machine’s state will only change between the dispatching of successive state events** (s14).

Related: QCOM API function qcom\_luaHookScript() (s11.15.4).

State events

Scripts should consider that new data fields may be added to state events over the life of a script. Scripts should not see this an exception by default, but just ignore the new fields.

Events

New data fields may be added to events over time. Scripts and associated systems should not see this an exception, but if the event is stored then store the new event data as well.

## Persistent Variables

*Design Requirements:*

* *There must be methods in which a QCOM user can have the machine store a limited amount of user data persistently (i.e. data is preserved across machine restarts and power failures)*
* *There is predicted demand for QCOM users needing the ability to make their persistent variables private or public (read-only). Users may need to store passwords or private keys for access to an external server or database or they may wish to share information with other users.*
* *Persistent variables made private must not be accessible by anyone including the QMA.*
* *Make the default PV private and if made public then it must be read-only (with respect to other users). Make users invoke specific API commands in order to share the data with other QCOM users.*
* *The design or hosting environment will take into consideration as necessary the fact that some implementations of NV memory in machines have well below typical memory r/w access times and is far more resource intensive. This is more of a design requirement relating to how the QCOM Lua Engine will be hosted by a machine which will take this into account.*
* *General event buffers / logs must be able to be saved as QCOM persistent variables, in the event a user wants to store an event buffer in PV space.*
* *Support for* ***nested tables****.*
* *The solution must be able to implement lists, stacks and queues and must also be suitable for the storage of a small number of certificates (being approximately 4-6kB each).*

QCOM Persistent Variables provides permitted QCOM users with a limited amount of non-volatile machine memory for the persistent storage of data. The machine must preserve QCOM user-created Persistent Variables across machine power fails and reboots. cp:[[155]](#endnote-112) The QLE Lua software driver implements PV API and will let the machine know of any Persistent Variable data updates on-the-fly.

QCOM Persistent Variables are suitable for a wide range of data such as events, meters, encryption keys, certificates, passwords and QCOM user state recovery information.

Implementation of QCOM Persistent Variables in QCOM machines is mandatory. cp:[[156]](#endnote-113)

The machine must make available a specified amount of its NV memory[[157]](#footnote-44) for the purpose of storing QCOM Persistent Variables. The total amount of NV memory a machine must make available for QCOM API persistent variables is **50kB[[158]](#footnote-45)**. cp:[[159]](#endnote-114) The machine may optionally provide a larger amount if desired. The QLE Lua software driver will ensure the limit is not able to be exceeded.

QCOM Persistent Variables are implemented via the QCOM PV Lua API. Refer section 11.17.

In a successful call to one of the pvCommit QCOM API functions, the QLE Lua software driver will message the machine and the machine must then make that data persistent, i.e. store in NV memory. The machine receives and restores all PV data in the form of **binary strings**, with the QLE LSD performing all conversions wrt the PV QCOM API functions.

QCOM users note: All updates to the original table/string in the user’s environment are only saved to NV memory each time the QCOM API PV commit function is invoked which is basically a replace operation overall. Each PV data item can only be updated in full, so individual persistent variable data items should be either kept small or updated infrequently.

*Design note:*

*An alternative design solution considered which would eliminate the need for the PV feature completely, was for the machine to maintain the entire QCOM Lua Engine’s state in NV* memory*. A feasibility study would be required to progress this concept. Related:* [*http://lua-users.org/wiki/PlutoLibrary*](http://lua-users.org/wiki/PlutoLibrary)[*https://github.com/fnuecke/eris*](https://github.com/fnuecke/eris)

### Privacy

All PV names and data must be kept **secret** by the machine other than to the owning user. There must be no facility in production machines to extract or examine PV NV memory contents unless you are the owning QCOM user. cp:[[160]](#endnote-115) Related: Section 10.2.6 on how QCOM users can share data if desired.

### Updates (Commits) of PVs

The storage methodology for persistent variables in NV memory does not have to be maintained with any type of error detection or correction methodology. However at a minimum, the machine must guarantee an all-or-nothing approach in its implementation of PV commit (update) operations. cp:[[161]](#endnote-116) Specifically:

If a machine is restarted (e.g. power outage) part-way through a commit operation, the machine upon next restart must either revert to the previous data state prior to the call to the QCOM API commit function, or resume the update to ensure full completion of the commit. cp:[[162]](#endnote-117)

A partially updated or corrupted PV must never result from a machine restart occurring at any time. cp:[[163]](#endnote-118)

After a call to the PV commit function and the machine returns control to the calling script, the update to the PV must be stored persistent in NV memory of the machine by this time. Any machine restart or interruption from this point onwards must never result in the PV from becoming corrupt, lost, or reverting to an old value. cp:[[164]](#endnote-119)

For more information and requirements refer to the example implementation of Persistent Variables in the QCOM 3 SDK.

### PV API Definitions

Refer Lua API definitions prefixed with ‘qcom\_pv’ (section 11.17) for detailed information

### PV Application examples: Storing an Event buffer

Example 1: Refer to the ‘qcom16’ user that implements the QCOM v1.6.x protocol that comes with QSIM 3. Here QCOM 3 PV’s are used to store the QCOM v1 event buffer.

Example 2:

The example described below illustrates how a QCOM user may store a FIFO-like buffer of events as QCOM Persistent Variables. Events with a schema similar to 13.5 are envisaged but with the nested event ‘data’ table serialised into a string. This example is applicable to the approach for storing any type of arbitrary buffer / queue / stack etc. of any arbitrary data table in QCOM Persistent Variables.

How-to:

* Store each individual event as an individual PV utilising some form of alpha-numeric or numeric index\* embedded in the name in the PV namespace for the buffer.
* Maintain a header table as a PV, which denotes all the information pertaining to buffer management e.g. head index, tail index, size etc.

To add an event to the buffer, the script should commit the event as a new PV first and then update the PV header table second. This approach ensures better integrity of the buffer in the event of an interruption such as a machine restart.

To delete an event, the script should update the header table first and then delete the actual event from the PV memory second.

*\*Both alpha only or alpha-numeric indexes could be used. Lua’s ability for table indexes to be either numeric or associative (alpha-numeric) provides some options for the approach to indexing.*

*A numeric only index enables the header table index management to use simple arithmetic functions and create indexes without any additional overhead. To have multiple buffers (queues etc.) the user could segregate each buffer into a numeric range. The event buffer uses indexes in the range 1000-1100 and a separate message stack in PV uses indexes in the range 2000-2200.*

*Alternatively, alpha numeric PV string indexes could be used (i.e. Lua associative table indexes). With this approach, there is extra overhead in making useable indexes. An event buffer uses indexes in the form “eb\_xxx” (where xxx is the event index number) and a separate message stack uses indexes in the form “ms\_xxx”.*

## User Created Meters

**V3.0.1 Downgraded to concept only – do not implement. This feature may no longer be required in lieu of Persistent Variable support (s10.10) which provides similar functionality.**

Another type of special persistent variable exists for use by authorised users in limited quantities. This feature allows a user to create and maintain a ‘meter’ to the same standards as defined in GMNS, i.e. as “critical memory”.

The meters provided to users by the requirements in this section must be maintained by the machine to the same requirements as “meters” in GMNS. cp:[[165]](#endnote-120)

A QCOM machine must make available a small amount of its GMNS compliant critical NV memory to QCOM users for the purpose of storing QCOM user created meters persistently across machine application restarts and power downs. cp:[[166]](#endnote-121)

The minimum amount of memory a machine must make available for QCOM user-created meters must be sufficient to store **16 meters**. cp:[[167]](#endnote-122)

The ‘meter’ must be a Lua integer *number* type. cp:[[168]](#endnote-123)

Proposed Lua API definitions are listed in section 11.18. Functions in this category are prefixed with *‘qcom\_userMeter’*

Meters as required by GMNS v10.3 are resource heavy in that they are stored in multiple locations within the machine and are CPU intensive to update. They are therefore limited in number and a user should only be given rights to create a limited number of meters when necessary. A Lua script should always in the first instance, utilise persistent variables (see above) unless risk or a requirement specifically mandates the use of a “meter”.

Meters maintained by the above API must be readable by all other Users on the machine with rights to the QCOM API meter read function. cp:[[169]](#endnote-124) Support for private user meters may be added in subsequent versions of QCOM if there is demand.

It is intended that these meters be used for mission critical purposes for meters and data requiring the highest level of integrity in the machine. To use them for example, for meters related to machine performance (e.g. individual game meter in an EGM) is considered overkill and QCOM Persistent Variable support should be considered instead.

## Applications and Examples

This section details a range of example applications using the QCOM 3 scripting engine. Also refer to section 34 regarding possible future functionality. There are no requirements in this section.

### General

QCOM 3 represents a generic solution for a highly secure, reliable and versatile remote interface specification for any computer or embedded system. To make a new specification, basically remove any non-applicable or unwanted function classes from the QCOM API and add new classes and functions as needed.

Related: Advanced Linked Progressive Prize Support section 26 - Virtual Machine (VM) solution.

### Random Number Generator

Refer to the QCOM 3 SDK which contains an example crypto based RNG for which a QCOM user could use in the following example.

### Externally Triggered Jackpot Systems

One application of a script enabled interface specification like QCOM 3, which is anticipated will prove popular, is in the implementation of what OLGR terms ‘externally triggered jackpot systems’[[170]](#footnote-46). Often the QCOM API will be used just to trigger the jackpot/prize, however whole standalone jackpot systems and arbitrary prize systems could be potentially hosted by a QCOM 3 EGM.

External jackpots are historically triggered by third party (to the EGM) jackpot triggering device hardware, however the QCOM Lua Engine allows a jackpot service provider to trigger jackpots within the actual EGM by downloading a script to the EGM’s QCOM Lua Engine to perform the task. This means that external jackpots can then be operated much like EGM triggered linked jackpots and achieve the integrity and reliability benefits that are inherent to EGM triggered jackpots. Some of these benefits include:

* Eliminating all latency / network issues concerning the triggering and awarding of jackpot wins. Traditional externally triggered jackpot systems can experience network latency problems which are difficult to overcome (especially WAN jackpots that are centrally triggered), as well as serious potential issues caused by network outages at critical moments. Issues in this area can lead to “walkaways” (where a player leaves the EGM before notification of a jackpot win on that machine has been received) resulting in lost or belated contributions to prize triggering algorithms and an increase in the chance of simultaneous wins.
* Jackpots being triggered by a QCOM script will reliably lockup the EGM immediately after a play and nothing (network latencies or dropouts) can prevent this from occurring.
* Reduces the amount of network bandwidth required to implement a jackpot system over a LAN or WAN.
* Making the implementation of linked jackpot systems much easier as there are less critical sections to have to handle e.g. network dropouts at critical times and practically eliminates the possibility of lost or duplicate contribution to a player’s chance of winning.
* Since external jackpots and its meters also potentially affect gaming taxation (depending on the jurisdiction), external jackpots can be optionally implemented by a trusted or SAA controlled user whose scripts are authenticated by another trusted third party (e.g. the QMA in the case of using the QCOM SAA).
* Eliminating the need to install additional hardware into each machine. External jackpot systems typically require the installation of additional hardware at the gaming venue, often into each machine. Not having to do this represents a significant cost saving.
* Some jackpot triggering algorithms / systems are potentially CPU intensive, especially when implemented in a group totaliser type device. The ability to implement a part of the system in each machine via the QCOM API is effectively distributed processing, spreading the load across many machines.

Example implementation:

The QCOM user would hook a script onto the PLAY\_COMMENCED state event.

The hooked script when executed as a result of a player commencing a play on the machine would use the bet amount from the event to trigger a jackpot or promotional prize. Following is a simple example of a jackpot triggering script for a fixed $10 prize that would be hooked as described:

function myJackpotTriggerCallBackFunction(args)

p = 1e-4 -- probability per cent bet of triggering the jackpot

-- example only – typ this value is set by a host

weightedP = p \* args.betamt -- multiply p by the play's bet

R = myrng.gen\_real() -- gen\_real returns [0...1.0)

if (R < weightedP) then -- hit?

-- Hit!!!

-- queue a System Lockup to notify the player

slt = {title = "Jackpot", message = "you have won $10"}

qcom\_slRequest(slt)

-- lets transfer it to the credit meter as well

-- Any transId will do for this e.g.

qcom\_ectAddCredit(10, "ExtJP", tostring(R))

end

end

Because QCOM 3 is a programmable interface specification, any possible conceivable jackpot triggering algorithm can be implemented via the QCOM API. The above example’s triggering algorithm is based on QCOM v1’s CRanE feature except that it is using a RNG providing numbers the real interval [0,1) instead of a 32 bit integer RNG as per QCOM v1’s CRanE feature. The overall implementation is the same regardless.

QCOM 3 allows for most centrally triggered jackpot systems to be converted to the equivalent of a machine triggered jackpot system and achieve all the benefits stated previously in this section.

### Offline Operation

This term has been mentioned previously when listing the benefits of a script enabled interface specification like QCOM 3 and warrants further explanation.

In QCOM, a machine intended for offline operation can be setup with scripts out of the factory or at commissioning that can implement an unlimited range of functionality and policies applicable to the target jurisdiction autonomously.

When operating a QCOM machine offline from a network / monitoring and control system, some minimal / base functionality must still be uploaded to the machine and implemented via an autonomous user (6.1.3) before the machine can successfully operate offline. This is because there is a range of parameters in QCOM that revert to a default value on machine restart as well as a need to implement any local jurisdictional requirements.

Minimal needed functionality required for offline operation would be:

* As a QCOM machine at factory defaults cannot payout credits (refer section 22) all QCOM machines intended for offline operation need an autonomous QCOM user that implements the desired functionality pertaining to credit redemption for the given venue/jurisdiction.
* If any power save functionality is required, this needs to be added via an autonomous QCOM user.
* Enabling the desired games and variations. (Needed once per RAM clear)
* Enabling play via the Play Enabled Flag. (Every restart of the machine)

Refer to default value columns in the QCOM summary spreadsheet – “Global Types” sheet for other possibilities.

Other possibilities:

* As mentioned previously one example could be the implementation of licensed gaming hours in EGMs. An EGM could feasibly be setup up with many years’ worth of licensed gaming hours (including special events) and could automatically enable and disable play accordingly. This would occur without ever being connected to a network.
* Promotions and external jackpots.

There are really too many possibilities to list them all, suffice to say all functionality is possible in an offline QCOM machine that do not inherently require the presence of a network server.

### Player Information Display (PID)

*Design Requirements:*

* *Equivalent functionality as per QCOM v1*
* *Reserve IP concerning types of dynamically downloaded PIDs*

QCOM has the following support options for PIDs:

Built-in PIDs

Some EGMs have PID templates built-in. There is a QCOM API PID selection function which allows an authorised user to select from a set of hardcoded PIDs built into the machine. Refer to the QCOM API summary spreadsheet for a list of qcom\_pid supporting class of functions.

PID Access Metering

The only other functionality concerning PIDs that QCOM v1 contains is a PID accessed meter. If a PID meter to the same functionality is desired in a QCOM 3 machine, implementation would be performed by an authorised user using QCOM user Meter support (10.11) with a script hooked into the PID state events (refer list of hook-able events table). This would trigger an event handler that would increment the meter equivalent to QCOM v1 PID accessed meter.

Custom PIDs

QCOM 3 permits an authorised QCOM user to add new menu items in the “i” button menu. Refer section 3.7. When accessed, the QCOM user then has full control over an interface information display on the machine. This permits the creation of fully customised PIDs. A QCOM user would use QCOM API to query game specific information and then dynamically generate a PID. This will allow for an infinite number of possible PIDs and since QCOM scripts are all remotely upgradeable, ensuring that changes can be made easily and cost effectively. This solution would save machine manufacturers the inconvenience of implementing a range of possible PID types for a corresponding range of jurisdictions.

A possible variation here is where the QCOM user downloads pre-made PIDs on demand from a remote database. (The database would primarily be indexed via machine manufacturer and game/variation UID.) This solves the issues concerning dynamically generating PIDs that are unable to be generated from information the QCOM API can ascertain.

Custom PID support also has an advantage over built-in PID support in that it can support for example, a jurisdiction that requires a PID that must change dynamically; e.g. a PID that depends on current denomination (e.g. Queensland), or what external jackpots the machine is participating in (e.g. Victoria) in addition to any other unforeseen requirements.

No PIDs support

PIDs may be completely disabled (or not even implemented) in a QCOM 3 machine.

### Player Accessible GUI / Information Displays

**Concept only.**

The ability to personalise a player’s experience on a gaming machine is significantly increased in QCOM 3. Authorised QCOM users will now be able to hook into (via extensions to the QCOM API) and extend the functionality of the user interface of the gaming machine via the existing “i” button functionality built into AU gaming machines (refer 3.7). This can be used to create arbitrary custom interactive GUI / information displays on the machine’s built-in display.

Example applications:

* Player Loyalty system support.

*For example, a player could query the status of their loyalty points and other related information*

* make bets on another gaming or wagering system[[171]](#footnote-47)
* Customised PIDs
* Any form of arbitrary information display.
* Also see examples in section 19

The above are only some examples of a virtually unlimited number of possible applications that could be offered. Any application requiring a user interface could now potentially be delivered via the machine. The main intention and advantage of this functionality is that it should alleviate the need for additional third party display devices to be installed in the machine. This represents a significant cost saving.

To facilitate the implementation of this feature, two solutions are being developed:

One option previously mentioned, is to extend the QCOM API with a GUI API. Refer QCOM API qcom\_info class of functions (section 11.28) as an example simple API in this regard.

Pros / cons:

*+ Low risk (provided the API is kept simple)*

*- The UI will only be as aesthetically pleasing as the supporting QCOM API allows e.g. it may not support image display and drawing (i.e. a text UI only)*

The second preferred option would be upon access to launch an embedded (bare bones) web browser, which when activated, can be used by a player to access desired functionality via a remote web server of the service provider.

Pros / cons:

*+ Inherently graphics rich*

*- Potentially higher risk due to the remote access and how hardened the browser is.*

*The initial/final solutions adopted will be determined after industry feedback is received on this particular application. The first option may be the safer approach until QCOM 3 machines are fully remotely upgradeable devices in order to permit patching (e.g. the embedded web browser API as necessary).*

Accessing

In the case of a gaming machine, to initially access the custom UI functionality, an authorised QCOM user now has the ability via the QCOM API to add new custom menu item options off the machine’s existing “i” button activated menu (refer 3.7). Once hooked in, when a custom “i” button menu option is activated by the player / attendant, the machine launches a fully customisable interactive UI display via one of the above methods. More than one QCOM user can append new menu options to the “i” button menu.

User input will be possible and the machine will provide the “close” button to allow the user to exit any custom UI / display at any time.

### Protocol Converter

From section 1.7 on benefits:

As a QCOM 3 machine is multilingual from the a network protocol perspective, as well as being dynamically programmable protocol wise, a QCOM 3 machine should be able to implement legacy protocols in order for the machine to integrate with pre-existing systems.

For new systems however, generally speaking, if you own both sides of the conversation (which you typically do in QCOM 3); it is considered to be more advantageous to define a new protocol, or use one you already own unless for example, there is an free / open source like protocol already available that predominantly meets your needs.

### Health Monitoring and Sanity Checks

Because QCOM scripts operate from within a virtual environment on the machine, this makes performing machine health monitoring and sanity checks (e.g. meter movements, auditing, balancing) possible from within a user script and still retain the same level of integrity as if these types of checks were being done from an external device.

### Credit Meter Limiting

This concerns functionality which disables physical credit input on the machine after the credit meter has reached or exceeded some arbitrary threshold. (In QCOM v1 this was the CRLIMIT parameter.)

In QCOM 3, arbitrary rules can be implemented on demand. To implement a QCOM v1 CRLIMIT for example, a QCOM user designated the task of implementing a credit meter limit would hook into the following events:

**BANKNOTE\_ACCEPTED**

**COIN\_TOKEN\_IN**

Upon receiving control as a result of hooking into the above events, the user can implement whatever credit meter limiting rules they like (or are authorised for) and would use the following QCOM API functions to enable/disable physical credit input on the machine:

qcom\_egmCreditInputDisable()

qcom\_egmCreditInputEnable()

Regarding ECT, each transaction is limited by MAXECT (as per QCOM v1, except in QCOM 3, there is a MAXECT per QCOM user). There is also a global ECT enable flag controllable via the QCOM API.

### Large Win Lockups

In QCOM 3, the enhanced System Lockup feature (s16) facilitates large win lockups. This eliminates three legacy parameters from QCOM v1: namely LWIN, NPWINP and SAPWINP.

The generic nature of the new methodology means:

* Large wins can be verify-only, partially paid out or fully paid out and controlled as desired by the authorised user.
* Different behaviour can be implementing by win type (e.g. non-progressive win, progressive win, linked or standalone)
* Any possible jurisdictional requirements pertaining to large win handling can be catered for.

A QCOM user implementing a large win lockup would likely hook into the TAKE\_WIN or PLAY\_COMPLETE state event in order to monitor for the need to throw a system lockup and use it to implement a large win lockup. Upon queuing up a system lockup, the QCOM user can decide whether the large win lockup is to be: verify-only, partially paid out, or fully paid out. The user should use suitable messages in the system lockup to indicate the intention here.

Related: Credit Redemption. Section 22.

### Residual Credit Removal Feature (RCRF)

Refer to other former OLGR publications (e.g. QCOM v1, the former OLGR Queensland appendix to GMNS & GMNS) for more information about a RCRF.

This feature (due to its limited use across all known jurisdictions and its application to machines with coin hoppers only), makes it a candidate for implementation within the QCOM Lua Engine in order to take the burden off machine manufacturers from implementing a feature that has only been adopted by only a limited number of jurisdictions.

Since the RCRF is a gamble-like feature, its meters will also potentially affect gaming tax calculations. Therefore it is recommended RCRF (or any QCOM user delivered service involving a gamble or prize trigger) be implemented as a standalone QCOM user dedicated to the role and where the scripts are also authenticated by a trusted authority or SAA. (Similar to external jackpots 10.12.3)

The QCOM user/party implementing the RCRF would primarily utilise the QCOM **System Lockup** feature (s16) to obtain the required input from the player. Implementing an RCRF also requires the creation of new meters to store total RCRF stroke, turnover and wins as critical meters. (Refer section 10.11 - User Created Meters)

To implement: Firstly the RCRF QCOM user would hook into the **COLLECT\_WITH\_CREDIT** and **HOPPER\_COLLECT\_EXIT** state events in order to determine if there was any residual credit upon any collect operation.

If there is no residual credit then no further action is taken.

If there is residual credit then the user will queue up a system lockup (s16) and use it to ask the user if they wish to gamble the credit for a chance to win a whole amount that can be paid by the hopper, or exit. (Refer to other formerly published OLGR and GMNS requirements documents for more details on how a RCRF functions).

If the user elects to gamble (re **SYSTEM\_LOCKUP\_RESPONSE** state event) the QCOM user would perform the gamble and thus determine the outcome.

If the gamble is lost, the QCOM user makes use of the **qcom\_ectSubtractCredit()** QCOM API function in order to remove the credit from the credit meter and exit the system lockup.

If the gamble is won, the user would use the **qcom\_ectAddCredit()** QCOM API function to round up the credit to the next whole value (coin/token) payable by the hopper, exit the system lockup and instigate a collect (via **qcom\_rcCollectPress()**) which should now result in a complete payout of the credit meter via the hopper, or payout directly (privilege permitting) via the **qcom\_hopperPayout()** QCOM API function.

Related: Credit Redemption – Gaming Machines - section 22.

*Note in QCOM 3, RCRF, the created turnover and win meters won’t be added to the machine’s built in total turnover and wins meters as per QCOM v1.*

# QCOM Lua API – Function Full Descriptions

*Note: almost all requirements in this section (by far the largest section of this document) are implemented as a part of the QLE Lua software driver. Related: section 33.1.*

This section contains full descriptions of selected\* QCOM API functions. If a function does not have a full description within this section, then refer to the QCOM API summary table / spreadsheet for the function’s description at this time.

Not all functions are mandatory at this time. Refer to the QCOM API summary table / spreadsheet for incept dates.

QCOM API functions are written in this document as **qcom\_functionName()**. This is for editing convenience only. In a real QCOM user environment in a machine, QCOM API functions are all contained in a Lua *table* called “*qcom”*. Thus individual QCOM API functions are actually invoked in the following way:

**qcom.*functionName(args)***

e.g.: qcom.idInterfaceVersion()

All QCOM API functions must be implemented as non-blocking and return immediately.

For more information regarding the QCOM API refer to section 10.5.

Related: generic sanity checks on arguments. Refer section 10.5.2.

For quick reference refer to the QCOM 3 API Summary spreadsheet.

*\* Functions which are too simple are excluded.*

## \_ID

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_idInterfaceVersion

### qcom\_idDeviceType

### qcom\_idMachinePlatform

### qcom\_idMachineID

### qcom\_idSetMachineID

### qcom\_idLogicUID

This API function returns the machine's UID.

This UID value, produced be the machine, must be directly related to the machine's primary logic board. (In a gaming machine it must be the logic board which determines game outcomes.) The return value must be a unqiue constant, indelibly hard-coded and extruded from inside the sealed logic area of the device, preferably derived from a non-socketed component on the machine's logic board. For example, it may be the machine's IEEE 802 network physical MAC address (if this satisfies the requirements above e.g. the network interface IC hosting the MAC address is soldered on the logic board). Alternatively, it could be a CPU ID (if supported and the same caveats above apply). This UID must never change for the life of a logic unit.

cp:[[172]](#endnote-125) (check entire above para).

**Call format:**

qcom\_idLogicUID()

**Return Value:**

*string*

Refer to the QCOM 3 summary spreadsheet for more information. The machine manufacturer must ensure the return value is unique across all their machines. cp:[[173]](#endnote-126)

### qcom\_idCommissionUID

This function returns the machine’s commissioning UID number that represents a specific commissioning instance / state of the machine.

After a machine RAM clear, upon the machine **Logic Area Seal Confirmation** function being successfully completed (refer section 4.4), the machine must choose a **160 bit** random number for the QLE Lua software driver to return via this function which will be used to uniquely identify the commissioning of the machine. cp:[[174]](#endnote-127)

**Call format:**

qcom\_ idCommissionUID()

**Return Value:**  *nil | hexstring*

The function will return *nil* if it hasn’t yet selected a random commissioning UID. Otherwise the return value must be the *hexstring* representation of the random number chosen.

The base return type for a set commissioning UID is *string;* specifically, a **40**-character *hexstring* denoting the machine’s commissioning UID.

### qcom\_idMfr

### qcom\_idMfr3

### qcom\_idOSversion

This function must return the operating system name and version of the application hosting the QCOM Lua Engine and QCOM API. The return format is open except that Microsoft Windows based machines must exclusively include the text "**Microsoft Windows**" and Linux based machines must exclusively include the text "**Linux**" in the return value. This information is also obtainable via the content auditing QCOM API function: **qcom\_cAuditCommonResults()**.

Microsoft Windows based machines should return a value equivalent to the output of command prompt command: **ver**

Linux based machines should return a value equivalent to the output of command prompt command: **uname -or**

cp:[[175]](#endnote-128) (check entire above para).

**Call format:**

qcom\_idOSversion()

**Return Value:**  *string*

## \_Network Management

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_networkIP

### qcom\_networkEthAddr

## \_Location Management

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_locSetCountryCode

### qcom\_locCountryCode

### qcom\_locSetCurrencyCode

### qcom\_locCurrencyCode

### qcom\_locSetStateProv

### qcom\_locStateProv

### qcom\_locSetVenueID

### qcom\_locVenueID

### qcom\_locSetVenueName

### qcom\_locSetVenueType

### qcom\_locVenueType

### qcom\_locVenueName

### qcom\_locSetVenueAddress

### qcom\_locVenueAddress

### qcom\_locSetFloorLocation

### qcom\_locFloorLocation

## \_NTP

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

Related: section 9.

### qcom\_ntpSetp

### qcom\_ntpEnable

### qcom\_ntpStatus

## \_Time

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

Related: section 9 - Timekeeping.

### qcom\_timeSet

Sets the current local time of the machine.

**Call format:**

qcom\_timeSet(dt)

*dt* This argument is a *table* similar to the return value from the Lua API function os.date() but comprised only of the following keys: *year, month, day, hour, min, sec, & msec*. The QLE Lua software driver will sanity check the type and range of each table value with respect to its key. Any superfluous keys will be ignored. Any missing keys must be auto-populated by the machine from the machine’s current local time.

If the argument passes checks, then the QLE Lua software driver will notify the host machine and the machine must set its local time to the equivalent time represented but the *table argument*. cp:[[176]](#endnote-129) The machine must also update hardware NV RTC with respect to this time. cp:[[177]](#endnote-130)

Changing the time on a machine via this function will:

* Affect the machine’s current values for:
  + Local time cp:[[178]](#endnote-131)
  + UTC time cp:[[179]](#endnote-132)
  + Unix epoch time cp:[[180]](#endnote-133)
* generate events. Refer section 9 - Timekeeping for more information about these events. cp:[[181]](#endnote-134)

Conversions:

UTC = (local date and time) – (time zone bias where east of GMT is positive)

Any timers, delays and timing functions used by the machine must not be affected by time or time zone changes (for example the QCOM 3 global: *machine operating time*.)

NB: To read the time QCOM users should use the Lua *os* library date and time functions provided.

**Return Value:** Boolean

*true* on success, *nil* on fail (e.g. missing, invalid argument, or spam).

Related: section 9 - Timekeeping; the state event **TIME\_CHANGED**; the event **MACHINE\_TIME\_CHANGED** and the *clib* function *mktime()*.

### qcom\_timeSetStrfmt

### qcom\_timeStrfmt

### qcom\_timeSetTimezone

Sets a new total time zone bias on the machine. It is used to instigate both geographical time zone and daylight savings time changes.

The new timezone bias must take effect immediately in the QLE and machine’s primary applications[[182]](#footnote-48) cp:[[183]](#endnote-135).

The QLE Lua software driver will only advise the host machine of a time zone change if the new time zone value is different from the current time zone value.

**Call format:**

qcom\_timeSetTimezone(integer)

The *integer* argument is the local time zone’s difference from GMT in **minutes**. May include a DST offset when applicable. Positive numbers denote eastward from GMT. Negative numbers adjust westward from GMT.[[184]](#footnote-49)

(Examples: **600** denotes AU east coast during winter; **660** in certain AU States during summer).

Conversions:

UTC = (local date and time) – (time zone bias)

Any timers, delays and timing functions used by the machine must not be affected by time or time zone changes (for example the QCOM 3 global known as the *machine operating time*.)

This function will affect the machine’s local time value. The machine’s current values for UTC and Unix epoch time must **not** change as a result of this function cp:[[185]](#endnote-136).

NB: To read the time, QCOM users should use the Lua *os* library time() and date() functions.

**Return Value:** Boolean

*true* on success, *nil* on fail (e.g. missing, invalid argument, or spam).

Related: section 9 - Timekeeping; the state event **TIME\_CHANGED** and the event **MACHINE\_TIME\_CHANGED**.

### qcom\_timeTimezone

### qcom\_timeSetOSD

### qcom\_timeOSD

## \_Timer

The QLE Lua software driver fully implements this API class of behalf of the host machine. The only contribution the host machine must make is the timely logging of the state event: **QLE\_ONESEC\_TICK**.

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_timerCreate

### qcom\_timerSetp

## \_Machine

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_machineAttendantRequired

### qcom\_machineSetLanguage

### qcom\_machineLanguage

### qcom\_machineLanguagesSupported

### qcom\_machineSetMeterDenom

**Call format:**

qcom\_machineSetMeterDenom(*integer* expint)

This function sets the denomination of all currency values/meters in the machine. This global value is denoted as *meterDenom*. *meterDenom* is set by the *expint* argument relative to the machine's set value of *CurrencyCode*, where:

1 meter unit (meterDenom) = *currencyCode* value x 10*expint*

For example, if the *CurrencyCode* indicated **Dollars** and expint = **-2** then the value of a single meter unit in the machine is **one cent** and meterDenom would be set equal to **0.01**.

As QCOM mandates currency values (global type *camt*) must be an integer value the machine's meter denomination should be set at the lowest common denomination that the machine has to deal with. This should take into consideration the machine's credit meter display, coin acceptor, banknote acceptor and hopper token denominations and (to some extent) any progressive jackpot display, all of which must be an integer multiple of meterDenom.

The argument *expint* must be an integer number. QCOM 3 does not support non-decimal currency

The machine’s *meterDenom* global value must be a write-once only item and must not be able to be set until after the machine's *currencyCode* parameter is set (refer section (4.7) Application Level Configuration).

**Arguments:**

*integer* expint

**Return Value:**  meterDenom : *number* | *nil, errmsg*

The value of *meterDenom* will be returned on success.

The values *nil, errmsg* will be returned on failure.

Once set, all subsequent calls to this function will return true if the argument agrees with the currently set value, otherwise it will return *nil, errmsg*.

Related:

* qcom\_egmSetCreditDenom (s11.19.6)
* qcom\_gameGetp (s11.20.7)
* qcom\_bnaMetersByDenom (s11.32.7)
* qcom\_caSetDenom (s11.34.7)
* qcom\_hopperSetDenom (s11.35.5)

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_machineMeterDenom

### qcom\_machineSetNumberDisplayFormat

### qcom\_machineSetCurrencyDisplayFormat

### qcom\_machineQueueReboot

### qcom\_machineQueueRestart

**Concept only; do not implement this function until further notice.**

### qcom\_machineQueueShutdown

### qcom\_machineQueueHibernate

**Concept only; do not implement this function until further notice.**

### qcom\_machineShuttingDown

### qcom\_machineSetPowerUpTime

**Concept only; do not implement this function until further notice.**

### qcom\_machinePowerUpTime

**Concept only; do not implement this function until further notice.**

### qcom\_machinePowerSaveEnter

### qcom\_machinePowerSaveExit

### qcom\_machinePowerSaveActive

### qcom\_machineInactivityTimer

### qcom\_machineSetFastBoot

### qcom\_machineRAMclear

### qcom\_machineInDebugMode

### qcom\_machineTakeScreenshot

**Concept only; do not implement this function until further notice.**

**Call format:**

qcom\_machineTakeScreenshot(pname, host)

|  |  |  |
| --- | --- | --- |
| **Argument** | **Type** | **Description** |
| pname | string | Screenshot filenames must be prefixed with the *pname* function argument. The machine must append xxx.png to the filename where x is a single digit 0…9  Denoting a machine display enumerate must start at zero. In a gaming machine, ‘0’ must denote the primary (play outcome) display, ‘1’ must denote a top art/pay-scale display screenshot. Any remaining displays may be numbered in any order. |
| host | string | Denotes the IP of the host to save the screenshots. Protocols to be supported are: ftp |

**Implementation of this function is not yet mandatory. Refer section 34.3 for more information.**

This function commands the machine to take a screen shot of all of its displays and forward them to the designated host.

The image file format must be PNG. If a manufacturer has any copyright concerns, it is acceptable for the machine to watermark, blur or degrade the image quality of the screenshot somewhat; however the end result must still be human-readable (i.e. any text can still be read and symbols / art still be uniquely identified)

The machine may enforce a cooldown period between successive screenshots. It is not required to perform another screenshot until the processing of the previous screenshot command has been completed.

**Return Value:**

integer | false, string

On success, the function will return the number of screenshots taken. On fail, the QLE Lua software driver will return false and an error message string.

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_machineOperatingTime

This function returns the number of seconds the machine's primary application has been running since the last full factory reset of the machine (i.e. since last RAM clear). This *machine operating time* must be maintained by the machine in NV memory and must be incremented and updated under a one second timer service. cp:[[186]](#endnote-137) *Machine operating time* must not be derived from any form of world / local time calculation or be affected by associated time changes in the machine. cp:[[187]](#endnote-138)

The *machine operating time* increment service routine must be started before the machine QCOM event logging service is started. cp:[[188]](#endnote-139)

**Call format:**

qcom\_machineOperatingTime()

**Return Value:**

*integer*

Related: QCOM event buffers, section 13. QCOM API timekeeping functions; section 11.5.

### qcom\_machineRand

### qcom\_machineReady

### qcom\_machineUpgradeGetVerify

### qcom\_machineUpgradeQueue

### qcom\_machineUpgradeGetp

### qcom\_machineUpgradeSetp

Refer section 30.4 for the above-named *upgrade\** functions.

## \_Peripheral Devices

### qcom\_peripheralSupported

**Call formats:**

qcom\_peripheralSupported()

qcom\_peripheralSupported(string)

This function can return a list of peripherals currently ‘supported’ by the machine in its software/firmware, or the ‘supported’ status of a single peripheral via the string argument.

Note: The term ‘supported’ is a loose term here and does not fully indicate the meaning of the *boolean* type return values. See below for more information.

**First call format**:

|  |
| --- |
| **qcom\_peripheralSupported()** |

**Return Value:** table

A *btable* must always be returned for this call format.

Possible table keys in the return value are:

"bna" banknote acceptor

"tp" ticket printer

"ca" coin acceptor system

"hopper” coin / token hopper

“ts” touch screen

“mm” mechanical meters

“reels” spinning reel mechanism

A given “peripheral” (shown above) must only have a value of *true* if the peripheral is supported by the machine in software / firmware and either the machine expects it to be present, or it has been detected by the machine.

This function’s return value must only change after a machine restart and before the QLE is started. (Meaning a machine must be restarted in order to add a new peripheral.)

A value of *true* also denotes that the applicable EGM meters will be present and readable via the QCOM API. For this reason once a peripheral has a non-zero meter then the machine must never remove the peripheral (stop expecting it to be present) unless the machine is subsequently RAM cleared. However, peripherals may still be disabled in hardware if they are no longer needed for example, by physically disconnecting them and then ‘quieting’ the fault. Refer section 20.1.

**Second call format:**

The second call format is as follows and allows the caller to query the ‘supported’ status of a single peripheral:

|  |  |  |
| --- | --- | --- |
| **qcom\_peripheralSupported(cn)** | | |
| **Argument** | **Type** | **Description** |
| *cn* | string | See above for the current list of possible peripheral strings. If this argument is provided then the support status of a single peripheral may be queried, e.g:  if (qcom\_peripheralsSupported("hopper")) then ... |

**Return Value:** true | nil

The return value is as per the table key values returned for the first call format but for a single peripheral.

Related:

Peripheral Devices Support, section 21.

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_peripheralConnected

### qcom\_peripheralStatus

## \_Security

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_secQMAcert

### qcom\_secAddSUAcert

### qcom\_secSUAcerts

### qcom\_secMachineCert

### qcom\_secSetUAAverifyCert

Related: Section 23.

### qcom\_secUAAverifyCert

Related: Section 23.

### qcom\_secLogicSealOk

**Concept only; do not implement this function until further notice.**

This function is applicable to all machines. It may be used to check if the machine’s logic area has been accessed since the machine was commissioned (via the logic seal confirmation function, refer section 4.4).

**Call format:**

qcom\_secLogicSealOk()

This function contains no arguments.

**Return Value:**

If there has been any detected logic area access since the creation of the device's private key at machine commissioning, then this function must return *false*. Otherwise the function must return *true* or *nil* if the machine has not been commissioned. This function is intended for high frequency use and must use minimal resources, so to implement, the machine must use a bit of the machine's eSeal NV memory (or power down logic circuit state in the event the machine does not implement an eSeal).

### qcom\_eSealPublicKey

**Concept only; do not implement this function until further notice.**

### qcom\_eSealVerify

**Concept only; do not implement this function until further notice.**

This function is applicable only to machines that implement an OLGR electronic seal.

*Design note: The current way this function operates is only a proposal at this time. Other approaches to challenging a machine electronic seal are still being considered. See below*

Calling this function requires the machine to sign (with the private key related to its electronic seal implementation), a secure hash result of a CSV string comprising of the following:

* the machine’s current date and time,
* a number representing the total number of calls to this function since last RAM clear,
* the return value of a call to qcom\_networkEthAddr(),
* the supplied function argument string capped to 8 ASCII characters max.

For example:

"Mar 14 2012 17:03:23,12345,12:34:56:67:89:0a,abcd1234".

**Call format:**

qcom\_eSealVerify(string)

The string argument will be capped at 8 printable ASCII characters. The string argument excludes commas and space characters and returns an error on fail.

**Return Value:**  string | nil

Returns nil on error (e.g. invalid argument, or if the private key pertaining to the machine’s electronic seal has not been created).

On success, the return value is the digital signature in PEM format.

This function is CPU intensive and may require a cooldown.

Related: OLGR Electronic Seal Minimum Requirements and Section 31.

## \_Digest

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_dgst

### qcom\_dgstCreate

### qcom\_dgstHMAC

### qcom\_dgstHMACcreate

## \_Encryption

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_encEncrypt

### qcom\_encDecrypt

### qcom\_encEncryptCreate

### qcom\_encDecryptCreate

## \_x509

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_X509decode

## \_Content Auditing

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

Also see section 27.

### qcom\_cAuditCommonStart

**Call format:**

qcom\_cAuditCommonStart(alg:string[, seed:hexstring])

This function kicks off a hash calculation of all content categorised as common content in the machine. If a seed is present then the hash calculation will be HMAC based. Once started, the audit cannot be aborted except by a machine restart (e.g. power fail). Calling the function again while a common content audit is currently in progress must restart the common content audit with the latest parameters. This function must return immediately in all cases. cp:[[189]](#endnote-140)

|  |  |  |
| --- | --- | --- |
| **Arguments** | **Type** | **Description** |
| *alg* | string | Hashing algorithm to use e.g. “sha”, “sha1”, “sha256”. The machine must support the same range of algorithms as the openssl dgst command. |
| *seed* | String: hexstring | Optional. The presence of a seed denotes a HMAC calculation must be performed. |

**Return Value:** true | nil, errmsg : string

When a common content audit is complete the machine must throw the state event:

**CAUDIT\_FIN\_COMMON**

Refer to section 27 on QCOM content auditing for more information on this function.

### qcom\_cAuditCommonResults

Refer to section 27 on QCOM content auditing for more information on this function.

### qcom\_cAuditGameStart

**Call format:**

qcom\_cAuditGameStart(gameid, category, alg:string[, seed:hexstring])

This function kicks off a hash calculation of all game content for the given game and content category. If a seed is present then the hash calculation will be HMAC based. Once started, a game content audit cannot be aborted except by a machine restart (e.g. power fail). Calling the function again while a game content audit is currently in progress must restart the game content audit with the latest parameters. This function must return immediately in all cases. cp:[[190]](#endnote-141)

|  |  |  |
| --- | --- | --- |
| **Arguments** | **Type** | **Description** |
| gameid | string | Denotes a game in the machine. Refer QCOM table of global types in the QCOM summary sheet |
| category | string | Either “cd” (code/data) or “media” |
| *alg* | string | Hashing algorithm to use e.g. “sha”, “sha1”, “sha256”. The machine must support the same range of algorithms as the openssl dgst command. |
| *seed* | String: hexstring | Optional. The presence of a seed denotes a HMAC calculation must be performed. |

**Return Value:** true | nil, errmsg : string

When a game content audit is complete the machine must throw the state event:

**CAUDIT\_FIN\_GAME**

Refer to section 27 on QCOM content auditing for more information on this function.

### qcom\_cAuditGameResults

Refer to the QCOM 3 summary spreadsheet and section 27 on QCOM content auditing for more information on this function.

## \_WWW

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_wwwGenerateCert

**Concept only; do not implement this function until further notice.**

### qcom\_wwwCert

**Concept only; do not implement this function until further notice.**

### qcom\_wwwStop

**Concept only; do not implement this function until further notice.**

### qcom\_wwwStart

**Concept only; do not implement this function until further notice.**

### qcom\_wwwStatus

**Concept only; do not implement this function until further notice.**

## \_Lua

### qcom\_luaUsedMem

**Call formats:**

qcom\_luaUsedMem()

The function will return the current total overall memory used by the QCOM Lua Engine. Equivalent to a call to Lua C API functions:

lua\_gc(L, LUA\_GCCOUNT, 0) + lua\_gc(L, LUA\_GCCOUNTB, 0) / 1000.0;

or the Lua function:

collectgarbage(“count”)

**Return value:** integer

Units are in kB.

Related: s11.30.15

### qcom\_luaErrors

Refer QCOM API Summary Spreadsheet.

### qcom\_luaPrintHistory

Refer QCOM API Summary Spreadsheet.

### qcom\_luaHookScript

**Call format:**

qcom\_luaHookScript(seventId, f | nil)

|  |  |  |
| --- | --- | --- |
| **Arguments** | **Type** | **Description** |
| *sEventID* | string | Denotes a QCOM state event.  Sanity check: Type *string* and the string must denote an actual QCOM state event. |
| *f* | function | Function to hook. May be *nil* which removes any previously hooked function for the user. Sanity check: Type *function* or *nil*. |

This function hooks the function denoted by the argument *f* onto the state event denoted by the *sEventID* argument. The function will then be executed by the QLE Lua software driver within the user’s jailed environment whenever the state event is subsequently triggered as per its definition by the host machine.

It is not possible for the same QCOM user to chain more than one function to the same state event. If this is attempted, the new function will simply replace the old.

It is permissible to hook the same function to multiple events.

The QMA and anon user are not permitted to hook scripts via this function.

**Return value:**

*Return true* on success, or *nil[, errmsg]* on failure (e.g. Sanity check fail (see above).

Persistence:

The QLE Lua software driver will clear all hooked scripts upon each restart. In other words, each QCOM user must re-instate all hooked scripts upon each restart of the machine’s application.

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_luaQCIAddCommand

Refer section 24.3 for this function’s full description.

### qcom\_luaEventTime

### qcom\_luaEventData

This function fetches data associated with the last dispatched QCOM State Event. This function is the only way for a QCOM user to retrieve the State Event associated data.

*The reason QCOM State Event data is retrieved this way and not e.g. as an argument passed to the function hooked to the state event (as per QCOM 3 drafts 1-3), is to ensure that any resources associated with the state event data (such as memory use) count towards the quotas of the QCOM user with respect to their State event hooked script. It is also more efficient in the case where the user isn’t interested in the state event data, just the event.*

**Call format:**

qcom\_luaEventData()

**Return value: *table***

The return value will be a Lua *table* consisting of keys only of type *string*. Table data may be of types: *boolean*, *number* and *string*.

Any other key, value pairs returned in the table depend on the State Event (seid). Refer to the QCOM summary Spreadsheet, State event sheet for a complete list.

Example:

The **COIN\_TOKEN\_IN** State event. A call to a qcom\_luaEventData() for a script hooked to this state event would return a *table* like:

{seid = “COIN\_TOKEN\_IN”, amt = 10000}

Related:

* Section 14.

### qcom\_luaSEID

### qcom\_luaPanic

### qcom\_luaPublish

This function allows the calling user to share data with one or more QCOM users. As each QCOM user operates in its own jailed environment within the single Lua state engine, this function is the primary way for a user to share data with other users. All shared data via this function is read-only to recipient QCOM users.

**Call format:**

qcom\_luaPublish(table)

The table must have the schema:

{

[username | "all"] = {data table to be shared for given username/key}

…

}

Table keys denote a list of usernames to share the table {} with. A username may be "all" in order to share with all resident QCOM users in the machine.

A user may share all Lua data types via this function excluding userdata and functions.

Once a table is shared via this function all subsequent updates to shared tables by the sharing user will be automatically visible by intended QCOM users without the sharing user having to call this function again.

Recipient QCOM users can only access the shared data via the **qcom.luaPublishGetValue()** function which provides read only access. All data returned via this function are values, or full copies in the case of strings.

If the sharing user adds sub-tables to an existing shared data table the subtables and data will be visible to recipient users with no futher action required.

It is possible to share data to currently non-existing users, e.g., in the event they are yet to be created at the time.

Typically, a QCOM user wishing to share data would call this function once each restart.

Example:

QCOM user "simple" shares table d2 to a user called "rpc" and another table d1 is shared to "all" users:

d2 = {a = 123, subt = {col = "black", id =123}} d1 = {name="test", age = 55}

assert(qcom.luaPublish({all = d1, rpc = d2}))

To read a shared value (all):

=qcom.luaPublishGetValue("simple", true, 100, "name") – returns “test"

To read a shared value (user ‘rpc’):

=qcom.luaPublishGetValue("simple", false, 100, "subt", "col") – returns "black"

Related:

* section 10.2.6

### qcom\_luaPublishGetValue

Refer QCOM API Summary Spreadsheet.

### qcom\_luaOverload

**Concept only; do not implement this function until further notice.**

**Call format:**

qcom\_luaOverload(fname, f, boolean)

|  |  |  |
| --- | --- | --- |
| **Arguments** | **Type** | **Description** |
| *fname* | string | A string denoting a QCOM API function name. The function must exist in the QCOM API. |
| *f* | string | The function to overload the function denoted by fname. The function must exist in the current environment. |
| *persistent* | boolean | Denotes whether or not to make the overload persistent across machine restarts. *true* = persistent; *false* = one-off |

This function overloads a function in the QCOM API with the function provided (f). The overload will not take effect until next restart of the machine/QLE. (Ensures the overload is inserted into each QCOM user’s environment without side effects.) The QLE Lua software driver will also apply the overload to all QCOM user environments.

The machine must save all persistent overloads (i.e. it will have to actually save the function in NV memory) [[191]](#footnote-50) allowing the QLE Lua software driver to automatically restore all persistent overloads upon each machine restart. This must be done before handing over control to any QCOM user scripts.

When a QCOM API function has been overloaded, when the overloaded function is called, the call includes a new argument inserted as the first argument, which is a variable of type function which is the original QCOM API function. This allows the original function to be chain called. (This will mean an overload actually overload the original function twice in implementation)

The QLE Lua software driver will not allow a QCOM API function to be overloaded more than once by this function.

**Return value:**

boolean [, string]

The return value will be true on success and false on fail. A optional error message string may also be returned upon failure.

Related: section 10.8

### qcom\_luaOverloadClear

**Concept only; do not implement this function until further notice.**

Related: section 10.8

### qcom\_luaDoString

**Call format:**

qcom\_luaDoString(s)

|  |  |  |
| --- | --- | --- |
| **Arguments** | **Type** | **Description** |
| s | string | s is a string representing the serialised Lua data to load. The first character of the string must be ‘{‘. The last character should be ‘}’ but this is not critical |

This function will safely load the string argument as serialised data table as Lua constructor code.

Compared to other serialisation methods available to QCOM users this function is CPU heavy (as it invokes the Lua compiler) and is intended for occasional use only.

The function is safe against all possible malformed arguments.

Any embedded functions in s will be ignored without error.

**Return value:**

* On success, the function returns a table representative of the string argument.
* On failure, the function returns nil, plus a string error message.

Related:

http://www.lua.org/pil/12.html (Explains concept of r/w Lua constructed data)

http://www.lua.org/pil/6.1.html (Shows how to protect variables and functions)

### qcom\_LuaBigNum

Refer QCOM API Summary Spreadsheet.

## \_Audit

### qcom\_auditReg

**Call format:**

qcom\_auditReg(t:table)

|  |  |  |
| --- | --- | --- |
| **Argument** | **Type** | **Description** |
| t | table | The ‘table’ argument defines a small user defined audit mode menu of up to 8 labels.  schema:  **{{label: string: 1…16: printable: unique**  **[, refresh: integer: secs]}, ... (max entries is 8)}**  ‘label’ denotes the page's label  ‘refresh’: number: secs. Optional. Only one page may have the refresh parameter. |

QCOM users privileged to this function may use it to register up to **8** audit mode pages for the display of QCOM user supplied text information within the machine's audit mode.

The machine must display each registered page on its own page in machine audit mode accessible using the page’s designated label. cp:[[192]](#endnote-142)

The QCOM user’s pages must be accessible under the QCOM user's existing required audit mode pages. Related s28. cp:[[193]](#endnote-143)

The ‘label’s denote each pages menu label.

Only one entry may have a refresh field which denotes an automatic page refresh every 'refresh' seconds.

The QLE LSD applies a cooldown of once per QCOM user per user restart.

The machine must throw the **AUDIT\_PAGE\_REQ** state event to request a specific page's latest data from the QCOM user when needed. In response to this state event, the applicable QCOM user may send the text to be displayed by calling the **qcom\_auditResp()** API function which generates an STH message in the name of the function.

In the associated audit mode display:

* The machine must automatically provide horizontal and vertical scrolling as required. cp:[[194]](#endnote-144)
* The machine must use a non-proportional font and each line of text must be left justified. cp:[[195]](#endnote-145)
* The machine must must refresh the page data each the time page is entered. cp:[[196]](#endnote-146)
* If the page also has a ‘refresh’ value associated with it, then the machine must also refresh the associated page’s text with this period. cp:[[197]](#endnote-147)

When the page is exited, the EGM may forget/recover the memory used for the text.

If the argument passes all QLE LSD sanity checks. The QLE LSD will throw the **'auditReg**' STH message with the format as shown in the QCOM Summary Spreadsheet.

The machine must not save anything concerning auditReg in NV memory; the QCOM user must call this function every user/machine restart to reinstate it. cp:[[198]](#endnote-148)

**Return Value:**

true | nil, errmsg

Related:

* The **AUDIT\_PAGE\_REQ** state event.
* Example usage: Refer qcom16 user (<qsim3>\qusers\qcom16)

### qcom\_auditResp

Refer QCOM API Summary Spreadsheet when no description is present.

## \_Persistent Variables

Also refer to section 10.10 for a full description of this feature.

### qcom\_pvCommit

**Call format:**

qcom\_pvCommit(pvname, pvtable) -- update / create pv

qcom\_pvCommit(pvname[, nil]) -- delete pv

|  |  |  |
| --- | --- | --- |
| **Argument** | **Type** | **Description** |
| *pvname* | string | The name of the PV to create / replace.  Max length is **16** bytes; machine to truncate if necessary. Printable characters only and must be a valid Lua identifier. |
| pvtable | Table | nil | PV data associated with *pvname*. There are restrictions on size, keys and data.   * Allowable PV table key (index) types are the *string* and *integer* number data types. * Allowable table values types are *boolean*, *number* and *string*. * Nested tables are not allowed.   (The QLE LSD converts this table to a binary string before sending to the host machine. From the QCOM 3 host machine’s perspective, all PV data are simply binary strings.) |

This function creates or updates the PV (as a whole) denoted by the pvname argument in the machine's PV NV memory. The first argument denotes the uid name of the PV in the PV memory space for the user and the second argument is the PV data represented by a table. Refer to the QCOM 3 SDK for restrictions and sanity checks pertaining to *pvtable*.

The function must not return until the update to NV memory is guaranteed to complete. cp:[[199]](#endnote-149)

If the PV denoted by the string argument does not exist, then it must be **created**.

If the PV already exists, then the QLE Lua software driver will **replace** the existing table provided in the second argument. For example; copy the table into spare NV memory and switch it in via a minimal sized critical section of code.

If the second argument is absent, equal to *nil*, or is an empty table, then the QLE Lua software driver will **delete** the PV denoted by the string argument from NV memory.

Refer 10.10.2 regarding requirements with respect to updates to PV’s in NV memory.

If two QCOM users use the same *pvname* string argument, then this will not result in any contention. The QLE Lua software driver will maintain a PV namespace per QCOM user to ensure this.

**Return Value:**

The return value will be an integer number denoting size in bytes of the string serialised table on success and *nil, errmsg* on fail. The reasons for a call to the function to fail are:

* Illegal *pvname*
* Missing argument *pvname*
* Out of memory
* Illegal pvtable (refer 10.10)

Refer to the QLE Lua software driver in the QCOM 3 SDK for sanity checks and *errmsg* return values.

### qcom\_pvCommit\_string

Another version of the previous function that simply accepts raw binary strings as the PV data (second argument).

Reiterating that the machine must support **binary** strings for PV data (i.e. strings that could contain non-printable characters in the range 0…255).

**Call format:**

qcom\_pvCommit(pvname, data:string) -- update / create pv

qcom\_pvCommit(pvname[, nil]) -- delete pv

### qcom\_pvRestore

### qcom\_pvRestore\_string

### qcom\_pvStats

### qcom\_pvSetp

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

## \_User Meters

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

Related: User Created Meters (s10.11).

### qcom\_userMeterCreate

### qcom\_userMeterUpdate

### qcom\_userMeterDestroy

### qcom\_userMeterRead

### qcom\_userMeterOwner

### qcom\_userMetersSetMax

### qcom\_userMetersAvail

## \_EGM Maintenance

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_egmSetMinMaxRTP

### qcom\_egmMinRTP

### qcom\_egmMaxRTP

### qcom\_egmSetMaxRTPDeviation

Dot not implement.

Refer to QCOM API Summary Spreadsheet’s functional description for more information.

Related:

* QCOM v1.x MAXSD.
* OLGR Jackpot Systems minimum requirements document.

### qcom\_egmMaxRTPDeviation

Dot not implement.

### qcom\_egmSetCreditDenom

### qcom\_egmCreditDenom

### qcom\_egmCreditInputDisable

### qcom\_egmCreditInputEnable

### qcom\_egmCreditInputEnabled

Related: **qcom\_playSetPEF()**

### qcom\_egmSetMaxBet

**Call format:**

qcom\_ egmSetMaxBet(f)

|  |  |  |
| --- | --- | --- |
| **Argument** | **Type** | **Description** |
| egmmaxbet | egmmaxbet | See below. Also refer table of global types |

This function sets the maximum permissible bet on the machine. Specifically, the machine must ensure that no total bet for a single play for any game on the machine exceeds this amount. (Excludes gamble feature bets.)

The machine must apply *egmmaxbet* dynamically during the player betting user interface for example by automatically *culling* the number of available lines, ways per line and credits bet per line etc. The machine is not required to cap at exactly *egmmaxbet*, just so long as it does not allow the value to be exceed.

For example if a game has an inherent bet range of 1….100 credits, then it must be theoretically possible to play the game at one credit if *maxegmmbet* is set to the value of a single credit.

This *culling* must occur even if it orphans or disables existing bet buttons on the machine’s UI. (This is primarily a concern for machine using physical bet buttons.) It is optional for the machine to display an explanatory message in this case, but it may be assumed that the maximum bet limit *egmmaxbet* will be a widely published figure. Ideally, machines should try to make *egmmaxbet* limit enforcement as clear and as user friendly as possible.

Games typically have a minimum bet (very commonly this is the value of a credit). If *egmmaxbet* is lowered below this amount then this must disable all betting in the game, thus making the game unplayable. *It is the applicable controlling authority/systems responsibility to ensure it does not enable a game then set egmmaxbet to a value that makes the game unplayable and if it does it is also their responsibility to display a suitable message on the machine.*

When a maximum bet limit is not applicable for the machine's current operating environment, *egmmaxbet* will be set to a very high value. Accordingly all games must also implement their own internal maximum bet limit per game (either by game design / betting user interface) if it is not intended to offer an infinite max bet facility for the game.

When *egmmaxbet* is changed, the new value must be applied immediately if the machine is in idle mode, otherwise upon next return to idle mode. The accepted value of *egmmaxbet* must be reflected immediately in any subsequent call to the QCOM API function *qcom\_egmMaxBet.*

Ante-bet Games, Feature Unlocks and *egmmaxbet*

Refer to the glossary for the definition of an ante-bet game that is applicable in this document.

If *egmmaxbet* is lowered to below a feature unlock threshold then the game must be still playable but without the feature. *It is the applicable controlling authority/systems responsibility to ensure it does not enable a game then set egmmaxbet to a value that makes the game unplayable or certain game features unattainable.*

**Regulators need to be aware that when they approve games with an ante-bet feature or a minimum bet to unlock game features (or with a greater then one credit minimum bet in general), that they are potentially limiting future policy options (or increasing the cost or ramifications thereof) in regards to changing the value of *egmmaxbet*.**

**Entities purchasing games (e.g. licensed gaming venues) with ante-bet features need to be made aware that the game may be potentially largely broken if future regulatory policy concerning maximum bet is decreased.**

Related:

* Section 11.20.7 qcom\_gameGetp(): *betthresholds*  value.

### qcom\_egmMaxBet

### qcom\_egmCreditMeter

### qcom\_egmBetMeter

### qcom\_egmWinMeter

### qcom\_egmMeter

### qcom\_egmMeters

**Call format:**

qcom\_egmMeters()

EGM meters, meterID’s and definitions the machine must support are listed in the QCOM 3 summary spreadsheet: ‘Meters’ worksheet. cp:[[200]](#endnote-150)

**Return Value:** key / pair table: meterID:string / meter current value:integer

Related:

* QCOM API function: qcom\_gameVarMeters

### qcom\_egmSetMeterDisplayFunction

**Concept only; do not implement this function until further notice.**

**Call format:**

qcom\_egmSetMeterDisplayFunction(f)

|  |  |  |
| --- | --- | --- |
| **Argument** | **Type** | **Description** |
| f | function | See below |

This function installs a call-back function which must be invoked by the machine whenever it needs to display its master meters. When a call-back function is setup via this function, the machine upon needing to display it master meters, instead of displaying its factory default list of master meters, must display the table of meters provided by the return value from the call to the call-back function.

Refer 15.5 for an introduction and additional information.

The QLE Lua software driver will invoke f at the privilege level of the original caller to this function.

Once the call-back function has been setup, the call-back function and the privilege level must be saved by the machine in NV memory and automatically restored after any machine or machine application restart. (NB the key to saving the function is the Lua string.dump() API function)

The call-back function must be invoked via a call to the Lua API function pcall(). Upon any error, or the call-back function return value is not a table, or the table is greater than **60** entries, then the machine must abort using the return value and revert to its hard-coded master meter display function.

The call-back function denoted by the above argument when invoked must return a table. Specifically it must return an associative indexed table of intended meters and values for use with the machine's master meters display in machine audit mode. Table keys must be strings and must denote the meter display name text of the corresponding value. The table values must denote the corresponding value of the meter represented by the table key represented as strings. This allows the call-back function to format the meter and denote units appropriately.

The machine must verify that returned table keys and values are all string types. On error the offending table entry must be ignored.

The machine must take the table returned from the call back function and display it as its list of master meters in its audit mode master meter display in the order given by the Lua pairs() iterator function.

The machine must truncate returned table keys to a maximum of **40** characters and truncate values to a maximum of **30** characters. Key, value pairs must be displayed one per line, with keys left justified and values right justified. The machine must use leader characters or alternating shading/colour, or some other method to assist the user in reading the display. If there are too many meters to display on a single page, then the machine must break the display up over multiple pages and provide a user interface to navigate between pages.

**Return Value:**

true = success, the call-back function was setup and saved

false = fail, the call-back function failed a sanity check and was not saved

*To maintain security and integrity of the machine’s master meters display in audit mode, it is recommended that the QMA does not give away privilege to this function without careful consideration.*

### qcom\_egmDisplayMeters

Refer QCOM API Summary Spreadsheet.

### qcom\_egmState

**Call format:**

|  |
| --- |
| qcom\_egmState() |
| This function has no arguments |

This function returns the EGM’s state similar in nature to QCOM v1 GSR STATE.

**Return Value:** string[, string]

The return value will be strings and represent the EGM’s state as per the following table:

|  |  |  |
| --- | --- | --- |
| **State**  **(1st return value)** | **Sub-state**  **(2nd return value)** | **Description** |
| “hopper collect” | nil |  |
| “gamble” | nil | Gamble / double-up |
| “idle” | nil | As per QCOM v1 |
| “idle” | “info” | (*feature on hold & may be changed to a flag / IsIn() like API function*) Related section 3.7 |
| “idle” | “reserved” | Machine reserve feature |
| “idle” | “pael” | (*feature on hold*) Player Accessible Event Log |
| “play” | nil | As per QCOM v1 |
| “play” | “feature” |  |
|  |  |  |
| “system lockup” | nil | Refer section 16. |
| “system lockup” | “pael” | *(feature on hold)* |
| “lp award” | nil | Linked Progressive award lockup |
| “take win” | nil | The EGM is waiting for the user to press take-win, gamble or repeat bet and play-on or adjust bet. As soon as any of these action commence, the EGM must exit the “take win” state |

The return value does not indicate any concurrent conditions on the machine such as: faults, door opens, audit or test mode as these are covered by other QCOM API functions. For example, if the machine was in-play but with a fault condition, then this function just indicates the “in-play” return value. The machine must not cause the return value of this function to change during a fault, door open, audit mode, or test mode access. cp:[[201]](#endnote-151)

Refer to the QCOM 3 state transition diagrams in the QCOM summary spreadsheet – “SE Diagrams” sheet. While state transitions are controlled by the host machine, they are sanity checked by the QLE Lua software driver to ensure they only transit as per the QCOM 3 SE Diagrams. If the QLE Lua software driver detects an illegal state transit, it will immediately shut down the QLE and send a panic message to the host machine which requires the machine lockup and restart. The QLE will never let QCOM users see an illegal state transition.

Related:

Illegal State Transitions section 14.2.1

QCOM State Events (s14)

qcom\_machineShuttingDown()

qcom\_machinePowerSaveActive()

qcom\_machineReady()

qcom\_egmDoorsClosed()

qcom\_egmDoorState()

qcom\_egmInFault()

qcom\_egmInQuietFault()

qcom\_egmInTestMode()

qcom\_egmInAuditMode()

qcom\_egmOK() –- no faults, all doors closed, not in audit/test modes

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_egmDoorsClosed

### qcom\_egmDoorState

### qcom\_egmInFault

### qcom\_egmInQuietFault

### qcom\_egmInTestMode

### qcom\_egmInAuditMode

### qcom\_egmOK

### qcom\_egmKeySwitchDisable

### qcom\_egmKeySwitchEnable

### qcom\_egmKeySwitchStatus

### qcom\_egmSPAMA

### qcom\_egmSPAMB

### qcom\_egmSPAMC

### qcom\_egmGPM

### qcom\_egmSMS

The function replaces what was the External Jackpot Information Poll (EXTJIP) display functionality in QCOM v1 with a more general-purpose function. It is now re-themed as a generic Short-Message-Service like display feature able to be utilised for any short message information type service / application.

|  |  |  |
| --- | --- | --- |
| qcom\_egmSMS(t) | | |
| **Argument** | **Type** | **Description** |
| *t* | table | Table of string. See below |

The single table argument must be an indexed table of string up to a maximum limit of 8 entries (with indices 1…8). The table may be sparsely populated. The QLE Lua software driver will ignore additional entries exceeding the limit or outside the range. Each string value must be **0...32** characters. If greater than this limit, the QLE Lua software driver will truncate the string back to the maximum. For missing entries, the machine must assume the values for those indices are unchanged. Only a value of “” will clear an entry.

The machine must display each string value on its primary display as per the QCOM v1 EXTJIP requirements where each string in the table is akin to a single entry display in a EXTJIP (refer QCOM for more information).

Updates

It will be very common for the machine to receive a new set of messages to display while it is currently in the process of displaying one or more string. In this case, the machine must ensure it updates any currently displayed entry/s immediately and as gracefully as possible. A QCOM user spamming this function must not cause some strings, or parts thereof, not to be displayed, i.e. the machine must ensure it continues where is was in the indexed set rather than restart the whole display cycle every time there is an update.

Example arguments

t = {“Big Bucks Mega $1,234,56”, “Bonus Points Jackpot 123,456”}

or

t = {“Draw tonight at 6pm”, “Closing at midnight”, ”Storms likely this evening”}

### qcom\_egmGambleSetp

### qcom\_egmGambleGetp

Refer QCOM API Summary Spreadsheet.

### qcom\_egmReserve

### qcom\_egmSetReserve

## \_Game Maintenance

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_gameCurrent

### qcom\_gameNum

### qcom\_gameList

### qcom\_gameMaxEnabled

### qcom\_gameLanguagesSupported

### qcom\_gameUID

### qcom\_gameGetp

This function returns properties of the game denoted by the function argument/s.

**Call formats:**

qcom\_gameGetp(gameid)

qcom\_gameGetp(gameid, key)

The first call format is:

|  |  |  |
| --- | --- | --- |
| qcom\_gameGetp(gameID) | | |
| **Argument** | **Type** | **Description** |
| *gameid* | gameid | Denotes a game in the machine. Refer QCOM table of global types in the QCOM summary sheet |

**Return value:**

Returns *nil* on error (e.g. gameid does not exist) otherwise an associative indexed table will be returned comprising of the following keys-value pairs for the game denoted by gameid function argument:

|  |  |  |
| --- | --- | --- |
| **Key\*** | **ValueType!** | **Description** |
| gameid | gameid | Refer table of global types. |
| gamename | gamename | Refer table of global types.  Related qcom\_gameName() s11.20.8. |
| gamever | gameVer | Refer table of global types. |
| gameyear | integer | Year of game design. Manufacturer assigned. |
| gvn | string:6 | The game’s GVN as per QCOM v1. Refer table of global types. |
| [themename] | string | Game Theme Name.  If the game is a part of a multi-game packaged theme of games, then this string must denote the theme name. Games that share one or more progressive levels **must** have a themename.  Example: The manufacturer might have a set of games under the theme‘travel mania’; with individual games like ‘Paris’, ‘Sydney’, ‘London’, etc. |
| [denomination] | integer | Credit denomination of the game when the game is a single denomination game. This value must state how many meter units (aka qcom.machineMeterDenom(); typically 1 cent ) make up 1 credit e.g. if the game is what we call a 10 cent game (i.e. 1 credit = 10 cents) then denomination = 10. |
| [denominations] | btable | Used when the jurisdiction allows a player selectable denomination game to appear as a single game in QCOM 3.  Re player selectable denomination games in QCOM 3. QCOM 3 supports a single game record having multiple denominations[[202]](#footnote-51), but it may still be a requirement in some jurisdictions that player selectable denomination games must be implemented as per QCOM v1.6, where each game denomination is reported as a separate game w.r.t QCOM.[[203]](#footnote-52)  Example: A player selectable game that offers 1,2,5,10 cent denominations; The denominations would equal: {[1] = true, [2] = true, [5] = true, [10] = true} |
| [maxden] | integer | NA / TBA  *The QCOM v1 MAXDEN control parameter is obsolete / removed in QCOM 3.* |
| minbet | integer | Minimum required bet to instigate a single play for the game in any given variation. In units of credits. |
| maxbet | integer | Largest supported possible bet for the game in any given variation. (This value must not be capped by *egmmaxbet*). In units of credits. |
| [paylines] | integer | Maximum number of pay-lines in the game, if applicable. *The QCOM v1 MAXLINES limiting parameter is obsolete / removed in QCOM 3. A QCOM user can optionally use the QCOM 3 paylines field to decide on suitability rather than the EGM hard-coding a rule.* |
| [betthresholds] | table | **WIP Do not use this property at this time.**  Bet thresholds.  Applicable to ante-bet games or games where certain game features are only accessible given certain betting options or thresholds. The table must be an associativelyindexed table, where each key is a feature name (type *string*)and each value (also type *string)* represents a minimum bet amount or pattern for which the feature denoted by the table key becomes accessible. In the string value, the units of any amount must be stated. The following unit abbreviations may be used: “cr” = credits; “cpl” = credits bet per pay-line; “l” = pay-line. |
| [notes] | string | Arbitrary manufacturer notes concerning the game. Include any game prerequisites or limitations that are not denoted or inferred by other game properties returned by this function. Related qcom\_playOKex(). |
| pnum | integer | The number of progressive levels in the game |
| [ptname] | string: printable incl. spaces; min length 1 | Progressive theme name. These are manufacturer assigned and are optional. If a progressive theme name is not applicable then the machine returns *nil* for this value. |
| … | … | The machine may also return any number of additional custom game properties in the *table* return value. Each game may also return a different or custom set of game properties**. \*\*** |

\*All keys are type string. [] denotes a conditional property.

! Refer to the “Global Types” sheet in the QCOM summary spreadsheet in order to resolve non-Lua types.

\*\*Restrictions and requirements pertaining to new game properties creation:

* OLGR must be advised well in advance of all new game properties
* Machine manufacturers must agree to make reasonable changes to their proposed properties when suggested by OLGR. *OLGR will make suggestions in order to either unify or collate properties of identical nature between manufacturers where possible, or to prevent contention issues (e.g. with key names) between manufacturers, or general suitability issues. OLGR will avoid making any recommendations if a recommendation represents a commercial disadvantage to any party and will only tend to make recommendations after a clear and open industry trend is emerging.*

The table object that is returned will be a copy generated in the QCOM user’s environment. In other words, if the user changes any value in the structure this will not affect any other QCOM user’s instance of the same table retrieved via the same function. Refer 10.5.4.

This function has a second call format as follows:

|  |  |  |
| --- | --- | --- |
| qcom\_gameGetp(gameid, key) | | |
| **Argument** | **Type** | **Description** |
| *gameid* | gameid | Denotes a game in the machine. Refer QCOM table of global types in the QCOM summary sheet |
| *key* | string | *Key* denotes the value in the game information table for the game denoted by gameid to return |

This format allows a single property and associated value to be retrieved

**Return value:**

*nil* on error

Otherwise returns the value for the given key, whatever type that may be.

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_gameName

### qcom\_gameGEF

How to display GEF disabled games on a QCOM 3 EGM:

In a QCOM 3 machine there, is no longer a requirement for a “no game enabled” display as per QCOM v1. Where a QCOM 3 EGM is trying to be QCOM v1 compliant, the intent is that the new QCOM 3 PEF feature / API will display the QCOM v1 ‘no game enabled’ condition along with QCOM v1’s other CDC conditions i.e. A QCOM user will implement the QCOM v1 ‘no game enabled’ display (i.e. the OLGR ‘qcom16’ user will implement it).

To display GEF disabled games in a QCOM 3 machine: cp:[[204]](#endnote-152)

* A simple text message saying ‘game disabled’ is acceptable. Cycling this text with other messages is also acceptable. The display here may be more prominent at the discretion of the EGM manufacturer.
* In a multigame EGM, re the **game select screen**, the approach used for displaying disabled games in QCOM v1 would be acceptable. Hiding the game on the game select display became the common approach here (where in QCOM v1 greying-out the game was the other option). If all games were disabled, a totally blank game select display would be acceptable, but it’s also ok if the EGM displays an extra message to explain the blank screen here if desired.
* RE Multi-denom games (where each game denom appears as a separate game in QCOM); the EGM may either hide disabled denoms from selection, or allow the player to still select disabled denoms and then display “This denomination of the game has been disabled” or similar. text message as per the first dot point above.

Related:

* **qcom\_playOKex()**

### qcom\_gameDisable

### qcom\_gameEnable

### qcom\_gamePCmeters

### qcom\_gameVarNum

### qcom\_gameVarCurrent

### qcom\_gameSetVar

This function requests that the EGM change the designated game's current variation. The change must not affect a game in play. cp:[[205]](#endnote-153)

If a game presents with multiple variations then support for variation hot-switching via this QCOM API function is mandatory (as per QCOM v1.x), even if the game has a progressive component (optional in QCOM v1.x). cp:[[206]](#endnote-154)

**Call format:**

qcom\_gameSetVar(gameid, varid)

|  |  |  |
| --- | --- | --- |
| **Argument** | **Type** | **Description** |
| *gameid* | gameid | Denotes a game in the machine. Refer QCOM table of global types in the QCOM summary sheet |
| *varid* | integer | Denotes a Variation ID number of the above game |

Initial prerequisites before a variation change request will be passed onto to the host EGM by the QLE LSD are:

* The EGM’s credit meter is currently equal to zero.
* The EGM is currently in either idle mode or a system lockup.

The QLE Lua software driver will ignore the request and return a failure error message if any of the above prerequisites fail at the time the function is called.

The EGM must not deny a variation change just because progressive component parameters would change. In this case the EGM must also change the progressive component parameters acordingly and carry over the contribution correctly. cp:[[207]](#endnote-155)

Upon processing the *gameSetVar* STH message, the EGM may still deny a variation change request for other reasons[[208]](#footnote-53) even after returning *true* via this function. If an EGM does deny a variation change after the fact in this way, it must indicate the reason in the associated **GAME\_VAR\_CHANGED** state event which has a *fail* key/field of type *string* which must indicate the reason. cp:[[209]](#endnote-156) All EGM manufacturer specific reasons for denying a variation change request must also be listed in the *notes* field of the game’s property table. Refer s11.20.7. cp:[[210]](#endnote-157)

Games with a different progressive configuration per game variation: cp:[[211]](#endnote-158)

For these games, when a variation switch is made active via **qcom\_gameSetVar()**, the EGM must change the games’s currently reported progressive levels properties[[212]](#footnote-54) with the new the properties (e.g. sup, pinc, ceiling, hrate) and automatically adjust affected properties in kind (e.g. prizeiw, atv, overflow, rtp). This action must not affect level properties such as e.g.: adjneg, adjpos, games, hits, mode, modes, pgid, plevs, pluid, ptm, setp, turnover, wins). A **PROGR\_CFG** state event must also be logged here (in order to advise the QLE LSD about the changes to all the progrcfg records properties).

Related:

* **qcom.gameGetVar().progrcfgvar** table which details the progressive properties for the given variation.
* Host machine state (hms) schema:
  + *hms->games[]->varlist[]->progrcfgvar*
  + Refer **hms\_schema.lua** in the QCOM 3 SDK
* For a progressive level that is also shared among multiple games:
  + changing one game variation will update the pluid for all games that share that level. For this reason, for these games must add a ‘var’ property to the progrcfg record so that it is clear what variation’s progr cfg is currently being read.
  + The EGM would also disable here until every associated game was set to the same variation. Related **PLAY\_OK\_EX** state event.

**Return value:** *true | nil, errmsg : string*

*true* if the initial function call passes all sanity checks and prerequisites.

*nil, errmsg:string* otherwise. Refer to the QLE Lua software driver in the QCOM 3 SDK for all possible failure RVs error messages.

**Related:**

* **GAME\_VAR\_CHANGED** state event (denotes success or fail).
* **VAR\_ENABLED** event (denotes a successful variation change).
* qcom\_playOKex() API function. Section 11.26.3. For example, when an EGM requires all games to be set at the same variation.
* **qcom\_gameVarMeters()** API function.
* QCOM v1.x Variation Hot Switching.

Refer to the QCOM Summary spreadsheet for more information on the above related items.

### qcom\_gameVarList

### qcom\_gameVarGetp

This function returns more detailed properties of a game variation for the game variation denoted by the function arguments.

**Call format:**

qcom\_gameVarGetp(gameid, varid)

|  |  |  |
| --- | --- | --- |
| **Argument** | **Type** | **Description** |
| *gameid* | gameid | Denotes a game in the machine. Refer QCOM table of global types in the QCOM summary sheet |
| *varid* | integer | Denotes a Variation ID number of the above game |

**Return value:**

The function will return nil on error.

On success an associative indexed table is returned consisting of the following key-value pairs:

|  |  |  |
| --- | --- | --- |
| **Key\*** | **ValueType** | **Description** |
| rtp | number | Minimum theoretical percentage RTP of the game variation excluding any progressive game component. |
| [stddev] | number | Expected standard deviation of the *rtp* excluding any progressive game component.  This value is for comparative purposes only, a voluntary declaration/estimate of the game variation’s RTP volatility and does not have to be an exact value e.g. it may be determined empirically during manufacturer fast-play testing and hardcoded here and should be accurate (stable) to 3 decimal places given sufficient play-testing. Round the least significant digit down.  For more information on how to calculate this value empirically refer to QCOM v1 “standard deviation display”.  *The QCOM v1 MAXSD as a limiting parameter has been removed in QCOM 3.* |
| [rtpmax] | number | Maximum theoretical percentage RTP of the game variation for any given bet. Report only if this value is different from the value of ‘rtp’ property above by more than 0.2%[[213]](#footnote-55). Again, exclude any progressive game component. |
| [rtptable] | table | **WIP Do not use this property at this time.**  A numerically and/or string indexed table. Optional. Each table key may represent either a bet amount (in credits) or game mode and the value is the corresponding RTP. Applicable to games with a non-linear rtp distribution.  Excluding any progressive game component. |
| [notes] | string | Arbitrary notes concerning the property table such as any variation prerequisites. If a variation has any prerequisites (excluding any rtp prerequisites) then the variation must have a notes property. |
| [progrcfgvar] | table | In QCOM 3, each game variation may have/apply a different hardcoded set of progressive level properties per game variation.[[214]](#footnote-56)  (The alternative is to allow progressive level properties to be set/changes externally via the QCOM API function **qcom.progrSetp()[[215]](#footnote-57)**.  Refer SDK hms\_schema.lua: *progrcfgvar* for the schema here. |
| … |  | The game manufacturer may add custom game variation properties to the table return value. Refer to qcom\_gameGetp() QCOM API function for caveats. |

\*All keys are type string

[] denotes a conditional property.

All properties reported must be constant values

*Todo review and clarify for games like blackjack*

### qcom\_gameVarMeters

Returns an associative indexed table of 'game' related meters for the specified game and variation. Refer to the QCOM 3 summary spreadsheet: ‘Meters’ worksheet, for defined 'game' key index string names. The corresponding data value for a given key is a number representing the current meter value.

**Call format:**

qcom\_gameVarMeters(gameid, varid)

|  |  |  |
| --- | --- | --- |
| **Argument** | **Type** | **Description** |
| *gameid* | gameid | Denotes a game in the machine. Refer QCOM table of global types in the QCOM summary sheet |
| *varid* | integer | Denotes a Variation ID number of the above game |

The EGM must keep separate variation meters for each variation in a game until the next RAM clear / factory reset. cp:[[216]](#endnote-159) This requirement is same as QCOM1.6.

Respective EGM master meters (refer qcom\_egmMeters()), will equal the sum of the all the variation meters provided the EGM’s game set has not changed since RAM clear (re downloadable game support). cp:[[217]](#endnote-160)

Game variation meters the machine must support are listed in the QCOM 3 summary spreadsheet: ‘Meters’ worksheet. cp:[[218]](#endnote-161)

**Return value:**

On success, an associative indexed table of 'game' related meters for the specified game and variation will be returned.

On failure, the function will return nil, errmsg: string.

Refer QCOM API Summary Spreadsheet for more information.

## \_Progressive Prize Support

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

Related: Chapter 25 on Progressive Prize Support.

### qcom\_progrNum

### qcom\_progrGamesList

### qcom\_progrMeters

### qcom\_progrTurnoverMeter

### qcom\_progrLastHit

### qcom\_progrList

### qcom\_progrGetp

Refer 25.2.1

### qcom\_progrSetp

### qcom\_progrSetPrize

### qcom\_progrResetLockup

### qcom\_progrPosAdj

Refer section 25.2.2.

### qcom\_progrNegAdj

Refer section 25.2.2.

### qcom\_progrModeChange

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

## \_Events

Refer section 13.

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_eventsGetIterator

### qcom\_eventsGetLatest

### qcom\_eventsGetLast

### qcom\_eventsStatus

### qcom\_eventsAddUserEvent

**Concept only; do not implement this function until further notice.**

### qcom\_eventsSetTTL

### qcom\_eventsSetSLx

### qcom\_eventsGetSLx

## \_ECT

Also refer to section 17.

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_ectIsEnabled

### qcom\_ectDisable

### qcom\_ectEnable

### qcom\_ectMaxECT

### qcom\_ectSetMaxECT

### qcom\_ectAddCredit

This function can add credit to the machine's credit meter.

The machine must allow an ECT instigated by this function to take place in all states other than:

* Those states for which the QCOM API function qcom\_egmOK() would return *false*.
* The “hopper collect” state. (Related: qcom\_egmState() )

If the machine is in a System Lockup (s16), to avoid potentially confusing the player, it is recommended that the QCOM user wishing to perform an ECT via this function, does not instigate the ECT unless the currently focused System Lockup display is also that of the QCOM user. Related: **SYSTEM\_LOCKUP\_TIMEOUT** state event.

There are two call formats for this function. The **first** call format is:

**Call format:**

qcom\_ectAddCredit(amount, reason, transactionID)

|  |  |  |
| --- | --- | --- |
| **Argument** | **Type** | **Description** |
| *amount* | integer | denotes the amount to be transferred to the machine’s credit meter |
| *reason* | String | An arbitrary transaction description for short term reference / display purposes. The *reason* must be 24 characters or less, truncated as necessary by the machine on function entry. If the argument equals the empty string then the API will substitute it with the calling user’s username string.*Examples: “Account xfer”, “Account: 321324”, “Jackpot ID:12345678 lvl 1”, “Ticket-in”, “RCRF”.* |
| *transactionID* | string | An arbitrary transaction ID string, 8 characters or less, truncated as necessary by the machine on function entry |

Before attempting to perform the operation, the QLE Lua software driver will first apply all the sanity checks listed in the QCOM SDK and then check the following conditions in order shown aborting on the first failure:

|  |  |
| --- | --- |
| **Condition** | **Return Values on failure** |
| ECT has been enabled via the QCOM API *qcom\_ectEnable()* function | nil, “ect disabled” |
| Ensure *transactionID* is not equal to the last successful *transactionID* for the calling QCOM user. | nil, “duplicate transaction” |
| A cooldown per QCOM user of **1** seconds applies to this function. | nil, “on cooldown” |
| The current return value of the QCOM API function qcom\_egmOK() would be *true*. | nil, “egm not ok” |
| The machine must not be in the “hopper collect” state. Refer: qcom\_egmState() | nil, “hopper collect” |
| Confirm the amount is not zero or negative | nil, “amount 0 or negative” |
| Confirm that the *amount* is less than user’s MAXECT limit set by the *qcom\_ectSetMaxECT* function | nil, “maxect” |
| Ensure the credit meter is capable of displaying the updated value. | nil, “amount too large” |
| The current return value of the QCOM API function qcom\_machineReady() would be *true*. | nil, “not ready” |

If all the above conditions are met, the machine must immediately (in the order shown below):

* Attempt to add the amount to the machine’s Credit Meter and Total Cashless In meter. (NB the EGM could have had an issue since the sanity checks were undertaken so it is possible for the ECT to still fail.)
* If successful:
  + Log the (ref s13.6.1) **ECT\_TO\_EGM** event to NV memory (denotes success).
  + Log the **ECT\_TO\_CM** state event (denoting success)
  + Log the **EVENT** state event for the **ECT\_TO\_EGM** event.
  + Instigate a short cash-in like sound that has a rising terminal (related:section 3.6) and add the credit to the credit meter.
* If not successful:
  + Log the **ECT\_TO\_CM** state event (denoting fail)

Any residual amount not of a whole credit value must still be added to the EGM’s credit meter and stored until it is either transferred off the machine, or rounded off (e.g. due to subsequent ECT transfers), or cancelled, etc.

If the whole rounded down credit meter value is zero and there is still residual credit, the EGM must update the credit meter display to display the residual credit amount in the units of the set currency (if not already being displayed).

**Return Value:**

The function will return *true* on success (note that success here does not mean the ECT will actually be successful (Refer to the next received ECT\_TO\_EGM state event for that information). On fail, the function will return *nil* plus one of the error strings listed in the above table. Refer to the QCOM SDK for additional sanity checks and error return values.

There are two call formats for this function. The **second** call format is:

**Call format:**

qcom\_ectAddCredit()

This call format takes no arguments.

**Return Value:**

The function will return a table containing details of the last successful ECT-to-CM for the calling user. The table schema will be as per the ECT\_TO\_CM state event data table schema. If there is no ECT yet for the QCOM user, then the function will return *nil*.

Related: Section 17 (ECT)

### qcom\_ectSubtractCreditAuthorised

This function authorises a QCOM user to subtract credit off the machine's credit meter for the specified amount via any of the QCOM ECT 'subtract' credit QCOM API functions. An authorisation lasts for the number of seconds denoted by the *timeout* argument. An ECT authorisation created via this function will not persist across a machine restart.

Note: Authorisation of ECTs is only applicable to credit redemption operations via **qcom\_ectSubtractCredit()** and **qcom\_ectTicketOutSubtractCredit()** QCOM API functions.

**Call format:**

qcom\_ectSubtractCreditAuthorised(username, camt, timeout)

|  |  |  |
| --- | --- | --- |
| **Argument** | **type** | **Description** |
| *username* | global type | Denotes the QCOM user being authorised to perform a ECT ‘subtract’ operation. |
| *camt* | global type | The authorised amount of the pending transfer |
| *timeout* | integer | Amount of time in seconds in which the user has to perform the ECT. This timer does not persist across machine restarts. |

Conditions for success:

* *username* must be a valid & existing QCOM username
* *camt* must not be less than or equal to zero.
* *camt* must be less than MAXECT.
* *camt* must be <= the current credit meter value.
* *timeout* must be in the range (1…limit). Refer to the QCOM SDK for the current limit.
* A previous authorisation must not be still outstanding.

Refer to the QLE Lua software driver in the QCOM 3 SDK for additional sanity checks, the order of checks and all error return values.

An authorisation will be automatically **terminated** by the QLE Lua software driver if its timeout expires, or the respective QCOM user declines it, or after a successful call by the authorised user to any of the ECT subtract QCOM API functions.

An authorisation must be automatically **terminated** by the machine (by it logging the **ECT\_FAILED** state event), if the credit meter decreases at any time during the timeout period. Refer **ECT\_FAILED** state event’s definition for more information.

It should be noted that because the QLE can lag behind the host machine and depending on how credit redemption is operated by the applicable QCOM users, it is possible for an ECT authorisation to be terminated by the host machine almost immediately in some rare circumstances if its state changes within a short time window.

**Return Value:**

On success:

* The QLE Lua software driver will increment the ECT subtraction serial number global value *ectsubsn* (refer table of global types) and then the machine must throw the **ECT\_AUTHORISED** state event.
* Return *true*

On failure:

* Returns *nil, string* where the string is an error string. Refer to the QCOM 3 SDK for possible error return values

**Related:**

* Section 17 on ECT
* Section 22 on Credit Redemption
* State events:
  + ECT\_AUTHORISED
  + ECT\_FROM\_CM
  + ECT\_FAILED
* qcom\_ectSubtractCredit()
* qcom\_ectTicketOutSubtractCredit()

### qcom\_ectSubtractCredit

This function can remove credit off the machine's credit meter provided certain conditions are met. See the conditions table below.

**Call format:**

qcom\_ectSubtractCredit(amount, reason, ectsubsn)

|  |  |  |
| --- | --- | --- |
| **Argument** | **type** | **Description** |
| *amount* | integer | This denotes the amount to be deducted from the machine’s credit meter. |
| *reason* | string | An arbitrary transaction description for short term reference / display purposes. The *reason* must be **24** characters or less, truncated as necessary by the machine on function entry. If the argument equals the empty string then the API will substitute it with the calling user’s username string. *Examples: “Account xfer”, “Account: 321324”, “Jackpot ID:12345678 lvl 1”, “Ticket-out”, “RCRF”.* |
| *ectsubsn* | integer | This value will equal the corresponding ECT authorisation serial number of the currently active authorisation via qcom\_ectSubtractCreditAuthorised() or the call will fail. |

The QLE Lua software driver will perform the following checks upon function entry (in order):

|  |  |
| --- | --- |
| **Condition** | **Return Values on failure** |
| The calling user must have a still active authorisation to perform the operation. Refer to the QCOM API function qcom\_ectSubtractCreditAuthorised(). | nil, “not authorised” |
| *ectsubsn* must equal the *ectsubsn* of the last thrown ECT\_AUTHORISED state event | nil, “invalid serial” |
| Amount <= the amount authorised in the corresponding call to qcom\_ectSubtractCreditAuthorised() | nil, “amount too large” |
| The machine must be in a System Lockup (s16), as well as a positive result from qcom\_egmOK(). | nil, “wrong state” |

If all the above conditions are met the QLE Lua software driver will (STH) message the machine; on receipt, the machine must immediately: cp:[[219]](#endnote-162)

If the amount in the STH message and machine state is ok:

* Update meters: Deduct the *amount* from the machine’s credit meter and add the *amount* to the machine’s Total Cashless Out Meter (**ectout**),
* Log the (ref s13.6.1) **ECT\_FROM\_EGM** event to NV memory.
* Log the **ECT\_FROM\_CM** state event.
* Log the **EVENT** state event for the **ECT\_FROM\_EGM** event.
* Instigate a short cash out sound that has a falling terminal (related: section 3.6).

If not ok:

* Log the **ECT\_FAILED** state event.

Return Values:

The function will return *true* on success otherwise, on fail the function will return *nil* plus one of the error strings listed in the above table.

**Related:**

qcom\_ectSubtractCreditAuthorised

Section 17 on ECT

Section 22 on Credit Redemption

### qcom\_ectSubtractCreditDeclined

### qcom\_ectTicketInAddCredit

In support of cash ticket-in and a part of the ticket-in sequence, this function is similar to the QCOM API function qcom\_ectAddCredit() in that it can also add credit to the machine’s credit meter.

To see the overall the ticket-in sequence refer to section 18.3.

There are two call formats for this function. The **first** call format is:

**Call format:**

qcom\_ ectTicketInAddCredit(camt)

|  |  |  |
| --- | --- | --- |
| **Argument** | **type** | **Description** |
| *camt* | Global type | Denotes the amount to be added to the machine’s credit meter. |

This call format when invoked by an authorised QCOM user must only add the given amount to the machine's credit meter and *ticketin* meter. The transfer must only take place if a ticket is currently in escrow. The QLE Lua software driver will only permit one transfer of credit via this function per ticket held in escrow. If the credit transfer takes place the ticket in escrow is stacked (retained by the machine).

The QCOM user invoking this function only has a limited time to invoke this function before the machine times-out and automatically ejects the ticket back to the player.

Before attempting to perform the operation, the QLE Lua software driver will apply all the sanity checks listed in the QCOM SDK and then check the following conditions in order shown aborting on the first failure:

|  |  |
| --- | --- |
| **Condition** | **Return Values on failure** |
| ECT has been enabled via the QCOM API *qcom\_ectEnable()* function | nil, “ect disabled” |
| A ticket is currently held in escrow. | nil, “no ticket in escrow” |
| The current return value of the QCOM API function qcom\_egmOK() would be *true*. | nil, “egm not ok” |
| The machine must not be in the “hopper collect” state. Refer: qcom\_egmState() | Nil, “hopper collect” |
| Confirm the amount is not zero or negative | nil, “amount is 0 or negative” |
| Confirm that the *amount* is less than user’s MAXECT limit set by the *qcom\_ectSetMaxECT* function | nil, “maxect” |
| Ensure the credit meter is capable of displaying the updated value. | nil, “amount too large” |
| The current return value of the QCOM API function qcom\_machineReady() would be *true*. | nil, “not ready” |

If all the above conditions are met, then the QLE Lua Software driver will queue a STH message and assume that there is no longer a ticket in escrow within the QCOM Lua Engine machine state. (This prevents duplicate cash ticket ECTs.)

Upon processing the STH message, the machine must start an attempt to stack the ticket. (NB the EGM could have had an issue since the sanity checks were undertaken so it is possible for the stack past the point of no return to still fail.)

Then if the stack is successful, or once there is no possibility that the machine will eject the ticket, the machine must:

* Attempt to add the amount to the machine’s Credit Meter and *ticketin* meter.
* If successful:
  + Log the (ref s13.6.1) **TICKET\_IN\_ECT** event to NV memory (denotes success).
  + Log the **TICKET\_IN** state event (denoting success)
  + Log the **EVENT** state event for the **TICKET\_IN\_ECT** event.
  + Instigate a short cash-in like sound that has a rising terminal (related:section 3.6)
* If not successful:
  + Log the **TICKET\_IN** state event (denoting fail)

Any residual amount not of a whole credit value must still be added to the machine’s credit meter and stored until it is either transferred off the machine, or rounded off (e.g. due to subsequent ECT transfers), or cancelled, etc.

If the whole rounded down credit meter value is zero and there is still residual credit, the EGM must update the credit meter display to display the residual credit amount in the units of the set currency (if not already being displayed).

**Return Value:**

The function will return *true* on success otherwise on fail, the function will return *nil* plus one of the error strings listed in the above table. Refer to the QCOM SDK for additional sanity checks and error return values.

There are two call formats for this function. The **second** call format is:

**Call format:**

qcom\_ectTicketInAddCredit()

This call format takes no arguments.

**Return Value:**

The function will return a table containing the details of the last successful ticket in ECT-to-CM for the calling user. The table’s schema will be as per the **TICKET\_IN** state event data table schema. If there is no ECT yet for the QCOM user, then this function’s call format will return *nil*.

Related:

* *ticketin* meter
* **TICKET\_IN\_ECT** event
* **TICKET\_IN** state event
* QCOM API function **qcom\_bnaRejectTicket()**
* Section 18 on TITO

### qcom\_ectTicketOutSubtractCredit

This function operates the same way as the QCOM API function qcom\_ectSubtractCredit() except for a few additional arguments which are needed for cash ticket printing and in what meters a successful call to the function updates and what events are logged.

If all the conditions for a sucessfult call to this function are met are met (refer qcom\_ectSubtractCredit() function) the QLE Lua software driver will (STH) message the machine; on receipt, the machine must immediately: cp:[[220]](#endnote-163)

If the amount in the STH message and machine state is ok:

* Update meters: Deduct the *amount* from the machine’s credit meter and add the *amount* to the machine’s **ticketout** and **ticketoutcount** meters,
* Log the (ref s13.6.1) **TICKET\_OUT\_PRINTING** event to NV memory.
* Log the **TICKET\_OUT\_PRINT\_START** state event.
* Log the **EVENT** state event for the **TICKET\_OUT\_PRINTING** event.
* Instigate a short cash out sound that has a falling terminal (related: section 3.6).

If not ok:

* Log the **ECT\_FAILED** state event.

Related:

* Section 22 on Credit Redemption
* Section 18 on TITO

## \_CancelCredit

### qcom\_cancelCredit

This function can remove credit off the machine's credit meter provided certain conditions are met. This function is similar in operation to qcom\_ectSubtractCredit(), the main difference is the meters which it updates are different.

**Call format:**

**qcom\_cancelCredit(amount, reason, transactionID)**

|  |  |  |
| --- | --- | --- |
| **Argument** | **type** | **Description** |
| *amount* | integer | denotes the amount to be deducted from the machine’s credit meter |
| *reason* | string | An arbitrary transaction description for short term reference / display purposes. *reason* must be 24 characters or less, truncated as necessary by the machine on function entry. If the argument equals the empty string then the API will substitute it with the calling user’s username string. *Examples: “Collect”,“Large Win”, “RCRF”.* |
| *transactionID* | string | An arbitrary transaction ID string, 8 characters or less, truncated as necessary by the machine on function entry |

The QLE Lua software driver will perform the following checks upon function entry:

|  |  |
| --- | --- |
| **Condition** | **Return Value on failure** |
| *transactionID* must not equal the last accepted *transactionID* for the calling user | nil, “duplicate transaction ID” |
| Amount <= Credit Meter Value. | nil, “amount invalid” |
| The machine must be in a System Lockup (s16) of the calling user as well as a positive result from qcom\_egmOK(). | nil, “wrong state” |
| Refer to the QLE LSD function implemention in module qcomapi.lua for other sanity checks and failure return values. | |

If all the above checks and conditions are met, then the machine must immediately upon processing the cancelCredit STH message (in order):

* Re-check the EGM’s state.
  + If not ok, then Log the **CANCEL\_CREDIT** state event denoting a failure.
  + If ok, then proceed below
* Update NV memory; all-or-nothing:
  + Deduct the *amount* from the machine’s credit meter,
  + Add the *amount* to the machine’s Total EGM Cancel Credit Meter and
  + Log the (ref s13.6.1) **CANCEL\_CREDIT** event to NV memory
* Once NV memory is irrevocably updated then…
  + Log the **CANCEL\_CREDIT** state event
  + Log the **EVENT** state event for the **CANCEL\_CREDIT** event

Related:

* Section 13.6.1
* Section 14.3 – State event timing.

**Return Value:**

The function will return *true* on success otherwise on fail, the function will return *nil* plus one of the error strings listed in the above table.

Also refer to section 22.

## \_System Lockup

### qcom\_slRequest

Queue or refresh a System Lockup on the machine for the calling user. Also refer to section 16 – “System Lockup”.

Precondition: if the QCOM user has invoked a slReset() for their system lockup, then calls to this function will fail (return an error) for that user until the next corresponding **SYSTEM\_LOCKUP\_CLEARED** state event.

Upon execution of this function and the argument passes sanity checks, the machine must enter the System Lockup condition if it is in idle mode and the machine is **ok**[[221]](#footnote-58) , otherwise if it is not already in a System Lockup it must **queue** the lockup request in NV memory and immediately enter the System Lockup upon next return to idle mode and the **ok** state. cp:[[222]](#endnote-164)

If the EGM was in either in the reserve feature (or power save mode) and a system lockup request is received, then the EGM must immediately automatically exit the reserve feature (or power save mode); i.e. log either **RESERVE\_EXIT** or **POWERSAVE\_EXIT** accordingly, then exit idle mode (**IDLEMODE\_EXIT**) and enter system lockup (**SYSTEM\_LOCKUP\_ENTRY**). cp:[[223]](#endnote-165)

If the machine is already in a System Lockup but not including a system lockup of the calling user, then the calling user’s system lockup must be added to the current set of system lockups.

If the machine is already in a System Lockup (including a system lockup of the calling user), then the parameters (as contained in the function argument) and display of that user’s system lockup must be replaced / refreshed with respect to the new parameters. Another **SYSTEM\_LOCKUP** state event must be thrown again for the user as well. cp:[[224]](#endnote-166)

While in a system lockup, if the machine becomes not ok58, then the SL display must not be reset, or updated (any new slRequests must be queued) or function (any SL related buttons must disable) until the machine becomes ok again.

None of the above scenarios must change the currently focused system lockup QCOM user. Only a human user at the machine must be able to do that. cp:[[225]](#endnote-167) (Refer section 16.1 for requirements and information on SL multi-user support.)

*Additional system lockup general requirements may be found at the end of this sub-section and in section 16.*

**Call format:**

**qcom\_slRequest(t)**

|  |  |  |
| --- | --- | --- |
| **Argument** | **Type** | **Description** |
| *t* | table | See below. The QLE Lua software driver will assume default values (see below) for any missing keys. Note, that table argument contents may be modified as a result of sanity checks performed. |

The argument of type table may have the following optional key-value pairs:

|  |  |  |  |
| --- | --- | --- | --- |
| **Key** (string)  *All key-pairs are optional* | **Value Type** | **Default**  (if nil) | **Max Length** |
| title | string | nil | title = username[[226]](#footnote-59) | 16 |
| subtitle | string | nil | nil | 32 |
| message | string | nil | nil | 160 |
| iconID | string | nil | nil | 256 |
| resetkeydisable | boolean | nil\* | nil | - |
| timeout | Integer | nil | - |
| yesbtn | boolean | nil | nil | - |
| nobtn | boolean | nil | nil | - |
| cancelbtn | boolean | nil | nil | - |
| okbtn | boolean | nil | nil | - |
| retrybtn | boolean | nil | nil | - |
| ignorebtn | boolean | nil | nil | - |
| abortbtn | boolean | nil | nil | - |
| attendantrequired | boolean | nil | nil | - |
| alertsound | boolean | nil | nil | - |
| fanfaresound | boolean | nil | nil | - |
| cashoutsound | boolean | nil | nil | - |

\**nil* means the value is missing/not provided and vice-versa. Also, for boolean values, the machine must consider *nil* to be the same as *false* logically speaking. It’s a Lua language facet.

The QLE Lua software driver will automatically truncate any *string* type parameter to its maximum length if the value is exceeded. Refer to the QCOM 3 SDK for other argument sanity checks and error return values.

Parameter descriptions:

**title**

This is the title of the system lockup to be displayed when active. This text must be the most prominent text out of the *subtitle* and *message* text data. The text must be centred.

**subtitle**

This is the subtitle of the system lockup to be displayed when active. The text must be centred.

**message**

This is the message body of the system lockup to be prominently displayed when the user’s particular system lockup is active. This text should be slightly less prominent than the *subtitle* text.

The machine must support a static text area of exactly **40 characters x 4 lines** for the display of the message. Each line display must be centred. The standard text display area means function callers (QCOM users) know exactly where the message line will wrap and the onus is on them to set messages that will display correctly in the space provided. cp:[[227]](#endnote-168)

**iconID**

A user must be able to have a custom picture displayed along with their system lockup. *Concept Only - do not implement iconID until further notice.*

**resetkeydisable**

If true, the machine must not allow the calling user’s SL to be cleared by any means other than the QCOM API system lockup reset function qcom\_slReset(). This must only affect the calling user’s SL and not affect machine key-switch operation for any other operation in the machine. cp:[[228]](#endnote-169)

If *false* or *nil*, then the machine must allow a QCOM user’s System Lockup to be cleared by either the qcom\_slReset() QCOM API function, or physical machine key-switch (provided the switch is not disabled (refer qcom\_egmKeySwitchDisable()).cp:[[229]](#endnote-170)

**timeout**

This parameter sets a view timer on the QCOM user’s system lockup in units of seconds. The machine must start the timer when the calling user’s SL is displayed to the player and continues to increment only while the players SL is displayed to the player and the machine is ok[[230]](#footnote-60). Once *timeout* is reached the machine must simply throw a **SYSTEM\_LOCKUP\_TIMEOUT** state event and do nothing else.

The view timer must be reset and restarted anytime a user’s active SL is refreshed via another slRequest. A machine restart must restart all system lockup view timers for those that have not have already expired. cp:[[231]](#endnote-171)

A timeout value of **nil** means no timer / abort the current timer if applicable.

A timeout value <= **zero** will be nil’ed by the QLE LSD.

*To ensure minimal resources are spent in the EGM in implementing these SL timers; it’s acceptable to implement using a timer based on a tick rate with a 1 second period, i.e., if a SL was displayed for less than 1 second, then there may be no tick in this period and the timeout value wouldn’t increment in this case. It intended SL timeouts should be able to be implemented for multiple QOCM user SL’s by using a single timer in the EGM.*

SL request button display flags

If a button flag is true (see table below), then during the user’s SL, the machine must (in addition to displaying other required SL text information), also display a corresponding player accessible button labelled as shown in the table below. cp:[[232]](#endnote-172)

|  |  |
| --- | --- |
| **SL Request Button Display Flag** | **Button text/label** |
| **yesbtn** | Yes |
| **nobtn** | No |
| **cancelbtn** | Cancel |
| **okbtn** | Ok |
| **retrybtn** | Retry |
| **ignorebtn** | Ignore |
| **abortbtn** | Abort |

The QLE LSD limits buttons to a maximum of **three** buttons and will return an error if more are requested.

If a player activates a SL displayed button then the machine must throw the **SYSTEM\_LOCKUP\_RESPONSE** state event indicating which button was chosen and then remove all \*btn flag related buttons from the applicable user’s SL and stay in the user’s SL. cp:[[233]](#endnote-173)

The SL may also still be reset via the permitted methods at any time without a response being given.

**attendantrequired**

This flag does nothing except that its value must be copied into the associated hook-able **SYSTEM\_LOCKUP** event. This allows users utilising system lockups to perform various functions requiring an attendant to also notify an attendant paging system. The idea is that the paging system will have a script hooked to the system lockup event and monitor this flag for possible attendant pages. Attendant paging system developers should also refer to section 22 for an alternative method.

Play Sound Flags

**alertsound**

If set, the machine must also emit a short alert (“ding”) like sound upon first display of the user’s SL lockup to the player. The volume level must be at the same priority as a fault condition on the machine. cp:[[234]](#endnote-174)

**fanfaresound**

If set, the machine must also emit a **short** (~2secs) (“tada”) like jackpot alert sound upon first display of the user’s SL lockup to the player. The volume level must be at the same priority as a fault condition on the machine. cp:[[235]](#endnote-175)

**cashoutsound**

If set, the machine must also emit a cash-out (“ka-ching”) like sound upon first display of the user’s SL lockup to the player. The volume level must be at the same priority as a fault condition on the machine. cp:[[236]](#endnote-176)

If more than one sound parameter/flag is present, then the machine must play the first it sees and ignore the others. cp:[[237]](#endnote-177) (NB A QCOM user can play multiple sounds in their system lockup by invoking another slRequest().)

Related: Standard sound proposal, refer section 3.6.

**Return Value:** *true | nil, string*

The function will return *true* on success; on fail, the function will return *nil* plus an error message of type *string*. Refer to the QCOM 3 SDK for sanity checks and error return values.

**Other System Lockup related requirements:**

As per QCOM v1, a machine in a SL must disable physical credit input (including accepting cash-in tickets into escrow), the collect button, betting and new game play for the duration of the lockup.

ECT in and out must be allowed during a SL. (FYI QCOM v1.x only allowed ECT in during a SL).

The machine’s credit meter must be clearly visible (and labelled) during a SL and updated in real time if for example, any hopper collects, ECT, Cancel Credit or Ticket out print events were to occur.

Audit mode and test mode must be accessible during a system lockup.

All required messages to be displayed by single System Lockup, must be displayed constantly (i.e. they must not be cycled on the display with other messages). To make this easier, it is acceptable for the system lockup display to overwrite the play outcome display area (e.g. game reels display), but it must not overwrite any meter displays or other text messages. Text messages must still be displayed during the SL with respect to the QCOM API functions: qcom\_egmSPAMA() & qcom\_egmSPAMB().

The QCOM SL state (including all queued SLs) must be stored in the machine NV memory and be automatically restored after any interruption (e.g. machine restart, fault condition, door open, audit mode, etc.).

The machine must not display “Call Attendant” or the similar message during a System Lockup unless the *attendantrequired*parameter above mandates it for a given user’s SL. cp:[[238]](#endnote-178)

In a low prominence position (using minimum permissible font size) the machine must display the text “**QCOMSL**” for the duration of all QCOM System Lockups. cp:[[239]](#endnote-179)

**Exiting the system lockup condition**

The machine must remain locked up in any given SL until reset by the qcom\_slReset() command or physical manual machine reset key-switch provided *resetkeydisable*is not set. Only once all SL’s for all QCOM users have been cleared (which must be cleared one at a time), the machine must **exit** from the System Lockup condition. cp:[[240]](#endnote-180)

Related: Refer to QCOM v1 documentation for example implementation screenshots of QCOM v1 compliant System Lockups, the aesthetics of which are applicable to QCOM 3 System Lockups.

Also refer to section 16.1 for requirements and information on SL multi-user support.

### qcom\_slReset

This command attempts to clear the calling user’s active[[241]](#footnote-61) System Lockup (SL).

**Call format:**

**qcom\_slReset()**

A QCOM user can only clear a SL that they created.

If the user does not have a SL active, then the user’s call to this function will do nothing.

If the machine is not ok[[242]](#footnote-62), then the call will do nothing. cp:[[243]](#endnote-181)

The reset command must not be queued in machine NV memory or similar.[[244]](#footnote-63) cp:[[245]](#endnote-182)

If a user has a SL active and the machine is ok62, then there must be *minimal reason* why the machine does not immediately remove the user’s SL. cp:[[246]](#endnote-183)

QCOM users should note that the “system lockup” state (wrt the return value of qcom\_egmState() ) will not change (i.e. the machine will not exit the SL state) if any other QCOM users have SL in progress. Refer System Lockup Multi-user support s16.1.

**Return Value:**

Returns *true* if the user has a system lockup to clear and the QLE state is qcom\_egmOK() at the time and the associated SendToHost message has been sent to the host machine for action. Note a success RV does not guarantee the lockup will be cleared, but in most cases it will; see footnotes. Upon any issue the return value will be *nil, errmsg* and the associated SendToHost message will not be sent to the host machine*.* Refer to the QCOM 3 SDK for all sanity checks and return values.

### qcom\_slStatus

This function returns the current status of the QCOM 3 system lockup feature. This function can return a *table* representing the set of QCOM users who have a SL either pending or active including full information about each SL.

**Call format:**

**qcom\_slStatus([username:string])**

**Return value:** *table | nil*

When the username argument is *nil*, the function will return an empty table if there are no SLs pending or active, otherwise a table of usernames will be returned (comprised only of user’s who have a SL active or pending) where each table value will denote the table argument from their last successful call to the slRequest() QCOM API function.

When the username argument is not *nil* and the username argument denotes a QCOM user who has a SL active or pending, then a table will be returned which is the argument from their last successful call to the slRequest() QCOM API function. Otherwise *nil* is returned.

If the human user has provided a response via the machines SL UI, then a response key / value will be included in the return value data.

For more information refer to the QCOM 3 SDK.

Related: section 16 – “System Lockup”.

## \_Play Control

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_playSetPEF

This function controls the machine’s overall Play Enabled Flag (PEF).

The QLE Lua software driver will maintain:

* a PEF per QCOM user and
* one overall PEF which will be equal to the logical AND of each resident QCOM user's individual PEF.

*This function replaces what was SEF & MEF functionality in QCOM v1.x.*

If the machine’s overall PEF is equal to ***false*** then physical credit input devices (including cash-in ticket acceptance into escrow), the bet adjustment UI and new play commencement on the machine must all be disabled. cp:[[247]](#endnote-184) Collect operations must still be available. cp:[[248]](#endnote-185)

If the machine was already disabled for the calling user, then the machine must simply refresh the calling user’s reason for disable string and update its display if necessary, to reflect any string change.

As the QCOM API function operates on its overall Play Enabled Flag; nothing excluding a RAM error condition on the machine, must prevent the machine from processing a playSetPEF() function call. cp:[[249]](#endnote-186)

**Call format:**

**qcom\_playSetPEF(boolean [, reason:string [, username]])**

**Arguments:**

**boolean** : new value for the calling user’s PEF.

**string** : reason for disable : for length requirements, refer to the QCOM 3 SDK or the *playSetPEF* SendToHost message data format in the QCOM 3 summary spreadsheet. This argument is only applicable if the boolean argument is false. If the string argument missing, an empty string, or invalid then the QCOM user’s username will be substituted by the QLE LSD. QCOM users should use reason strings that are indicative as to which user is responsible e.g. “system offline” should not be used,

**username** : only the QMA may include the username argument.

**Return value:** none

**UI/Display Requirements**

When the overall machine PEF is equal to false, all QCOM users whose individual PEF flag is also false must have their *reason* string displayed in full by the EGM during idle mode where egmState() returns the two values *“idle mode”, nil.* cp:[[250]](#endnote-187) The PEF play disabled display must be updated in near real time as possible\* in a window with the title **“Play Disabled”**. cp:[[251]](#endnote-188) The play disabled display window must be displayed statically. cp:[[252]](#endnote-189)

Out of all the idle mode sub states, the display of the play disable condition must be the lowest priority. Refer QCOM API function **qcom.egmState()** section 11.19.20. cp:[[253]](#endnote-190) This means that for example the following EGM idle mode functionality: help and info, PID’s, power save, a reserved feature, rules display and PAEL, must all still be available while the EGM is play disabled. The play disabled display must be temporarily removed while any other EGM idle mode functionality is accessed.

\*It is accepted that some EGMs have a limit on the maximum UI refresh rate. This is acceptable provided the refresh rate period does not exceed **one** second. cp:[[254]](#endnote-191)

The EGM must also incorporate any currenlty active **qcom.playOKex()** disable reasons into this ‘play disable’ display. cp:[[255]](#endnote-192)

Acceptance of the overall legibility of the play disabled display is at the discretion of the CEO OLGR. cp:[[256]](#endnote-193)

Related:

* PLAY\_DISABLED, PLAY\_ENABLED state events
* PEF global type.
* Section 16 – “System Lockup”.
* Section 16.3 - “System Lockup vs Play Disable”.
* Maximum number of QCOM users: section 5.
* **qcom\_egmCreditInputDisable()**
* **qcom\_playOKex()**

Note that both this function and **qcom\_egmCreditInputDisable()** can disable physical credit input on the machine. This means that both the PEF and the CIEF (credit in enabled flag) must be enabled before the host machine may allow physical credit in. cp:[[257]](#endnote-194)

### qcom\_playPEF

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_playOKex

This function provides information on the status of any manufacturer specific prerequisite conditions for play the EGM may have.

An EGM may optionally have several manufacturer specific prerequisite conditions that when conditions are not satisfied, temporarily disble the EGM in the same way as a PEF = false condition disables the EGM; refer playSetPEF() above. cp:[[258]](#endnote-195)

This function may be used at the discretion of the machine manufacturer with prior approval from the OLGR cp:[[259]](#endnote-196) (for the purpose of standardising ‘*reason’* strings).

**Call format:**

|  |
| --- |
| **qcom\_playOKex(giveReasons)** |

**Arguments:**  *giveReasons* : boolean  
  
If the *giveReasons* argument is *true,* there will be an additional table in the return value which denotes the set of manufacturer specific prerequisites and their status before the EGM can be made playable.

**Return Value:** boolean[, table]

The second return value above of type *table* will only be present if the *giveReasons* argument equals true and the EGM has one or reasons / conditions to provide. Otherwise the second return value will be *nil*.

The table return value will be a single level Boolean value table indexed by keys of type *string* where each string denotes a manufacturer specific prerequisite required before the EGM can be fully operational. It will have the following schema:

{condition:string = true | false, ...}

The 'condition' string above must be in plain English. A corresponding value of *true* means the condition is satisfied and *false* means the condition is not satisfied.

The first return value will be the logical **'AND'** of all the boolean values in above table in the EGM. If the EGM has no prerequisite conditions the first return value must be *true*.  
      
Example return values and conditions:

false, {

"This EGM requires 5 installed games" = true,

"All games must be enabled" = true,

"Games must all be set to the same variation" = false,

"Required hardware: wheel of awesome" = true

}

true, {

"This EGM requires 5 installed games" = true,

"All games must be enabled" = true,

"Games must all be set to the same variation" = true,

"Required hardware: wheel of awesome" = true

}

Any game specific conditions must also be stated in qcom\_gameGetp().’notes’ field. cp:[[260]](#endnote-197)

Related:

* The **PLAY\_OK\_EX** state event which indicates an update to this function’s return value has occurred.

## \_PID

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_pidEnable

### qcom\_pidEnabled

### qcom\_pidList

## \_Custom User Interface

### qcom\_infoAddMenuItem

**Concept only; do not implement this function until further notice.**

This function appends a new menu item to the machine's "i" button menu with the title denoted by the first string argument.

**Call format:**

**qcom\_infoAddMenuItem(caption, callback)**

|  |  |  |
| --- | --- | --- |
| **Argument** | **type** | **Description** |
| *caption* | string | “i” button menu item string to display.  If string is nil then this must remove any previously set menu item for the given user. Each QCOM user may only append one menu item. |
| *callback* | function | This is a call-back function which the machine invokes anytime the player/user enters input. Any user input is provided as function arguments. |

Each QCOM user may only append **one** menu item max to the machine’s “i” button menu. If two different QCOM users use the same menu item caption then the machine must implement both as separate menu items.

Menu items added by QCOM users must not persist across machine restarts and power fails. If the QCOM user is deleted or ‘quarantined’ (refer 5.8) then their respective menu items must be automatically removed by the machine.

Once setup via this function, any subsequent player/user activation of the “i” button menu items must cause the machine to throw the **IDLEMODE\_INFO\_ENTRY** state event and then launch a UI display (see requirements below) on behalf of the owning QCOM user. The owning QCOM user may then control content via other qcom\_info class QCOM API functions.

**Return Value:** *true* on success, *nil* on fail

QCOM User UI Display Requirements

The UI display must be exactly **80x25 characters in size** using a **non-proportional** character font display methodology. The display must be **prominent** and must cover more than **50%** of the hosting display device’s display area.

The machine must clear the display on launch.

The machine must implement a close button / methodology that can be activated at any time by the player via a suitable and intuitive action. When activated, the machine must throw the **IDLEMODE\_INFO\_EXIT** state event.

The machine must provide the player with an interface which allows navigation and activation of any QCOM user added “buttons”. Refer QCOM API function

An idle timeout must be implemented as per section 3.7.

Related:

* “i” button requirements (3.7)
* Refer 10.12.6 for more information and possible applications as well as PID support (section 10.12.5).
* QCOM state events
  + **IDLEMODE\_INFO\_ENTRY**
  + **IDLEMODE\_INFO\_EXIT**.

Todo: handle caption collision

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_infoUIclrscr

**Concept only; do not implement this function until further notice.**

### qcom\_infoUIgetnum

**Concept only; do not implement this function until further notice.**

### qcom\_infoUIgetstring

**Concept only; do not implement this function until further notice.**

### qcom\_infoUIhotspot

**Concept only; do not implement this function until further notice.**

### qcom\_infoUIprint

**Concept only; do not implement this function until further notice.**

## \_Health

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

All functions in this class excluding than qcom\_healthFreeMem() are optional to implement.

### qcom\_healthFreeMem

### qcom\_healthTemperatures

### qcom\_healthVoltages

### qcom\_healthFanSpeeds

### qcom\_healthDrives

## \_User Maintenance

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_userCreate

### qcom\_userDelete

This function can delete a QCOM user from the machine, if invoked by an authorised user (typically it is envisaged that this will be the QMA). A QCOM user cannot self delete via this function.

**Call format:**

qcom\_userDelete(username)

|  |  |  |
| --- | --- | --- |
| **Argument** | **type** | **Description** |
| username | Global type | Denotes the QCOM user being deleted.  Refer to the QCOM 3 SDK for this argument’s sanity checks. |

If the username is a current QCOM user on the machine, then their account on the machine and any associated data must be deleted unless stated otherwise.

The QCOM Lua Software will ensure:

* The user is gracefully shutdown.
* It is no longer possible for the QCOM user to log into the machine (23) using their login credentials
* Any user commands added to the QCOM Command Interpreter will be removed.
* The user is deleted in the QCOM 3 Lua state.
* The “**userDelete**” message is logged. Refer QCOM 3 Summary spreadsheet: SendToHost worksheet for required machine actions here.

**Return Value:**

Returns true on success otherwise on error, the function will return *nil* plus an error message string. Refer to the QCOM 3 SDK for possible error messages.

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_userDeleteSelf

### qcom\_userQuarantine

### qcom\_userIsQuarantined

### qcom\_userSetUAApublicKey

Refer section 23.

### qcom\_userSetMyUAApublicKey

Refer section 23.

### qcom\_userUAApublicKey

Refer section 23.

### qcom\_userEnable

### qcom\_userDisable

### qcom\_userEnabled

### qcom\_userSetDiskQuota

Refer section 5.3.

### qcom\_userDiskStats

### qcom\_userSetMemoryQuota

Sets the maximum amount of machine memory the given user may utilise with respect to its scripts and execution thereof.

If the user's scripts exceeds this limit, then the QCOM user will be quarantined by the QLE Lua software driver.

**Call format:**

qcom\_userSetMemoryQuota(username, B)

|  |  |  |
| --- | --- | --- |
| **Argument** | **type** | **Description** |
| username | Global type | Denotes the QCOM user being created. |
| B | integer | Memory limit for user in bytes.  This value must be saved by the machine in NV memory. |

**Return Value:** true | nil, string

Returns true on success otherwise on error, the function will return *nil* plus an error message string. Refer to the QCOM 3 SDK for possible error messages.

**Related:** Section 10.2.1.

### qcom\_userMemoryStats

This function retrieves memory information and stats for all QCOM users.

**Call format:**

qcom\_userMemoryStats()

**Return Value:** table

The return value will be of type *table* where the keys are of type *username* (refer table of global types) and each sub-table adheres to the following minimum schema:

{

-- each value below scope is wrt the encompassing QCOM user.

memlimit : *integer*, -- as set by qcom\_userSetMemoryQuota()

memused : *integer* -- current memory used

-- NB machines may return more fields than shown above

}[username]

Where:

* Values denoting an amount of memory are in units of bytes.
* Values only have to be current up to the time the encompassing script started execution as a result of the triggering QCOM state event.

Related: s11.15.1

### qcom\_userSetFullName

**Concept only; do not implement this function until further notice.**

### qcom\_userFullName

**Concept only; do not implement this function until further notice.**

### qcom\_userSetPrivilege

This function sets or resets whether a given QCOM user is privileged to invoke a specific QCOM API function, or QCOM Command Interface function.

The QCOM API privilege status for each resident QCOM user in the machine, must be stored by the machine in NV memory. QCOM user API privileges must be automatically restored every machine restart.

**Call format:**

qcom\_userSetPrivilege(username, funcs, b)

|  |  |  |
| --- | --- | --- |
| **Argument** | **type** | **Description** |
| username: string *| “anon”* | Global type | Denotes the QCOM user being created. |
| *funcs* | table | string | “all” | May be either a string, or table of string, or a boolean table; that denotes a set of QCOM API and QCI function names being granted or removed privilege with respect to *username* and *boolean* arguments e.g. {“egmSPAMA”, ”machineQueueReboot”[, …]}, or  {egmSPAMA = true, machineQueueReboot =true[,…] }, or “all”, or “egmSPAMA”. |
| *b* | boolean | A value of *true* grants the user privilege to the functions denoted by the second argument, false revokes it. |

Revoking a privilege to call a function for a user must only take effect on next user / machine restart. Generally it should be rare for the QCOM user to lose a privilege once it is given.

**Return Value:** true | nil, string

Returns true on success otherwise on error, the function will return *nil* plus an error message string. Refer to the QCOM 3 SDK for possible error messages.

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_userPrivileges

### qcom\_userSetReadyState

### qcom\_userWhoAmI

### qcom\_userWho

### qcom\_userLoggedOn

### qcom\_userList

### qcom\_userSetSAAcert

### qcom\_userSAAcert

### qcom\_userLoadScripts

### qcom\_userSetScripts

### qcom\_userScriptHashes

### qcom\_userRestart

This function restarts the calling or denoted QCOM user. Only the QMA may supply a username argument allowing it to restart a specific QCOM user. The function returns control to the calling script and allows that script to finish before the restart of the user occurs. NOTE QCOM user memory use is temporaily doubled when restarting an already running QCOM user.

Typically, this function allows a QCOM user to automatically restart themselves after a successful call to the QCOM API function **qcom\_userLoadScripts()** or the QCI *userloadscripts* command. Refer section 24.2.

Note QCOM users that utilise TCP, UDP, or UART resources may\* need time to shutdown those resources before they can be restarted successfully. For these users, it may\* better to shutdown these users first (via the *shutdownuser* QCI command), wait a few seconds, and only then restart the user. \*This can be avoided if the QCOM user is specifically programmed to wait a few seconds after starting up before trying to acquire these types of resources.

A restart is comprised of a user shutdown (if not already shutdown), then a deletion and recreation the QCOM user's jail, followed by an script package extraction, load, compilation and finally execution the QCOM user’s latest *qinit.lua*.

**Call format:**

qcom\_userRestart([username])

Related:

* QCI commands *restartuser & userloadscripts*. Refer section 24.2.
* **USER\_RESTART** state event

**Memory use warning: this function causes a temporary spike in memory use for the target user (roughly x2) when performed on a QCOM user that has already been running. The QMA (or delegated authority) must not privilege this function to any QCOM user unless multiple user restarts (without machine reboot) has been tested and the necessary memory requirements are established.**

FYI Technical notes re the above warning. (This is not mandatory reading):

* When you first start a QCOM user, it starts up and its memory use will quickly reach its average typical operating value.
* In the QLE LSD, i.e. in Lua, memory used is allocated instantly. However freeing memory is unpredictable in time. This is because Lua, like Java, and similar languages that manage memory on behalf of the programmer, use a complex memory garbage collector that uses an algorithm to free orphaned memory objects over time. As a result, recovering no-longer-required memory is not instantaneous, but is somewhat unpredictable.
* When a QCOM user that has already been running is restarted via this function, their entire jail (aka memory objects used) is cleaned out and replaced. However, this orphaned memory is freed over the next few seconds by the garbage collector (the time this takes depends on how busy the QLE LSD and the Lua garbage collector is, and how much memory is currently in use and how much is in an orphaned state).
* This means there will be a short period following a user restart, in the case the user was already running, where the user’s memory use roughly doubles, then settles back down to its average typical operating value again.
* Out of all QCOM API functions, this is the only function that has an unexpected memory hit.

### qcom\_userShutdown

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_userIsShutdown

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_userAnonSetPass

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

Related: section 5.1.

### qcom\_userAnonPass

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_userCPUstats

This function returns an approximate amount of total CPU usage for all resident QCOM users.

**Call format:**

qcom\_userCPUstats()

This function requires no arguments.

**Return Value:**

The return value will be a table of table where the indices are usernames (e.g. t[username]). Each value will be of type *table* comprising of the following key, values:

|  |  |
| --- | --- |
| **Key** | **Value** |
| cpulimit | Type *userCPUquota*. As set by the QCOM API function qcom\_userSetCPUquota(). |
| cputotal | Type *float*, units are milliseconds*.* The total CPU time spent executing the respective user’s scripts. |
| cpupeak | Type *float*, units are milliseconds*.* The time of the longest running state event handler script seen by the machine since last power up. |

The minimum precision of the above values must be within a microsecond.

Related: CPU Usage, Monitoring and Control s10.2.3.

### qcom\_userSetCPUquota

Sets the maximum amount of time that any single script handler for the given user (as triggered by a QCOM state event) may execute for

If the handler execution time exceeds this limit, then the QCOM user will be quarantined by the QLE Lua software driver.

**Call format:**

qcom\_userSetCPUquota(username, limit)

|  |  |  |
| --- | --- | --- |
| **Argument** | **type** | **Description** |
| *username* | Global type | Denotes the QCOM user being created. |
| limit | integer | CPU limit for user in units of milliseconds.  This value must be saved by the machine in NV memory. The QLE Lua software driver will message the machine each time this function is called.  *Note that interruptions during execution of a QCOM user script (e.g. by another process) will typically also be added to the user’s CPU quota. This will be taken into account by the QCOM users with privilege to this function. For more information refer to section 10.2.3.* |

**Return Value:** true | nil, string

Returns true on success otherwise on error, the function will return *nil* plus an error message string. Refer to the QCOM 3 SDK for possible error messages.

**Related:** Section 10.2.3.

### qcom\_userInstructionStats

This function returns the Lua instruction quotas for all resident QCOM users.

**Call format:**

qcom\_userInstructionInfo()

This function has no arguments.

**Return Value:**

The return value will be a table of table where the indices are usernames (e.g. t[username]). Each value will be of type *table* comprising of the following key, values:

|  |  |
| --- | --- |
| **Key** | **Value** |
| quota | Type *integer*. Instruction quota/limit as set by the QCOM API function qcom\_userSetInstrucitonQuota() for the respective QCOM user. |

Related: User Lua Instruction Quota (s5.5)

### qcom\_userSetInstructionQuota

Sets the maximum amount of Lua instructions that any single script for the given user may execute before returning.

If the script exceeds this limit, then the QCOM user will be immediately quarantined by the QLE Lua software driver (s5.8).

Related: User Lua Instruction Quota (s5.5).

**Call format:**

qcom\_userSetInstructionQuota(username, limit)

|  |  |  |
| --- | --- | --- |
| **Argument** | **type** | **Description** |
| username | Global type | Denotes the QCOM user. |
| limit | integer | This value denotes the maximum number of Lua instructions that may be executed per user script as executed as a QCOM State Event handler.  The QLE Lua software driver will reject any value below **50** and above **5000**. (NB A typical value for production machines will be close to the minimum and depends on the role the QCOM user is performing).  Refer to the QCOM 3 SDK for other applied sanity checks for this argument. This value must be saved by the machine in NV memory. |

For more information on how to apply *limit*, refer User Lua Instruction Quota (s5.5).

**Return Value:** true | nil, string

Returns true on success.

On error, the function will return *nil* plus an error message string. Refer to the QCOM 3 SDK for possible error messages.

## \_RemoteControl

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_rcSetAutoPlay

### qcom\_rcAutoPlay

### qcom\_rcButtonState

**Concept only; do not implement this function until further notice.**

### qcom\_rcButtonPress

**Concept only; do not implement this function until further notice.**

### qcom\_rcPlay

**Concept only; do not implement this function until further notice.**

### qcom\_rcRandomResponse

**Concept only; do not implement this function until further notice.**

### qcom\_rcResetFault

### qcom\_rcResetLockup

### qcom\_rcResetKey

### qcom\_rcGoFault

### qcom\_rcCollectPress

## \_Banknote Acceptor Maintenance

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_bnaGetp

### qcom\_bnaSetEnableFlag

### qcom\_bnaFirmwareID

### qcom\_bnaFirmwareUpgrade

Refer to section 21.2.

### qcom\_bnaNoteStatus

### qcom\_bnaSetNoteStatus

### qcom\_bnaMetersByDenom

### qcom\_bnaRejectTicket

Section 18 on TITO.

## \_Ticket Printer Maintenance

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_tpGetp

### qcom\_tpFirmwareID

### qcom\_tpFirmwareIDUpgrade

Refer to section 21.2.

### qcom\_tpMeters

## \_Coin Acceptor Maintenance

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_caGetp

### qcom\_caSetEnableFlag

### qcom\_caFirmwareID

### qcom\_caFirmwareIDUpgrade

Refer to section 21.2.

### qcom\_caSetDiverterAutoFlag

### qcom\_caDenom

### qcom\_caSetDenom

## \_Hopper Maintenance

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_hopperGetp

### qcom\_hopperSetDefaultRefillAmount

### qcom\_hopperRecordRefill

### qcom\_hopperDenom

### qcom\_hopperSetDenom

### qcom\_hopperPayout

## \_uart

The QCOM 3 UART API provides QCOM users with access to serial ports on the machine. Serial port support has been added to QCOM 3 in order to aid with the migration of existing legacy protocol machines to QCOM 3 and Ethernet.

QCOM 3 machines must provide access via this API class to a minimum of **two** serial ports that are externally accessible to its secured processor cabinet. cp:[[261]](#endnote-198)

The QCOM 3 UART API defined in this section is able to be implemented using the native Windows OS serial port API. There is no requirement or need to alter or design the existing Windows serial port device driver in order to implement QCOM 3 UART support.

For Linux OS based machines, the native Linux serial port programming API (as currently documented) does not appear to have a fast enough response time and therefore a substitute Linux serial port API or possibly modified device driver may be required in order to implement QCOM 3 UART support. However Linux is highly tweakable and it wouldn’t be surprising if the response time of the native Linux serial port API was able to be improved.

Basic UART object returned by a call to **qcom.uartOpen()**:

**UART Object:**

Member functions:

* setState()
* getState()
* write()
* close()
* free()

\*rx thread

\*rx thread

Tasks:

* Waits on rx-data based on timeouts
* Generates **QLE\_UART\_RX** state events which are queued and passed to the QLE thread, and ultimately to the user’s set rx call-back function.
* Exits on free(); self deleting

The rx thread must be based on two timeouts: cp:[[262]](#endnote-199)

* An overall internal thread loop cycle timeout set at **500msecs** (defines the worst response time to a call to the UART close member function)
* A *read interval timeout* (rit) set by the owning QCOM user via the **setState()** UART object member function.

Because the thread waits on rx-data using timeouts, the above arrangement results in fairly average/volatile response times wrt poll – response protocols e.g. **2…30+ msecs** depending on the machine h/w and OS. These kinds of response times are not good enough for some legacy protocols (such as QCOM v1), therefore QCOM 3’s UART API also has special support for protocols requiring fast response times and programmed parity.

Accordingly, the remainder of this section describes two special features added to QCOM 3’s UART support / API which provides reliable fast response times (always under **5 msecs**) and programable parity bit packet framing.

QCOM 3 UART support special features are:

* Low level UART Lua script for packet framing (esp. parity bit packet framing)
* Queueable Tx message packets that can be sent on demand by the above script via a qwrite() UART API member function.

**Fast response times via tiny embedded Lua interpreter in low level I/O thread**

If there is a requirement[[263]](#footnote-64) for a reliable fast response time on the serial port object (e.g. QCOM v1.x requires a 5msec poll response time and a 0.5729 msecs byte-byte max time), then the QCOM user will try to achieve this by supplying a small Lua script in the call to **qcom.uartOpen()**. This script, used for UART message framing, must be executed inside the rx thread and the machine must alter the thread to wait on a specific number of bytes read (the default being 1 byte at a time) in addition to the timeouts. cp:[[264]](#endnote-200) The result is that the response time is faster on average and less volatile. A response time ~ 2-3msecs on average is typical. The reason for this improvement is that the UART thread is no longer just waiting on timers for rx bytes; but a specific number of bytes received for which the response time is far quicker.

Note: The above response time improvement will generally only occur in machines where the UART rx threshold is attached to a CPU interrupt or similar arrangement.

Refer to the QCOM SDK / qcom v1.6 protocol app (qcom16) for an example QCOM user that uses a UART message framing script in the UART object.

More information on this technique and requirements follow.

**QCOM 3 UART API may be script enabled.**

The QCOM 3 UART API is about what you would expect from a UART API except for one useful feature which gives the QCOM user the option of supplying a small Lua script for use within a stand-alone tiny Lua state instance inside each UART object’s read thread.

The script's purpose is to perform any desired low-level serial port IO for the UART object and **permits fast response times** for the implementation of arbitrary UART based protocols (such as QCOM v1.6.x) by:

* Allowing serial messages to be received and responded to within the same thread.
* Avoiding relying on read timeouts (with are fairly volatile).
* Allowing OS UART read operations to return on an exact number of bytes. (The response time for which is far quicker than using timeouts).
* Finally the script also drastically reduces the number of **QLE\_UART\_RX** state events in the QLE because the script can hold bytes and collate message packets.

When a UART object is scripted enabled by the QCOM user then the UART object returned by a call to **qcom.uartOpen()**: becomes:

**UART Object:**

Member functions:

* setState()
* getState()
* write()
* close()
* free()
* **qwrite()**
* **qcancel()**
* **poke()**

\*rx thread

\*rx thread

Thread as before waits on rx data based on timeouts **but also reads on an exact number of bytes (default = 1).**

Additional tasks in addition to basic tasks:

* Tiny stand-alone Lua state instance
  + QCOM user supplied script
* Every rx byte is passed to the script’s LPB() function and the return value tells the thread what to do next.

**An implementation of required QCOM 3 UART support is provided in the QCOM 3 SDK.**

Host machine requirements:

For each UART object for which the QCOM user also provides a script via the QCOM 3 API function qcom.uartOpen(), the host machine must also create a tiny Lua state instance within the uart object’s thread as per the requirements below and load the user’s script into it.

* The Lua state must have absolutely no libs loaded into it. (Without any libs, the initial Lua state will be less than 3 kilobytes in size and will not perform any syscalls)
* Compile and load the QCOM user supplied script into the state.
* Perform a single full garbage collection then disable garbage collection in the Lua state.
* Provide a single C Lua function to the Lua state with the following Lua function call format: **send(msg:string)**. This function is used by the QCOM user’s UART script to send sideband messages back to its rx call-back function in the QLE for special events like low level CRC errors, packet framing errors and UART LSR register events such as UART overrun errors. This function when invoked by a QCOM user script within the UART object’s tiny Lua instance, the machine must send the provided string message argument to the **QLE UART** dispatch receiver function **dispatchUARTevent()** in qle.lua via a **QLE\_UART\_RX** state event with the length property equal to **-1**. Refer to the QCOM 3 SDK example module *qlua232thread.cpp* for example implementation code. The code of interest is associated with the C function ***sendSideBandMsgToQLE()***. Related: **QLE\_UART\_RX** state event description in the QCOM 3 summary spreadsheet.

A Lua state created as per the above, will be a pure software VM and will not perform any system calls.

The resulting Lua instance and script will be tiny (~2-3kB), fast and could potentially be run under interrupt or by low level comms IO driver in the host machine.

The script’s LBP() function (see below) which the machine must call with all UART rx data as an argument (see below) must be executed by the host machine as close to the hardware layer as possible

Once the com port object is able to RX, the host machine must:

* Call the script's LBP() function for each received byte on the associated serial port with the required arguments.
* Process the LBP() script's return value as per below.
* The host machine must execute the script's LBP() function as close to the hardware layer as possible.
* Periodically check the Lua state's memory use and if greater then **50kB** then close the com object.

QCOM user requirements pertaining to any supplied low level comms IO script.

The script cannot (as in the QLE Lua software driver and host machine will constrain the script to the below):

* Access any Lua libraries whatsoever (not even the Lua concatenate (‘..’) operator will be available).
* Have any loops or goto's. (The QLE Lua software driver will ensure the relevant keywords cannot be present even as a part of a comment, string or keyname) \*
* Have any recursion. \*
* Have exactly **one** function declaration and that function must be called 'LBP'. See below for more information on the LBP function \*
* Be more than **7kB** in size (wrt the original script string - including comments).\* Related qcomapi.lua : qapi\_uartscrlimit
* Once loaded, the script cannot use new Lua state memory over time as there is no garbage collection. (The host machine must auto close the port object if script memory usage reaches the limit mentioned in the previous list above)

(\* The QLE Lua software driver will apply these constraints to the user’s script. See qcomapi.lua : api\_uartScriptSanitise(s) )

**LBP() function**

For a QCOM user UART script to function, it must contain a single function called **LBP()** which must have the following call format:

function LBP(b : integer : RS232 steam byte[, lsr : integer : UART Line Status Register])

Where:

b: rx byte : integer.

lsr: integer.

If the com port object is set to use serial LSR character insertion and parity checking then the host machine must indicate if a parity error has occurred on the byte denoted by the first argument by setting bit 4 of lsr argument.

The host machine may optionally indicate other error supported by the UART’s LSR if desired.

(FYI The LSR argument is as per a legacy 16550 UART hardware Line Status Register)

As required above, the machine will push every received UART byte through this function allowing the QCOM user to formate rx bytes into message packets to be then sent onto the the QLE and received via their UART rx call-back function. This drastically lowers the number of QLE state events associated with UART support by allowing UART bytes to be formated into meaningful packets by the QCOM user’s LBP script first.

Return Value:

btr : integer [, rvc1 : integer [, rvc2 : integer [, rvc3 : integer [, rvc4 : integer ] ] ] ]

Where:

btr : integer: number of bytes the machine must try to read next.

Before another system read of the port takes place, the machine must finish pushing all rx bytes to date through LBP(); thus a btr return value may be overwritten by any subsequent calls to LBP() and never used.

[rvcX] : integer : max 4 :

Commands what the UART objects RX thread must do the rx byte.

The host machine must process the commands in order from left to right, where:

0 = means do nothing (default).

1 = means hold the rx byte.

2 = means send all currently held rx bytes to the QLE / qcom user's RX callback func. for assoc. uart object. Then also perform a 3 below.

3 = means clear all currently held bytes.

4 = means transmit via the serial port a previously queued write (if any). See **qwrite()** UART object member function.

Related:

* For a working example QCOM UART API implementation that QSIM 3 also uses, refer to the 232 related ‘cpp’ modules in the QCOM 3 SDK.

**QCOM 3 API UART class of functions:**

### qcom\_uartSetp

This function assigns limits and parameters pertaining to UART communications for a given QCOM user.

**Call format:**

**qcom\_uartSetp(table)**

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| interface | int | Denotes the serial port to update settings for  table: 1…nports (min 2). The value of nports is set by the machine and readable via uartGetp(). |
| username | Global Type | QCOM user username priviledged to open this serial port. |
| maxcbrate | int | **default = 1, min = 1, max = 256**. Sets the maximum number of uart rx call-backs per second for the given serial port via the **QLE\_UART\_RX** state event. Related: the UART object’s rxcallback function parameter defined in section 11.36.3. |

### qcom\_uartGetp

Returns the values as set by the qcom\_uartSetp() function for all serial ports in the machine for use by the QLE.

**Call format:**

**qcom\_uartGetp()**

**Return Value:** *table | nil, string.* Refer to the QCOM SDK for failure error messages.

Refer to the QCOM summary xlsx for the returned table schema.

### qcom\_uartOpen

Creates and returns a new serial object for a serial port.

**Call format:**

qcom\_uartOpen(table)

|  |  |  |
| --- | --- | --- |
| **table argument** | | |
| **key** | **value** | **Description** |
| interface | integer | 1…(typ 2)  This denotes which serial port interface in the machine to open. |
| script | string | Optional. See below |
| Low level communications pre-processor script.  The string must denote a Lua script for use as low level communications pre-processor. Refer to section 11.36 above for more information and requirements.    For an example pre-processor script, refer to the QCOM 3 SDK and qcom16 app (file: q16\_232.lua) that is provided with QSM 3. This is the pre-processor script used for the implementation of the QCOM 3 - QCOM v1.6 Protocol app. | | |
|  |  |  |

**Return Value:** table | nil, string

On failure the function will return *nil, string* where the string must denote a nature of the failure e.g. “*no ports available*” (this message text may vary) in the event more ports than the machine supports are attempted to open.

On success the function will return a table library comprised of the following functions:

* **setState()**
* **getState()**
* **write()**
* **close()**
* **free()** *– private; no QCOM user access*

The member functions below require a script enabled UART object to work:

* **qwrite()**
* **qcancel()**
* **poke()**

See following sub-sections for each member function’s functional description.

Note: UART reads operations are facilitated via a **call back** function set via the setstate() member function. The host machine must dispatch these to the QLE Lua software driver via the dispatch function *dispatchUARTevent()* with a **QLE\_UART\_RX** message event ID. cp:[[265]](#endnote-201) Refer to the QCOM 3 summary spreadsheet for event specifics.

The machine must not start sending through serial port rx-data state events for a UART class object until after the first call to the setState() member function (see below).

The required default settings for a newly opened serial port must be: cp:[[266]](#endnote-202)

* No flow control.
* Binary mode.
* 8 bits, 1 start bit and 1 stop bit.
* No parity
* Ignore breaks
* No translation
* Read / write buffer sizes of **16kB** each.
* DTR = enabled
* RTS = enabled
* DSR sensitivity = false

### UART object member functions

qcom.uartOpen() returned UART object member functions

* **setState()**
* **getState()**
* **write()**
* **qwrite()**\*
* **qcancel()**\*
* **poke()**\*
* **close()**
* **free()** - private

\*this member functions are only present when a script is associated with the UART object.

* + - 1. **setState()**

**Call format:**

setState(table)

|  |  |  |
| --- | --- | --- |
| **table argument** | | |
| **key** | **value** | **Description** |
| baudrate | integer | The desired baud rate in units of bps.  Min 300, Max 115000. |
| checkparity | bool | If true then parity checking must be performed. If enabled, then parity errors (and optionally other UART line status changes) must be inserted into the rx data stream in the same manner as the Windows API **IOCTL\_SERIAL\_LSRMST\_INSERT** control code.  It is acceptable for Linux based machines to only insert parity errors into inserted line status bytes.  FYI A UART’s LSR parity bit mask is 0x04. |
| parity | string | One of “none”, “odd”, “even”, “mark” or space” |
| esc\_char | string | Length = 1.  If set, this must enable error insertion into the rx data stream (see checkparity above). The string also denotes the escape character to use with error insertion. (It is acceptable for Linux based machines to only support 0xFF as the escape character, overriding the supplied esc\_char value with 0xFF in this case.)  If this field is not present then no error insertion is used (default). |
| rit | integer | Read interval timeout. \* see below.  1…1000 msecs (host machine must apply.)  Creation default must be **1000** msecs.  The applied accuracy / granularity of this timeout is not expected to be particularly great and its performance is expected to vary between operating systems. Machines are not required to improve on this and are permitted to simply map this value to the most equivalent value in the machine’s operating systems serial port API.  QCOM users note: To achieve very fast reponse times, QCOM users should use a low level communications pre-processor script and primarily always read an exact number of bytes. |
| rxcallback | function | The function if supplied will be called based on the set read timeout values below at a maximum rate of no more than ***maxcbrate*** times per second. This rate may be changed for a given QCOM user via the **qcom\_uartSetp()** QCOM API function. If the maximum rate is exceeded then no data must be lost; the rx data must be concatenated together in a single call back.  The rx call back function single argument will be a table with a schema as per the **QLE\_UART\_RX** state event data. |

* All table key/values are optional. Any missing key/value pair tells the machine to retain the existing value for that key/value.
* Unknown key/values must be ignored by the machine.
* **\* rit** must be implemented as per the *read interval timeout* property as documented in the Windows API **COMMTIMEOUTS** structure.

**Return Value:** true | nil, string

On success the function will return true.

On failure the function will return *nil, string* where the string must denote a nature of the failure.

* + - 1. **getState()**

This function will return the table as per the setState() member function.

**Call format:**

getState()

**Return Value:** table

The table schema will be populated are per the setState() member functions table argument but excluding rxcallback info.

Note however that the machine may add additional properties to the returned table such as ports usage stats. For example: rx byte count, tx byte count. When adding additional properties, please check the QCOM 3 SDK example C source code to see if a property field name can be adopted for consistency.

* + - 1. **write()**

The function will attempt to write the binary string argument to the serial port. This operation must be non-blocking.

**Call format:**

write(string)

**Return Value:** true | nil, string

Note that the host machine is not required to buffer writes. If a QCOM user performs a write before the previous write has finished transmitting then this behaviour may yield unexpected results (such as bytes transmitted in wrong order or not at all).

On success the function will return *true.*

On failure, nil plus an error message string will be returned. Refer to the QCOM 3 SDK for possible error messages.

The average response time[[267]](#footnote-65) using this function is expected to slower and more volatile than the qwrite method below. Anywhere from 2-30+ msecs for example. This is because there may be a thread switch between the thread/process that is receiving the byte and the QLE thread/process. For fast response times use the qwrite method instead.

Related: See serial port defaults section above for max write buffer size.

* + - 1. **qwrite()**

This UART member function is only present in the UART object when the object was created with a valid Lua script provided as an argument for the uartOpen() QCOM API function.

Pre: For this function to have any effect (and not return an error), the UART object must have been successfully created with a *script* argument representing a low level comms, byte pre-processor. cp:[[268]](#endnote-203)

The function must queue the binary string argument to be written to the serial port. cp:[[269]](#endnote-204) The write to the serial port must only take place when the low level comms pre-processor script (LBP) returns a specific command value. A queued write must occur from the same process/thread as the process/thread that is reading from the UART. This process/thread must reside as close to the machine’s hardware layer as practical, or otherwise located in an elevated priority thread dedicated to the COM port object’s read / queued-write operations. cp:[[270]](#endnote-205)

Once a qwrite call has been successful, additional qwrite()’s must fail until the last qwrite has actually been instigated by the machine or cancelled via the qcancel() member function (see below). cp:[[271]](#endnote-206)

**Call format:**

qwrite(string)

**Return Value:** true | nil, string

On success the function will return *true.*

On failure, nil plus an error message string will be returned. Refer to the QCOM 3 SDK for possible error messages.

Related: See serial port defaults section above for max write buffer size.

* + - 1. **qcancel()**

This UART member function is only present in the UART object when the object was created with a valid Lua script provided as an argument for the uartOpen() QCOM API function.

This member function cancels any previously qwrite() queued message, if any. cp:[[272]](#endnote-207)

**Call format:**

qcancel()

**Return Value:** true | nil, string

On success the function will return *true.*

On failure, nil plus an error message string will be returned. Refer to the QCOM 3 SDK for possible error messages.

* + - 1. **poke()**

This UART member function is only present in the UART object when the object was created with a valid Lua script provided as an argument for the uartOpen() QCOM API function.

If the UART object has been successfully setup with a pre-processor *script* (refer uartOpen()), then this function allows the caller to set simple global integer variables within that script’s environment. This allows the respective QCOM user some lightweight control over the script’s operation.

One example of use of the poke() member function is the QCOM 3 qcom16 app[[273]](#footnote-66) which uses poke to set/change the QCOM v1.6 poll address within the pre-processor scripts environment.

**Call format:**

poke(varname:string, value:integer)

The *varname* argument must be a valid Lua variable name. 4 chars max. length.

A poke must take effect no later than the next call to the pre-processor script.

Once a poke call has been successful, additional pokes will fail until the last poke has been actioned within the pre-processor scripts environment.

**Return Value:** true | nil, string

On success the function will return *true.*

On failure, nil plus an error message string will be returned. Refer to the QCOM 3 SDK for possible error messages.

* + - 1. **close()**

Signals the host machine to close the serial port. This function must be non-blocking. cp:[[274]](#endnote-208)

The close function must signal

* the uart thread to terminate if started; else the close function must free the com port handle itself. cp:[[275]](#endnote-209)
* the QLE Lua software driver by logging a **QLE\_UART\_RX** state event with a message of length set to 0. (In response the QLE Lua software driver will call the free() member function. cp:[[276]](#endnote-210)

Note that depending how the host machine implements serial communications, and while the function always returns immediately, it may take some time (msecs) before the serial port is actually closed (i.e. to terminate the UART thread) and allowing the port to be re-opened. Machines must take no longer than **500msecs** before the same serial port is able to re-opened. cp:[[277]](#endnote-211)

**Call format:**

close()

**Return Value:** true | nil, string

On success the function will return *true.*

On failure, nil plus an error message string will be returned. Refer to the QCOM 3 SDK for possible error messages.

* + - 1. **free()** - private

While the host machine must always supply this member function, the QLE Lua software driver makes it private for its use only; i.e. no QCOM user access to this function.

When free() is called by the QLE Lua software driver (which it does in response to a *dispatchUARTevent()* dispatched **QLE\_UART\_RX** message event ID with a message length of **zero**), the host machine must free the serial port resources\* provided it has been closed first, else it must return an error RV.

\*The free function must:

* Free any internal memory structure associated with the serial port object. cp:[[278]](#endnote-212)
* Clear the Lua up-value which was used to associated the Lua serial object with the C/C++ serial object. cp:[[279]](#endnote-213)

Refer to the QCOM 3 SDK re the above.

This function must be non-blocking. cp:[[280]](#endnote-214)

**Call format:**

free()

**Return Value:** true | nil, string

On success the function will return *true.*

On failure, nil plus an error message string will be returned. Refer to the QCOM 3 SDK for possible error messages.

## \_udp

User Datagram Protocol (UDP) communications functions. Refer section 12 for rmore information.

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_udp

Creates and returns an initially unconnected UDP object.

**Call format:**

**qcom\_udp([table])**

**Arguments:**

t:table Optional. The table may contain the following optional key / values:

ipv6:boolean

If ipv6 = true then the socket object returned must utilise the IPv6 address family / protocol.

If ipv6 is *false* then the socket object returned must utilise the IPv4 address family / protocol.

**Return Value:** *table | nil, string*

On success the return value will be of type *table* which denotes the applicable UDP API as defined in section 12 with respect to the given function arguments provided.

On failure the return value will be *nil* plus a suitable error string. Refer to the QCOM SDK for failure error messages. The function may also return machine specific error messages with respect to the specific IP API it utilises.

### qcom\_udpSetp

**Call format:**

**qcom\_udpSetp(username, table)**

This function contains all the parameters from the **qcom\_tcpClientSetp()** function. Refer to section 11.38.2 for more information concerning this function’s arguments. Note that the properties set by this function only apply to UDP socket objects / API.

The following additional table parameters (specific to the QCOM UDP API) must also be supported by this function:

srcports[] : integer : max **8**

The *srcports* parameters must be a numerically indexed table of source socket ports (type *number*) for example:

srcports = {24711, 24712, 32000}

Ports must be in the range **22000…65535.**

The machine must ensure that the QCOM user is only able to bind() UDP sockets on the source ports listed in the *srcports* parameter. cp:[[281]](#endnote-215) (The QLE LSD provides a global Lua function called **qapi\_socketSourcePortAvail(username, srcport)** the machine may call for this purpose, or it can check this itself as privileged source ports per qcom user are also saved by the machine in its NV memory. Refer STH message: ***udpSetp\_srcports***)

Example:

assert(qcom.udpSetp(qcom.userWhoAmI(), {maxobj = 2, mtu = 1000, maxbps = 500, maxiops = 2, srcports = {24711, 24712, 32000}}))

### qcom\_udpGetp

Returns the values as set by the qcom\_udpSetp() function for the given QCOM user.

Note however in the return value the *srcports* table will be converted to a *btable* (a set) of source ports for example:

srcports = {[24711] = true, [24712] = true, [32000] = true}

**Return Value:** *table | nil, string.* Refer to the QCOM SDK for failure error messages.

### qcom\_udpGetStats

**Incept date of this function is: TBA.**

## \_tcpClient

TCP/IP client communications functions. Refer section 12 for more information.

Refer QCOM API Summary Spreadsheet for the following functions where no description is present.

### qcom\_tcpClient

**Call format:**

**qcom\_tcpClient([table])**

**Arguments:**

t:table Optional. The table may contain the following optional key / values:

ssl:boolean

If ssl = true then the machine mustcreate and return a TCP/IP based Transport Layer Security (TLS) v1.2 (RFC 5246) secured client socket object (of type *table*)

ipv6:boolean

If ipv6 = true then the socket object returned must utilise the IPv6 address family / protocol.

If ipv6 is *false* then the socket object returned must utilise the IPv4 address family / protocol.

**Return Value:** *table | nil, string*

On success the return value will be of type *table* which denotes the applicable TCP/IP API as defined in section 12 with respect to the given function arguments provided.

On failure the return value will be *nil* plus a suitable error string. Refer to the QCOM SDK for failure error messages. The function may also return machine specific error messages with respect to the specific IP API it utilises.

### qcom\_tcpClientSetp

This function assigns limits and parameters pertaining to TCP client communications for a given QCOM user.

This function also applies to both the insecure and secure client socket API but not to UDP socket object API / objects.

**Call format:**

**qcom\_tcpClientSetp(username, table)**

|  |  |  |
| --- | --- | --- |
| **Argument** | **Type** | **Description** |
| username | Global Type | QCOM user username. |
| t | table | See below. Any missing keys will retain their current value (see below). Extra keys will be ignored. Refer to the QCOM SDK for all sanity checks and limits pertaining to this table and its keys and values.  Changes must affect only future socket objects created. |

The table t is an associative indexed table and may be comprised of the following keys and values:

“**maxobj**” : Lua type *integer,* **default = 0, min = 0, max = 4**. The maximum number of socket objects the QCOM user with *username* may have at any time. Refer QCOM API function qcom\_tcpClient(). The QLE Lua software driver enforces this limit.The QLE Lua software driver will also impose a machine wide client socket object limit mention in section 12.1.

“**mtu”** : Lua type *integer,* units = bytes, **default = 1000, min = 1, max = 10000**. This value denotes the maximum size in bytes, of any message in a QCOM API write operation. If a QCOM user tries to exceed this limit in any write operation (s12.2.8), the write function must return an error without sending any data. If a message was received which exceeds this value,then the machine must spread the message over several reads. Note, this value is **not** related to the MTU of any underlying transport protocols whose MTU values should be left at default, (Please advise OLGR if these default MTU values will not be approximately ~1-2kB in size.)

“**maxbps**” : Lua type *integer*, units = bytes per second, **default = 500, min = 10, max = 10000**. This value is the maximum rate in which the QCOM user with *username* may transmit and receive bytes on each socket object. This limit must not count any underlying protocol bytes.

“**maxiops**” : Lua type *integer*, units = I/O per second, **default = 5, min = 1, max = 300**. This value is the maximum rate in which the QCOM user with *username* may perform QCOM API read / write operations on each socket object. This limit must not count any underlying protocol IO.

Implementing maxbps / maxiops

The machine must throttle the socket object’s bps and I/O to within the set rates for each QCOM user. In implementing *maxbps*, the machine may allow the *maxbps* limit to be exceeded by one read / write operation and then throttling future read / write operations until the user’s bps falls back below the limit.

If a QCOM user tries to write bytes while their socket object quota is currently exceeded, the write operation must return an error and not perform the write. Refer s12.2.8 for more information.

If a QCOM user receives data while their socket object quota is currently exceeded, then the machine must simply delay reading new data from the underlying communications API in proportion to the set limits, which will put back pressure on the external sender.

The machine must store the above settings for each QCOM user in persistent / NV memory.

**Return Value:** *true | nil, string*

The return value is true on success and nil on failure plus a suitable error string. Refer to the QCOM SDK for failure error messages.

### qcom\_tcpClientGetp

Returns the values as set by the qcom\_tcpClientSetp() function for the given QCOM user.

**Call format:**

**qcom\_tcpClientGetp([username])**

**Return Value:** *table | nil, string.* Refer to the QCOM SDK for failure error messages.

### qcom\_tcpClientGetStats

**Incept date of this function is: TBA.**

This function retrieves statistics pertaining to client socket communications for the given QCOM user. All parameters set by the qcom\_tcpClientSetp() API function are also echoed back here containing their currently set values.

**Call format:**

**qcom\_tcpClientGetStats(username, table)**

|  |  |  |
| --- | --- | --- |
| **Argument** | **Type** | **Description** |
| *username* | Global type | QCOM user username. |

**Return Value:** boolean, [string] | table

The return values are true, table on success and false, string on failure; a suitable error string must be supplied on failure.

The table return value will be an associated table containing the following key, value pairs.

For example (example only – refer to the QCOM SDK for actual data to be returned):

* **“username”** : string
* **“txb”** : integer : Total bytes tx
* **“rxb”** : integer : Total bytes rx
* **“iocount”** : integer : Total send / receive operations
* **“connections”** : integer : The total number of connections made
* **“activeconnections”** : integer : The current number of currently active connections

# QCOM Internet Protocol API

The QCOM Lua API provides a number of IP communication APIs defined in this section for use by privileged QCOM users.

All the functions in the QCOM communications API’s must be implemented as non-blocking.

*Note: At this time a communications “server” API have not been provided (i.e. a QCOM user cannot host a server on a machine; it can only be a “client”). This is to reduce the costs of implementing QCOM in a machine. Also, server APIs may also introduce additional risks and demands on machine resources that must be carefully managed before permitting. Communications server APIs for QCOM machines may be considered for future QCOM releases or sooner, depending on industry feedback over time.*

The QCOM 3 API currently supports two IP based protocols:

* TCP (clients only; both secure and non-secure)
* UDP

The QCOM Lua IP API is based on the Berkeley socket API, and the openssl API for SSL connections.

Communications objects may be created by privileged QCOM users by calling:

**qcom\_udp()**

**qcom\_tcpClient()**

Once a QCOM user is finished with a communications object, to free the object and associated data and resources, the user must invoke the free() method on the communication object*.* This must cause the object and associated resources to be freed by the machine. This must also gracefully close any connection associate with the object. Refer to the QCOM SDK for example code.

Parameters concerning each available communications API, for each QCOM user may be setup via the following functions:

**qcom\_udpSetp()**

**qcom\_tcpClientSetp()**

Refer sections 11.37.2 & 11.38.2 respectively.

The settings the above functions control protect the machine from being over utilised by any given QCOM user granted privilege to the QCOM communications API. This is achieved by being able to control the amount of network bandwidth available and the number maximum client connections for any given QCOM user. The communications setup functions are intended to be privileged out to a commercially neutral QCOM user[[282]](#footnote-67) (typically the same user that creates QCOM users accounts, i.e. the QMA).

The QCOM communications API operates using a number of call-back functions which are discussed in more detail in the following sections. They are setup via respective calls to the following TCP object member functions:

**setConnectCallback()**

**setRxCallback()**

**setDisconnectCallback()**

**setErrorCallback()**

**setCanWriteCallback()**

In QCOM 3 call-back functions are executed similar to any other user script hooked to a state event. (Related 14.6.1). The State Event ID’s (global type: *seid*) the machine must use with respect to the above set call-back functions are listed in the QCOM 3 SDK. Refer to the QCOM 3 SDK for the required arguments to be provided to each type of call-back function.

Socket Defaults

The equivalent of the following Berkeley socket options must be set for all QCOM Communications API sockets:

**SO\_KEEPALIVE = false**

**SO\_LINGER = false**

**SO\_OOBINLINE = true**

**SO\_RCVBUF = 8192**

**SO\_SNDBUF = 8192**

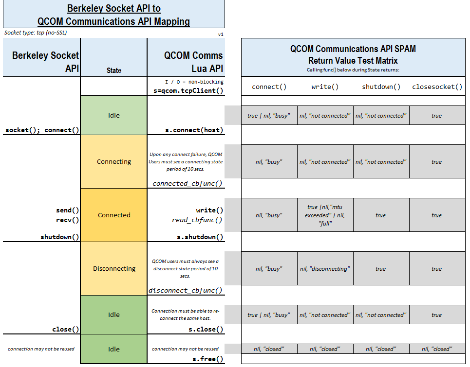
TCP Defaults

The equivalent of the following Berkeley TCP options must be set for QCOM Communications API sockets created as listed below:

**TCP\_NODELAY = false**

*If there is sufficient demand from QCOM users, a TCP related privilege will be created that allows a specific QCOM user to toggle* ***TCP\_NODELAY*** *as desired****.***

Other



Refer to the QCOM Summary Spreadsheet “TCP” sheet for QCOM Communications API diagrams relating to socket state and API return values.

## Connection Management

QCOM users using the communications API must note:

There are QCOM user and global cooldowns on socket API function calls relating to socket object creation, connects and reconnects. These cooldowns exist in order to limit state events and CPU use in the machine.

The QCOM Lua software driver implements a QCOM user owned overall socket object limit of **8**. The machine may increase this limit if desired. This limit is coded in the SDK module: machine.lua. Related: Per QCOM user socket object limit. Refer section 11.38.2.

## TCP Client Socket API

The QCOM API function **qcom\_tcpClient**() must create and returns a TCP/IP client socket object (type *table*). The following functions must reside within the returned client socket object:

socket()

connect(ip, port)

setConnectCallback(function)

setRxCallback(function)

setDisconnectCallback(function)

setErrorCallback(function)

setCanWriteCallback(function)

send(message:string)

close()

free()

QCOM socket Lua API functions must never block. If the operation was applicable at the time the function is invoked, then the machine must ensure the operation attempts to complete in the background. The machine must invoke the call-back functions to advise the QCOM user of when any potentially blocking operation has been completed and the result thereof.

### socket()

This function calls the equivalent of the Berkeley API socket function.

**Call format:**

**socket()**

**Arguments:**

None.

**Return Value:** true | nil, errmsg:string

### connect()

This function may instigate a connection attempt to a remote host denoted by the *host* argument.

In the implementation of this function, the machine must perform the equivalent of a call to the Berkely socket function **connect()**.

In Linux, if connect fails, the state of the socket is considered unspecified. Portable applications should close the socket and create a new one for reconnecting.[[283]](#footnote-68)

**Call format:**

**connect(host, port)**

**Arguments:**

The *host* argument is of type *string* and may be an IPv4 or IPv6 address.

Refer to the QCOM 3 SDK for required argument sanity checks.

**Return Value:**

On success, the function must return the Boolean value of ***true***.

On failure, the function must return the values **nil**, **string**; where the string is the applicable error message shown in quotes above. Refer to the QCOM 3 SDK for required error messages.

Any issues concerning the subsequent connection attempt must be sent by the machine to the error call-back function as set by setErrorCallback().

**Related:** Refer to the QCOM Summary Spreadsheet “TCP” sheet for QCOM Communications API state diagrams and other possible return values.

### setConnectCallback()

This function sets up a call-back function which must then be called each time a successful connection is made on the associated communications object.

**Call format:**

**setConnectCallback([f | *nil*])**

**Arguments:**

f : *function* : Denotes the call-back function.

The QLE Lua software driver will call the function as defined above with a single argument of type table. The table data contains all the **QLE\_IP\_CONNECT** state event data. Refer to the QCOM 3 summary spreadsheet or SDK for more information.

Any return values received from the call-back function are be ignored by the QLE Lua software driver.

If the argument is missing or *nil* then this must clear any previously set call-back function*.*

The machine’s implementation of this function must check the following items; aborting with a failure RV upon any issue:

* There is at least one argument (“too few arguments”).
* The argument is not of type *function* (“argument is wrong type”).

Related: generic sanity checks on arguments. Refer section 10.5.2.

**Return Value:**

On success, the function must return the Boolean value of ***true***.

On failure, the function must return the values **nil**, **string**; where the string is the applicable error message shown in quotes above.

The call-back function must be called for a client socket just after the connection to the server is successfully opened. The socket must also be immediately available for writing when the call-back function is called.

### setRxCallback()

This function sets up a call-back function which must then be called each time data is received associated with the communications object with respect to throttling parameters set by qcom\_tcpClientSetp().

**Call format:**

**setRxCallback([f | *nil*])**

**Arguments**:

f : *function* : Denotes the call-back function.

The QLE Lua software driver will call the function as defined above with a single argument of type table. The table data contains all the **QLE\_IP\_RX** state event data. Refer to the QCOM 3 summary spreadsheet or SDK for more information.

Any return values received from the call-back function are be ignored by the QLE Lua software driver.

If the argument is missing or *nil* then this must clear any previously set call-back function*.*

The machine’s implementation of this function must check the following items; aborting with a failure RV upon any issue:

* There is at least one argument (“too few arguments”).
* The argument is of not type *function* (“argument is wrong type”).

Related: generic sanity checks on arguments. Refer section 10.5.2.

Even if there is no rx call-back function registered, rx data must still read from the network by the machine, counted towards the users rx quota (which may then cause a subsequent refusal to read from the network layer) and then discarded. cp:[[284]](#endnote-216) Rationale: This approach will be less complex & work to implement in the machine, plus there is already a per quser ***maxiops*** rate limit etc to implement that should negate the need for the user to also be able manually control this. (For UDP socket objects FYI; QCOM users can manually perminently shutdown UDP socket object rx).

**Return Value:**

On success, the function must return the Boolean value of ***true***.

On failure, the function must return the values **nil**, **string**; where the string is the applicable error message shown in quotes above.

**Call-back Function Arguments:**

The following arguments must be passed to the call-back function in the order shown:

msg : *string* : message data

len :integer : message data length

### setDisconnectCallback()

This function sets up a call-back function which must then be called each time a session associated with the communications object disconnects.

**Call format:**

**setDisconnectCallback([f | *nil*])**

**Arguments**:

f : *function* : Denotes the call-back function.

The QLE Lua software driver will call the function as defined above with a single argument of type table. The table data contains all the **QLE\_IP\_DISC** state event data. Refer to the QCOM 3 summary spreadsheet or SDK for more information.

Any return values received from the call-back function are be ignored by the QLE Lua software driver.

If the argument is missing or *nil* then this must clear any previously set call-back function*.*

The machine’s implementation of this function must check the following items; aborting with a failure RV upon any issue:

* There is at least one argument (“too few arguments”).
* The argument is not of type *function* (“argument is wrong type”).

Related: generic sanity checks on arguments. Refer section 10.5.2.

**Return Value:**

On success, the function must return the Boolean value of ***true***.

On failure, the function must return the values **nil**, **string**; where the string is the applicable error message shown in quotes above.

### setErrorCallback()

This function sets up a call-back function which must then be called each time an error occurs for the following reasons:

* Unable to connect (TCP only)
* Rx & Tx errors that don’t indicate a graceful shutdown

The machine must pass an error message to the call-back function as function arguments (see below).

**Call format:**

**setErrorCallback([f | *nil*])**

**Arguments**:

f : *function* : Denotes the call-back function.

The QLE Lua software driver will call the function as defined above with a single argument of type table. The table data contains all the **QLE\_IP\_ERROR** state event data. Refer to the QCOM 3 summary spreadsheet or SDK for more information.

Any return values received from the call-back function are be ignored by the QLE Lua software driver.

If the argument is missing or *nil* then this must clear any previously set call-back function*.*

The machine’s implementation of this function must check the following items; aborting with a failure RV upon any issue:

* There is at least one argument (“too few arguments”).
* The argument is not of type *function* (“argument is wrong type”).

Related: generic sanity checks on arguments. Refer section 10.5.2.

**Return Value:**

On success, the function must return the Boolean value of ***true***.

On failure, the function must return the values **nil**, **string**; where the string is the applicable error message shown in quotes above.

**Call-back Function Arguments:**

The following arguments must be passed to the call-back function in the order shown:

msg : *string* : error message

The above error message must start with the string “unable to connect”. It may be supplemented with API specific information.

### setCanWriteCallback()

This function sets up a call-back function which must then be called each time the socket object is available for writing after any period of write unavailability.

Accordingly the machine must call any set can-write call-back function in the following two scenarios:

* After a successful connection (TCP) or bind (UDP) has occurred and the socket object becomes available for reading and writing.
* After any previous failed call to send()/sendto() that had an error return valueand a subsequent select() operation by the machine indicates the socket is available for writing.

**Call format:**

**setCanWriteCallback([f | *nil*])**

**Arguments**:

f : *function* : Denotes the call-back function.

The QLE Lua software driver will call the function as defined above with a single argument of type table. The table data contains all the **QLE\_IP\_CANWRITE** state event data. Refer to the QCOM 3 summary spreadsheet or SDK for more information.

Any return values received from the call-back function are be ignored by the QLE Lua software driver.

If the argument is missing or *nil* then this must clear any previously set call-back function*.*

The machine’s implementation of this function must check the following items; aborting with a failure RV upon any issue:

* There is at least one argument (“too few arguments”).
* The argument is not of type *function* (“argument is wrong type”).

Related: generic sanity checks on arguments. Refer section 10.5.2.

**Return Value:**

On success, the function must return the Boolean value of ***true***.

On failure, the function must return the values **nil**, **string**; where the string is the applicable error message shown in quotes above.

### send()

Writes data via the TCP client socket with respect to the QCOM user’s TCP client settings (refer section 11.38.2).

In the implementation of this function, the machine must perform the equivalent of either a call to the Berkely socket function **send()** or SLL API function **SLL\_write()** depending on whether the socket is using SLL or not.

**Call format:**

**send(string)**

**Arguments**:

Type *string;* the message data to send via the socket object. (NB, Lua supports binary strings.) Max message length is limited by the ***mtu*** property as set via qcom\_tcpClientSetp().

The machine must automatically handle OS socket API partial writes on behalf of the user.

The machine’s implementation of this function must check the following items; aborting with a failure RV upon any issue:

* There is at least one argument (“arg missing”).
* The argument is not of type *string* (“arg must be string”).
* The socket object is not in the connected state or in the process of disconnecting (“not connected” or “disconnecting” respectively).
* The string argument length is not zero (“string arg is 0 length”)
* The string argument length is greater than the “mtu” (as set by the **qcom\_tcpClientSetp()** QCOM API function in section 11.38.2) (“mtu exceeded”).
* The write buffer still has space (“full”)

Related: generic sanity checks on arguments. Refer section 10.5.2.

**Return Value:**

On success, the function must return the Boolean value of ***true***.

On failure, the function must abort and return the values **nil**, **string**; where the string is the applicable error message shown in quotes above. Where the error concerns a third party or custom socket API error, the API specific error string must be returned.

**Related:** Refer to the QCOM Summary Spreadsheet “TCP” sheet for QCOM Communications API state diagrams and other possible return values.

### close()

This function must result in the associated socket object being closed.

If the socket is currently connected, the TCP socket connection must be gracefully shutdown by the machine first and then closed. As the QCOM Communications API must be non-blocking; the close must occur in background in this case.

If the socket has never successfully established a connection and is idle, then the close must complete before the function returns.

If the socket is in the process of connecting then, this process must gracefully abort if possible, or otherwise complete the connect process (success or fail) and then gracefully automatically close the connection immediately after. As the QCOM Communications API must be non-blocking; the close command must be queued to occur in background in this case.

If the socket is already in the process of closing then the function must return *nil, “disconnecting”*.

**Related:** Refer to the QCOM Summary Spreadsheet “TCP” sheet for QCOM Communications API state diagrams.

Once closed, the socket object must be able to be reused.

**Call format:**

**close()**

**Arguments**: none.

**Return Value:**

Returns *true* on success and *nil, errmsg* on any error. Refer to the QCOM SDK for error return messages.

### free()

Gracefully closes and frees a socket object.

A QCOM user must be able to call this function on a socket objected anytime. It is accepted that this operation may not always be instantaneous in the machine.

When the machine has completed the close / free operation, it must log the **SOCKET\_FREED** state event which:

* Advises the QLE Lua software driver, allowing it to update its internal state accordingly.
* Indicates to QCOM users that they can create a new socket object without fear of reaching their maxobj quota.

*Design note: The free function was designed this way in order to simplify host machine programming and also to provide the host machine with a fire-and-forget method of freeing all QCOM user socket objects, especially needed in the case of a qcom.userDelete(). It also simplifies QCOM user communications programming in that they don’t have to deal with ‘try again’ return values typical to non-blocking socket I/O, and it ensures their maxobj count only decrements as a part of the call to free() (when called at the appropriate time) and not some indeterminate time later after the socket shutdown first.*

QCOM users note: to ensure you don’t run out of socket objects (w.r.t their qcom\_tcpSetp() set **maxobj** value) there should be one call to free() per call to either of:

* qcom\_udp (s11.37.1), or
* qcom\_tcpClient (s11.38.1)

**Call format:**

**free()**

**Arguments**: none.

**Return Value:**

Returns *true* on success and *nil, errmsg* on any error. Refer to the QCOM SDK for error return messages.

**Related:**

* Refer to the QCOM Summary Spreadsheet “TCP” sheet for QCOM Communications API state diagrams and other possible return values.
* **SOCKET\_FREED** state event.

## Secure Client Socket API

The QCOM API function **qcom\_tcpClient({ssl = true})** mustcreate and return a TCP/IP based Transport Layer Security (TLS) v1.2 (RFC 5246) secured client socket object (of type *table*).

The secure client socket object returned must include all the functions in the previous section as well as the following additional functions:

addTrustedCert()

getPeerCert()

verifyCert()

getCiphers()

setCiphers()

Before returning the object, the machine must automatically remove any known weak ciphers at design time. Related: <https://weakdh.org/sysadmin.html>

### addTrustedCert()

This function adds the trusted certificate/s or public keys thereof into the object. This must be done by the user before the connection attempt to a secure host is made.

It is optional to use this function to set any trusted certificates / public keys, as a secure connection can still be made to a host if no trusted certificates / public keys have been loaded. In any case, the host’s certificate may be retrieved manually during a session and manually verified by the user via the verifyCert() function.

Certificate store persistence.

The case the trusted cert store must persist across multiple connection sessions using the same secure client object. cp:[[285]](#endnote-217)

Explanation: The trusted certificate store must kept a part of the client socket object in an SSL\_CTX object (or the equivalent), so only when the socket is free()’d will the cert store is destroyed. For example; using a single client socket object instance, after close()’ing a connection (which releases the socket id), it must be possible to get a new socket() ID and successfully make a connect() again without having to add the trusted certificate again.

**Call format:**

addTrustedCert(cert)

**Arguments:**

*string* cert

The *cert* argument should be either an x509 certificate or public key of type *string* in PEM format denoting one or more trusted certificate authorities.

The machine’s implementation of this function must check the following items; aborting with a failure RV upon any issue:

* There is at least one argument (“arg missing”).
* The argument is not of type *string* (“arg must be string”).
* The string length must be less than **10kB** characters (“arg too long”).
* The client socket is not able to load the host cert because it is in the wrong state; e.g. already connecting or connected. (“busy”)
* addTrustedCert() must only be allowed to be successfully called once per client socket object (“max reached”)
* If there is an issue within the encoded PEM string, then it is permissible to return a secure client API specific error message.

**Return Value:**

On success, the function must return the Boolean value of ***true***.

On failure, the function must return the values **nil**, **string**; where the string is the applicable error message shown in quotes above.

Related:

* Openssl C API functions:
  + SSL\_CTX\_get\_cert\_store()
  + X509\_STORE\_add\_cert()

### useCert()

Sets a client certificate to use for the secure client object. This machine must apply the certificate and private key to the SSL CTX within the secure client object. If successful, the machine must also apply the certificate and private key to the SSL object if one currently exists.

Before returning, the machine must also check the consistency of a private key with the corresponding certificate loaded into ctx and return the error if there is a problem.

**Call format:**

useCert(cert, privkey)

**Arguments:**

cert : string : PEM

privkey : string : PEM

Refer to the QCOM 3 SDK for required argument sanity checks.

**Return Value:**

On success, the function must return the Boolean value of ***true***.

On failure, the function must return the values **nil**, **string**; where the string is an applicable error message. Refer to the QCOM 3 SDK for required error message return values.

Related:

* Openssl C API functions:
  + SSL\_use\_certificate()
  + SSL\_CTX\_use\_certificate()
  + SSL\_CTX\_use\_PrivateKey()
  + SSL\_use\_PrivateKey()
  + SSL\_CTX\_check\_private\_key()

### getPeerCert()

This function must return the peer certificate of an established connection.If openssl is being used, this is equivalent to a call to openssl API function **SSL\_get\_peer\_certificate()**.

**Call format:**

getPeerCert()

This function has no arguments

The machine’s implementation of this function must check the following items; aborting with a failure RV upon any issue:

* The client socket is not able to load the host cert because it is in the wrong state; e.g. not connected or disconnecting. RV = nil, (“not connected” or “disconnecting” respectively)

**Return Value:** string | nil, string

On success, the function must return a string comprised of the peer (host) certificate in PEM format.

On failure, the function must return the values **nil**, **string**; where the string is the applicable error message shown in quotes above. Secure socket library API errors or similar may be returned for the error string for errors not covered here.

### verifyCert()

This function must verify the remote host’s certificate during an established connection against any trusted CA certificates or public keys loaded previously if any (via the addTrustedCert() function) if any. The verifyCert() function, or its underlying secure socket API equivalent, must not automatically be called upon a connection being established as there are many non-critical certificate issues that still allow a connection to be made. It must be a decision for the QCOM user whether or not to continue with a connection that fails verification.

This function allows the QCOM user to manually verify a certificate for critical and non-critical issues at their discretion.

Note, the QCOM user should first manually ensure a certificate was actually presented via getPeerCert() as this function could return *true* in the case where the peer did not present a certificate at all.

Related: openssl API function: **SSL\_get\_verify\_result()**.

**Call format:**

verifyCert()

This function has no arguments

The machine’s implementation of this function must check the following items; aborting with a failure RV upon any issue:

* The client socket is not able to load the host cert because it is in the wrong state; e.g. not connected or disconnecting. RV = nil, (“not connected” or “disconnecting” respectively)

**Return Value:** *true | nil, string*

The function must return *true* on success, or *nil, errmsg:string* on fail.

On failure, an error message string must be returned as to the reason why the verification failed. The error message string must be one of the strings returned by the openssl API function **X509\_verify\_cert\_error\_string()** (e.g. “self signed certificate” or "certificate has expired", etc.)

Machines that utilise other secure socket APIs must map their errors onto the equivalent openssl API error string.

Related: openssl API header file x509\_vfy.h:X509\_V\_ERR\_\*

Except for critical errors, the machine must take no action if a certificate fails verification. It is a decision for the QCOM user as to whether or not to continue or terminate the session if a certificate fails verification but where a connection is still made.

### getCiphers()

Returns the list of available SSL ciphers from the context stored in the client socket object, sorted by preference.

Related: The openssl API function:

const char \*SSL\_get\_cipher\_list(const SSL \*ssl, int priority);

**Call format:**

getCiphers()

This function has no arguments

**Return Value:** *string | nil, string*

The function must return *string* on success, or *nil,string* on fail.

Example success return value:

“ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES256-GCM-SHA384:DHE-RSA-AES128-GCM-SHA256:DHE-DSS-AES128-GCM-SHA256:DHE-DSS-AES256-GCM-SHA384:DHE-RSA-AES256-GCM-SHA384:ECDHE-RSA-AES128-SHA256:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES128-SHA:ECDHE-ECDSA-AES128-SHA:ECDHE-RSA-AES256-SHA384:ECDHE-ECDSA-AES256-SHA384:ECDHE-RSA-AES256-SHA:ECDHE-ECDSA-AES256-SHA:DHE-RSA-AES128-SHA256:DHE-RSA-AES128-SHA:DHE-DSS-AES128-SHA256:DHE-RSA-AES256-SHA256:DHE-DSS-AES256-SHA:DHE-RSA-AES256-SHA:AES128-GCM-SHA256:AES256-GCM-SHA384:AES128-SHA256:AES256-SHA256:AES128-SHA:AES256-SHA:SRP-DSS-AES-256-CBC-SHA:SRP-RSA-AES-256-CBC-SHA:DHE-DSS-AES256-SHA256:SRP-DSS-AES-128-CBC-SHA:SRP-RSA-AES-128-CBC-SHA:DHE-DSS-AES128-SHA:DHE-RSA-CAMELLIA256-SHA:DHE-DSS-CAMELLIA256-SHA:CAMELLIA256-SHA:DHE-RSA-CAMELLIA128-SHA:DHE-DSS-CAMELLIA128-SHA:CAMELLIA128-SHA:DES-CBC3-SHA:”

Related: https://weakdh.org/sysadmin.html

The failure error string may be secure socket API specific.

### setCiphers()

This function must set the list of available ciphers for the object’s SSL CTX object using the control string argument provided. If the object currently has an SSL object then it must set the ciphers on that as well.

When operating on the SSL CTX, it will be possible to successfully call this function anytime while the client socket object exists other than while in the process of freeing it. Changes will take effect on each new established connection\session and will not affect an existing connection/session.

**Call format:**

setCiphers(string)

The format of the string argument can be found at:

<https://www.openssl.org/docs/manmaster/man3/SSL_CTX_set_cipher_list.html>

Related: The openssl API functions:

int SSL\_CTX\_set\_cipher\_list(SSL \*ssl, const char \*str);

int SSL\_set\_cipher\_list(SSL \*ssl, const char \*str);

The machine’s implementation of this function must check the following items; aborting with a failure RV upon any issue:

* There is at least one argument (“arg missing”).
* The argument is not of type *string* (“arg must be string”).
* The string length must be greater than **4** characters (“arg too short”).
* The string length must be less than **1,000** characters (“arg too long”).
* For secure socket API related issues it is permissible to return the secure client API specific error message.

**Return Value:** *true | nil, string*

The function returns *true* on success, or *nil,string* on fail.

On failure, the function must return the values **nil**, **string**; where the string is the applicable error message shown in quotes above. Secure socket library API errors or similar may be returned for the error string for errors not covered here.

Related: https://weakdh.org/sysadmin.html

## UDP API

The QCOM API function **qcom\_udp()** creates and returns (initially **unconnected)** UDP objects. In QCOM 3’s UDP implementation, the UDP source port must be bind()’ed and to a source port that has been privileged to the user. Refer to section 12.4.2 for more information on setting the source port.

Within this object, the following functions are defined:

socket()

bind()

connect()

setBroadcast()

setRxCallback()

setErrorCallback()

setCanWriteCallback()

sendto()

send()

shutdown()

close()

free()

QCOM socket Lua API functions must never block. If the operation was applicable at the time the function is invoked, then the machine must ensure the operation attempts to complete in the background. The machine must invoke the call-back functions to advise the QCOM user of when any potentially blocking operation has been completed and the result thereof.

Simple UDP example (no error checking):

udp = qcom\_udp()

udp.setRxCallback(myRxFunc) -- optional

udp.bind(27446)

udp.sendto(”hello world”, "192.168.1.11", 7447)

udp.close()

udp.free()

### socket()

This function calls the equivalent of the Berkeley API socket function.

**Call format:**

**socket()**

**Arguments:**

None.

**Return Value:** true | nil, errmsg:string

### bind()

**Call format:**

**bind(srcPort : integer)**

This function must **bind** a UDP socket for rx & tx operations.

In the implementation of this function, the machine must perform the equivalent of a call to the Berkely socket function **bind()**. The QCOM 3 UDP API requires all UDP sockets to be binded (whereas in Berkely sockets this is not mandatory FYI.)

**Arguments**:

*srcPort*

The *srcPort* argument denotes the source port to use for the UDP socket object. *srcPort* must be in the range **22000…65535** and must also have been privileged to the calling QCOM user via the QCOM API function qcom\_udpSetp() (refer s11.37.2).

The function must return an error if the port is not privileged or already in use. Refer to the QCOM API for required sanity checks and error messages.

The machine’s ephemeral port range[[286]](#footnote-69) must not overlap the allowable QCOM 3 source port range above.

**Return Value:** true | nil, errmsg

Refer to the QCOM API for required sanity checks and error messages.

### connect()

For call format and arguments, refer section 12.2.2.

For a UDP object, a successful call to the underlying OS connect() function, merely establishes a default destination address that allows the user to call the **send()** method. Also, any datagrams received from an address other than address specific and port specified by the call to connect will be discarded.

For a UDP object, connect may be called again to change the destination address.

Use of connect is optional but is recommended when applicable as it can help prevent rx spam.

### setBroadcast()

**Call format:**

**setBroadcast(boolean)**

**Arguments**:

The *boolean* argument turns *on* or *off* the machine equivalent to the **SO\_BROADCAST** Berkeley socket option provided the QCOM user pertaining to the socket object has also been privileged to do so with respect to the socket’s source port. Refer QCOM API function qcom\_udpSetp() (s11.37.2).

The default at socket object creation must be *off* (0).

**Return Value:**

Returns *true* on success and *nil, errmsg* on any error. Refer to the QCOM SDK for error return messages.

### setRxCallback()

As per section 12.2.4.

### setErrorCallback()

As per section 12.2.6.

### setCanWriteCallback()

As per section 12.2.7.

### sendto()

Attempt to write data via the UDP socket with respect to the QCOM user’s UDP socket settings (refer section11.37.2). In the implementation of this function, the machine must perform the equivalent of a call to the Berkely socket function **sendto()**.

**Call format:**

**sendto(message:string, IPaddress:string, destPort:integer)**

**Arguments**:

**message**

The message data to send. Max message length is limited by the ***mtu*** property set via the QCOM API function qcom\_udpSetp().

**IPaddress**

A string denoting the destination IPv4 or IPv6 address.

**destPort**

The destination socket port of the UDP message being sent.

Refer to the QCOM SDK for required argument sanity checks.

The machine must automatically handle OS socket API partial writes on behalf of the user.

**Return Value:**

Returns *true* on success and *nil, errmsg* on any error. Refer to the QCOM SDK for error return messages.

### send()

**Call format:**

**send(string)**

This function will fail unless the UDP socket object has connected via a prior call to the **connect()** method.

### shutdown()

This function must instigate a **read data** shutdown on the associated socket object equivalent to the Berkeley socket API function:

shutdown(socket, SD\_RECEIVE);

This cannot be undone for the given socket descriptor.

**Arguments**:

None.

**Return Value:**

Returns *true* on success and *nil, errmsg* on any error. Refer to the QCOM SDK for error return messages.

### close()

This function must close the socket. A UDP socket object must be able to be reused.

**Call format:**

**close()**

**Arguments**: none.

**Return Value:**

Returns *true* on success and *nil, errmsg* on any error. Refer to the QCOM SDK for error return messages.

### free()

Refer to section 12.2.10.

# Events

A QCOM machine will generate a range of **machine events** during normal operation. For example: fault events, door state changes and significant events like Cancel Credits etc. This section deals with QCOM event definitions, event storage and retention, the QCOM event API and reporting.

QCOM 3 machine events are defined in the QCOM summary spreadsheet – “QCOM 3 Table of Machine Events” – ‘events’ sheet.

QCOM 3 machine events directly relate to QCOM v1’s events. Many QCOM 3 machine events share the same defintions as the equivalent QCOM v1 event definition.

All machine events that occur while the QLE is running are sent to the QLE LSD via the **EVENT** state event.

Note:

* QCOM 3 **machine events** are NOT **state events** (s14); they are two entirely distinct and mutiually exclusive classes of events logs in QCOM 3.
* In this document, “state events” will always be referred to as “state events”; any other mention of an event in the context of a buffered or logged event is a reference to a machine event under this section.
* The main differences are that **state events** are numerous and ephemeral, whereas **machine events** are less frequent by definition and are stored in the machine NV memory for a period.

Related:

* Refer to section 10.7.1 for when the machine may start logging events on a power up.
* Refer section 4.4 for then the machine may start logging events after RAM clear.
* The following chapter on state events.

*High Level Design requirements:*

* *Event retention. The machine must retain a copy of all generated QCOM events for machine delivered display / reporting / diagnostic purposes for as long as possible until overwritten.*
* *If events are being forwarded to a host, the copy of the event in the machine may serve as a backup until each event is confirmed at having reached a fault tolerant destination host.*
* *Via suitable event definitions, keep the number of new events logged during normal operation to a* ***minimum*** *and consider rate limiting methods on events susceptible to spamming and runaway.*

*Design requirements:*

* *Suitable for implementation in direct memory mapped NV memory (i.e. memory which does not have a memory manager, heap, or garbage collector.*
* *Events must have a serial number.*
* *Events must be auditable (e.g. via serial and [chained hash])*
* *Machine generated events must be read-only to users.*
* *[the ability for a user to create a sub-event log managed by the machine but equally auditable] i.e. each QCOM user may have their own event log*
* *[the ability for a user to add custom events into the event log as a special privilege]*
* *Where user events are possible, it must be able to distinguish between machine generated events and user generated events. A user must not be able to fake a machine generated event.*
* *The ability for users to subscribe to events of interest and for the user to be able to account for receipt of all of them. Users can setup machine events for which they receive call-backs on via hooked scripts.*
* *The ability for a permitted user to generate their own events and add to the machine’s event log.*
* *A range of events need to be accessible / shared between users; e.g. cashless events & progressive jackpots. The solution must have multi-user support.*
* *Support RUGMS and multi-game machines.*
* *Previously logged events must be recoverable by a user so long as they have not yet been over-written.*
* *Give consideration to event spamming/runaway in their design and management.*
* *In the longer term, possibly provide support for a range of event forwarding protocols (e.g. possibly rsyslog, RELP):* 
  + *machine facilitated methods / protocols for sending events securely and/or reliably to one or more hosts*
  + *protocols utilised must have 100% reliability*
  + *Ability to feed events directly into a remote database*
* *Some events may require a higher level integrity/security/longevity. For example significant progressive win or cash transfer events may benefit from the ability to be authenticated and/or held in memory for longer.*
* *Events must be prioritised. Spamming a low priority event must not cause important events to be overwritten before their time.*
* *The machine must be able to store the last occurrence of each event type at a minimum.*

*Design mandates:*

* *New events must be defined with consideration to keeping trigger frequency to a minimum. Events prone to high frequency may better as state events (refer s14).*

## The QCOM Event Buffer

A QCOM machine must have one persistent QCOM event buffer maintained in the NV memory of the machine[[287]](#footnote-70) for the storage of QCOM events defined under section 13.6. cp:[[288]](#endnote-218)

The QCOM event buffer size must be capable of storing exactly **255** **events**. cp:[[289]](#endnote-219)

A machine must log new events to the QCOM event buffer as per each event’s definition (refer 13.6). cp:[[290]](#endnote-220)

The QCOM event buffer must be updated via a methodology makes it immune w.r.t power dispruptions that could cause an incomplete or corrupt update, or that results in invalid event indices. cp:[[291]](#endnote-221)

The QCOM 3 SDK contains a C & Lua example implementation of the QCOM Event buffer.

The QCOM event buffer may be parsed via the QCOM API provided. Refer section 11.22.

Event forwarding.

A QCOM 3 machine has no hard-coded methodologies for the forwarding of events over the network. Functionality in this area is added by authorised QCOM users in order to meet their own operational / business needs and any local requirements. For more information refer section 13.8.

## Event Retention

The QCOM 3 SDK contains a C & Lua example implementation of the QCOM Event buffer that implements the requirements in this sub-section.

QCOM events must be retained by the machine for as long as possible before being overwritten by new events as permitted by the requirements in this section. Starting at factory defaults, the machine event log fills to capacity and then it remains full for the operational life of the machine.

QCOM events must only be overwritten by the machine, one event at a time, in order to make space for each new QCOM event logged. Events must only be overwritten by the machine once the QCOM event buffer is “full” (however what defines an event full buffer is a little unorthodox based on the requirements, see below).

Note that neither QCOM users nor the QMA can directly delete or purge QCOM events.

**QCOM 3 proposes a new prioritised approach with respect to the order in which old events are be overwritten by new events in the QCOM event buffer described in the following paragraphs.**

For every event generated by a QCOM machine, there must be **two** parameters (defined below) which control how long the machine will retain the particular event in the machine’s QCOM event buffer before overwriting it with a new event. They are:

* Store Last x (SLx)

There must be one SLx value per event category. Refer QCOM summary xlsx : ‘Event-Cat’ worksheet for the list of event categories.

For example, if SLx was set to 20 for ‘cash’ type category events (e.g. Cancel Credit events), then the machine must always ensure it retains (at a minimum) the last 20 ‘cash’ category events for the machine. **An event must not be overwritten by a new event while it is still a part of the ‘store last x’ quota, no exceptions.**

Each event category has a factory default SLx value (refer QCOM summary spreadsheet for default values). However, the default value for any given event category may be changed by an authorised user via the QCOM API qcom\_eventsSetSLx() function.

* Time To Live (TTL)

There is one TTL value per event in the QCOM event buffer.

TTL is in units of seconds; however the value does not refer to a time ‘interval’, instead it denotes a specific **machine operating time[[292]](#footnote-71)** endpoint, the MOT at which the event ttl is considered expired. Once an event’s TTL has been exceeded by the machine ‘operating time’, then the event becomes eligible to be overwritten by new events, i.e:

*if (event.ttl < qcom\_machineOperatingTime()) then*

*-- the event is ‘aged’ and thus may now be overwritten by new events as long as it is not also still a part of SLx quota.*

On event creation, the machine could initialise an event’s TTL for example:

*newEvent.ttl = qcom\_machineOperatingTime() + defaultEventTTL*

The default TTL (*defaultEventTTL*) interval period for all events must be **zero**. This default value cannot be changed and must be hardcoded into the machine. However, the actual TTL value for any event in the event buffer may be changed by an authorised QCOM user via the QCOM API function: qcom\_eventsSetTTL(). It permits important events to be retained by the machine for longer.

**The primary rule concerning event TTL is that a machine must not overwrite an event whose TTL has not yet expired, bar one exception, see \* below.**

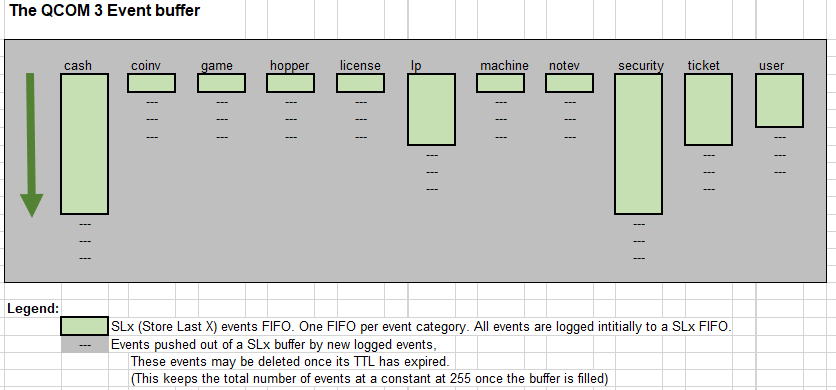
During normal operation (once the QCOM event buffer is filled), when a new QCOM event is generated, the machine must, in the first instance, search and overwrite an event in the buffer whose SLx & TTL parameters have both expired. When the machine searches for an event to overwrite, the search must start from the oldest event and work forward (wrt event serial numbers, not timestamps) stopping at the first event whose SLx & TTL parameters have both expired. cp:[[293]](#endnote-222)

Edge case: once an event category’s SLx FIFO is filled, each new logged event in the same category will push an old event out of that SLx FIFO. This old event must be included in the above search for an expired TTL. cp:[[294]](#endnote-223)

\* There is one exception whereby the machine may overwrite an event whose TTL has not yet expired. This occurs when all TTL values for events in the event buffer (which are also not a part of a SLx quota) are yet to expire. In this situation, the machine must find and overwrite the event whose TTL is closest to expire**.** cp:[[295]](#endnote-224) When there are multiple events with equal TTL values, the machine must pick the oldest (wrt event serial numbers, not timestamps). cp:[[296]](#endnote-225)

QCOM users can discover the event that was last overwritten via the QCOM API function qcom\_eventsGetLast().

**A machine must never overwrite, under any circumstances, an event which is still a part of a “store last x” (SLx) quota**.*The QCOM API function qcom\_eventsSetSLx() handler must ensure that the total number of SLx values are less than the total event buffer size. Refer section 11.22.7 for more information.*



### Smart event retention using TTL

The main purpose and benefit of the above two parameters and requirements is to provide the option of implementing a prioritised approach to the process of overwriting old events with new events in the QCOM event buffer. It allows the machine to retain some events much longer than others, based not only on the event type, but also specific event data content, at the discretion of an authorised QCOM user.

The remainder of this subsection is all examples relating to smart event retention using TTL.

For example, it is possible for a QCOM user authorised to set event TTL’s, to hook a call-back function into all the cash movement events (e.g. cancel credit, ECT, etc.) which implements a weighted approached to assigning an event TTL based on the cash amount in the event. Thus an ECT / CC event of $1 can be assigned a much smaller TTL than an ECT / CC event of say $9,000. Thus allows more critical events to be retained even during periods of heavy event load or spamming.

Following is an example Lua script that could be hooked to ECT event generation which assigns an event TTL in proportion to the amount of the ECT:

function setECTEventTTL(event)

local tm = 7257600 -- ~3 months in seconds

local TTL = event.data.amount / 1000000 \* tm

qcom\_eventSetTTL(event.sn, qcom\_machineOperatingTime() + TTL)

end

With the above function, an ECT of $1.00 would have a TTL of 725 seconds and an ECT of $10,000 would have a TTL of approximately 3 months. *An expired TTL does not necessarily mean the event will be deleted immediately, as events are only overwritten by the generation of new events. If there are no new events, then currently stored events will stay in the log indefinitely. An expired event TTL simply means the event is eligible for deletion.*

The above was a formulaic approach to setting a TTL for Cancel Credit events. A threshold based approach may also be considered.

Yet another example of smart event retention (via custom TTL prioritisation based on event content) for large win or linked progressive award events. A custom setEventTTL function could be utilised for those event types which use both the “event.data” win amount and the “event.data” probability field (in the case of LP awards) in order to set a suitable TTL for the event. This can have significant benefit to integrity and security by retaining critical events for significantly longer for verification purposes. In some cases, for very high value or improbable jackpot events, a TTL in the order of years may be prudent. There would be no risk in these events polluting the event buffer due to how infrequent they are on the machine.

For example, a game may have a linked jackpot of $1,000,000.00 where the likelihood of the event occurring in the life of the machine is known to be very low. Setting this event’s TTL in the order of 5 years ensures evidence of the event is preserved in the machine for an adequate period. It would be impossible for a user to force the event to be purged by spamming the generation of events, as the event’s TTL will protect the event from being overwritten.

In the case of door accessed events, processor door access events could be given higher TTL than for example, a belly door. An attacker trying to remove evidence of processor door access can’t do this by simply spamming other events as they can only purge out processor door events by generating even more door access events, which would draw even more attention.

Finally, a custom event TTL function as mentioned in the examples above could also have the capability of indefinitely storing events with suspect or corrupt data for diagnostic purposes. For example sanity checks on events may be performed locally on the machine by a call-back script and if a suspect event is found, the event can be assigned an elevated TTL for longer retention for subsequent diagnostic purposes.

### Closing remarks on event retention

SLx and TTL assignment functions can be adjusted dynamically as required. If there is an operation or regulatory requirement to store certain events longer or via different rules, then SLx and TTL assignment functions can be easily adjusted.

QCOM users also have the option of maintaining their own event buffer by copying events into their user environment on the machine. Events can be saved by individual QCOM users via Persistent Variable support (refer 10.10).

It is expected for security reasons that the role of managing SLx and TTL assignments functions will typically stay with the QMA. This role, if delegated should only be given to a commercially neutral[[297]](#footnote-72), trusted QCOM user.

*Design notes related to this approach to event retention:*

* *The event buffer fragmentation - as the size of the QCOM event buffer is quite small, there will be no need for any defragging as the CPU impact will be a negligible event in any scenario.*
* *The implementation is more complex than for example a vanilla circular or FIFO buffer. To assist in development, example implementations in both “C” and Lua source code is provided in the QCOM 3 SDK.*

## Event Buffer Full Condition

With QCOM’s smart event retention methodology, a traditional event buffer full condition does not strictly exist in QCOM 3 (like for example in QCOM 1.x).

The most applicable situation to an event buffer full with respect to the QCOM event buffer prescribed in this section, is when the machine starts overwriting events whose TTL has not yet expired.

The machine must keep a tally of the number of times it has overwritten an event whose TTL has not yet expired (refer *hms.eventb.fullcnt* and is reported via the qcom\_eventsStatus() API function) and must also indicate each occurance in the **EVENT** state event via the *\_wasfull* field. cp:[[298]](#endnote-226)

The normal operating mode is that the machine’s QCOM event buffer fills to capacity and then remains full, from which point on, one new event means one event overwritten.

Under QCOM, each user may implement any desired functionality concerning any arbitrary defined event full condition / event / lockup and action to take, according to any local requirements or operational needs (e.g. play disable based on some user defined event full condition). This allows for a more flexible approach to event management across a wide range of jurisdictions and possible operating modes.

The same applies to any possible action taken upon an arbitrarily defined QCOM event buffer high level threshold detection (e.g. warning event / play disable). The detection methodology and action to take is able to be arbitrarily implemented by concerned / authorised QCOM users. The machine has no hard-coded functionality in this regard, which is intended.

*Unless the machine is in an offline (or standalone) mode of operation, it is recommended that there always be an event buffer high level threshold which disables the machine’s primary function (e.g. play). The QMA should ensure that this is implemented by either themselves or a delegated QCOM user.*

*QCOM users also have the option of making copies of any events of interest via their QCOM persistent variable memory pool if they wish to store an event for longer or using a different methodology than has been setup by the QMA or delegated user.*

## Event Buffer QCOM API

The QCOM event buffer QCOM API provides functions to parse through events in the QCOM Event Buffer and adjust values for event TTL and event category SLx.

Refer “*qcom\_events*” prefixed QCOM API functions for more information.

## QCOM event schema

When QCOM users working with QCOM events via the QCOM API, events are represented by a Lua data structure in the form of an associative indexed *table* (the Lua standard *table* data type) which is defined as follows:

Key Lua Type Description

sn integer Machine global event serial number

csn integer Event serial number with respect to event category

tt integer Time the event was logged. Units of seconds.

The value must be a POSIX compliant Epoch time[[299]](#footnote-73) and must work correctly with the Lua os.date() library function. cp:[[300]](#endnote-227)

bias integer Local time bias wrt tt.

Units of minutes. Same units and offset direction as per the QCOM API function qcom\_timeSetTimezone() s11.5.4.

This amount when added to ‘tt’ must give the timestamp of the event when it was logged in the machine in local time.

To get the local time QCOM users can do the following for example:

In a table : **os.date("!\*t", tt + bias\*60)**

As a string: **os.date("!%c", tt + bias\*60)[[301]](#footnote-74)**

eventid string Event ID

cat string event category. Refer QCOM 3 xlsx: ‘Event-Cats’ worksheet.

ttl integer Time To Live

This value is in terms of a machine operating time endpoint that denotes the MOT at which the ttl expires.

Units must be in seconds.

*When hooking onto an event at creation, the user’s call-back script may see the default TTL or an adjusted TTL value depending on the order of execution of hooked scripts.*

data table | nil arbitrary / extended event data as key-value pairs

data = nil for many event types; Cancel credits and jackpot events are examples of events with *data*. The specific *data* will depend on the event type. Refer QCOM 3 summary xlsx – Events worksheet for all event data.

Notes:

* QCOM 3 machines must implement all the above fields cp:[[302]](#endnote-228) plus a few additional fields will be required (e.g. *next*, *prev* event indexes) in order to fully implement the QCOM event buffer. In this regard, a full Lua QCOM event schema may be found in the QCOM 3 SDK module: ***hms\_schema.lua***.
* Systems should not have to store in the long term the fields: csn, cat, ttl.

## QCOM Event Definitions

The majority of QCOM v1 events will have an equivalent in QCOM 3. Refer to the QCOM 3 summary spreadsheet document: “events” worksheet for a full list of event types and data definitions as well as obsolete / removedevents with respect to QCOM v1.

### The meaning re ‘log the’.

Throughout this document and the QCOM 3 summary spreadsheet, when the term ‘***log the <event\_name>***’ is used with respect to a QCOM event defined under this section (s13). This is not a single operation. It requires the machine to:

1. Update NV memory first, i.e., Log the <event\_name> event to the (s13 defined) NV event buffer and save it to NV RAM. This must be an all-or-nothing operation.
2. If applicable, log any associated state event to the machine’s state event FIFO (e.g., **DOOR\_OPEN** state event) as state events must be logged first. Also refer section 14.3 for examples.
3. Finally, log the **EVENT** state event (containing the event name and associated data) to the machine’s state event FIFO which will ensure the QLE Lua software driver’s copy of the machine event buffer is also updated.

Related: Section 14.3 - State Event Timing

## Implementation Notes

This section deals with possible methods of implementation of a QCOM event buffer.

The QCOM 3 SDK contains a C & Lua example implementations of the QCOM Event buffer.

The machine **must** not use event timestamps for any purpose other than informative purposes. Any reference to the oldest / newest / next / previous event, or parsing the event buffer chronologically must be with respect to event serial numbers.

The machine should have some form of master enumerated **event definition list** (one entry per event type) with at least the following fields:

* Event ID
* Event Display Descriptor (human readable – refer QCOM summary spreadsheet)
* [Number of times each event has been logged]

The machine should have some form of master **event category list** (one entry per event category) with at least the following fields:

* Event Category ID/string descriptor
* The current SLx for the given event category
* RAM clear default value for the event category
* Number of times events for the given category have been logged (used to generate the event category serial numbers)

The machine should have some form of buffer of **events** (i.e the actual main event buffer) with each event containing the following fields in addition to most of the fields in the QCOM Event Schema listed in section 13.5:

* Index / pointer of next most recent event in the buffer (with respect to the event serial number, not the event timestamp)
* Index / pointer of previous most recent event in the buffer (with respect to the event serial number, not the event timestamp)

Finally the machine will encompass this in a master structure or class which would have at least the following fields:

* Index / pointer to the most recent event in the buffer (with respect to the event serial number, not the event timestamp)
* Index / pointer to the oldest event in the buffer (with respect to the event serial number, not the event timestamp)
* A once-off number of events counter, used as the event buffer populates itself the first time after a machine RAM clear
* Global event counter (used to generate the next event serial number)

An array-based event buffer is a suitable approach for the implementation of the QCOM event buffer, being especially friendly toward direct memory mapped NV memory types (i.e. no heap or memory manager). Given the required methods needed to parse through the buffer, both next / prev pointers / indices (i.e. two way lists) make for the most efficient implementation.

In the implementation of the "which event to overwrite" decision process in relation to maintaining SLx event number quotas, it is a requirement that the decision process **must** take into account the category of the incoming event. (This behaviour is being mandated to ensure deterministic behaviour across all manufacturer implementations of the QCOM event buffer which allows for automated testing of QCOM event buffer implementations.)

Example QCOM 3 event buffer implementations in both “C” and Lua source code is provided in the QCOM 3 SDK.

## Event Forwarding

QCOM users (aka gaming related service providers) will typically require methods which forward events generated by the machine or their own scripts, onto a remote server or host. In QCOM the intent is to offer multiple event forwarding methodologies via the QCOM API which may be used at the discretion of the individual QCOM user. This allows any service provider to meet their specific requirements or operational needs.

This section briefly lists what options are available for the purpose of event forwarding.

Available options:

* QCOM users can utilise all available communications API in the QCOM Lua Engine for the purpose of forwarding events.
* In the longer term, it is feasible that the QCOM Lua Engine may make available a number of database APIs allowing a QCOM user to connect to and dump events directly into a database of a remote system.

Each option will be available with or without encryption / authentication.

Other possible methods of event forwarding considered were:

* https publishing of events on the machine’s QCOM web server
* Addition of an scp Lua API / wrapper
* Addition of an rsyslog (event forwarding service) Lua API / wrapper
* Addition of an RELP (Reliable Event Logging Library) Lua API / wrapper

## Event SPAM Reduction

This section relates to a generic proposal for event SPAM reduction. Event Spam Reduction is independent to the event storage methodology contained earlier in this chapter.

A problem with gaming machine monitoring is the amount of events that can be generated by the machines during operation. Depending on event definitions, the large number of events potentially generated can be a burden and a cost to any associated system. Some event types can also be intentionally spammed.

One potentially large source of events from gaming machines are various types of machine **door** open/close events and other ‘paired’ type events, able to be spammed either intentionally or as result of faulty hardware. The intent of this section is to define and propose **a hysteresis feature** **on door closure type events** for the purpose of reducing the frequency of such events that a machine can log in any given timeframe but without any loss of machine security.

The feature as defined in this section has the following benefits**:**

* It reduces the number of events in the machine and associated systems (i.e. reduces storage costs)
* It does not reduce machine integrity or security.
* It has the ability to filter out superfluous events created by faulty hardware (e.g. fault switches and difficult to close doors) or intentional spamming.
* Implemented in software so it can be changed if need be.

The requirements in this section are currently **only mandatory** for all types of machine **door** open/closed events[[303]](#footnote-75) cp:[[304]](#endnote-229), however this may be extended to other event types susceptible to spam in the future.

The remainder of this section will only refer to door events for the purpose of illustrating the requirements.

The primary requirement is as follows:

After a door has been opened on the machine, upon next door closure detection, the machine must not log the door closed event or any further door open events, or resume operations as a result of the door closure, until the door has been continually detected in the closed state for a timeout period. This timeout period must be set at no less than **2 seconds**. cp:[[305]](#endnote-230) This does not apply to door closures that may occur while powered off, or during boot, up until the point door monitoring has commenced by machine software.

If the door is physically re-opened during the time-out period, this must reset the hysteresis timer for the given door. The door closure event is not logged nor is a new door open event logged. From the event buffer’s point of view, the door is deemed to be open for the entire time, i.e. there is only a single door open event recorded. cp:[[306]](#endnote-231)

However despite the hysteresis, the actual physical state of the door must still be indicated to the person operating the door in real-time on the machine’s built in display. cp:[[307]](#endnote-232)

Accordingly, from the point of view of the person operating the door, upon closing the door, a message or indicator should appear on the machine’s display asap / near real-time, along the lines of “**door closing…**” or similar message or clear indication that the door is closing[[308]](#footnote-76) for the duration the timeout period. If the door is opened during the timeout period, then the machine must immediately remove the message and return to the door open state. Once the timeout period has expired, the message must be removed and the machine may only then resume programmed operations as a result of a door closure (provided no other doors are open). cp:[[309]](#endnote-233) optional

The above real-time in-machine physical door state display may be useful. As without this feedback, the ‘lag’ created by the ‘hysteresis’ may be confusing to a human operator.

If the machine loses power during the timeout period, the EGM must consider itself as powered down with the door open. cp:[[310]](#endnote-234)

Only once a door has been consistently closed for the time-out period, may a machine log a door closed event for the door and allow logging of new door open event again for the given door. cp:[[311]](#endnote-235) The door closed event time-stamp and serial number must correspond to the time in which the event is actually finally logged to the event buffer after the hytersis timer has exipired. cp:[[312]](#endnote-236) This requirement overrides any other requirement in this document concerning event timestamps having to correspond to the events defnition.

Example:

1. Door open at time 0 seconds. EGM logs this as per the event defintion.
2. Door close at time 1 second. EGM **does not** log the door closed events yet and starts a 2 second countdown. The EGM still considers (logically speaking) the door to still be opened during this countdown.
3. Door open at time 3 seconds. EGM **aborts** the 2 second countdown. It’s should be like the door closed never occurred in the EGM w.r.t. QCOM.
4. Door close at time 10 seconds. Again, the EGM **does not** log the door closed events yet and starts a new 2 second countdown. The EGM still considers (logically speaking) the door to still be opened during this countdown.
5. After 2 more seconds, if nothing has changed wrt the door state (i.e. it’s still closed), the EGM finally logs the door closed event with the current time.

Implemented correctly, door open/close events always pair (i.e. alternate 1 to 1).

NB In this section the word “log” means, for example w.r.t a main door open is the actual events logged. Example: main door open; the order of operations here should be:

1. Log the **MAIN\_DOOR\_OPENED** machine event to the NV event buffer (s13) and save it to NV RAM.
2. The **DOOR OPEN** state event is then logged to the state event FIFO.
3. The **EVENT** state event (containing the **MAIN\_DOOR\_OPENED** NV event) is then logged to the state event FIFO.

Related: s13.6.1.

# State Events

*Note: almost all requirements in this section are implemented as a part of the QLE Lua software driver. Related: section 33.1.*

*Design Requirements:*

* *The ability to attach a script to a state event to be executed upon each occurrence by the machine.*
* *Multiple users must be able to hook onto the same state event.*
* *The ability for a script to detect that is has not been run out of context.*

QCOM 3 state events are unrelated to the events in section 13.

State events must be generated by the machine as defined by the QCOM 3 specification as the machine executes its primary application. State events must be FIFO buffered in ordinary RAM (i.e. do not use NV memory for the state event FIFO). The machine’s QLE thread must read from the buffer and dispatch to the QLE Lua software driver where any hooked QCOM user event handlers will execute. The machine’s QLE thread should sleep while there are no state events in the state event buffer. cp:[[313]](#endnote-237)

QCOM 3 state events are defined (i.e. when to push into the FIFO) in the QCOM summary spreadsheet – “QCOM 3 State Events” worksheet.

State events may be deleted any time after the QLE thread has finished with them.

The state events / state event FIFO must not persist across machine restarts. cp:[[314]](#endnote-238)

Note: QCOM 3 **state events** are NOT **machine events** (s13); they are two entirely distinct / mutually exclusive classes of events in QCOM 3. In this document, “state events” will always be referred to as “state events”; any other mention of an event in the context of a buffered or logged event is a reference to a machine event under section 13.

* State events are not fault conditions or lockups and are **not** logged to the NV event buffer referred to in section 13.
* The only link between state events and section 13 events, is that the **EVENT** state event is used to carry events defined under section 13 into the QLE.

State events have two purposes:

* To keep the copy of the machine state held inside the QLE Lua state instance up to date. (It is this machine state that QCOM users “see”)
* QCOM user script progression. By allowing QCOM users to hook onto state events as execution triggers for their scripts.

QCOM users may attach (or hook) a script to QCOM state events if given privilege to the QCOM API function below.

qcom\_luaHookScript()

For more information refer section 11.15.4.

When a given state event is triggered (as per its definition) and sent to the QLE by the machine, the QLE Lua software driver will ensure that all QCOM user scripts hooked to that event will be executed by the machine back-to-back.

State events must be dispatched in the QCOM Lua Engine in the order in which they occur.

At this time no guaranteed order of script execution (with respect to multiple hooked users on the same event) implemented by the QLE Lua software driver.

NB: The the QLE Lua software driver ensures the host machine’s state appears frozen to QCOM user scripts during execution. From the perspective of QCOM users, the state of the host machine will only change between successive state events. Refer section 10.1 for more information.

Meters will balance

The QLE Lua software driver will ensure that whenever control is passed by the machine to a hooked script, all machine meters accessible to QCOM users will be in a “balanced state” i.e. any meter audit formulas applied will not fail e.g. the machine’s credit meter will equal the sum of relevant machine meters and progressive prize values should be able to be reconciled against relevant progressive prize meters.

Call-back functions

In the implementation of the QCOM 3 state event dispatcher, there will be some special types of state events which are created and dispatched to specific QCOM user’s handler scripts. This is in relation to “call-back” functions concerning a number of QCOM 3 APIs. Call-back functions are used to avoid QCOM API functions from blocking (e.g. QCOM Communications API - section 12) and other QCOM API functions that require a call-back function as a parameter. These types of special state events are listed in the QCOM State Event Summary table. These state events have special properties such as the destination QCOM user and call-back function and sometimes additional application specific data.

Related: qcom\_egmState() s11.19.20.

## State Event Definitions

QCOM 3 state events are defined in the QCOM summary spreadsheet – “QCOM 3 State Events” – ‘state-events’ sheet.

## State Event Transitions

For this section, please refer to the state event diagrams in the QCOM summary spreadsheet – “SE Diagrams” sheet which show all state events and how they must transit.

**These three diagrams represent a set of requirements regarding state event transitions as seen by QCOM users.**

With respect to the first state transition diagram, no state transitions may occur if the machine is not ok[[315]](#footnote-77).

From the point of view of a QCOM user, the machine must setup the QLE to the same state (ref qcom\_egmState(), s11.19.20) after a power fail or restart.

The QLE Lua software driver is specifically **hardcoded** to ensure that the transition order of state events and states reported by qcom\_egmState(), as seen by QCOM user scripts, is always exactly as permitted by the QCOM 3 state diagrams. Refer to the QLE Lua software driver in the QCOM 3 SDK for a full Lua implementation of the sanity checks required.

### Illegal State Transitions

An illegal state transition in this document refers to state transitions as seen by QCOM users as a result of the host machine logging state events. It is accepted that the QCOM 3 machine’s application has its own “states” and likely a lot more of them; for the definitions of QCOM 3 machine states events (specific to EGMs) it is intended these host machine application states can be mapped easily onto the required QCOM 3 states seen by QCOM users inside the QLE.

Allowable state transitions as seen by QCOM users are shown in the QCOM 3 state diagrams located in the QCOM 3 Summary spreadsheet.

Illegal state transitions are detected by the QLE Lua software driver based on state event received from the machine.

In other words an illegal state transition in QCOM 3 essentially means that the host machine has logged state events in an order for which the QLE Lua software driver considers illegal.

If the QLE Lua software driver receives state events from the host machine that would direct it to perform what it considers an illegal state transition, the QLE Lua software driver will stop dispatching events to QCOM users (until next restart) and send the ***“panic”*** message back to the host machine. In response to the panic message, the machine must then behave as per the ***“panic”*** message handler’s description in the QCOM 3 Summary spreadsheet. cp:[[316]](#endnote-239)

Illegal state transitions are only ever likely to occur during the implementation of QCOM 3 in a machine and are not expected to occur in production.

This condition is treated as critical because most QCOM user’s scripts will rely on QCOM state event transitions occurring exactly as defined in this specification. The QLE Lua software driver will never let QCOM users see what it considers to be an illegal state transition as this will cause QCOM user scripts to get confused.

Related:

* The QLE Lua software driver contains all the state transitions sanity checks that are performed by it based on the three state transition diagrams.
* qcom\_egmState() s11.19.20.

## State Event Timing

To ensure the requirement that QCOM user scripts do not have to be machine manufacturer specific (refer s10.1), QCOM state event timing (i.e. when state events trigger) is critical and must be precisely defined.

State event timing is typically included as a part of the event’s definition (refer to the QCOM summary spreadsheet), however additional state event timing related rules are summarised in this sub-section.

Refer to the state event diagram in the QCOM summary spreadsheet – “SE Diagrams” sheet, state transition diagrams (there are three). **The state transitions shown in these diagrams are a requirement.** The machine must only generate state events in the possible orders displayed in the diagrams. The QLE sanity checks this, refer previous section on illegal state transitions.

* State events associated with meter updates

If there are any meter updates associated with the trigger of a state event, then the meters (as seen by QCOM users) will be updated **prior** to dispatching the state event to any QCOM users. This is handled by the QLE provided in the QCOM 3 SDK.

* State events associated with machine state changes

If there are any machine state changes associated with the trigger of a state event, then the machine state (as seen by QCOM users) will be updated **prior** to dispatching the state event to any QCOM users. This is handled by the QLE provided in the QCOM 3 SDK.

* State events sharing the same trigger definition

If there are two or more state events which share the same trigger definition, then the order in which the state events must be thrown will be clarified in each case within the state events definition. Refer to the QCOM 3 summary spreadsheet for state event definitions.

* State events vs events (section 13)

In a QCOM 3 machine, each new machine event (re section 13) must generate an **EVENT** state event. When a state event and event (re section 13) both have the same trigger definition (e.g. cancel credit) then the machine must log the state event (e.g. **CANCEL\_CREDIT**) before the corresponding **EVENT** state event. The two state events must be logged back-to-back by the machine in the same process/thread. cp:[[317]](#endnote-240)

See examples on the next page.

Examples:

**Cancel Credit**; i.e. successful call to the QCOM API function qcom\_cancelCredit() occurs. Order of QCOM 3 related actions & events: (copy of this e.g. in s11.24.1)

1. Update NV memory; all-or-nothing
   1. The machine updates applicable NV meters. (i.e. deduct amount from credit meter and add it to CC meter and commit to NV memory)
   2. Log the **CANCEL\_CREDIT** machine event to the event buffer (s13.1) and commit to NV memory.
2. Once the machine’s NV memory is irrevocably updated then…
   1. Log the **CANCEL\_CREDIT** state event and data.
   2. Log the **EVENT** state event (containing the **CANCEL\_CREDIT** machine event & data).

The above operations, once started, **must complete as if it were all a single instantaneous / uninterruptable operation in the machine**, even for example if a door open, or fault condition etc. occurred during the process. cp:[[318]](#endnote-241)

Note: a power down / CPU reset is permitted to interrupt operations not affecting NV memory updates (i.e. point 2 above)

**Door open event**; order of QCOM 3 related actions & events:

1. Log the **DOOR\_OPENED** machine event to the event buffer (s13.1) and commit to NV memory.
2. Once the machine’s NV memory is irrevocably updated then
   1. Log the **DOOR\_OPENED** state event.
   2. Log the **EVENT** state event (containing the **DOOR\_OPENED** machine event).

**QSIM generated examples:**

Start up QSIM 3 and simulate a main door open, the following examples what **state events** must be generated in this case:

5 7.657 dispatchStateEvent() : DOOR\_OPENED {door="main"}  
6 7.657 dispatchStateEvent() : EVENT {\_epochtime=1560228196,\_sn=20,\_cat="security",\_mot=13233,\_eventid="MAIN\_DOOR\_OPENED",\_ttl=86400}

Upon a subsequent main door close:

7 13.047 dispatchStateEvent() : DOOR\_CLOSED {door="main"}  
8 13.047 dispatchStateEvent() : EVENT {\_epochtime=1560228202,\_sn=21,\_cat="security",\_mot=13239,\_eventid="MAIN\_DOOR\_CLOSED",\_ttl=86400}

## Sync events

Sync events are **state events** that have been designated as such in QCOM 3. State events designated as sync events are indicated in:

* the QCOM 3 Summary spreadsheet:
  + “state-event” &
  + “Sync SE's” (includes use/test cases for sync events) &
  + state diagrams worksheets; and
* the QCOM 3 SDK: **syncevents.lua** module.

Only a small subset of overall state events designated as sync events.

Sync state events exist in QCOM 3 to allow QCOM users to process and react to state events **before** the machine continues a *specific operation*. The *specific operation* here is the machine’s processing of any further human user input and any state changes that would change the return value of the QCOM API function qcom.egmState() (s11.19.20).

Example: On an EGM with credit, the player pushes collect and immediately pushes a play button. The collect press with credit generates a **COLLECT\_WITH\_CREDIT** state event. Now because in QCOM 3 **COLLECT\_WITH\_CREDIT** is a sync event, the machine must ensure all QCOM user scripts hooked to the **COLLECT\_WITH\_CREDIT** finish executing before the it processes the play button press**.** This allows a QCOM user to direct the machine into for example a hopper collect where the play button push would then be ignored.

Use cases for all sync events may be found in the QCOM 3 Summary spreadsheet – “sync SE’s” worksheet.

The need for sync events stems from the fact that QCOM 3 state events are intended to be buffered and processed in a dedicated QLE thread which can result in the QLE lagging behind the machine somewhat (msecs). Sync events give the QCOM 3 QLE thread a change to catch up at certain times.

As soon as a sync state event is logged by the machine, it must temporarily suspend its human UI input\action processing (or the equivalent, or other designated action[[319]](#footnote-78)) and any state changes that would change the return value of the QCOM API function qcom.egmState() (s11.19.20) until all scripts hooked to the sync state event have finished executing which is indicated to the machine when it processes the corresponding ‘*syncevent*’ sendToHost message. cp:[[320]](#endnote-242)

Typical sync times (the time it talks the QLE to finish executing scripts on a given state event) are in the order of **micro-seconds or less**.

If sync events implemented correctly, then sync event delays will and must not affect things like machine animations, sound playback etc. In the worse scenario, e.g. if the QLE was lagging very badly (e.g. a test script will be provided to help test this)\*, it might be possible on some older machines for a human user to notice a slight delay before the machine responded to their action.[[321]](#footnote-79) *(****\*****But* ***i****t is expected on most machines that this won’t be a condition that is even possible to create without a specifically crafted development code base.)*

In summary, sync events should only negligibly delay the machine’s processing of user interface (UI) events/actions and nothing else. In other words, a thread that posts a sync event in the machine doesn’t wait, but until the QLE Lua software driver has finished processing the sync event (after which it will send a message, see below), the machine must process no further human-based UI input/actions in the interim, or otherwise change state.

The QLE Lua software driver indicates that is finished processing scripts pertaining to a given sync state event by sending the following message back to the machine:

“syncEvent” : {seid:string} -- seid denotes the sync event descriptor

Example (same as above but more detail). The state event **IDLE\_MODE\_ENTRY**; it is a sync state event. So what happens is:

* EGM enters “idle mode” and logs the state event **IDLE\_MODE\_ENTRY**.
* The EGM may not act on human user input (e.g. press a play button) or any action or that would cause the EGM to leave idle mode until it receives the following QLE to EGM message back from the QLE:

**syncEvent {seid=”IDLE\_MODE\_ENTRY”}**

## QCOM Event Dispatcher

The QCOM Lua Engine is event driven and as such there needs to be what is termed a ‘QCOM event dispatcher’ provided by the host machine. The QCOM event dispatcher must be a FIFO buffer and is suggested it be run in its own thread. The QCOM event dispatcher simply waits for new state events on the machine to occur then dispatches them onto the QLE Lua software driver via a Lua C API protected mode Lua function call[[322]](#footnote-80). It sleeps otherwise. The act of logging a new state event in the machine should wake up the thread.

Refer to the QCOM 3 SDK for an example QCOM event dispatcher written in “C”.

Below is the list of state event receiver functions available in the QLE Lua software driver

dispatchStateEvent(seid:string, t:table)

dispatchSocketEvent(seid:string, t:table)

dispatchUARTevent(seid:string, t:table)

dispatchOneSecTick(mot:number)

dispatchQMAscript(scrfunchash:string, username:string)

dispatchUserFunc(username:string, funcname:string[, argstr:string])

luaL\_dostring(string)

Related: Refer to the ‘state-events’ worksheet in the QCOM 3 summary spreadsheet for state event to receiver function associations and requirements.

Upon receipt of each state event, the QLE Lua software driver will **execute** any hooked scripts in the respective QCOM user’s script jail / environment and given level of privileges. Related section 10.1.

QCOM Event Dispatcher CPU Utilisation

While the QCOM event dispatcher is waiting for new events to occur it must use negligible CPU time. Ideally its thread should sleep when idle.

QCOM Event Dispatcher host process priority

Refer section 10.1.

Event Dispatch Latency

When a new state event is triggered as defined, which has a script attached while the QCOM event dispatcher was idle, the **desired** maximum delay before the first hooked script begins execution is under **50 milliseconds\***. Exceeding this value when the machine is busy should be acceptable.

Final acceptability of the machine’s **average State Event dispatch time[[323]](#footnote-81)**: If a machine’s average reading is above the limit, but it is evident in machine software and given its current hardware that the machine is not needlessly or purposely delaying the dispatching of QCOM State events, then approval for any given level of average latency will be granted.

\****Note that given the way QCOM 3 now works this value is far from critical. This value will not be finalised until OLGR takes some readings from some real gaming machines.***

Related: QCOM user can see the latency via the QCOM API function qcom\_luaEventTime().

QCOM event dispatcher initial event dispatch order

The QCOM event dispatcher must dispatch scripts initially in the order in which the associated event was triggered.

NB There are no requirements pertaining to the order of execution of multiple scripts hooked to the same event. However there is likely to be control over this in the long term. This order (if any) is controlled by the QLE Lua software driver.

## Special Types of State Events

### Call-back Functions

All call-back functions in the QCOM 3 API are executed similar to any other user script hooked to a State Event. This ensures that the resources utilised by the call-back function count towards the applicable QCOM user (aka the user’s CPU, memory, and instruction quotes). This pertains to:

* The QCOM 3 Communications API (section 12)
* The QCOM 3 UART API (section 11.36)
* Any QCOM API function that accepts an argument of type *function*.

### Internal-Use Only State Events

In the implementation of QCOM 3 the machine will need to throw a number of state events classified as *internal-use-only*. Internal-use-only in that a QCOM user must not be able to hook onto it. Examples:

* The loading and execution of a QCOM user’s start-up / initialisation script (refer section 10.7.1 and QCOM 3 SDK: QLUAE\_LOAD\_USER\_SCRIPT)
* All call-back functions listed in the previous subsection.

All internal-use-only state events start with the prefix of “**QLUAE\_**”.

Refer to the QLE Lua software driver in the QCOM 3 SDK for the complete list of required internal-use-only state events.

## State event buffer full

This section concerns machine requirements relating to:

* How large does the machine’s state event FIFO must be.
* Actions that must occur if the machine’s state event FIFO becomes full.

The QCOM 3 state event buffer should spend most of its time empty under normal operating conditions.

The main contributor of state events in a QCOM 3 machine is IP & UART rx data, note however these sources of state events are throttled.

Ignoring rx data state events; sync events (s14.4) ensure that the machine’s state event buffer does not need to be very large at all.

The machine’s state event FIFO buffer must be sufficiently large to ensure it does not become full even during heavy operation. cp:[[324]](#endnote-243)

If memory is of no concern in the machine, then the machines state event buffer size must be set to **400** events. For machines with limited memory, a much smaller value may be used with permission. cp:[[325]](#endnote-244)

The minimum required size of the machine’s state event FIFO buffer will depend somewhat on the machine’s performance characteristics, notably CPU speed and the speed of its RAM and NV memory store. Each QCOM 3 machine model will undergo benchmarking before a suitable smaller size can be decided for that machine model. OLGR will provide scripts that will help with this task.

If the machine’s detects its state event FIFO buffer is full, the machine must panic as per a software exception and require a restart to fix. cp:[[326]](#endnote-245)

To guarantee the FIFO can never get full under any circumstances, it is acceptable for the machine to implement the following:

If the machine’s state event FIFO current event count exceeds a threshold of maxsize minus **15** events, the state event FIFO logging function may deny the logging of QCOM 3 communications rx data state events with a *denied/try again* return value until the buffer drop back below the threshold. In this case, the machine’s QCOM 3 IP thread should just sleep a short period then try again. For UART threads it is acceptable just to lose the data in this case.

If the above is implemented then the madchine must record statistics how often a state event was denied and display in QCOM related page in EGM audit mode.

Other than what is described above, there must be no other adverse side effects during a state event FIFO full condition in a machine. cp:[[327]](#endnote-246)

# Meters

Definition of “Meter”: refer QCOM v1.

*Design requirements:*

* *Hardcoded QCOM meters will be defined and available at the lowest level. Meters which are the sum of other meters will not be mandated as a hard-coded meter.*
* *QCOM must allow new “meters” (adhering to critical meter requirements) to be created by authorised users which may be the sum of any combination of lower order meters, events, states, or other accessible data.*
* *Allow direct access to commonly needed meters and single meter access via the QCOM API.*
* *Any QCOM API function associated with “Meters” must have the word “Meter” in its function name.*
* *(on hold) The ability to customise a master meter list for the default machine meters display in machine audit mode (re localisation support). This may consist of:*
  + *A call-back function to a meter display script*
  + *The ability to change meter labels (re localisation support) e.g. “banknote”, “bills”, “notes” all mean banknotes. “drop”, “token”, “coins” are other terms used in varying jurisdictions*
  + *The ability for all users to access the localised master meter list*

Support for “Meters” in QCOM 3 is straightforward. Meters are divided into categories which are accessible through associated functions throughout the QCOM API. When parsing through the QCOM API (section 11), any function concerning ‘Meters’ will have the word ‘***Meter*’** embedded in its function name.

## Meter List and Definitions

Refer to the **QCOM 3 Meters** and **QCOM 3 Meter Definitions** summary tables located in the QCOM Summary spreadsheet for the list of meters and associated definitions.

The timing of meter updates is as per QCOM v1 as applicable.

Some meters have been discontinued in QCOM 3, however discontinued meters can still be implemented by QCOM users or the QMA on demand via QCOM persistent variable – meter support (10.11).

## Meter Representation

All versions of Lua support 64 bit double precision floating point numbers which means that with respect to integer maths using floats in Lua there is approximately a limit of 100,000,000,000,000 before any loss of precision starts to occur. QCOM users should keep this in mind as well as the Lua reference manual section on *coercions and conversions*.

Since in Lua v5.3, there are now two *number* types in Lua; one is equivalent to a ‘*double’* in “C” (i.e. 64 bit double precision floating point number) and the other (new to Lua in v5.3) is a 64 bit signed integer number type. For the integer type, this provides a maximum positive integer value of 9,223,372,036,854,775,807 (=2^63-1). Refer to the Lua reference manual for more information.

Machine “meters” as seen by QCOM users inside the QLE Lua software driver are represented by the Lua **64 bit signed integer** *number* type unless the specific meter definition states otherwise.

Since ‘meters’ are typically positive numbers, this gives an effective meter range in QCOM 3 machines of 0…231.

Meters stored in the host machine (outside the QLE Lua software driver) must support a positive range greater than or equal to Lua’s 64 bit signed integer *number* type positive range. cp:[[328]](#endnote-247) I.e. 263-1 = **9223372036854775807**.

It is acceptable if the machine supports a higher maximum meter value.

## Meter Rollover

The meter range is now so large that the issue of meter rollover is arguably irrelevant. Thus a machine utilising 63 bit or higher meters is not required by this specification to have any code to manage a meter rollover.

Meter-too-large sanity check

Machines must implement a **meter-too-large** sanity check on its **credit meter** upon start-up and once per play equivalent to: cp:[[329]](#endnote-248)

if ((absolute\_meter\_value > x) then

a RAM error must result on the machine.

The value ‘x’ may be assigned by the machine manufacturer at this time.

## Self Audit

QCOM 3 EGM must perform a self audit as per the requirements in QCOM v1.6.x. cp:[[330]](#endnote-249)

## Audit Mode Master Meter Display

**Concept proposal only: do not implement the feature in this section at this time.**

Jurisdictions sometimes have specific and differing requirements pertaining to the machine’s master meter display in audit mode. The feature proposed in this section helps cater for possible variants in those requirements.

In QCOM 3, it is possible for an authorised QCOM user to hook into the machine’s audit mode master meter display function and replace the machine’s default master meters list with a different set. Thus when an attendant enters machine audit mode and views the machine master meters, they may see a custom set of meters which for example, would be suited to their regulatory environment e.g. a machine that has a RCR type feature might use this to add RCR meters to the meters display.

This feature should only be privileged out as required to a single authorised user in order to maintain the security and integrity of the machine machine’s master meter display. Typically, it is expected that the privilege to this function will remain with the QMA at all times.

In many cases the customisation simply amounts to changing meter labels and the meter display order.

The relevant QCOM API function is

**qcom\_egmSetMeterDisplayFunction()**

Refer to section 11.19.18 for more information on this QCOM API function.

# System Lockup

*Design Requirements:*

* *At least the equivalent functionality as per QCOM v1 System Lockup*
* *Also see “enhancements” below*

*Required knowledge:*

* *This section assumes the reader has a basic understanding of QCOM v1 System Lockup feature.*

The System Lockup feature in QCOM 3 is the same System Lockup (SL) feature in QCOM v1 but with some significant enhancements.

SL has many possible uses. In QCOM v1 it was mostly used by external jackpots systems to notify a player that they have won a prize. In QCOM 3, SL is now also an integral part of credit redemption (s22).

The following enhancements have been made to SL in QCOM 3:

* Support for simultaneous System Lockups instigated by different QCOM users.
* The credit meter must be clearly visible (and labelled) during a SL.
* SL can now throw a state event after a specific user’s SL has been viewed for a timeout period.
* The ability to refresh a SL in progress with new data without leaving the SL. This will allow for the seamless chaining of questions and statements in a single SL.
* Credit meter deductions (excluding hopper collects) must only be possible while a machine is in a SL. This makes SL an integral part of QCOM 3 ECT (refer section 17), Cancel Credit and Ticket Out (TITO – refer section 18) solutions.

Requirements while a machine is in the SL state are currently located at the end of section 11.25.1.

Summary of SL related functionality in QCOM 3:

* State Events:
  + SYSTEM\_LOCKUP\_ENTRY
  + SYSTEM\_LOCKUP\_EXIT
  + SYSTEM\_LOCKUP
  + SYSTEM\_LOCKUP\_TIMEOUT
  + SYSTEM\_LOCKUP\_CLEARED
  + SYSTEM\_LOCKUP\_RESPONSE
* QCOM API Functions:
  + qcom\_slRequest()
  + qcom\_slReset()
  + qcom\_slStatus()

The above SL related QCOM API functions are described in full in section 11.25.

## Multi-user support

In QCOM 3, it is now possible for multiple authorised QCOM users to queue a system lockup and each be displayed/serviced simultaneously.

Accordingly, it is required that the machine facilitate multiple simultaneous System Lockups by displaying them in either a tabbed display window, or otherwise label the system lockup display with “x of y” and provide the player/attendant with the ability to cycle between each System Lockup via the machine’s button panel or touch screen interface. (See below for example illustrations). cp:[[331]](#endnote-250)

The machine’s physical key-switch (if enabled during the lockup for a particular user’s SL – this is a SL parameter function call) must only operate on the currently visible System Lockup. cp:[[332]](#endnote-251)

If the machine uses a tabbed display then each tab should contain the SL’s *title* parameter. As each SL is cleared, the respective tab must disappear from the display. cp:[[333]](#endnote-252)

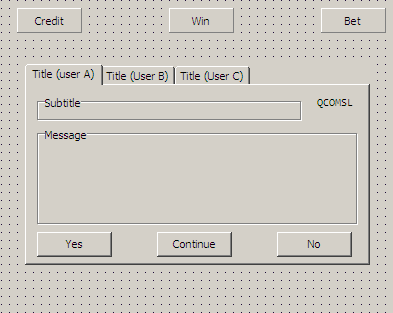


Figure 1 Tabbed Method

* *The dotted area represents the machine primary game display. The system lockup sits on top of this.*
* *The Credit, Win and Bet buttons represent the machine’s credit, win and bet meters respectively.*
* *The “Yes”, “Continue” and “No” buttons may not be present in every SL depending on the parameters provided in the last call to the SL request QCOM API function.*
* *Also refer to QCOM v1 documentation for examples of how a system lockup should overlay the machine’s play display*
* *The above image does not reflect the requirement for the SL text (excluding QCOMSL) to be centred.*
* *The text “Title”, “Subtitle” and “Message” is to illustrate the example and not to be displayed on the actual machine.*

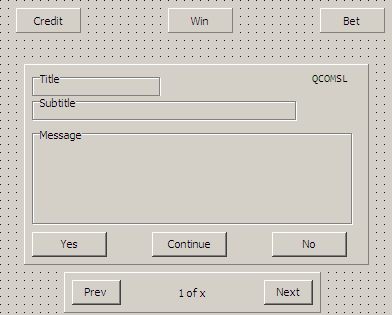


Figure 2 "x of y" Method

* *The above image does not reflect the requirement for the SL text (excluding QCOMSL) to be centred.*
* *The “prev” / “next” button may be more verbose if desired e.g. “Previous lockup” / “next lockup”. Or “lockup 1 of x”*
* *The text “Title”, “Subtitle” and “Message” is to illustrate the example and not to be displayed on the actual machine.*

There may be other acceptable methods for displaying multiple SL’s not listed here. Approval of alternative methods is at the discretion of the CEO OLGR. cp:[[334]](#endnote-253)

Acceptance of the overall legibility of the SL display is at the discretion of the CEO OLGR. cp:[[335]](#endnote-254)

## Orphaned System Lockups

If a user is deleted or quarantined (refer 5.8) whilst an SL is pending or active for that user, then the machine must automatically reset that QCOM user’s pending or active SL.

## System Lockup vs PEF Disable

System Lockup and PEF disable (refer s11.26.1) both disable all physical credit input devices, the bet adjustment UI and new play commencement on an EGM.

Credit input via QCOM API ECT methods are still possible in both.

Differences

If the EGM has credit, then during a system lockup the player cannot press collect whereas in a play disable condition they can press collect and instigate a credit redemption.

In QCOM, System Lockup defined as a discrete state and not a part of idle mode. The play disabled condition however is just a simple software flag (abbrev. as PEF in QCOM) that can be changed at any time via the QCOM API (s11.26.1), but where the disabled condition i.e. PEF = false must only displayed during EGM idle mode state.

System Lockups are a higher priority than a PEF disable in that the EGM must exit a play disabled display in EGM idle mode in lieu of a system lockup or hopper collect.

A PEF disable in QCOM 3 is similar to Communications Disabling Conditions in QCOM v1.x, the main difference is the play disable condition/s in QCOM 3 are now generic and the reason/s for play disables are controlled by authorised QCOM users via the QCOM API.

# ECT

*Design Requirements:*

* *At least the equivalent functionality as QCOM v1.*
* *Multi-master support.*
* *The ability of transfer credit a fraction of the current credit meter off the machine (QCOM v1 could only ECT the whole credit meter amount).*
* *Minimise memory requirements concerning many ECT events.*

ECT stands for Electronic Credit Transfer. QCOM ECT is used to transfer credit to or from a machine’s credit meter. Example applications are:

* Cashless gaming (i.e. account based gaming) (add / subtract credit)
* External jackpots (add credit)
* Arbitrary reward and prizes (add credit)
* TITO (system based TITO) (add / subtract credit)
* RCRF (10.12.11) (add / subtract credit)
* Arbitrary purchase of goods and services (subtract credit)

QCOM ECT is only concerned with the transfer of credit directly to or from a machine’s credit meter. QCOM makes no assumption as to what the ECT peer (with respect to the machine) may be, as it could be a player account, a jackpot system or other type of service. The generic ‘reason’ fields present in each QCOM ECT transaction, must be populated by respective QCOM user.

QCOM’s privilege system allows individual QCOM users to be granted rights to perform ECTs to and/or from the machine’s credit meter. Each permitted user also has their own set of ECT limits and set of meters.

ECT to the machine’s credit meter (adding credit) is straight forward, all that is required is permission to the applicable QCOM API functions (below) and it’s essentially just a matter of the QCOM user calling the function.

ectAddCredit()

ectTicketInAddCredit()

ECT from the machine’s credit meter (subtracting credit) requires an extra layer of authorisation because it will be not common for multiple QCOM users (cash ticket out system provider & a cashless out system provider) to contend over any particular credit out event. Thus every subtract credit event requires individual authorisation. This allows for a QCOM user in the role what is called a ‘credit redemption manager’ to make the decision on which QCOM user receives authorisation for any particular credit out event. Refer section 22 on credit redemption for more information.

The following requirement from QCOM v1:

*“In an EGM, the last amount transferred (if not zero) via the QCOM ECT API, either to or from the machine’s credit meter, must be displayed on the EGM while in idle mode in real time until commencement of the next play or ECT… ”*

…is not being carried forward as not all jurisdictions may want this. If this functionality is desired it is suggested that it could be implemented via the new player accessible event log display (refer section 19) or via qcom\_egmSPAMA[B]().

The QCOM API functions concerning ECT are defined in section 11.23. Also refer to the QCOM API summary spreadsheet document.

The EGM must maintain total machine ECT-to-CM and ECT-from-CM meters.

A machine display of ECT related events can be found via the machine’s event buffer (s13) display in audit mode (s28).

Related:

* Credit Redemption – section 22

# TITO

TITO stands for Ticket In / Ticket Out. This section describes and summarises QCOM support for TITO.

## Summary of methods

Below is a summary of events and methods associated with QCOM TITO for quick reference purposes.

State Events:

COLLECT\_WITH\_CREDIT

TICKET\_OUT\_PRINT\_START

TICKET\_IN\_ESCROW

TICKET\_IN

Events:

TICKET\_OUT\_PRINTING

TICKET\_PRINTER\_PAPER\_OUT

TICKET\_PRINTER\_PAPER\_JAM

TICKET\_PRINTER\_GENERAL\_FAULT

TICKET\_PRINTER\_FAIL\_PRINT

TICKET\_PRINTED

TICKET\_PRINTER\_INK\_LOW

TICKET\_PRINTER\_PAPER\_LOW

TICKET\_IN\_ECT

TICKET\_IN\_TIMEOUT

TICKET\_IN\_ABORTED

TICKET\_IN\_REJECTED

Primary QCOM API functions:

qcom\_ectSubtractCreditAuthorised

qcom\_ectTicketOutSubtractCredit

qcom\_ectTicketInAddCredit

qcom\_bnaRejectTicket

Supporting Ticket Printer QCOM API functions:

qcom\_tpGetp

qcom\_tpFirmwareID

qcom\_tpMeters

Other related QCOM API functions:

qcom\_locSetVenueName

qcom\_locVenueName

qcom\_locSetVenueAddress

qcom\_locVenueAddress

## Ticket-Out Sequence

*The key to ticket-out operation is the qcom\_ectTicketOutSubtractCredit() QCOM API function (s11.23.11). This function can only operate while the machine is in a System Lockup (section 16).*

The ticket-out sequence would typically begin with the player pressing the collect button on the machine while the machine has credit. This action would throw the **COLLECT\_WITH\_CREDIT** state event.

In a single operator environment i.e. no credit redemption manager (refer s22), the QCOM user in the role of the TITO service provider would hook into the above state event and upon occurrence, direct the machine into a System Lockup via the relevant System Lockup QCOM API function (refer section 16).

Once the machine is in the System Lockup, the QCOM user can retrieve a ticket authentication code and data from the TITO host (via one of the available communications APIs QCOM provides). Once the TITO host has authorised the TO operation, the QCOM user (or a CRM) can authorise the transfer via the qcom\_ectSubtractCreditAuthorised() QCOM API function and then the user may instigate the transfer of credit and the physical ticket print via the qcom\_ectTicketOutSubtractCredit() QCOM API function. If the machine instigates the print, the machine must throw the **TICKET\_OUT\_PRINT\_START** state event and log the (ref s13.6.1) **TICKET\_OUT\_PRINTING** event.

Unless a fault condition occurs while trying to print the ticket, this ends the ticket-out operation.

If the QCOM user (or TITO host) decides on an alternative action (i.e. not to print a ticket) the QCOM user may utilise the QCOM API and System Lockup features to give the user adequate feedback and take a different course of action.

In a multi-operator environment, instead of waiting for the **COLLECT\_WITH\_CREDIT** state event, the QCOM user in the role of the TITO service provider would wait for the **ECT\_AUTHORISED** state event. The procedure is the same as before after that.

### Cash Ticket Layout

The cash ticket print layout is as per QCOM v1 requirements. Refer to the latest version of QCOM v1 for more information.

## Ticket-In Sequence

If the banknote acceptor is enabled and a player inserts a ticket, the machine will throw the **TICKET\_IN\_ESCROW** state event.

If no further action takes place, the machine must eject the ticket after a timeout period of **10** seconds and log the (ref s13.6.1) **TICKET\_IN\_TIMEOUT** event.

If the QCOM user acting in the role of the TITO service provider decides to credit the ticket to the machine, the user will use the QCOM API function **qcom\_ectTicketInAddCredit()** (refer s11.23.10) in order to credit the value of the ticket to the machine’s credit meter. To be successful this must occur within the machine’s ticket in timeout period. Upon successfully adding the credit to the credit meter the machine must throw the **TICKET\_IN** state event abd log the (ref s13.6.1) **TICKET\_IN\_ECT** event.

If the QCOM user acting in the role of the TITO service provider decides **not** to credit the ticket, the user will use the QCOM API function **qcom\_bnaRejectTicket()** to return the ticket to the player. This action must cause the machine to log the (ref s13.6.1) **TICKET\_IN\_REJECTED** event.

## Ticket Details / Layout

Ticket Details / Layout are as per QCOM v1.

# Player Accessible Event Log

*In this section, “player” means a person with physical access to the QCOM machine.*

**Possible future functionality; do not implement until further notice.**

This section defines requirements for a Player Accessible Event Log (**PAEL** for short) for use with gaming machines. The Player Accessible Event Log feature provides players and machine attendants with access to a special on-demand, event log display on the machine.

*Design Requirements:*

* *Provide a player accessible event log (i.e. publicly accessible by any physical machine user)*
* *Provide a multi-user capable player messaging system which does not burden the machine’s already limited capacity to display messages. Similar/related in functionality to QCOM v1’s SPAM A&B & GPM functions.*
* *QCOM users require privilege to post events to the log*
* *All Users can query the log, i.e. see other users entries*
* *Provide an auto erase timeout parameter on individual events*
* *A flush all command*

*Benefits:*

* *This feature has the potential to significantly reduce and streamline the increasing number of messages cycling on the machine’s display e.g. QCOM v1 and other gaming machine standards sometimes mandate a message to be displayed until the ‘start of the next play’. This feature could be utilised in those cases. It can be seen as a QCOM multi-user extension to QCOM’s (v1’s) single user SPAMA&B and General Promotional Message.*
* *This feature has the potential to replace requirements that required a “display-until- next-play” type message.*
* *Dev-note: This feature has the potential to replace “reason” type arguments used in the QCOM API.*
* *It may help reduce machine attendant callouts as information in the log may assist a player in answering their own questions.*

*Caveats:*

* *Depending on specific event content, it is theoretically possible that some events in the log may not be desirable for access by subsequent players.*

*Risks*

* *The risk of spam may require that the feature be controlled or throttled.*
* *The feature should never be over utilised as the user will start ignoring the events if there are too many generated*
* *Some content could raise privacy concerns and should be used with caution regarding specific player activity.*

Some examples of possible applications are:

* Venue broadcasts/notices of player interest e.g.
  + venue messages e.g. EOD close warnings
  + brief news items
  + promotions (as permissible)
* Advertising (regulatory environment permitting)
* An alert list – important notices, e.g.
  + “This machine’s banknote acceptor Is not working”
  + “Closing early today – 5pm”
  + “Fire system test today at 10am”
  + “This machine is closing for maintenance at 4pm”
* Value added service support
  + Reasons for machine disables (play/credit input etc.) and other events and errors or special messages
  + Reasons for ECTs, Cancel Credits & Jackpots (See risks above)
  + Denoting the machine’s participation in a related product or service such as an external jackpot, e.g. “This machine is enrolled in “Mega Millions Jackpot” - Play for a chance to win”, or “Mega Millions Jackpot $12,345,67” *(ref EXTJIP)*
  + WAN Jackpot win notification e.g. “Mega Millions was won today at Brisbane No Name Leagues Club”
  + TITO support e.g. error messages
  + LP support; reporting of LP related ECT’s / clarifying when multiple occur.

The feature is expected to be used for short term events only, i.e. event life expectancies typically around a few minutes on average.

## Accessing

The Player Accessible Event Log (PAEL) must be accessible an option offered when the standard “i” button is activated on the machine. Refer section 3.7 for “i” button specific requirements (The “i” button is currently utilised for PIDs displays in some AU gaming machines).

Access to the PAEL log by this method and requires no special permission other than a physical presence at the machine. cp:[[336]](#endnote-255) It must take no more than two separate actions on the machine’s UI in order to access the log. cp:[[337]](#endnote-256)

QCOM users will also be able to access all stored events in the log via the QCOM API.

## Appearance

The PAEL display must not be visible during normal operation on the machine.

When accessed, the PAEL must pop up on the machine’s current display (akin to QCOM system lockup display – ideally with some background transparency). The default event display order must be time-stamp sorted, from top to bottom, newest events last. One exception here is that events that were logged with no time-stamp display (notime) must all be displayed grouped together at the top in the order in which they were logged. cp:[[338]](#endnote-257)

The total number of events the log must support/display is **25** and they must all fit onto the machine’s display at once i.e. no need for a UI to scroll the display. cp:[[339]](#endnote-258)

Each event must be displayed on a single line as follows: cp:[[340]](#endnote-259)

hh:mm:ss <Event text>

or, if logged with the *notime* flag, then:

<Event text>

There is no date display as time-stamped PAEL events are intended for short term events only, i.e. less than 24 hours.

The following items must remain visible during the PAEL display:

* Credit, bet and win meters cp:[[341]](#endnote-260)

The event log display must be closed if the player activates the “i” button again or other visible button labelled ‘exit’. cp:[[342]](#endnote-261)

If an event’s text string is too long to display on a single line, then the event string must be repeatedly scrolled left and right at the rate which allows the user to comfortably read the message in a reasonable timeframe. A **5-10** character per second scroll rate (preferably a smoothed scroll) would be ideal. cp:[[343]](#endnote-262)

If events are added or deleted from the PAEL while on display then the display must automatically refresh to reflect any changes in real time. cp:[[344]](#endnote-263)

## Posting events

Only authorised QCOM users will be able post new events to the log via the QCOM API (Refer section 7). The events themselves and their text based content are generic and can provide for all kinds of information. Event content can be optionally controlled by the QMA if desired.

**Call format:**

qcom\_paelLogEvent(arg:table)

Where arg is a *table* of named arguments as follows:

|  |  |  |
| --- | --- | --- |
| qcom\_paelLogEvent(arg:table) | | |
| **Argument name\*** | **Type** | **Description** |
| *s* | string | Event text (max - **100** characters) |
| *[notime]* | bool | If notime = true then no time-stamp must be displayed along with this event when displayed. Default is *nil* (time displayed) |
| *[ttf]* | integer | Time to flash “i” button (in MOT secs). Default 0. Stop flashing is the event is deleted and no other events still require the “i” button to flash. |
| *[fud]* | integer | Flash “i” until event is deleted and no other events still require the “i” button to flash. Default *nil* |
| *[fuv]* | bool | Flash “i” until event is viewed or deleted and no other events still require the “i” button to flash. Default *nil* |
| *[ttl]* | integer | Time to live TTL (in MOT secs). The machine must auto-delete the event after this time has expired. |
| *[voo]* | bool | Auto erase after 1st view (i.e. view-once-only). Default *nil* |
| *[eonp]* | bool | Auto erase on next play start. |
| *[priv]* | bool | Event is intended for current player eyes only and must be auto-purged upon any invocation of the qcom\_paelPurgePrivate() QCOM API method. Default *nil*. |

\* [ ] means argument is optional

The **return value** is an integer number representing the new event’s UID which can be used for manually deleting the event at a later time. On error, the function must return *nil*.

When one or more new events are posted to the log by an authorised user, the machine’s “i” button may be required to pulse-sate or flash in order to alert the player/user that a new event has been posted to the log. An overlaid unread event count is acceptable. In addition, if the “i” button was flashing or had unread events when selected by the player, then the player event log option appearing under the “i” button menu must also be more prominent than other menu options in order to help direct the player to the log.

Upon adding a new event and the PAEL is full (with respect to the user’s maximum event limit) then the machine must **auto-delete** the user’s oldest event to make space.

## Persistence

Events in the log must not be stored in the machine’s NV memory and all PAEL events must be cleared upon any restart. cp:[[345]](#endnote-264)

The time for which an event exists in the log is settable upon the initial logging of the event as well as several other control parameters. See previous section.

If a QCOM user is deleted then the machine must delete all events for that user at that time.

## Configuration

The following QCOM API function sets parameters concerning the PAEL on a per QCOM user basis. Typically the QMA will retain sole privilege to this function.

qcom\_paelSetp(

userID

integer, -- Maximum event string length (capped at max)

integer, -- Maximum TTL in secs

integer , -- Maximum number of events (default 0)

integer -- priority

)

## Other supporting QCOM API functions:

Purging:

qcom\_paelPurge(eventID)

*A user must only be able to purge events they logged via the above function*

qcom\_paelPurgeAll()

*It is expected that the above function will remain in control of QMA. Restarting the machine also clears all events in the log.*

qcom\_paelPurgePrivate()

*The above function purges all events denoted as ‘private’ e.g. called when a new player card is removed / inserted into the machine.*

*Design note: Possible additional functionality under consideration for future expansion of this feature are:*

* *QCOM API for read-only access.*
* *Possible stats for performance monitoring:*
  + *Total # events logged.*
  + *Q full event & count.*
* *New event logged QCOM state event.*
* *The ability to pre-load event message text via a dedicated function, then event logging is a select-from-list approach. This may be useful when the message content needs to be strictly controlled. This feature can be layered on later by an authorised QCOM user or QMA (good example of the dynamic nature of QCOM 3).*

# Fault Conditions

**QCOM 3 Definition: “Fault Condition”** (same as QCOM v1)

A fault condition means an event has occurred on the machine where either a hardware component(s) of the machine can no longer perform the function(s) for which it was designed, or some action has taken place which the machine considers to be illegal behaviour at the time. Door opens and stacker removed conditions are not to be considered fault conditions under this specification[[346]](#footnote-82).

The following requirements apply to fault conditions on QCOM 3 machines:

When a new fault condition occurs on the machine, the machine must suspend what it was doing (e.g. game play for gaming machines), disable credit input on applicable peripheralsand prominently display the new fault condition on its primary display device and queue the **FAULT\_CONDITION** state event.

All active faults on the machine must be prominently displayed initially each time the machine enters audit or test mode.

The machine must remain suspended in the fault condition until the fault is “**acknowledged”** which is defined by either of the following actions:

* A physical user activates a key-switch on the machine while the machine’s UI is focused on that fault.
* A QCOM user successfully invokes the QCOM API function rcResetFault() for a given fault.

Once a fault has been **acknowledged**, then the machine may clear the fault anytime from that point on that the machine detects or believes otherwise the fault has been rectified.

Related:

* **FAULT\_CLEARED** state event

## Fault Condition Backgrounding / Quieting

Support for fault condition backgrounding is **optional** for machine manufacturers.

If a fault still persists after acknowledgement (see previous section), a machine may optionally depreciate the fault condition so it now only appears in audit / service mode of the machine and keep only the associated peripheral disabled until such time as the fault is rectified. This feature is termed **fault condition backgrounding or fault quieting**.

If there are no other issues suspending the machine, then the machine may continue normal operation in all other respects but with any the devices still with backgrounded faults disabled.

It is at the discretion of the machine manufacturer as to which fault conditions may be backgrounded, if any.

Special cases concerning fault condition backgrounding:

* If a fault concerns a peripheral device that is required for game play on a gaming machine (e.g. a mechanical reel device) then that fault must not be able to be backgrounded.
* If the fault concerns the core machine or a peripheral that is required for the integrity of the machine (e.g. RAM error / low NV memory battery / primary display failure / mechanical meters disconnected / over-temperature) then the fault must not be able to be backgrounded.

Certain issues with specific peripheral devices very suitable for fault quieting. For example:

* Banknote Stacker Full
* Touch Screen Faults (where an alternative input device is available)

**Fault Condition Backgrounding and the QCOM API**

The QCOM API function **qcom\_egmOK()** (s11.19.27) does not include quieted faults.

To find out what quieted faults there are on the machine, a QCOM user can use the QCOM API function **qcom\_egmInQuietFault()** (s11.19.24) to retrieve a list. This information is also available from **FAULT\_CLEARED** state events.

**Related:**

* Refer to term “quiet” throughout the document.
* **FAULT\_CLEARED** state event.

## Door Events

Refers to all doors open / close events including stacker removed / returned which are also treated like a door under this specification.

Door events are not defined as fault conditions under this specification however the machine’s reaction to both is the same.

While any machine doors are open, the machine must, must for the duration, suspend all human-at-machine based functionality (betting, play, collect and system lockup) other than access to audit/test modes and associated functionality. This is the same reaction as per any fault condition.

All door opens must be reset by corresponding door closure. If a higher level of disable is required, then a QCOM user will instigate a System Lockup.

Power off processor/logic door monitoring as per QCOM v1 requirements is still **mandatory** for QCOM 3 machines.

Related:

* **DOOR\_OPEN** & **DOOR\_CLOSED** state events.
* **qcom\_egmDoorState()** QCOM API function description. (List of door descriptor strings)

### Logic door access

*Design Requirements:*

* *In the life QCOM 3 machine, the goal is that a full commissioning and logic door access will only occur once (assuming the machine only has one owner and excluding component failures inside the sealed logic area).*

For logic door access **before** performing a ‘confirm’ via the Logic Area Seal Confirmation function (s4.4), the machine must do nothing other than display a status update of the door status on the machine displays when applicable. cp:[[347]](#endnote-265)

For logic door access **after** performing a ‘confirm’ via the Logic Area Seal Confirmation function (s4.4), the machine must behave as per section 20.2 above. cp:[[348]](#endnote-266) (QCOM users have the option of treating this event with greater severity as required / depending on risk.)

Proposed additional requirements if the machine is required to implement an electronic seal (s31):

For any detected logic door access **after** a ‘confirm’ operation via the Logic Area Seal Confirmation function has been performed (s4.4):

* The machine must immediately destroy all secrets (refer section 8 for examples) located in persistent storage; or otherwise the ability to access them (e.g. if the machine utilises a key store/safe or similar arrangement[[349]](#footnote-83), it is acceptable for the machine to just destroy the master secret key).
* The machine must lockup.
* Only a full factory default / RAM clear / machine recommissioning must reset a machine from a logic door open event lockup. This RAM clear procedure must require logic area access to instigate.

# Peripheral Device Support

In this section, peripheral devices refer to:

* Coin acceptors (includes coin diverter) (class prefix:“ca”)
* Coin hoppers (class prefix: “hopper”)
* Banknote acceptors (encompasses stacker and ticket-in) (class prefix: “bna”)
* Ticket Printers (class prefix: “tp”)
* Touch Screens (wrt the machine’s primary display; abbrev: “ts”)
* Spinning Reels (mechanical; addrev “reels”)
* Mechanical Meters (abbrev “mm”)

QCOM permits all peripheral devices to be optional. In order to cater for unforeseen changes in a machine’s operating environment, machine manufacturers must not hard-code the requirement for the presence of any given peripheral unless it is essential to the machine’s core functionality. None of the known peripheral devices listed above would fall into the category of essential.

Peripheral - levels of support

A QCOM 3 machine may have many more peripherals than listed above. For these, QCOM provides varying levels of support. For minimal support, QCOM 3 may define some **events** the machine must log for it (ref s13). Support may be increased for a given perhiperal by adding provision for it in the QCOM API ***peripheral*** class of functions (e.g. ts, mm & reels. Refer section 11.8); and possibly (as applicable) the **qcom\_egmDoorState()** QCOM API function. Finally, if more support is warranted, then the peripheral may be given its own QCOM 3 API **class** of functions (e.g. bna, ca, hopper & tp).

Similar to QCOM v1, support for new peripheral devices (re section 11.8) may only “appear” to the QLE / QCOM users upon machine / QLE restart.

The QCOM API has a class of functions for each major peripheral type. Minor peripheral devices do not have a function class associated with them, for example if a peripheral only requires basic QCOM support/functionality and as such does not have any meters or special monitoring and control requirements, it will not have a class of functions associated with it.

Below are the current peripheral devices that have a QCOM API function class:

Peripheral Device QCOM API Function Class Prefix

Banknote Acceptors qcom\_bna

Coin Acceptors qcom\_ca

Coin Hoppers qcom\_hopper

Ticket Printers qcom\_tp

Before calling any peripheral related function in a class, QCOM users should first ensure that the machine actually supports the peripheral in software via the QCOM API function **qcom\_peripheralSupported()** refer section 11.8.1. Failing to do so will result in a Lua error return value for those functions. Refer to the QCOM SDK for more information.

The machine must automatically compensate for any given scenario of non-responsive or disconnected peripheral devices. Specific requirements include:

* If a hopper is removed / disconnected and the coin acceptor is still connected then the machine must automatically divert coins to the cashbox.
* If a coin acceptor is disconnected, hopper payouts must still be possible via the qcom\_hopperPayout() QCOM API function. *It is the responsibility of the QCOM user with privilege to the function to apply logic here*.

## Coin / Token Hoppers

This sub-section lists specific requirements related to hoppers.

QCOM machines must not automatically resume a hopper pay-out after an interruption during the pay-out, e.g. door access, fault, power etc. cp:[[350]](#endnote-267) The hopper pay must terminate upon return from the interrupting condition. A hopper payout may be restarted by pressing collect after the condition is cleared. This is to allow a patron to be easily hand-paid in the event of a recurring fault.

QCOM v1 endeavoured to distinguish hopper jams from hopper empty events. Since this was never very accurate QCOM 3 simply treats these two events as a single condition. Refer **HOPPER\_JAMMED\_OR\_EMPTY** event.

Hopper refill support must be as per QCOM v1 latest version requirements.

Coin diverter to hopper/cashbox operations must be primarily dependent on the hopper full probe and not the current hopper level meter.

Illegal coins/token out in relation to a hopper runaway must not change the machine’s credit or coins-out meters. Illegal coins/token out are metered in the **HOPPER\_RUNAWAY\_AMOUNT** event.

## Peripheral Firmware Upgrades

In QCOM 3, peripheral device firmware may be upgraded over the network via the QCOM API, or via a USB port in the machine, or via a peripheral manufacturer specific methodology.

Regarding firmware upgrades via the network

Some peripheral devices may have a QCOM API function associated with them of the form below which facilitates upgrades to the device’s internal firmware over the network.

Support for this feature is currently **optional** for machine manufacturers but only with respect to peripheral types and models that generally cannot support the feature in the opinion of OLGR. However, for banknote acceptor devices, support for this feature is **mandatory**. cp:[[351]](#endnote-268)

Upgrade support via the network must be indicated via the peripheral’s QCOM API ‘Status’ function. cp:[[352]](#endnote-269)

QCOM API supporting functions

Call format:

**qcom\_<xxx>FirmwareUpgrade(URL: string, SHA256:hexstring)**

<xxx> above denotes the device type. For example, **qcom\_bnaFirmwareUpgrade()** (section 11.32.4).

In validation type peripheral devices (e.g. banknote and coin validators), this function is also to be used to update the given peripheral device’s validation data-set.

This function when invoked must cause the machine to instigate a download attempt of the designate peripheral device’s firmware package and/or validation data.

The function must never block; all download attempts must be conducted in the background by the machine.

A machine is required to be able to perform downloads to multiple peripheral devices **concurrently**. cp:[[353]](#endnote-270)

The machine must **abort** any downloads / upgrade operations if the machine is restarted during the process. cp:[[354]](#endnote-271)

The machine must display a warning message if the machine’s hardware is at risk if interrupted during critical moments of an upgrade. cp:[[355]](#endnote-272)

Arguments

The first argument denotes a URL of the firmware upgrade package. Max length sanity check is **300** characters.

The minimum set of protocols to be supported by the machine is **http** and **ftp**.

The second argument must be a *hexstring* representing a sha-256 hash of the package denoted by the first argument for verification by the machine. This argument will be superfluous to any manufacturer embedded authentication data contained in the downloaded package.

Peripheral, or machine manufacturer authentication data (i.e. digitial signatures) embedded in the downloaded package is **mandatory** and must be checked as valid by either the peripheral or the machine before applying the update. cp:[[356]](#endnote-273) No QCOM authorities (for example the SUA), apply to firmware upgrade packages.

The QLE Lua software driver will apply sanity checks to the arguments and return an error on fail, see below.

**Return Values:**

The return value must be a *true* | *nil, string*

On failure, (i.e. a failure of one or more of the arguments during the initial argument sanity checks, or upgrades not supported for the given device), the function will return *nil, string;* where the string indicates the reason for the error.

On success the function will return *true* and then commence a background attempt at downloading and authenticating the firmware upgrade package for the given device.

Once the attempt at downloading has been completed (success or fail) the machine must throw the QCOM state event **PERIPHERAL\_UPGRADE** which contains more information concerning the final outcome of the download. Refer to this state event full description for more information.

The machine must automatically **abort** an upgrade attempt after a protocol error or a timeout with a host occurs. Time-out values may currently be set at the discretion of the machine manufacturer but must be reasonable.

The machine must sanity **check the ongoing download package size** during the download. If it exceeds a reasonable size threshold (set at the machine manufacturer’s discretion[[357]](#footnote-84)) then the download must be also **aborted** with an error.

The format of the downloaded firmware upgrade packages is at the discretion of the machine or device manufacturer, however it is expected that the primary content will be encrypted and able to be authenticated via a digital signature by either a machine or peripheral device manufacturer’s public key before being applied.

After being downloaded and successfully verified, upgrades must be queued up for activation on next machine **restart**. cp:[[358]](#endnote-274)

**Under no circumstances must the QCOM machine have any provision whatsoever to ‘execute’ any part of the contents of the upgrade package. All execution of the entire package contents must only occur within the target peripheral.**

Peripheral firmware upgrades via USB

When peripheral upgrades are initiated from a machine USB port, the machine won’t know for certain if the package is a machine software upgrade package, or a firmware upgrade package until the machine actually processes the upgrade package. However, the **MACHINE\_UPGRADE\_STATUS** state event is required to be logged for machine software upgrade package downloads only. In this case,the machine should use something in the filename (e.g. a specific file extension) to denote which type it is just for the purpose of whether the **MACHINE\_UPGRADE\_STATUS** is logged or not. The machine should not assume however that the filename correctly does correctly denotes the type of upgrade when it comes time to process the file package contents. The worse-case scenario is a filename incorrectly denotes what type of upgrade it is. The only side effect of this must be that the **MACHINE\_UPGRADE\_STATUS** state event will be logged or not logged when it shouldn’t be. In the end however, the machine events (listed below) must logged correctly as the machine has since verified and parsed the upgrade package file:

* **MACHINE\_UPGRADE\_READY**
* **MACHINE\_UPGRADED**
* **PERIPH\_FW\_UPGRADE**

Related: s30.5

# Credit Redemption – Gaming Machines

*Design Requirements:*

* *Support multiple service providers relating to credit redemption, e.g. ECT out system, Ticket out system, and attendant paging systems.*
* *Support an arrangement where both a card based gaming system (ECT) and TITO system can run simultaneously*
* *Allow the implementation of any conceivable rule set concerning credit redemption (infers that a parametric control approach is limiting)*

*Design Note: In QCOM v1, credit redemption worked via several parameters which would cause the machine to instigate different credit redemption methodologies as a result of the current values of the following control parameters: COLLIM, TICKET and ECT cashless mode (CMode) flags.*

Credit Redemption in QCOM 3 concerns the process of deducting credit from the machine’s credit meter for any purpose other than for playing a resident game on the machine. Typically the credit is being deducted to be redeemed by the player via an available method such as:

* a hopper pay,
* cash ticket out, or
* transfer of funds to a player account, or
* cancel credit aka a cheque or manual payment.

The destination of the deducted credit is arbitrary (out-of-scope).

In a QCOM 3 machine, the credit redemption process may be instigated when a player presses the collect button, or when the QCOM API qcom\_rcCollectPress() command is invoked by a permitted QCOM user (e.g. a card based gaming service would use this QCOM API function).

When the collect button is pressed (or the QCOM API function *qcom\_rcCollectPress()* is invoked by an authorised QCOM user) and the machine has credit, no faults, all doors closed and is in idle mode, the machine must throw a **COLLECT\_WITH\_CREDIT** QCOM state event and allow the QLE Lua software driver to execute any hooked scripts. cp:[[359]](#endnote-275)

**If there are no scripts attached to the above state event, then pressing collect on the machine will have no effect,** i.e., a player cannot collect their credit off the credit meter. **This is the factory default behaviour in a QCOM 3 machine.** In some operating environments being able to ignore or change the response to EGM collect button presses, does suit some operating modes[[360]](#footnote-85).

The idea is that a permitted QCOM user will hook a handler script to the above state event which when executed upon a player collect, can make an decision, to either queue up a hopper pay, cancel credit (via System Lockup), perform a ticket print, or ECT Credit Meter deduction (via System Lockup), based on all factors such as the existence of hardware any and local or regulatory requirements. This method allows for any possible set of credit redemptioncontrol requirements to be implemented.

The following QCOM API functions are typically made available to the QCOM user granted credit redemption control:

* qcom\_slxxx (System Lockup class of functions). Refer section 16.
* qcom\_ectSubtractCreditAuthorised()
* qcom\_ectSubtractCredit()\*
* qcom\_cancelCredit()\*
* qcom\_ectTicketOutSubtractCredit()\*
* qcom\_hopperPayout()
* qcom\_rcCollectPress() – utilised for system instigated payouts

If a Credit Redemption Manager (see next section) is in user then also:

* qcom\_luaPublish() (Section 11.15.10)
* qcom\_luaPublishGetValue() (Section 11.15.11)

\*For these functions to operate, the machine must be in QCOM System Lockup, otherwise, they will return a failure code. In QCOM 3, the System Lockup feature is a key tool in credit redemption.

## Support for Multiple Service Providers

Some service providers relating to QCOM machines need to deduct credit from a QCOM machine. Examples may include an account based gaming service provider (wanting to transfer credit from a machine’s credit meter to a player account), or a TITO service provider (wanting to transfer credit from a machine’s credit meter to a ticket database host and have the machine provide the player with a cash ticket).

Under QCOM 3, a gaming venue can have both an account based gaming system and a TITO system from different service providers.

Support for multiple service providers needing to deduct credit from machines is a more complex scenario to manage than the opposite case of where there are multiple service providers wanting to **add** credit to a machine. This is because when there are multiple, potentially non-cooperating service providers relating to credit redemption, it creates a situation of contention over a single resource in the machine that being the credit amount up for redemption.

A machine may start out in an operating environment with only hopper collects and cancel credits. Say this role is assigned to QCOM user A. Subsequently the venue purchases a TITO system which brings in QCOM user B. Theoretically, user A must relinquish control of the role of credit redemption to user B in order to avoid contention over who performed the credit redemption. Finally the venue adds an account based gaming systems introducing user C. Now the situation becomes; how does B co-exist with C?

Typically, the main issue to resolve is how to decide which provider receives the credit whenever a ***COLLECT\_WITH\_CREDIT*** is instigated.

In order to manage all possible modes of operation, the concept of a QCOM user in the role of a **Credit Redemption Manager** (CRM) is introduced.

The role of the **CRM** is to:

* Provide the **default** credit redemption facility to any business or regulatory requirements in the absence of any other credit redemption service provider.
* Recognise, prioritise and **authorise** appropriate control to any third party credit redemption service provider on demand.
* Perform any **cancel credit** manual payments in the event no credit redemption service providers respond to a credit redemption.
* Perform any manual payments associated with **large wins**.
* Maintain and share ECT meters per service provider.

The CRM should be commercially neutral (with respect to machine credit redemption)**.**

The CRM is a virtual role in QCOM in that it does not have to exist at all and has no special meaning to the machine. The role is created by the specific QCOM API privileges granted by the QMA.

As the CRM should typically be commercially neutral (with respect to machine credit redemption), the most inherently suitable entity for the role of the CRM is the QMA as it is already a requirement for the QMA to be a commercially neutral party with respect to the delivery of all gaming machine related services.

The CRM role is easy to implement and can be facilitated as a fully autonomous QCOM user (refer section 6.1.3) typically installed by the QMA.

The CRM would be set up as a QCOM user dedicated to the CRM role and be assigned exclusive privileges to the following QCOM API functions:

qcom\_ectSubtractCreditAuthorised()

qcom\_hopperPayout()

qcom\_cancelCredit()

qcom\_machineAttendantRequired()

qcom\_slxxx (System Lockup class of functions)\*

qcom\_luaPublish()\*\*

qcom\_luaPublishGetValue()

\*In the event the CRM has to perform a cancel credit.

\*\*The lua publish functions above are used by the CRM and any service providers for communication. The CRM defines a simple protocol (which is all that is required) that the service providers must adhere to. For example, service providers ‘publish’ their interest in being a candidate for the next credit redemption event and the CRM reads this upon a ***COLLECT\_WITH\_CREDIT*** event and then makes a decision based on its current operating environment as to which user to authorise for the redemption. These ‘simple protocols’ (based on sharing data) will end up being very generic and will eventually become a part of the QCOM 3 standard. CRM scripts for various operating environments may be found in the QCOM 3 SDK.

CRMs, being closely related to the QMA, have a knowledge of the machine’s operating environment with respect to credit redemption as well as the expected QCOM users to be included in the credit redemption process.

On the service provider side, the privileges a particular type of service provider needs in order to deduct credit will vary. For example, an account based gaming service provider would typically have privileges to the following QCOM API functions:

qcom\_luaPublish()

qcom\_luaPublishGetValue()

qcom\_rcCollectPress()

qcom\_slxxx (System Lockup class of functions)

qcom\_ectSubtractCredit()

A TITO service provider would have privileges to the following QCOM API functions:

qcom\_luaPublish()

qcom\_luaPublishGetValue()

qcom\_slxxx (System Lockup class of functions)

qcom\_ectTicketOutSubtractCredit()

Refer to the associated QCOM API full function description for more information on each function.

### Credit redemption process for an account based gaming service provider

The credit redemption process for an account based gaming service provider in an operating environment consisting of multiple service providers vying for credit redemption would typically be as follows:

On start-up the service provider invokes the qcom\_luaPublish() method and shares a table of a specific schema with the CRM. For example:

qcom\_luaPublish({all = {crm = {eoi = true|false}}})

When a player inserts their card (into the SP’s card reader device), the QCOM user of the account based gaming service provider sets the *eoi* key value in the shared table to *true* which tells the CRM that they wish to be considered the candidate for the next credit redemption (**COLLECT\_WITH\_CREDIT**) event on the machine.

In addition, an account based gaming service provider can also instigate a **COLLECT\_WITH\_CREDIT** themselves by invoking the qcom\_rcCollectPress() method, e.g. in case a player removes their card with on the machine.

Subsequently on the next **COLLECT\_WITH\_CREDIT** event (for which the CRM will have a handler hooked), the CRM determines if the service provider should redeem the credit and if this is the case, the CRM authorises it via a call to qcom\_ectSubtractCreditAuthorised(). This throws the state event **ECT\_AUTHORISED** which the account based gaming service provider can hook onto which then gives them a limited time to transfer off the credit via a call to qcom\_ectSubtractCredit().

If the call to qcom\_ectSubtractCredit() fails, or the authorisation times-out, or the service provider’s host becomes unavailable, then the CRM can catch this (**ECT\_FAILED**) and try the next candidate if any. If none are found or ready, the CRM will handle the credit redemption; either as a hopper pay or a cancel credit (depending on the operating environment).

The account based gaming service provider can invoke eoi = false or ectSubtractCreditDeclined() at any time it no longer wishes to deduct credit from the machine.

### Credit redemption process for a TITO system

This process is identical to the credit redemption process for an account based gaming service provider except for the following differences:

* A TITO system typically does not instigate a credit redemption. It always lets the player instigate it via the collect button.
* The TITO system uses the QCOM API function qcom\_ectTicketOutSubtractCredit()to subtract credit instead of qcom\_ectSubtractCredit() function.
* As long as the TITO system is on-line, the QCOM user relating to the TITO system would keep the expression of interest (*eoi*) on at all times and only remove it when the TITO host is detected as offline or if a problem is noted with the machine’s ticket printer. (The CRM could monitor the machine’s ticket printer as well if desired.).

### Credit redemption process for other systems

This process would be primarily as per 22.1.1.

In each case, there is a decision relating to if the other system / service provider can instigate a collect (i.e. privilege to the qcom\_rcCollectPress() function).

The other consideration is that only a TITO system typically uses the qcom\_ectTicketOutSubtractCredit(), while all other systems deducting credit would use the generic qcom\_ectSubtractCredit() function.

## Attendant Paging System Support

This section concerns QCOM credit redemption and third party attendant paging system support (i.e. when the attendant paging system is not also playing the role of credit redemption methodology controller).

Attendant paging systems need to know about all credit redemptions that require the presence of an attendant (e.g. Cancel Credit).

Options for support include:

* The paging system hooks onto the System Lockup event and monitors the attendant requiredflag (refer section 11.25.1). (This relies on third parties to correctly set the flag)
* When the paging system can depend on the QCOM CRM (which should always be the case), it can simply monitor for the **ATTENDANT\_REQUIRED** state event.
* The paging system may be able to predict cancel credits requiring an attendant based on local regulatory requirements.

## Residual Credit Removal Feature

Refer section 10.12.11 for more information on how a Residual Credit Removal Feature (refer GMNS) would be implemented in a QCOM 3 machine.

Refer to other former OLGR publications (e.g. the former OLGR Queensland appendix to GMNS) for more information on RCRF in general.

# QCOM User Account Access (UAA) (SSL/TLS solution)

This chapter defines the methodology that must be provided by the machine which allows remote clients (QCOM users) to securely connect with the machine and interact with the QCOM Command Interpreter (see next chapter, s24).

The primary purpose of the UAA service is QCOM user maintenance; such as creating QCOM users and the initial installation and subsequent upgrades of their script packages also known as QCOM 3 apps.

During normal operation of the machine, the UAA service typically would be seldom used, like once per RAM clear per QCOM user on average. The intent is to keep the UAA as simple as possible to implement. The QCOM user framework should always be utilised to facilitate any desired ongoing network based functionality in QCOM 3 machines during both test and production.

A QCOM machine must host an IPv4 / IPv6, secure SSL / TLS based **listen** service to the requirements in this section. cp:[[361]](#endnote-276) This is referred to as the **UAA service**.

The UAA service must not be started until after the machine’s logic seal confirmation function has been completed (s4.4). cp:[[362]](#endnote-277) Once the seal is confirmed, then the UAA service must automatically start up and subsequently each time the machine boots. cp:[[363]](#endnote-278)

A successful connection and login by a remote client must place the QCOM user into a QCOM Command Interpreter (QCI). cp:[[364]](#endnote-279) Refer to the next chapter for more information on the QCI.

The QCOM UAA service must listen on IPv4 and IPv6 on port **47417**. cp:[[365]](#endnote-280)

The UAA service must utilise the machine’s UAA certificate for authentication by connected clients. cp:[[366]](#endnote-281) Refer section 8.2.

The QLE Lua software driver will limit the maximum number of connections equal to the maximum number of QCOM users (refer s5) plus **2**. Once the maximum number of connections is reached then new connections will cause the QLE Lua software driver to drop an existing *anon* user connection.

Refer to the next chapter concerning UAA session spam handling.

## Client Authentication Methodology

The machine’s UAA service must support two types of client authentication methods:

* Password based authentication.

Password based authentication **must only be used for anon user logins**. The **“password:**” prompt (required text), must only result when the remote client does not present an SSL certificate. cp:[[367]](#endnote-282) (The anon password is required to login as an anon\_\* user where \* will be the socket ID number.

Related: The anon password may be changed via the QCOM API function qcom\_userAnonSetPass(), s11.30.33.)

Related: **QLUAE\_QCI\_ANON\_TRY** state event.

* SSL client public key verification via client presented certificate.

(SSL client public key verification must be used(see below) by the machine for all other QCOM user logins when the remote client presents a certificate. cp:[[368]](#endnote-283) (Typically the QMA installs a key for user authentication upon user creation using the QCOM API function **secSetUAAverifyCert()**.) NB A QCOM user’s UAA public key is set via the QCOM API function **qcom\_userSetUAApublicKey()**, s11.30.6). The QLE Lua software driver uses these public keys to determine which specific user is trying to log in and will respond to the UAA service with either the **QCIE\_USEROK** or **QCIE\_USER\_REFUSED** message.

Related: **QLUAE\_QCI\_USER\_VERIFY** state event.

The final decision of whether to accept or reject a UAA remote client connection lies with the QLE Lua software driver.

The host machine must not care about any certificate metadata such as: the certificate expiry date, who the issuer/signer, etc. cp:[[369]](#endnote-284)

Checkpoints:

The machine must ensure the EGM’s UAA listen service is configured to always ask for a client cert via the SSL API. cp:[[370]](#endnote-285)

If the newly connected client does **not** present a cert (regardless of whether a UAA verify cert is set or not) then the EGM assumes an anon connection attempt and must request the anon password and log the result via the **QLUAE\_QCI\_ANON\_TRY** state event. cp:[[371]](#endnote-286) The QLE will check the anon password and reply with either **QCIE\_USEROK** or **QCIE\_USER\_REFUSED** messages.

If the client does present a certificate, then regardless if a UAA verify cert is set or not cp:[[372]](#endnote-287):

* the SSL\_accept() handshake must fail (connection refused) if the client can’t prove they have the private key for the cert presented. The machine won’t have to consider the certificate in this case as the SSL API won’t even reveal the certificate in this case and must just terminate the session. cp:[[373]](#endnote-288)
* if the SSL\_accept() handshake is successful (i.e. the client proved they have the private key for the certificate), then the **QLUAE\_QCI\_USER\_VERIFY** state event must be logged cp:[[374]](#endnote-289) The decision to keep the connection is now up to the QLE LSD; see below FYI.

FYI QLE LSD logic re the **QLUAE\_QCI\_USER\_VERIFY** state event processing:

*{cert=<string:PEM>, handle=<integer>, address=<string>, verifyResult=<integer>, verifyErrmsg=<string>}*

* If no UAA verify cert is set, then don’t care about the value of *verifyResult*; so long as the QLE public key in the cert matches a resident user then respond with the **QCIE\_USEROK** message, else respond with the **QCIE\_USER\_REFUSED** message.
* If UAA verify cert is set, then if the value of *verifyResult* is 0 (all ok) then then respond with the **QCIE\_USEROK** message, else respond with the **QCIE\_USER\_REFUSED** message.

*Design notes:*

*Certificate based authentication suits QCOM’s ‘new user’ introduction methodology because there will be no private key/password exchange in the user account creation process. It avoids having to embed a password in the user introduction script which would be a security issue).*

## The QCOM UAA Service Configuration

This section lists any specific configuration for the QCOM machine UAA service that is not able to be inferred from the requirements already stated in this chapter.

The machine’s UAA service must only accept SSL/TLS client connections. cp:[[375]](#endnote-290)

It is recommended that the machine’s UAA service’s equivalent of a IP **listen()** back log be set to **1**. cp:[[376]](#endnote-291)

It is recommended that the machine’s UAA service’s client certificate verification depth limit is set to **3**. cp:[[377]](#endnote-292)

There are two modes of operation in the machine’s UAA service with respect to client certificate verification:

* No client certificate verification
* client certificate verification

Client certificate verification against a CA is optional. The regulator or their delegate for a given jurisdiction will typically decide whether to use client certificate verification. [[378]](#footnote-86)

Which mode is determined by whether a QCOM user UAA CA cert has been installed in the machine, this is done via the QCOM API function **qcom\_secSetUAAverifyCert()**. If a client verification certificate has been set via this function, then the EGM must only accept connections from clients who present a cert that has been signed by this certificate. If a client verifification certificate has not been set, then the EGM must still accept connections, however, the machine must immediately drop the connection if the user can’t prove they have the private key corresponding to the public key in the certificate (this would typically be performed automatically by the TLS API the machine uses as a part of the TLS/SSL handshake).

The machine’s UAA service must be configured and started initially with no peer verification UAA certificates. cp:[[379]](#endnote-293) A UAA certificate used for peer authentication in this regard must be able to be added (or removed) at any time via the QCOM API function **qcom\_secSetUAAverifyCert()** s11.9.5. cp:[[380]](#endnote-294) The machine’s UAA service must not care if a certificate for client verification has been set or not cp:[[381]](#endnote-295) (it’s designed to be optional in QCOM 3); encrypted connections will be made either way. The machine’s SSL/TLS library shouldn’t care either (at least openssl’s SSL/TLS library doesn’t care) so there should be nothing for the machine’s UAA service to specifically implement here; its absence or otherwise of a peer verification certificate should be automatically handled by the machine’s SSL/TLS library.

The UAA’s listen service must always request client certificates for the purpose of trying to identify either an anon login attempt (aka no certificate presented by client), or for sending the client’s certificate to the QLE LSD for public key verification (aka a QCOM user is trying to connect). cp:[[382]](#endnote-296) Related: the QCOM user’s expected public key is known to the QLE LSD because it would have been setup via the **qcom\_userSetUAApublicKey()** QCOM API function upon user creation.

The machine’s UAA service must only perform the following actions with respect to peer client certificates:

* Ensure any peer client certificate is readable if presented and can make a secure connection with it. (The machine SLL/TLS library should be doing this) cp:[[383]](#endnote-297)
* If a secure connection is made, then extract the client certificate for which it must send to the QLE Lua software driver via the message: **QLUAE\_QCI\_USER\_VERIFY**. The QLE Lua software driver will verify the certificate’s public key against any public keys set via the QCOM API function **qcom\_userSetUAApublicKey** (s11.30.6). This allows the QLE Lua software driver to match a given public key with a specific QCOM user for use with the QCI (next chapter). The QLE Lua software driver will respond to the machine’s UAA service with either the **QCIE\_USEROK** or **QCIE\_USER\_REFUSED** message.

In the case a remote client does not present a certificate, the UAA service must assume it is a potential QCOM *anon* user login (refer section 5.1) and request the anon user password for verification. Once a password attempt is received the machine’s UAA service must log the **QLUAE\_QCI\_ANON\_TRY** state event. The QLE Lua software driver will check the password and respond with either the **QCIE\_USEROK** or **QCIE\_USER\_REFUSED** message accordingly.

Refer next chapter for more information about the QCI.

Related:

* QCOM API function qcom\_userAnonSetPass(), s11.30.33.
* QCOM API function qcom\_userSetUAApublicKey() (s11.30.6).
* QCOM API function qcom\_secSetUAAverifyCert() s11.9.5.
* Chapter 24.

## Summary of machine’ UAA service.

The machine implemented UAA listen service main tasks are:

* Listen for new remote client connections and make secure connections as per the requirements in this chapter.
* Ensure the remote client can make a secure SSL/TLS connection. The machine’s SSL/TLS library should be taking care of this. (The machine’s UAA service need not care about client certificate content; it must send all received client certificates onto the QLE Lua software driver via the **QLUAE\_QCI\_USER\_VERIFY** state event and wait for a response from the QLE Lua software driver.)
* When a secure client connection is made and the remote client does not present a certificate, then it must request a password from the remote client and log the **QLUAE\_QCI\_ANON\_TRY** state event and wait for a response from the QLE Lua software driver.
* Generate all **QLUAE\_QCI\_** prefixed state events as per their definition in the QCOM 3 summary spreadsheet document. Refer to the QCOM 3 summary spreadsheet.
* Buffer and process QCI messages logged by the QLE Lua software driver as per the requirements in this chapter as the QCI message description. (Refer to the QCOM 3 summary spreadsheet: ‘QCI’ sheet for the list of QCI messages that the QLE Lua software driver may send.)
* Format and send the login message to the remote client in response to the **QCIE\_USEROK** message from the QLE Lua software driver.

# QCOM Command Interpreter

Note: almost all requirements in this section are implemented as a part of the QLE Lua software driver. An endnote indicates a machine requirement to be implemented. Related: section 33.1.

This section contains all the requirements pertaining to the QCOM Command Interpreter which the interface that must be used for successful UAA service (see previous chapter) connections. cp:[[384]](#endnote-298)

The ‘QCOM Command Interpreter’ (QCI) refers to a QCOM specific dedicated command interpreter as defined in this chapter that a machine must connect a QCOM user with when the user logs onto the machine using the connection / data transport methodology defined in the previous chapter.

**The QLE Lua software driver implements the QCI interface defined in this section.**

The QCOM 3 machine’s UAA service must simply relay received messages and IP related state changes with respect to the remote computer (via **QLUAE\_QCI\_** prefixed state events) to the QLE Lua software driver which will perform all the command processing. The QLE Lua software driver will respond with **QCIE\_** prefixed messages back to the machine’s UAA service for which machine’s UAA service must action.

Related:

* QCOM 3 Summary Spreadhseet: QCI sheet
* **QLUAE\_QCI\_** prefixed state events
* The QCOM 3 SDK for related QLE Lua software driver source.

The QCOM Command Interpreter must not display a command prompt.

When a user successfully logs into a machine, the QLE Lua software driver will send the QCI message **QCIE\_USEROK** to the machine’s UAA service and log the state event **USER\_LOGON.**

The machine’s UAA service must use the information in the **QCIE\_USEROK** message to format and send a message formatted as shown below to the respective remote client. cp:[[385]](#endnote-299)

**Login message format:**

*[ ] above means conditional; <> above denotes a field with data that needs populating.*

Connected:

QCOM API Version: <qcom\_idInterfaceVersion()>”

Time: <yyyy-mmm-dd hh:mm:ss>

Last login: <yyyy-mmm-dd hh:mm:ss>

From: <ip address>

QMA: <QMA Certificate Fingerprint. Refer Section 4.5>

Device: <qcom\_idDeviceType()>

MID: <qcom\_idMfr3()>  
SERIAL: <qcom\_idMachineID()>

[Machine is in demonstration mode] *If qcom\_machineInDebugMode() eq true*

Each line of the login message will be followed by line feed / carriage return characters (“**\n\r**”) and all text messages sent by the machine during the session.

If the user is in quarantine, then the user will be disconnected immediately by the QLE Lua software driver.

While a QCI session is active for a given QCOM user, all standard output (*stdout* via the Lua print() function, refer 10.2.10) for the user will also be echoed to user’s QCOM Command Interpreter session in the machine by the QLE Lua software driver. The machine must pass this output through to the remotely connected peer. cp:[[386]](#endnote-300)

Inactivity time-out. The QLE Lua software driver will automatically terminate a QCI session if it is inactive for **180** seconds.

If a QCOM user attempts to log in more than once (i.e the user is already logged in), then both connections will be gracefully closed by the QLE Lua software driver but not before a final termination message (refer SDK) is sent. The user may then try to log in again at their discretion.

During each QCI session, the machine’s UAA service must apply a maximum read line processing rate. cp:[[387]](#endnote-301) A minimum max rate of at least **1** command per second is required. cp:[[388]](#endnote-302) Higher values are acceptable at the machine manufacturer’s discretion and risk.

The machine must truncate command lines received from remote clients to **512** characters per command line. cp:[[389]](#endnote-303)

Command lines read from remote clients must be truncated to the character before the first non-printable[[390]](#footnote-87) character if any. cp:[[391]](#endnote-304)

Related:

* QCOM state events USER\_LOGON & USER\_LOGOFF.

## Command Privileges

Individual commands in a QCI session (see next section) will not be available to any QCOM user unless the QCOM user is specifically granted the privilege via the QCOM API function: qcom\_userSetPrivilege (s11.30.18).

The anon user (s5.1) persistently has full access to the QCI command set.

The QLE Lua software driver does not permit the QMA to access the UAA service or QCI.

## Commands

The QLE Lua software driver implements the following commands defined in this sub-section for use within the QCOM Command Interpreter. Commands are case sensitive.

### qmaloadcert

**Syntax:**

**qmaloadcert {url=<string> [,hash=<hexstring>]}**

{*Argument format is a Lua table declaration*}

This command allows the user to download the QMA’s self-signed x509 certificate to the machine. The QLE Lua software driver will only permit this command to be successfully executed once per local factory reset / full RAM clear of the machine.[[392]](#footnote-88)

The *url* string argument key-value denotes a URL of the certificate which must be in the PEM file format. The *hash* string argument key-value is a SHA256 hash of the PEM file. The machine must verify the hash is correct (if provided) before accepting the certificate.

The minimum set of protocols to be supported is **http** and **ftp**.

Examples (NB: the *file:* protocol option used below is only available in qsim)

qmaloadcert {url="file://.\\QMA\\QMA\_sscert.pem"}

qmaloadcert {url="file://.\\QMA\\QMA\_sscert.pem", hash = "c075c10d43552f0c2e3282c30f359d243bb85b39902bdc09f439b97413626dc0"}

**Return values:**

* On success, in that a download attempt was started; once the download attempt is finished (success or fail), the QLE Lua software driver will output “**qmaloadcert:complete:**” followed by the following lua serialised text data with the format:

qmaloadcert:{success=<boolean>, filesize=<integer>, hash=<hexstring> [,errmsg=<string>]}

* On failure, in that a download attempt was not started; the QLE Lua software driver will output an error string begin with the text “**qmaloadcert:error:**” followed by an error message. Refer to the QLE Lua software driver in the QCOM 3 SDK for possible error messages.

Related:

* Section 4.5 the QMA and x509 fingerprint generation.
* QCOM API function qcom\_secQMAcert() s11.9.1

### qmacertfingerprint

**Syntax:**

qmacertfingerprint

This QCI command returns the fingerprint of the QMA’s self-signed x509 certificate in the machine. This QCI command is basically the output from the following *openssl* command line call:

openssl x509 –sha256 -in **qmacert.pem** -noout -fingerprint

Example response from the QLE Lua software driver:

qmacertfingerprint:SHA256:E0:1D:27:FD:B4:15:84:B6:A0:DF:29:78:31:96:39:30:9E:54:3E:94: B4:15:84:B6:A0:DF:29:78:31:96:39:30

If the machine does not yet have a QMA certificate loaded, then the QLE Lua software driver willreturn the following string:

“**qmacertfingerprint:error:no certificate found**”

*The QMA certificate will also be published and retrievable on the machine’s QCOM www server.*

### qmaexecscript

**Syntax:**

**qmaexecscript {url=<string> [,hash=<hexstring>]}**

{*Argument format is a Lua table declaration*}

This command causes the QLE Lua software driver to request the machine to attempt to download in the background a SMIME signed lua script text file denoted by the *url* field and verify it against the installed QMA certificate. If successful, the machine must pass the script onto the QLE Lua software driver which will then execute the script at the privilege level of the QMA as an internal state event within the QCOM Lua Engine. The applicable state event here is **QLUAE\_DOWNLOAD\_COMPLETE**. Refer to the QCOM 3 SDK for more information.

The maximum file size is set by the QLE LSD at 100,000 bytes. Related: qle.lua: *qmaexecscript\_maxsize* value.

The machine does not need to save scripts downloaded via **qmaexecscript** in NV memory; once the script has been executed by the QLE Lua software driver it will also delete the script from memory.

Example (openssl) of how to sign a QMA script:

**openssl** **cms** [-md sha256] **-sign** -in qmaScript.lua -nodetach -out qmaScript.lua.sig -binary -signer QMAsigner1\_cert.pem -inkey QMAsigner1\_privkey.pem

The minimum set of protocols to be supported is **http** and **ftp**.

The script will be executed as per any script hooked to a state event in the QLE Lua software driver.

**Return values:**

* On success, in that a download attempt was started; once the download attempt is finished (success or fail), the QLE Lua software driver will output the text string “**qmaexecscript:download:verification:complete:**” followed by the following lua serialised text data with the format:

qmaexecscript:{success=<boolean>, filesize=<integer>, hash=<hexstring> [,errmsg=<string>]}

* On failure, in that a download attempt was not started; the QLE Lua software driver will output an error string beginning with the text “**qmaexecscript:error:**” followed by an error message. Refer to the QLE Lua software driver in the QCOM 3 SDK for possible error messages.

Related: section 10.7.3.

### userloadscripts

**Syntax:**

**userloadscripts {url=<string> [,hash=<hexstring>]}**

{*Argument format is a Lua table declaration*}

The QLE LSD on receipt of this command will pass the arguments provided to the QCOM API function qcom\_ userLoadScripts() in the username of the invoking QCOM user. Refer to the QCOM 3 Summary spreadheet for more information on this function.

This command will not be available if the user is logged in as an anon user (s5.1).

**Return values:**

* On success, in that a download attempt was started; once the download attempt is finished (success or fail), the QLE Lua software driver will output the text string “**userloadscripts:complete:**” followed by the corresponding thrown **USER\_LOADSCRIPTS** state event data (minus the *seid* key-value and serialised as a single line lua constructor) with the format:

userloadscripts:{username=<string>, success=<boolean>, filesize=<integer>, hash=<hexstring> [,errmsg=<string>]}

For example:

userloadscripts:{filesize=11766,hash="52f03bda386e767b694bddbb366f083a47a5300e52430cca47790f6496280384",success=true,username="simple"}

* On failure, in that a download attempt did not start; the QLE Lua software driver will output an error string beginning with the text “**userloadscripts:error:**” followed by an error message string. Refer to the QLE Lua software driver in the QCOM 3 SDK for possible error messages.

The minimum set of file transfer protocols to be supported is **http** and **ftp**. cp:[[393]](#endnote-305)

Related:

* The QCOM API functions **qcom\_userLoadScripts()** (11.30.27) and **qcom\_userSetScripts()** (s11.30.28).
* QCOM user script file archives and associated requirements (s5.3.1)

*Design Requirements:*

* *Consider version control? – e.g. no downgrading allowed.*
* *Consider Time-outs and other arguments.*
* *Consider the need to apply size quotas for the given QCOM user at download.*

### restartuser

**Syntax:**

**restartuser**

The QLE Lua software driver will perform a call to the QCOM API function **qcom\_userRestart()** for the invoking QCOM user. Refer section 11.30.30.

**Return values:**

restartuser:ok | restartuser:error:<error message>

This command is implemented as a part of the QLE Lua software driver via the qcom.userRestart() API function. Refer to the QLE Lua software driver in the QCOM 3 SDK for the possible error messages.

### shutdownuser

**Syntax:**

**shutdownuser**

The QLE Lua software driver will perform a call to the QCOM API function **qcom.userShutdown()** for the invoking QCOM user. Refer section 11.30.31.

**Return values:**

shutdownuser:ok | shutdownuser:error:<error message>

Implemented as a part of the QLE Lua software driver via the QCOM API function **qcom.userShutdown()**. Refer to the QLE Lua software driver in the QCOM 3 SDK for the possible error messages.

### dostring

**Syntax:**

**dostring <string>**

This commandallows the user to *load* and execute (*pcall*) the given string argument as a string of Lua script in the user’s environment. The script represented by the string argument must be compiled and executed via a state event in the machine’s QCOM event dispatcher.

Machine developers: The QLE Lua software driver will prepare the string argument for compilation, execution and then call the C-Lua function qle\_dostring(). Refer to this function’s functional description in the QCOM 3 Summary Spreadsheet and the example source code in the QCOM 3 SDK for requirements.

The *string* argument must be no more than **100** characters long and be comprised of only printable ASCII characters.

Any error messages with respect to the *load()* or *pcall()* of the string argument must be echoed back to the QCI session. Refer to the QLE Lua software driver in the QCOM 3 SDK for the error string format and content.

A QMA will generally not grant a QCOM user privilege to this command as it is intended more for test and development.

If a QCOM user has an SAA (refer section 6.3), then the QLE Lua software driver will automatically disable this command for the user.

### lua

**Syntax:**

**lua**

This command when invoked causes all subsequent received QCI commands to be treated as if they were the argument of a *dostring* command (see previous QCI command). All requirements with respect to the *dostring* command operation and arguments must be applied.

In order to exit this mode the remote peer must transmit the text “**exit**”.

A QMA will generally not grant a QCOM user privilege to this command as it is intended more for test and development.

If a QCOM user has an SAA (refer section 6.3), then the QLE Lua software driver will automatically disable this command for the user.

### docmd

**Syntax:**

**docmd <commandname:string> [<argstr:string>]**

Refer next section on Custom Commands.

**Return value:**

* On failure to invoke a command, the QLE Lua software driver output will be “**docmd:error:command not found**”.
* On success (a user function was found and will be invoked), QLE Lua software driver will output one line confirming the command name was found (“**docmd:found**”) followed by zero or more lines of text of whatever the invoked function may have print()’ed.

### help

Syntax:

**help**

Invoking this command in the QCI must return a list of QCI commands implemented by the machine. Each line output must list one command, just the command name and no other syntax. Each line must be **\n\r** delimited. The output must also include any created custom user commands (see next section), these commands must follow on from the list of hardcoded QCI commands.

### exit

Syntax:

**exit**

Invoking this command in the QCI must exit the session, logging the user out.

Related: QCOM state event **USER\_LOGOFF**

## Custom Commands

A QCOM user can add new commands to the QCOM Command Interpreter (if privileged to the facilitating QCOM API function below). This allows a QCOM user to extend the functionality of the QCOM Command Interpreter to meet their individual needs while also staying within their QCOM SAA approval (if an SAA is set for the user). One likely application of this will be to facilitate custom Remote Procedure Calls via the machine’s UAA service.

To create a new QCOM Command Interpreter command, a QCOM user may use the following QCOM API function:

**Call format:**

qcom\_luaQCIAddCommand(string, function[, global:boolean])

Added custom QCI commands must not be persistent across machine restarts and therefore must be reinstated by QCOM users after each reset of the machine application.

**Arguments**

The **first argument** of type***string*** denotes the function name as per how it will be invoked within the QCI. The function name must be a valid Lua identifier. To prevent collisions with existing QCI commands or other QCOM user QCI commands, the QLE Lua software driver will verify that the function name argument is prefixed with the calling user’s username, i.e. QCOM user “smrtone” can only add functions starting with the prefix “smrtone”.

The**secondargument** *of type* ***function*** is a user supplied function accessible within the user’s environment which will be called whenever the function name (first argument) is invoked within the QCI via docmd.

For the user supplied ***function*** to receive any argument string that may be been also supplied in the **domcd** command line, the function should call the QCOM API function qcom\_luaEventData() to retrieve it. If any arguments were also supplied they will be in an *argstr* key in the qcom\_luaEventData() table return value.

For example:

If the user setup a custom QCI **docmd** command via:

qcom\_luaQCIAddCommand(“GetMeters”, myfunc)

where *myfunc* was:

function myfunc()

local t = qcom\_luaEventData()

print(“---My GetMeters called: ”..t.argstr)

end

Then a subsequent QCI command line of:

docmd GetMeters stroke, turnover, wins

Will result in the output:

---My GetMeters called: stroke, turnover, wins

The full table schema of a user **docmd** function that calls qcom\_luaEventData() is:

{username:string, seid=”QLUAE\_DOCMD”[, argstr:string]}

Finally, the optional**thirdargument**of type***boolean*** of qcom\_luaQCIAddCommand() which named *global* in the call format. If set to *true*, it makes the function visible to all QCOM users when logged into the QCOM Command Interpreter. If *false* it will be available to the calling user only.

When the global flag is set, to avoid contention with any other user’s commands, a globally shared command name will be prefixed with the username of the function. For example, if a user called ***herbit*** invokes:

qcom\_luaQCIAddCommand(“GetMeters”, myfunc, true)

will create a **docmd** function for all users with the name “**herbit-GetMeters**”. Note the added hyphen as well. Upon any QCI session **docmd**, the QLE will search the QCI session user’s list of docmd’s first and if none found it will search the global list of **docmd** commands.

Notes:

* The resources used by a global docmd function count towards the user that calls it (and not the user that created and shared it). Accordingly a user should never call a globally shared function of a given user unless they trust that user.
* A user that creates a global docmd function can restrict the users that can invoke the command by placing a filter on the *username* key-value pair that is supplied as event data with every docmd function call.

**Return Value**

qcom\_luaQCIAddCommand returns *true* on success and *nil, errmsg:string* on failure. In the event of a failure, an optional second return value of type string denoting an error message may be returned.

# Progressive Prize Support

Progressive prize support in QCOM 3 is similar to progressive game support in QCOM v1, however the new QCOM 3 interface means more operating modes and encryption / authentication of progressive network traffic are now possible.

This section deals with EGM triggered SAP and LP prizes.

Note: The other primary type of linked jackpot (i.e. the opposite of machine triggered) referred to in this document as **external jackpots/prizes[[394]](#footnote-89)** is not dealt with or applicable to this section. External jackpot systems in QCOM 3, as per QCOM v1, are largely invisible to a QCOM 3 machines and utilise the system lockup feature and QCOM ECT to transfer wins to the machine credit meter. A QCOM user can optionally provide external jackpots with a greater presence on the machine such as in EGM audit mode reporting and by utilising the machine as a jackpot display (refer qcom\_egmSMS aka QCOM v1’s EXTJIP feature).

*Design Requirements:*

*The design requirements for basic EGM triggered progressive prize support in QCOM 3 are:*

* *Encompass all functionality as per QCOM v1 progressive support*
* *Support simultaneous hits of different jackpot levels as well as multiple jackpot hits within a single play.*
* *Support a wider range of jackpot types (QCOM advanced jackpot support, when released will support any jackpot concept that exists now or in the future. See next chapter)*
* *“What you see is what you win” (as per QCOM v1; the prize a player is entitled to is snapshot upon a revealed hit regardless of any outstanding contributions)*
* *Simultaneous win policies as per QCOM v1*
* *Per level totalisation for maximum flexibility*
* *Make no assumptions regarding prize triggering and prize totalisation methodologies that may be utilised e.g. do not make assumptions about the presence of parameters or calculations pertaining to jackpots. Related: QCOM Advanced Jackpots.*
* *Support multi-game machines and RUGM*
* *Allow the use of secure network authentication. (For example network based Message Authentication Code methods and protocols or VPNs)*
* *Support strong encryption / authentication*
* *Support and promote third party progressive jackpot displays and systems.*

## Terminology

Refer QCOM v1 and OLGR Jackpot System Minimum Requirements documents.

## Configuration and Control

As per QCOM v1, a machine with one or more games with progressive prizes must have a reasonably sane set of factory default progressive level properties.

A game may have 0…8 progressive levels (i.e., max 8), each of which may be SAP only, LP only, or both. This is as per QCOM v1.

**New to QCOM 3:** Where QCOM v1 supported progressive levels being shared across all progressive games in a multi-game machine, in QCOM 3, progressive levels may be shared between games without any constraint other than a given progressive level must appear in a game no more than once.

Each progressive game must have and maintain a current progressive configuration and progressive meters. This is as per QCOM v1.

Depending on the game / game designer, the current progressive configuration may be able to changed via **qcom\_progrSetp()** QCOM API function (i.e., progressive levels are customisable by an external host), or a game variation change via the **qcom\_gameSetVar()** QCOM API function (i.e., a progressive set/cfg per game variation which is hardcoded into the game). Similar again to QCOM v1.

If a game does have a varying set of progressive level properties per game variation, then the number of progressive levels across all those game’s variations must be equal. This is as per QCOM v1.

LP progressive prize levels must allow their level parameters to be modified via the QCOM API function **qcom\_progrSetp()**. This is **optional** for SAP only level’s, but support is recommended as it saves costs in adding new progressive levels later to an already approved game. This is as per QCOM v1.

**New to QCOM 3:** Progressive games may use new totalisation methods (aka custom / new level properties), may trigger multiple progressive prizes in any given instant / play.

Progressive games level’s mode (LP/SAP) may[[395]](#footnote-90) be switchable via the **qcom.progrModeChange()** QCOM API function. The only way to change a progressive level’s mode must be via this function. cp:[[396]](#endnote-306)

### Progressive Level Properties

QCOM 3 manages and controls progressive prizes (or levels) on a gaming machine via the following set of manufacturer assigned properties:

At the **game** level, the QCOM API function qcom\_gameGetp() (refer 11.20.7) reports the following read-only properties pertaining to progressives:

* Progressive Theme Name (ptname)\*
* Number of progressive levels in the game (pnum)

\*Progressive theme names are manufacturer assigned and are optional. If a progressive theme name is not applicable then the machine returns *nil* for this value.

At the **progressive** level, the QCOM API function **qcom\_progrGetp(*pluid*)**(s11.21.7) is the primary method for configuring progressives and returns the following properties per level:

| **Key\*** | **type** | **Description** |
| --- | --- | --- |
| pluid | string: lua\_ident global type | Progressive level UID.  Every progressive level must have a manufacturer assigned pluid which must be a unique within the EGM.  If two or more games in the machine share a progressive level prize, or externally linked, then the pluid must be the same value across the shared level in those games and be unique across all games.  A pluid must never equal any *gameid* in the machine.  Proposed naming convention:   * When a level is not to be linked / shared, then pluid be an underscore separated concatentation of *gameid* and *plevs* properties e.g. “BigBucks\_mega”. * When a level is to be shared / linked, then pluid be a underscore separated concatentation of *ptname* and *plevs* properties. |
| plevs | string: lua\_ident global type | Progressive level name.  e.g. “mega”, “major”, “mini”, “minor” etc. |
| ptm | string: printable incl. spaces; min length 1 | Prize Triggering Methodology.  This is a manufacturer assigned string. The value is arbitrary but must uniquely denote how the progressive jackpot/prize is:   * made available as a part of the game, * varied with respect to changes to dependencies, * triggered and * awarded   This property primarily concerns linked progressives. It may help service providers determine if two jackpots can potentially be linked. If two levels with the same *plev* (see previous key above) on two different machines have the same value for the *ptm* property, then the two levels theoretically should be able to be linked together provided any other dependencies match.  *Example values: “vanilla”, “TrilliumV1”, “VGB2stage” etc. The jackpot product / marketing name may often be most suitable.* |
| [var] |  | Game variation of this progr. configuration  Must be present if a game has a progr configuration per game variation. |
| mode | string | The currently set mode of progressive prize level; i.e. “sap” / “lp” |
| modes | btable | Supported modes of operation of the level. Possible modes are currently “sap” and/or “lp”.  A progressive level that is able to be mode switched must contain all properties as if it was a sap, even if set to lp. cp:[[397]](#endnote-307) Refer QCOM 3 summary spreadsheet ‘Progr’ worksheet for applicability information. |
| games | btable | The set of games in the local machine sharing this level. Machine manufacturer controlled. Table keys are of type *gameid*, values must be equal to true. |
| [pgid] | string | If a progressive level supports the “lp” mode of operation, then always include this field and set it to “” (an empty string) as its default value.  This field denotes the externally assigned linked progressive group ID for the level. The machine does nothing with this value, except report the set value back as a property in this table and display in machine audit mode for verification purposes. |
| [hrate] | number | Hit Rate. The theoretical probability (p) that the jackpot prize/level will be won for a bet equal to the value of one meter unit. Refer QCOM API function qcom\_machineMeterDenom().  If the value of this property may change (e.g. for different bets) then don’t report this property.  HRATE must be > 0.0 and < 1.0 |
| [atv] | number | The theoretical Average Trigger Value of the progressive prize/level. If the value of this property may change (e.g. for different bets), then don’t report this property.  See note 1 below. |
| rtp | number | Minimum theorectical percentage Return to Player for the progressive level.  It is accepted that the reported value of rtp may change. Any special behaviour here should be reported in the *notes* property below.  See note 1 below.  Related: 11.20.17:rtp |
| [rtpmax] | number | Maximum percentage RTP for the progressive level. Report only if this value is different from the value of ‘rtp’ property above.  It is accepted that the reported value of rtp may change. Any special behaviour here should be reported in the *notes* property below |
| [rtptable] | number | A numerically and/or string indexed table. Optional. Each table key may represent either a bet amount (in credits) or game mode and the value is the corresponding RTP. Applicable to levels with a non-linear rtp distribution. For use at the machine manufacturer’s discretion. |
| pinc | number | Percentage Increment |
| sup | camt | Start-up (reset) value of the progressive level. |
| [hinc] | number | Percentage increment of the next jackpot reset value, facilitated by incrementing the overflow meter property. Related: qcom v1.x AUXRTP |
| ceiling | camt | Ceiling. Units are *camt* (refer QCOM table of global types) |
| vis | boolean | Visibility flag.  If true, then the level’s prize value is generally persistently displayed to the player.  If false, then the level’s prize is designed as a hidden prize. (i.e. the prize value is not displayed to the player until won). |
| turnover | camt | The total amount turnover that has contributed to the progressive level since last factory reset. This value must be the same for all levels for a given game. Excludes any adjustments wrt adjpos | adjneg properties.  In QCOM v1 this was known as the “LP Turnover Meter” |
| hits | integer | The total number of hits since last factory reset |
| wins | camt | The total number of wins since last factory reset |
| prizeiw | camt | The prize value which would be awarded to the player if the jackpot were won this instant |
| [liab] | camt | Applicable to SAP levels only.  This value is the current “value” / liability of the progressive component of the level.  This property must denote the amount to carry over in the event the jackpot level or game is decommissioned.    In a vanilla progressive level this property would equal the result of the calculation:  *prize – sup + overflow*  See note 1 below. |
| overflow | camt | Overflow. Any amount intended to be added to the next jackpot reset amount after the next hit. Typically incremented as a result of the ceiling being reached.  Also may include the hidden increment if one exists; refer *hinc* property. |
| adjpos | camt | SAP only.  The amount of positive adjustments to the jackpot level via the QCOM API function provided in the progr QCOM API class. |
| adjneg | camt | SAP only.  The amount of negative adjustments to the jackpot level. (This is a positive number) |
| setp | btable | Denotes to QCOM users the set of progressive level property key names whose values may be changed via a call to the **qcom\_progrSetp()** function.  Allowable keys: **pgid, hinc, pinc, sup, ceiling**  *Contact OLGR for approval before adding new properties to a setp table not listed above.*  *setp* examples:  Typical vanilla SAP & LP capable, or LP only level:  **{pgid=true, pinc=true, sup=true, ceiling=true}**  For a SAP-only reconfigurable level:  **{[hinc = true,] pinc=true, sup=true, ceiling=true}**  For a SAP-only, read-only level:  **{}**  The *setp* table for LP capable levels must always contain all a level’s primary properties (e.g., pinc, sup, ceiling) so that the QCOM user in the role of the LP tolaliser can tell the EGM what level properties it is using and FYI wrt other QCOM users. Related: QCOM v1 customSAP feature/flag |
| lasthit | table | A copy of the last **PROGR\_AWARD** state event’s data for this level when last hit/won. *Nil* otherwise. If this level’s *hits* meter is greater than 0, then this property must be a table |
| lastcont | table | A copy of the last **PLAY\_COMMENCED** state event’s *pcont*  data for this level. *Nil* otherwise. If this level’s turnover meter is greater than 0, then this property must be a table |
| [notes] | string | Arbitrary notes concerning the progressive level property table (e.g. prerequisites, dependencies and behaviour).  Example:  “Prize only able to be won at max bet. RTP varies in inverse proportion to bet however the base game compensates in kind”  *(NB the example above is not indicative of a prize that may be suitable to OLGR or any particular jurisdiction)* |
| … | … | Custom level properties.  The game designer may also add with permission, additional progressive level properties in the return value. Each progressive level may return a different set of properties**.**  Anyrestrictions/requirements to adding new properties are as per the qcom\_gameGetp() QCOM API function.  Game designers must not add **secrets** to progressive level properties. Please contact OLGR if a progressive level is being designed with properties requiring secrecy. (Secrets in progressive levels are indicative of a deterministic prize triggering algorithm of which some jurisdictions have restrictions on.) |

**Refer to the QCOM 3 summary spreadsheet, ‘progr’ worksheet, for a summary of the above properties which contains some additional information.**

[ ] above means the property is conditional and its inclusion depends on the properties associated definition and game designer.

Note 1: This property must be able to be calculated based on a formula using progressive level properties returned. The game designer should add new custom properties as required to make this happen. The formula used to calculate the property may vary depending how the level is designed to operate, which also may vary from level to level. If the required formula isn’t a “vanilla” progressive level formula, then it must be included as a note in the *notes* property.

Most properties in LP levels are FYI only. A simple echo back of what the LP totaliser has set for it. (This is as per QCOM v1.)

It is a requirement that the machine return sufficient properties which:

* permits reconciliation of the properties against the current progressive jackpot value (prizeiw).

Not every property need be involved in the calculation (e.g. ceiling), however the reconciliation formula must involve turnover, hits and wins metered properties. For example in a vanilla progressive, the reconciliation formula would typically be:

prizeiw + overflow = turnover x pinc + (hits+1) x sup – wins + adjpos – adjneg

*NB In subsequent versions of QCOM, the machine may report the actual formula used as a string in the return value to a call to qcom\_progrGetp().*

* are pertinent to the operation and behaviour of the jackpot, e.g. Ceiling value

Changes to SAP and LP level configuration properties may be performed via the QCOM API function:

qcom\_progrSetp()[[398]](#footnote-91)

Refer to the QCOM API summary for more information on this function.

### SAP Current Amount Adjustments

The following QCOM API methods are available to perform positive and negative adjustments to a SAP current value:

qcom\_progrPosAdj()

qcom\_progrNegAdj()

They are made separate functions as each type of operation has a different level of risk.

These functions are not applicable to LP levels as the current amount is inherently controlled and adjustable by the LP prize totaliser service provider.

Refer to the respective API functions and STH message descriptions in the QCOM summary spreadsheet for more information and requirements.

## Progressive Prize Meters

These are values returned by the machine after a call to the QCOM API **qcom\_progrMeters()** function.

The meters returned by this function are a total for the game. For totals for the Progressive level UID *(pluid)*, use the **qcom\_progrGetp()** QCOM API function.

|  |  |  |
| --- | --- | --- |
| **key** | **type** | **Description** |
| turnover | camt | The total amount turnover that has contributed to the progressive level since last factory reset. This value must be the same for all levels for a given game. Excludes any adjustments wrt adjpos | adjneg properties. In QCOM v1 this was known as the “LP Turnover Meter” |
| hits | integer | The total number of hits since last factory reset |
| wins | camt | The total number of wins since last factory reset |
| … | … | The machine may also return any number of additional custom meters in the table return value. Each game may return a different or custom set of game properties**.** Anycaveats to this are as per **qcom\_gameGetp()** QCOM API function. |

## QCOM API Progressive Function summary

Refer to the “*progr*” class functions in the QCOM API summary sheet.

## LP Prize Support and Operation

QCOM 3 (as per QCOM v1), assumes that simultaneous winners will be paid in full the amount shown to be won on the EGM, making LP amount resets a priority for LP totalisers to minimise this window.

### LP Prize Contributions and Totalisation

For an LP prize to operate, a jackpot totaliser service needs the amount of turnover applicable to the prize on a regular basis. In QCOM v1 this was the purpose of the “LP Turnover Meter” which was queued for transmission by the EGM upon every play.

In QCOM 3, LP prize totalisation is performed by an authorised QCOM user associated with the jackpot totaliser service. In order for the user to receive contributions in the same frequency and approach as per QCOM v1, the user would hook into the **PLAY\_COMMENCED** state event and call the qcom\_progrTurnoverMeter() QCOM API function to retrieve the applicable total turnover. This amount may then be sent over the network by the user to the jackpot totaliser service on a protocol of the service provider’s choosing.

As per QCOM v1, it is the job of the jackpot totaliser service provider, to calculate (based the total applicable turnover) the progressive increments.

### Prize Control

A prize totaliser service’s other role is to broadcast / synchronise the latest prize amounts over the network. This is done periodically for all prizes it maintains and immediately as a high priority for each successfully authenticated LP prize event.

Prize broadcasts / synchronisation may be via a protocol of the totaliser’s choosing. Encryption and message authentication is at the discretion of the server provider with respect to any regulatory requirements.

The QCOM user associated with the prize totaliser service in a machine would, upon receiving an update to an applicable prize value, use the qcom\_progrSetPrize() QCOM API function to set the latest LP prize values for applicable LP prizes in the machine.

### Awarding LP Jackpots

As per QCOM v1, upon a machine triggering a LP award for a given level, the amount the machine is required to award **must be locked** to the last value it received for the level as set via the QCOM API function **qcom\_progrSetPrize()**.cp:[[399]](#endnote-308) Also, as per QCOM v1, for security reasons, there must be no ability in QCOM 3 for a linked jackpot system to deny or change the amount the machine will award (and meter) once the machine has triggered a win. cp:[[400]](#endnote-309)

In QCOM 3, the use of the LP lockup state in QCOM 3 is also still mandatory (as per QCOM v1). However, in QCOM 3, multiple LP awards per play/spin/feature/lockup may be triggered simultaneously (back-to-back) for a single LP award lockup.

When a QCOM 3 machine triggers a LP award, it must log the (ref s13.6.1) **PROGR\_AWARD** state event followed by the associated **EVENT** state event **(LP\_AWARD** event) before logging another progressive award’s events.[[401]](#footnote-92) cp:[[402]](#endnote-310)

When the machine enters a LP award lockup state, it must throw the **LP\_AWARD\_LOCKUP\_ENTRY** state event and wait for a **qcom\_progrResetLockup()** QCOM API call in order to exit the lockup. cp:[[403]](#endnote-311) This method must be the only way to exit the LP award lockup state (lp\_award). cp:[[404]](#endnote-312) [[405]](#footnote-93)

It is the decision of the QCOM user associated with LP prize totalisation service provider as to whether to ECT the LP win amount to the machine’s credit meter.

The EGM must update its *lpwins* EGM meter and its *lpwin* game and variation meters upon logging the **PROGR\_AWARD** state event. cp:[[406]](#endnote-313)

*Note: LP wins continue to be metered separately as LP prizes are often paid by a third party via a trust account and this requires special consideration in some jurisdictions (such as Queensland) with respect to how tax is calculated. If a SAP needs to be paid by a third party (it awards very large prizes for example), then it should be setup and operated as a LP comprised of a single game.*

*Note: As LP awards are not added to the player displayed win meter, this means that LP wins still cannot be gambled (double ups) etc. (This is as per QCOM v1).*

**QCOM User LP Award Detection**

To detect a LP award, a QCOM user could hook onto either the **PROGR\_AWARD** state event[[407]](#footnote-94), or the **EVENT** state event (which follows) and look for the **LP\_AWARD** event type.

Once a new **PROGR\_AWARD** / **LP\_AWARD** event is identified, typically the event and details would be forwarded over the network by the QCOM user associated with the jackpot totalisation service to the jackpot totaliser host for processing. As soon as the prize totaliser host service sanity checked and authenticated the event, it would reset the prize pool, immediately broadcast the new reset prize values and inform their QCOM user on the machine of any special instructions (such as to ECT the prize to the machine’s credit meter).

If there were any issues with the LP award, the QCOM user associated with the jackpot totalisation service would leave the EGM locked up in the LP award lockup state.

**Multiple LP awards during a single play**

As mentioned above, a QCOM 3 machine may trigger multiple LP awards per play, spin, or feature. The awards may be any combination of staggered (a LP lockup state for each[[408]](#footnote-95)) or simultaneous (a single LP lockup state for multiple), or for the same prize level vs different prize levels.

## Specific arrangements

### Multi-variation progressive games

**Multi-variation progressive games with different progressive level configurations per game variation.**

In QCOM v1, the above was supported at selection of a VAR at commissioning, but once set, the variation (and therefore the progressive level properties) couldn’t be changed without a RAM clear, as QCOM v1 required variation hot-switching had to be disabled for games with this feature.

In QCOM 3, a progressive level cfg per game variation is also permitted (as per QCOM v1). However, in QCOM 3, variation hot switching must always be possible (without RAM clear) cp:[[409]](#endnote-314) meaning progressive level properties may change with game variation changes. When this occurs, the EGM must automatically manage the carry-over of any progressive level contributions and similar values when a level’s properties are changed either by a game variation change, or via the QCOM API function **qcom.progrSetp()[[410]](#footnote-96)**.

The suitability and timing of a game variation change of a progressive game that has been in play (that is one that could also result in a backward jump in a progressive prize value) is a matter for the regulator / operator.

### Hardcoded vs non-hardcoded progressive level properties

LP levels fall into the category of non-hardcoded progressive level properties as their properties are set via externally via the LP totaliser service/device / operator. This means this section is a discussion about SAP level properties.

EGM hardcoded progressive level properties:

+ avoids operator error; ensures that exactly what the game designer intended can only be used.

- More cost to add more progressive sets/configurations post approval, as the game must be reworked, reapproved & updated.

EGM non-hardcoded progressive level properties pros/cons are the opposite of the above. Related: EGM non-hardcoded progressive level properties are configurable via the QCOM API function **qcom.progrSetp()**

# Advanced Linked Progressive Prize Support

**This section pertains only to machine triggered linked jackpot systems and is still in the concept stage.**

*The design requirements for advanced EGM triggered linked jackpot support for QCOM 3 are:*

* *As per SAP prizes in EGMs; allow LP prizes with unique jackpot totalisation and/or trigger methodologies to be not dependent on third parties for the implementation those new methodologies in operation.*
* *Make no assumptions regarding prize triggering and prize totalisation methodologies that may be utilised and support all current and future methods without knowing what they are.*

*Please note that a feasibility study of QCOM 3 advanced LP prize support for use in the State of Queensland’s with respect to applicable policy and regulatory environments for machine gaming has yet to be completed.*

The problem QCOM 3 advanced LP prize support solves

In some markets (those with QCOM v1 machines), there exists a very diverse range of SAP prizes, but in contrast, most LP prizes are predominantly ‘vanilla’ in design. This is often as a result of an operating environment where creative facets of LP prize totalisation are operated by parties who are not also the LP game designer. QCOM v1.x linked progressive support and also QCOM 3’s basic jackpot solution both have this limitation. QCOM 3’s advanced LP prize support addresses this limitation and as a result we should see a much greater diversity in LP prize arrangements in QCOM machine markets

As an example to illustrate the concept of QCOM 3 advanced LP prize support, below is a comparison of QCOM progressive prize types which show how QCOM 3 advanced LP prize support differs from traditional LP prizes in terms of **who-does-what**: (The 1st party of the EGM manufacturer, the 2nd party is the machine owner)

|  |  |  |  |
| --- | --- | --- | --- |
| **Tasks relating to the operation of a *LP prize arrangement*** | **SAPs (FYI)** | **Existing QCOM LP prize support** | **QCOM 3 advanced LP prize support** |
| Prize / game designer | EGM | EGM | EGM |
| Prize parameter configuration | 3rd party | 3rd party | 3rd party |
| Adding / removing machines from the link | 3rd party | 3rd party | 3rd party |
| Enabling / Disabling the link and EGMs on it | 3rd party | 3rd party | 3rd party |
| Prize totalisation and resetting | EGM | 3rd party | EGM |
| Triggering the prize | EGM | EGM | EGM |
| Monitoring the link / EGMs | 3rd party | 3rd party | 3rd party |

There is actually very little change required in the overall, but it comes with significant advantages in return.

Embodiments of QCOM 3 advanced LP prize support

There are several solutions for the delivery of QCOM 3 advanced LP prize support are being considered. Namely:

* **Black Box (BB) solution**. This term denotes the solution using EGM manufacturer provided discrete totaliser devices / embedded systems.
* **Virtual Machine (VM) solution**. This term denotes the solution using EGM manufacturer delivered scripts representing a totaliser application for hosting by third parties machines in virtual scripting environments / virtual machines. (Similar in many respects to how the QCOM 3 scripting engine operates in a machine.)
* **Peer-to-Peer Totalisation (P2P) solution**. Where each machine on the link is a full LP totaliser. This solution has very high redundancy but is complicated to implement.

In all cases:

* Totalisation machines (either embedded, virtual, or built into the EGMs themselves) should be evaluated and approved as regulated gaming equipment.
* Totalisation of machine triggered linked jackpots when done right requires negligible network bandwidth, memory, CPU and support. QCOM 3 will may publish requirements in this regard in subsequent versions.
* A simple monitoring and control interface will be made available via any participating machine on the link via the QCOM 3 API. This includes information needed for jackpot display systems.
* Significant events concerning the totaliser[[411]](#footnote-97) are stored in every machine on the link, accessible via the QCOM 3 API. Minimising events is a design requirement.
* Totalisation machines must have a secure network based
  + - www UI and
    - console UI,

adhering to QCOM 3 minimum requirements (tba).

* While protocols may vary (see below), output from jackpot totalisers such as events, configuration and meters will be a QCOM 3 standard.

More information with respect to each solution:

* BB Solution.

The totaliser and EGMs communicate via a network protocol of the manufacturer’s choosing. The protocol utilised must be evaluated and approved (as a part of the overall totaliser evaluation and approval) and must meet the upcoming minimum requirements (principles) for QCOM 3 jackpot totalisation.

*In a variation of this solution, a nominated EGM on the link could act as the totaliser here. (This saves another hardware approval as any EGM is more than capable of performing the job of a LP totaliser.) The nominated EGM becomes the master, so if it goes down then another machine would have to take over. I’ll spare the reader the logical flow-on discussion here into LP totalising mirroring across EGMs and automatic fall-over, as they are all already established techniques in computing. This is seen in on-line multiplayer gaming. Google “host-migration”. Also refer to* [*https://en.wikipedia.org/wiki/Game\_server#Peer-to-Peer*](https://en.wikipedia.org/wiki/Game_server#Peer-to-Peer) *Note, a LP totaliser is a far easier application to do this with than your average computer game (where there are hundreds of variables that need sharing, syncing as well as totalising; including physics engine states and variables).*

* VM Solution.

In this solution, QCOM 3 will define a jailed scripting environment for LP totaliser scripts and a party suitable to the regulator provides the totaliser VM host machines[[412]](#footnote-98). This is similar to the way QCOM 3 defines environments for QCOM 3 users (e*xcept there will not necessarily be any requirement for multi-user support. In comparison, a LP totaliser environment will be far smaller than a QCOM 3 user environment.*) The environment includes and interface to the hosting party allowing them to perform all the “3rd party” tasks in the LP arrangement while letting the game use whatever method it desires to totalise and control/reset the prize amounts. The host environment includes methods which allow the totaliser VM scripts to fault tolerantly save its state on demand, communicate[[413]](#footnote-99), as well as use encryption. Similar to many of methods a QCOM 3 machine provides to QCOM users as a part of this specification.

The VM solution is recommended for existing QCOM operating environments for linked jackpots as it maintains existing operating arrangements for the links. The totaliser VM scripts can be considered just another a jackpot parameter in the evaluated and approved set of parameters for a given jackpot.

* P2P Solution.

This solution is strictly only for the talented. Again existing established methods in computing would be used to realise this solution. Special issues concerning totalisation exist around: who is the most current; synchronisation of the current amount; whose current amount does a display sign use (it can’t go backwards and all machine displays of current amounts must match.)

The EGMs communicate among themselves via an EGM manufacturer network protocol of the manufacturer’s choosing.

The specific requirements for each solution (excluding the P2P solution) will be published in future releases of this specification document.

# Content Auditing

*Design Requirements:*

* *Support a wide range secure hashing algorithms*
* *Support for seeded (HMAC) & non-seeded hashing*
* *While the hashing algorithm in any given machine will be hardcoded (at least in the initial specification), the specification must have ability to easily upgrade the secure hashing algorithm for new machines*
* *Variable and extensible audit reports*
* *Facilitate high level componentised software auditing to facilitate component swapping e.g. RUGMs i.e.*
  + *A game can be swapped out and the operator can still easily verify the EGM’s hash*
  + *Efficiently support an infinite number of permutations of a large pool of games*
  + *Support the efficient management of upgrades to individual machine components (e.g. BIOS upgrades)*
* *The ability to produce hashes to suit a wide range of regulatory environments in which the machine is located.*
* *Give consideration to the hashing down to any desired level of detail.*
* *Support for optional components/peripherals (e.g. Note/coin acceptors)*
* *The ability to detect a failure to report one or more expected components.*
* *As games take the longest to report a hash, the ability to hash by game and enable games as they report a correct hash result.*
* *Promote but not mandate reconciliation with PSDs.*
* *Consideration to missing extra result detection*
* *Protect the manufacturer’s IP by not exposing (e.g. down to the file level) the full file manifest of the device.*

*Design Scope Limits:*

* *Does not automatically discover missing content*
* *Does not verify dependencies*

refers to what was termed Program Hash Request/Hash responses in QCOM v1. In QCOM v1, there was simply one single combined overall HMAC-SHA-1 result returned for the entire device for any given *seed* or program hash request. In QCOM 3, a categorised approach to content auditing is taken and is broken down into **three** categories:

* **Common content**
* **Game content**
* **Peripheral content**

Machine content is categorised primarily by the machine manufacturer however the breakdown may also need to be approved by a regulator and there are certain requirements as follows.

There must be no duplication of content hashed across all categories. cp:[[414]](#endnote-315) i.e. if some content is included in one hash, then it must not be included again as a part of another hash in the same or any other category.

In gaming machines, a game change must not affect any of the other hashes returned by the machine (only the game hash results must change) and conversely, a change to a machine’s BIOS must not affect the hash returned for a game.

Hashing across all categories must constitute a total content verification/audit of the machine. cp:[[415]](#endnote-316)

A content audit in progress must be aborted by the machine if the machine is interrupted during the procedure (e.g. power down). cp:[[416]](#endnote-317)

Hashing Algorithms:

The hashing algorithms that must be supported by QCOM 3 machine are as per the openssl dgst command. cp:[[417]](#endnote-318)

HMAC must also be supported.

Combining multiple hash results:

Generally in QCOM 3, hash results should not be combined and be individual items in a returned hash table via the respective QCOM API \_cAudit class function. However where a content hash result is the combination of multiple hash calculations, then the hash result must be a product of:

* a chained hash, or,
* a hash operation across individual hash results (all calculated in full), or
* A XOR sum of each partial hash result. This last approach must also include some form of hash of a manifest of individual results XOR’d in, or some other technique which ensures that by simply including an item twice, it does not exclude the item from the overall combined result. (Property of XOR)

Hash Seed/Result Format:

The Lua *string* base data type is used to convey seeds and hashes via the QCOM API. Refer to the QCOM global types: *hexstring* and *HMACseed*.

Hash results must be formatted as lowercase hex-strings, least significant byte first, zero padded (to the hashing algorithm width) and contain no other formatting (e.g. spaces, 0x prefix etc.).

SHA256 / HMAC-SHA256 quick example for verification purposes:

Data**(**text **string)** **=** "abc"

seed**(**hex **string)** **=** "abcdef0000"

-- Note; any number of trailing 00’s in the seed above will not change HAMC-SHA result

SHA256**(**Data**)** **=** "ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad"

HMAC**-**SHA256**(**Data**)** **=** "7fb6acfd763f1673c0fdcb26c723ce23bc02b5c6655dba6fdbc49a320e3db250"

*QCOM Web Interface – Optional Concept/Proposal*

*Machine Manufacturers have the option of implementing a full manifest dump on the QCOM www interface. Refer 29 for more information.*

Three categories (API methods) concerning QCOM 3 machine content auditing:

* **Common content**
  + Content: BIOS’s, Operating System, generic applications etc.
* **Game content**
  + Two content types;
    - game application software / data, and
    - game sound/graphics.
* **Peripheral content**
  + Content: Coin/Note acceptors etc.
  + Firmware / Validation set version *string* reporting (content not hashed)

The QCOM API concerning content auditing must be implemented as non-blocking.

QCOM API functions concerning machine content auditing are listed in section 11.13.

Content audit hash results must not persist across a machine restart.

## Common Content Auditing

The definition of what is common content is not strict and is largely at the discretion of the machine manufacturer. It is permissible that there may be no common content for a given machine if the content is encompassed within another category e.g. as a part of application / game content.

Examples of common content:

* Machine BIOS,
* Bootstrap,
* Operating System (OS) and
* other shared or common application code or content in the machine.

In a gaming machine, common content would encompass the RNG and the generic portion of game outcome control.

When the machine receives a common content hash request (QCOM API function **qcom\_cAuditCommonStart()**), the machine must return a table of results within a timeout period[[418]](#footnote-100) via the **CAUDIT\_FIN\_COMMON** state event. The QCOM API function **qcom\_cAuditCommonResults**() may be used by a QCOM user to get the results in a Lua table. Each entry in the returned table must comprise of a manufacturer assigned human readable string **UID** and a corresponding string hash result. Example of an entry:

|  |  |
| --- | --- |
| **UID** | **Hash Result** |
| "Logic Unit OS Build;v12.01.01;2012" | "95fbd1c83ffb8ef34b3da249324015a5eb83d1210c06af01406bc1389165a45c" |

The number of entries in the table is machine/model/manufacturer, however it may also need approval in some jurisdictions. An example of typical entries in a machine might be as follows:

* Logic Board – BIOS
* Logic Board - Bootloader
* Logic Board - CPLD
* Logic Board – Operating System (OS)
* Logic Board - Application
* IOBoard - BIOS
* IOBoard - CPLD
* IOBoard – Firmware

The table may contain as many entries as required by the machine model (or none at all if the content is covered under another category e.g. application / game content hashing).

In Queensland, the list must be approved once per platform. The reason for having multiple entries is to allow for the possibility of a more efficient approval process for the upgrades of individual components of an overall machine to occur and also to allow a degree of interrogation as to which component failed an audit check.

Correlation between the returned table entries to actual physical PSDs in the machine is helpful and the UID may include location ID if desired, but this is not mandatory if it is not convenient for a given machine type. For example, many PSD’s may have virtual images inside which are better treated as individual entries in the table rather than an overall entry of the entire PSD. However if a table entry does correspond to a whole physical PSD then its p.c.b. location number must also be a part of the entry’s descriptor, e.g.: “Logic BIOS;v01.01.01;2014;U15” (where U15 is the board location).

In summary, the following functions are specific to common content auditing:

**qcom\_cAuditCommonStart(alg:string[, seed:hexstring])**

The above function initiates a hash based calculation of all content categorised as common content in the machine using the specified hashing algorithm. Refer section 11.13.1 for more information.

When a common content audit is complete, the machine must throw the state event **CAUDIT\_FIN\_COMMON**.

**qcom\_cAuditCommonResults()**

The above function retrieves the last completed common content audit results (even while the machine is currently calculating a new set). If no results have been generated yet, the function will return a table comprised of only uid values.

The return value must be associative and numerically indexed table where each numeric key value is an associative index table formatted to the following schema (Lua):

rv **=** **{**

bvn **= string,** platform **= string,**

**[**seed **=** **string,]** -- Applicable to HMAC only

alg **=** **string,**

**{**uid1 **=** **string,** hash1 **=** **string},**

**{**uid2 **=** **string,** hash2 **=** **string},**

**...**

**{**uidn **=** **string,** hashn **=** **string},**

**}**

|  |  |
| --- | --- |
| **Key** | **Value** |
| bvn | The game’s BVN as per QCOM v1. Refer table of global types in the QCOM 3 summary spreadsheet. |
| platform | Type *string*. This string denotes the platform or product name of the host machine. It must be indicative to number of expected *uid* entries in the return value and their respective *component ‘name’s* (see *uid* just below) i.e. ‘*platform*’s value must change if either of the above changes. |
| seed | Type *hexstring*. HMAC seed value. Its presence denotes a HMAC calculation was performed. |
| alg | Type string = “SHA256”. Denotes the hashing algorithm used by the machine. |
| uid | Type *string*. Manufacturer assigned unique identifier. The *uid* denotes a common component UID. A *uid* value must be unique with respect to all other *uid*’s in the machine’s *platform*.  Max length is **80** characters.  Allowable characters: **a-z A-Z 0-9\_-;.(){}** plus the space character.  Semicolons are restricted for use as field separators, see below.  The uid and each character following the ‘;’ character must start with a letter or number character.  The value must encapsulate the following information in the order shown:  **ComponentName;version;year[;location][[419]](#footnote-101)**  The *componentName* field in this string must be a human readable string (i.e. not be encoded or a hash) with respect to the machine’s *platform*.    The *uid* must also contain software *version* and *year* information. Physical *location* information is optional, but is encouraged when the information could be useful (such as in cases where the component corresponds to a physical device in the machine or p.c.b. locale). If location can be multiple physical locations, then just generalise it so that the location text does not change from machine to machine e.g. “USB\_internal” instead of say “USB3”. |
| hash | A *hexstring* (refer QCOM global types) denoting the hash result of the content denoted by UID. The length must correspond to the hashing algorithm used.  Default *nil, see below.* |

If there are no prior results yet calculated to report, then the function must return the table with *nil* for all the *hash* values. This allows a user to obtain a content list event before an audit has been completed.

Example, if the common content audit result was as below for the *seed* “1fd5cd550cb81fa352cba77e0eb86754e03b3aec”

|  |  |
| --- | --- |
| **Common component UID (uid)** | **Hash result (hexstring)** |
| seed | 1fd5cd550cb81fa352cba77e0eb86754e03b3aec |
| platform | Gen12 |
| alg | SHA256 |
| Logic BIOS;v7.12;2011;U14 | 6c47fb643bf3537be9fef88690109d3dc19b90132364765b8688346c16a4a41e |
| Logic CPLD;v6.00;2009;U67 | e97f23390299b4fa531a58d23174e15d4ea5c356f712872939f18c383957175d |
| Logic OS WindowsXP;v12.01.01;2012 | 075af7066000bf7bf36df5637845d50789594a1f31888b1d03966f6fa24b8c1c |
| Logic Main APP;v3.01.03;2011 | a1bca0109941cc5a2416d6076f1cc25184ded52a096fcc0ac1511b612085078e |
| Logic audit APP;v4.1.23;2011 | 98473b0bc4260e726f7b48c893373715880af41ac3b460e017ecaaf36e0e9a71 |
| IO – BIOS;v3.2.1;2011;U4 | 23ca4b584baea29c0b03ddb17455436e8965968e1be417c3db11d40be5522e69 |
| IO – Firmware;v6.00.03;2011;USB0 | 15c11f6746eebb84f6a1796e65684f1eff61207aa796e6317461146607961b7e |

Then with respect to the example table above, if:

hr = qcom\_cAuditCommonResults()

was invoked, then the variable *hr* would equal (in Lua):

hr **=** **{**

platform = “Gen12”**,**

seed **=** "1fd5cd550cb81fa352cba77e0eb86754e03b3aec"**,**

alg **=** "SHA256 "**,**

**{**uid **=** "Logic BIOS;v7.12;2011;U14"**,**

hash **=** "6c47fb643bf3537be9fef88690109d3dc19b90132364765b8688346c16a4a41e"**},**

**{**uid **=** "Logic CPLD;v6.00;2009;U67"**,**

hash **=** "e97f23390299b4fa531a58d23174e15d4ea5c356f712872939f18c383957175d"**},**

**{**uid **=** "Logic OS WindowsXP;v12.01.01;2012"**,**

hash **=** "075af7066000bf7bf36df5637845d50789594a1f31888b1d03966f6fa24b8c1c"**},**

**{**uid **=** "Logic Main APP;v3.01.03;2011"**,**

hash **=** "a1bca0109941cc5a2416d6076f1cc25184ded52a096fcc0ac1511b612085078e"**},**

**{**uid **=** "Logic audit APP;v4.1.23;2011"**,**

hash **=** "98473b0bc4260e726f7b48c893373715880af41ac3b460e017ecaaf36e0e9a71"**},**

**{**uid **=** "IO BIOS;v3.2.1;2011;U4"**,**

hash **=** "23ca4b584baea29c0b03ddb17455436e8965968e1be417c3db11d40be5522e69"**},**

**{**uid **=** "IO Firmware;v6.00.03;2011;USB0"**,**

hash **=** "15c11f6746eebb84f6a1796e65684f1eff61207aa796e6317461146607961b7e"**},**

**}**

The presence of a seed denotes a HMAC based calculation was requested.

The absence of any field relating to a hash means that a content audit has not yet been completed since the last power up.

*While the overall naming convention is with the manufacturer, consideration should be given the use of certain standard string identifiers which must be a part of the descriptor for certain items e.g. Operating systems must include the string “OS” & “Windows”, “Linux”; “BIOS”, “IO” etc.*

## Game Content Auditing

Game content auditing is specific to gaming machines and similar applications. Game content auditing must be able to be performed on a per game basis. Game content is sub-divided into two categories:

* Category “cd” - game specific code, pay-scale, rules and reel strips layout etc.
* Category “media” - game media such as sound and graphics.

For each of the two types above there is a single hash result returned by the applicable API function (i.e. hashing is performed one game at a time).

A summary of the Lua API concerning game content auditing is as follows:

**qcom\_cAuditGameStart(gameID, category, alg[, seed])**

This function initiates a hash calculation of the specified category game content in the machine. Refer section 11.13.3 for more information.

When a common content audit is complete, the machine must throw the following state event:

**CAUDIT\_FIN\_GAME**

Results:

**qcom\_cAuditGameResults(gameid, category)**

The above QCOM API functions retrieve the last completed game content audit results. The return value must adhere to the same requirements as the return value for a call to **qcom\_cAuditCommonResults()**QCOM API function.

An example for the “**cd**” category is as follows:

|  |  |
| --- | --- |
| **Game UID**  (refer global type: *gameuid*) | **Hash result** |
| Lots of Bucks;v03.01.00;2014 | 773541266f0b9d8af068869d44ac4ea1f6d1ba88dc6ec007449f3e0915a9464b |

If with respect to the example table above:

**rv = qcom\_cAuditGameResults(gameid, “cd”)**

The variable *rv* would equal in Lua:

rv **=** **{**

seed **=** "1fd5cd550cb81fa352cba77e0eb86754e03b3aec"**,**

alg **=** "SHA256”,

**{**

uid **=** "Lots of Bucks;v03.01.00;2014"**,**

hash **=** "773541266f0b9d8af068869d44ac4ea1f6d1ba88dc6ec007449f3e0915a9464b"

**},**

**}**

The presence of a seed denotes a HMAC based calculation was requested.

**Category 2 game content**

An example for the “**media**” category is as follows:

|  |  |
| --- | --- |
| **Game component UID** | **Hash result** |
| Lots of Bucks;mv01.00.00;2014 | f3908e3da84ef31cead327fc670577c7335f6e379109e7d188f93ceee7dd46e0 |

If with respect to the example table above:

**rv = qcom\_cAuditGameResults(gameid, “media”)**

The variable *‘rv’* would equal in Lua:

rv **=** **{**

alg **=** "SHA256"**,**

**{**uid **=** "Lots of Bucks;mv01.00.00;2014"**,**

hash**=** "f3908e3da84ef31cead327fc670577c7335f6e379109e7d188f93ceee7dd46e0"**},**

**}**

Note the QCOM content auditing API inherently permits common content audit and a game content audit to be run simultaneously[[420]](#footnote-102). The machine must support this scenario. cp:[[421]](#endnote-319)

*Design note: In addition, the above approach to game content auditing would also allow a more broken-down approach in the results, such as separate entries in the return value for sound, graphics and game definition. This is not permitted at this time but if demand ensues this will be facilitated.*

## Peripheral Content Auditing

Peripherals are treated separately because they are typically optional components in a machine and also may or may not be responsive at the time of an audit being undertaken. The machine can continue to operate without these peripherals being on-line.

Only peripherals of concern are dealt with by QCOM (such as peripherals that deal with the validation of currency). Peripherals currently of interest here are Coin/Note validators. At this time all other peripherals in a gaming machine fall into a ‘don’t-care’ category.

In QCOM 3, each peripheral of interest to content auditing will have its own version query function, for example:

qcom\_bnaFirmwareID**() -- banknote version ID descriptor**

qcom\_caFirmwareID**() -- coin acceptor**

qcom\_tpFirmwareID**() -- ticket printer**

These function’s return values are compatible with the NADS field in QCOM v1.

The *string* **return value** must comprise only of only the following ASCII printable characters: **a-z A-Z 0-9\_-,.(){}** including the space character . The first character and each character following the comma character must be an alphanumeric character. The comma character is restricted for use within the string as a field separator, see below.

The *string* **return value** must encapsulate the following information in the order shown:

*Manufacturer Name,Model,Firmware ID [,Currency Validation Set Version if applicable]*

An example return value for a note acceptor would be:

"NotesOrBust,fiteon-model,FW(AUS)-10223-SS,VS003-03V140-10-AUS"

If any field cannot be ascertained by the machine, or is not applicable, then should remain empty, e.g. if the note acceptor does not report a model then:

"NotesOrBust,,FW(AUS)-10223-SS,DS003-03V140-10-AUS"

*Design Notes: Peripheral Content Auditing Rationale*

*Note that QCOM’s current peripheral auditing API provides version reporting only (i.e. no content hashing). This can be easily added given demand but was passed on for the following reasons:*

* *Coin and note acceptor content is at low risk for attack.*

*Even the best attacks on firmware in these types of devices will always attract attention because a reconciliation failure of the next physical currency audit will inevitably occur following an attack. In addition, as only an insider will have the specialist knowledge and tools required to make an attack, it focuses potential suspects to a very small pool of people making firmware attacks risky for these types of devices. Finally, because the gaming machine also places sanity checks on reported incoming amounts from these devices in both the amounts concerned and frequency thereof, the scope and size of any attacks may be limited further.*

* *The misconception that a hash result gives additional security.*

*Unless day-zero trust for a device is established and maintained, hash results for that device provide no more security benefit than a declaration of version ID. While the seeded hash request approach to hashing does force an attacker to intentionally program their way around this (thus possibly leaving evidence behind), in light of point 1 above, there is a question of cost benefit here pertaining to note/coin validators.*

*In conclusion, unless a peripheral/device is also physically security sealed and the seals are also monitored and known to be intact, the trust level of any SHA results transmitted by the peripheral is very low. As auditable and accountable physical security sealing is resource expensive and should not be done without significant benefits.*

## Related Conventions

Regulators (or delegated authority in the role of machine content verification) typically maintain a file repository of approved machine content to use for verification against the machine via the QCOM content auditing API. Machine manufacturers typically submit machine content as raw files to the repository when approvals are granted.

This section defines a **file and file naming convention** the manufacturer must use when submitting files to the above type of repository.

There must be one file per {uid, hash} as reported by the machine across the entire QCOM content auditing API. A file when hashed must correspond to one of the hashes returned by the machine via QCOM content auditing API.

The filename format must be:

<mid>-<hash>[.<omfe>].q3c

Where:

<mid> is the **3** character manufacturer ID. Refer qcom\_idMfr3() s11.1.9.

<hash> is a **40** character **SHA-1**[[422]](#footnote-103) of the *uid* field defined in section 27.1 to which the file corresponds.

<omfe> is optional at the machine manufacturer’s discretion. If present must be a **3** character filename extension denoting the underlying file type.

Finally, the filename extension must always be **.q3c** (which stands for “QCOM 3 Content”).

Example filename:

**abc-6daf365ea16d8b0907d84cd8a909c4e6942f5300.img.q3c**

Where a manufacturer is required to submit a build process to a regulator or testing authority (i.e. compile / build machine image/application from source code) the build process submitted must automatically also produce the applicable .q3c files.

## Other Notes Concerning Content Auditing

*This sub-section is not mandatory reading – for discussion purposes only. There are no requirements in this sub-section.*

Chain-of-trust content control and authentication

Some machines implement a chain-of-trust type approach to content control and software upgrades/authentication.

For example in some machines there is code executed at machine start-up such as a BIOS or boot loader, which uses a store of trusted certificates (or just one) to ensure that from that point on the machine will only execute modules and code which authenticate. (There is also typically some manufacturer specific DRM present to protect the boot device and its trust store but this is out-of-scope for QCOM.)

OLGR has no objection to this type of DRM in QCOM 3 machines. In relation to this, in machines where OLGR can also be a trusted signatory (i.e. modules in the machine will not load or execute unless the module has a authentic signature from both the machine manufacturer and OLGR) then this potentially means in operation that the only content hash data that must be verified regularly is the BIOS or boot loader (the root authority in the machine), the remainder of hash content data returned becomes for-information-only and does not need checking on a day-to-day basis.

Machines are encouraged to publish certificates on their www interface for verification purposes.

Related: 6.2 Software Upgrade Authorities (SUA)*.*

*Related: OLGR publication - Principles for Remotely Upgradeable Gaming Machines - Package signing/verification requirements*

Dependencies

Dependences in machine software (and hardware) components are typical. In some jurisdictions, regulators approve specific arrangements of machine software components; often third parties such as monitoring operators are required to ensure that only approved combinations of content are used in production machines. It would be better if the need for this could be avoided as the side effects can be complicated and lead to more costly approval processes and management.

If machines could automatically detect incompatible or obsolete arrangements and react accordingly, then the need for a regulator to approve specific arrangements and have them verified could be diminished or eliminated. Some IQ in the area here in machines may not difficult to implement with a little extra planning during machine software design.

For example, take a simple EGM with the architecture: BIOS->OS->BaseApplication->GameData. If the BaseApplication had a function, resident since day-zero, which checks GameData for any embedded *not-before* BaseApplication version information, then a new game that required a specific base-application version could be detected in a production machine even one running an older base application version. The machine could gracefully prevent the arrangement from being enabled. Arguably the regulator does not need to worry about adverse side effects from the wrong base being used with the game as the machine will prevent the arrangement from going live in the first place. The risk moves from one of integrity to a short term availability issue.

The summary regarding this discussion on dependencies is that it is recommended that QCOM 3 machines have software compatibility detection built into core software components wherever possible.

# Machine Audit Mode

This section outlines audit mode requirements related to the implementation of QCOM 3 in a machine.

Because of QCOM 3’s multi-user support and its programmable nature, each QCOM 3 machine inherently has a remote audit mode interface built in. This allows for a minimal set of QCOM 3 machine audit mode requirements.

Audit mode of a QCOM 3 machine may (at the discretion of the machine manufacturer), include a display any or all QCOM related global type values where it does not also constitute a security risk.

Specific items that must be displayed in machine audit mode are: cp:[[423]](#endnote-320)

* Time and related values (must be displayed in real-time)
  + The machine’s current
    - local date and time
    - UTC date and time
    - Time zone bias (units: +/-hh:mm); relative to GMT; where east is positive
    - Unix epoch time value[[424]](#footnote-104)
* A log display of the last **20** calls (min) to **print()** by the QLE Lua software driver since start up. Related: section 10.2.10 & *stdout*.
* A list of all certificates and their associated fingerprints held by the machine. Refer section 8 for more information.
* For each QCOM user, display:
  + The user’s username.
  + A SHA256 fingerprint over the user’s UAA public key PEM formatted string (if present).
  + If the user is quarantined or not.
  + The user’s last .lua source code hash as calculated by the QCOM Lua software driver. (Refer to the **USER\_READY** state event for the hash value.)
  + A log of each user’s the last **20** calls to **print()**. Related: section 10.2.10 & *stdout*.
  + The contents of all the files inside the user’s script archive with the file extensions .txt and .lua. This is primarily to ensure that any software license messages required to be displayed by QCOM users, are able to be displayed on the machine in order to satisfy the terms of those licenses. This requirement is also useful for diagnostics and transparency reasons.
  + Any QCOM user defined pages. Refer s11.16.1.
* The text “QCOM 3” somewhere discrete in the top level audit mode display.
* QCOM Event Buffer (s13).
  + Must have the ability to display all events, or just events in a single category.
  + The display must be sorted on the global event serial number.
  + The event timestamps (the time the event was originally logged) must be displayed in a standard human readable format in local time. (Do this for all timestamp displays on the machine excluding x509 timestamps which are UTC)
* Refer section 32.3 - Audit mode relating to the MDP.
* The status of the QCOM 3 QLE software driver:
  + Panicked : boolean
  + Last error : string (if any)
  + Heartbeat count : integer (refer QLE heartbeat message).

The machine may display more information as desired. See QSIM 3 – QCOM user GUI for examples.

For any item displayable in machine audit mode that can change in real-time that is not automatically updated in real-time by the machine, then this must be indicated as such with either a message, or the presence of a manual refresh button. cp:[[425]](#endnote-321)

All units of all items displayed in audit mode must be displayed where applicable or not already indicated by the descriptor. cp:[[426]](#endnote-322)

For the display of any timestamps, the timezone must also be stated unless the timestamp denotes a local time which requires no timezone clarification.

Machines much not not display in audit mode secrets and private data such as: cp:[[427]](#endnote-323)

* Private keys
* QCOM user environment variables and persistent variables.

If is acceptable for the machine to optionally display in audit mode:

* Anything the anon user is privileged to see via the QCOM API.
* Any data returned by QCOM API read-only functions that are marked as 0 or blank risk in the QCOM 3 summary spreadsheet.
* Performance statistics concerning the QLE, QCOM users, UART / IP use.

Related:

* QCOM machine commissioning: section 4.
* Diagnostics, section 10.2.12.
* Privacy, section 10.2.4.

## Possible Machine Audit Mode Future Functionality

Provisions in this sub-section are not currently mandatory unless stated otherwise.

**Remote access to machine audit mode (discussion only – this is not a requirement)**

In the future, accessing EGM audit mode via the traditional physical interface at the machine on its built-in display, may be just one method of a number of methods to access machine audit mode.

Another expected approach to machine audit mode access in the future may be via remotely over the network (either wired or wireless access), via remote computers or hand held portable devices.

OLGR has no objection with QCOM 3 enabled machines providing this feature at this time.

In addition to the inherent advantages pertaining to audit mode remote access, another significant advantage is that depending how it is implemented; audit mode operations could potentially be performed without disrupting the machine’s core function (e.g. “play” in the case of a gaming machine).

In the longer term there may be an initiative to make access to machine audit mode and all associated functionality, available via both the traditional built-in audit mode display on the machine and remotely available via a network. To save on development resources, it would be sensible to propose that the machine utilise the same UI and program code to deliver audit mode functionality whether its access is over the network or directly via the machine.

Just as traditional EGM audit mode access requires a key, remote access to audit mode will also require some form of access control mechanism e.g. username / password or public key techniques (TLS) access control methods. The QCOM API would be enhanced with an additional function that could setup either of the access control methods mentioned previously. (Single user support to start and multi-user support if demand ensues.)

One possible scenario envisaged is that attendants with Wi-Fi enabled portable devices would be able to access a machines audit/test modes and any other diagnostic function of the machine via an application (or generic web browser) on a third party or personal device.

**QCOM User delivered custom functionality and content via QCOM machine audit mode (proposal)**

In the longer term it is proposed that QCOM make available methods whereby a QCOM user can deliver customised interactive pages within a suitable location in machine audit mode via a set of QCOM API and call-back functions. Examples of applications in this regard are: an audit mode display for an external jackpot in which the machine participating; user script and service diagnostics; configuration and control of custom QCOM user delivered services.

It is envisaged that permitted QCOM users may be granted a “home” page within a designated area in audit mode on the machine for example located under a menu hierarchy such as:

Audit mode Main (Top) Menu->QCOM->Users-><user-list>->QCOM User Custom User Interface

When accessed, the machine’s display and input (preferably via an on-screen keyboard or UI) control is transferred to the given user via the supporting defined QCOM API and call-back functions. The machine must implement an exit command (which cannot be overridden by a QCOM user). Apart from that, the machine user interface within this specific area / page display of machine audit mode is now within custom control of the respective QCOM user.

Initially the interface will be text based but a simple ***http*** driven interface would also be possible down the track, where the QCOM user could push out http formatted data for the machine to display. This approach allows the qcom user to interact with the human interfacing with their audit mode page.

This idea is also extensible to the concept of giving QCOM users the ability to deliver a custom UI to players on gaming machines. Related: section 10.12.6.

**Last Play Recall – long term storage facility**

***This concept is proposed as mandatory for QCOM 3 gaming machines.*** cp:[[428]](#endnote-324) ***Feedback is sought.***

QCOM 3 gaming machines must have a facility in audit mode that allows an attendant to move an historical play, currently in the machine’s last play recall log, to long term NV memory in the machine. A minimum of **five** plays must be able to be held in the machine’s last play recall long term storage. Plays must be held in long term storage until manually overwritten by an attendant by the saving another play to long term storage. Attendants must be able to select which play in long term storage to overwrite.

The advantage of this feature is that the machine may be returned to operational play without the risk of a saved play of interest from being lost. *(At the time of writing, last play recall games will traditionally be lost over time if a machine is permitted to return to play. This feature eliminates this problem.)*

# QCOM WWW Interface

***This section is a concept / work in progress as possible future functionality – i.e. nothing in this section is currently mandatory.***

HTTP and related protocols are a convenient set of established protocols for the potential publication over a network, of a range of machine related data under this specification.

Note however that the QCOM 3 WWW interface is **not** intended to become a fully-fledged set of machine protocol/s providing application rich functionality as this is the purpose of the QCOM Lua Engine (s10). Also, the QCOM WWW interface is **not** intended to be an interface that is visually or functionally rich, as keeping the QCOM WWW interface simple helps keeps the cost and risk to a minimum.

*Design requirements:*

* *Support http and https interfaces*
* *No reliance or dependence (i.e. a mandate to use) on a Certificate Authority. Machine self-signed certificates will be used.*
* *Serve content which is natively able to be easily processed by a computer but with the option of making it easily human readable, e.g. JSON\*, or Lua declaration script (is under consideration but less likely), or XML based pages. (\*JSON is the most likely to be the mandated format)*
* *It’s time that machine audit mode was available via both the machine’s built-in UI and via the network.*

The QCOM 3 WWW interface must be capable of delivering all content via both http and https. The JSON file format is proposed to be the primary default format of delivered pages unless stated otherwise.

The machine’s web server may be running at all times provided the machine logic seal has been confirmed via the ”Logic Area Seal Confirmation Function” (s4.4). A newly RAM cleared machine will not have a certificate initially and therefore will deliver content only via http. Once the machine’s seal has been confirmed (s4.4) it must automatically create a certificate and automatically restart the web server and enable https content delivery. Related: section 8.3.

Any further requirements in this section mentioning the delivery of http content must also deliver the same content via https once the web server is setup with its self-signed certificate.

The https protocol once enabled must allow only TLSv1 or newer protocols that utilise strong encryption. For example, if apache v2 server was being used then the respective configuration settings would be:

**httpd.conf**

SSLProtocol all -SSLv2 –SSLv3

SSLCipherSuite ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES256-GCM-SHA384:DHE-RSA-AES128-GCM-SHA256:DHE-DSS-AES128-GCM-SHA256:kEDH+AESGCM:ECDHE-RSA-AES128-SHA256:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES128-SHA:ECDHE-ECDSA-AES128-SHA:ECDHE-RSA-AES256-SHA384:ECDHE-ECDSA-AES256-SHA384:ECDHE-RSA-AES256-SHA:ECDHE-ECDSA-AES256-SHA:DHE-RSA-AES128-SHA256:DHE-RSA-AES128-SHA:DHE-DSS-AES128-SHA256:DHE-RSA-AES256-SHA256:DHE-DSS-AES256-SHA:DHE-RSA-AES256-SHA:AES128-GCM-SHA256:AES256-GCM-SHA384:AES128-SHA256:AES256-SHA256:AES128-SHA:AES256-SHA:AES:CAMELLIA:DES-CBC3-SHA:!aNULL:!eNULL:!EXPORT:!DES:!RC4:!MD5:!PSK:!aECDH:!EDH-DSS-DES-CBC3-SHA:!EDH-RSA-DES-CBC3-SHA:!KRB5-DES-CBC3-SHA

SSLHonorCipherOrder on

Related: https://weakdh.org/sysadmin.html

The QCOM web server must provide the following functionality:

* A default page
* [Audit Mode Access. (Including last play recall. Access control would be required, refer 28.1)]
* Access the all EGM’s public keys
* Access to a list of all certificates

It is expected that the functionality delivered by the machine’s web server will increase substantially over time.

## Default page

The result from directing a browser to *http://<machine address>/* is at the discretion of the machine manufacturer. However, any content returned from browsing to content delivered via the above URL must be suitable for all jurisdictions and suitable for a potentially insecure network (with respect to privacy or security) e.g. information is that externally physically visible such as make, model, serial number etc. If the machine has no content to offer, then redirecting to the default qcom URL (see below) is also acceptable)

## Machine Audit Mode Access

Implementing access to machine audit mode via the machine’s www service is optional.

If implemented, machine audit mode access should be accessible via a hyperlink from the default www page.

There must be some security in relation to accessing audit mode commensurate to the security a physical audit mode key currently represents. For example a password should be adequate in most cases (assuming read only access is provided to a typical set of machine audit mode information). Todo: A function in the QCOM API will be provided that allows a venue to change the password.

The www UI pertaining to machine audit mode access is at the discretion of the machine manufacturer.

Related: Refer section 28.1.

## /qcom

All QCOM 3 www functionality is implemented within the URL:

http://<EGMs IP address>/qcom

All URL’s with the above prefix are reserved for use by QCOM (all other possible URLs are for use by the EGM manufacturer).

For the remainder of this section any term starting with ‘/’ infers a prefix of:

http[s]://<machine IP address>

For example ‘/qcom’ refers to the URL: http[s]://<machine IP address>/qcom

The content on the /qcom page is intended to be local public information (i.e. information a person could ascertain from being in visual range of the machine). It would represent all of the items on a machine that may be ascertained by its external appearance with no specific interaction taking place (other than game play e.g. if the device is a gaming machine) bar a few exceptions (e.g. public keys and certificates).

The default page at /qcom must be /qcom/qcom.json in the JSON file format according to the following schema:

*The JSON* *schema is tba – An informal / unformatted list of content is proposed here is as follows:*

Machine ID Number

Machine Logic UID number

QCOM Interface version

Current local Date and Time

Manufacturer ID

Machine Model ID / Name

UAA Public Key

[Gaming machine specific items

Credit Meter Value

Door status

Games

No of games

Game…

Game name

Denomination

Side plate details (tba)

]

Country

Location

Venue Name

## /qcom/users

This location must contain a subdirectory in the name of each registered QCOM user on the machine. No default page is required to be implemented or a “*permission denied”* or similar error may be returned.

## /qcom/users/<username>

Navigating to /users/<username> will (provided the QCOM user with username exists) show a directory containing:

* A copy of the user’s self-signed certificate (if the user has uploaded on to the machine) in PEM file format (.pem)
* QCOM Lua Engine statistics for the user (tba)
* A ‘www’ subdirectory

The purpose of the user’s directory is to allow QCOM users an easy way to publish data on the machine’s QCOM web server. When browsing a user’s www page, a QCOM user hooked script will be invoked which will deliver all of the content to be delivered by the machine’s web server.

## /qcom/certs

This location displays a list of public keys and certificates held by the machine. This also includes any certificates made from the machine’s certificate request. The certificates must be published in PEM format.

## /qcom/lua

The default returned page must display some overall global QCOM Lua Engine statistics such as memory/CPU utilisation/global counters (tba).

In addition:

/qcom/lua/luaerrors.txt must return a file containing the machine current QCOM Lua Engine error table. Refer section 10.2.11 for the file format.

## /qcom/audit

Browsing to this location accesses a wide range of QCOM related variables and parameters. A JSON formatted page is likely (tba). A login (username & password) may be required to access.

## Event Log Access

Event log access will be a part of audit mode access. See last section. Schema is tba

## Other

Other future possible pages under consideration for possible publishing via the machine’s QCOM www interface are:

* Publish [selected - tba] resident QCOM user scripts. *QCOM users are advised that scripts uploaded to an EGM (especially non-SAA controlled scripts) may be viewable by third parties in EGM audit mode for diagnostic purposes. Particularly scripts of QCOM users that get ‘quarantined’ for diagnostic purposes (refer 5.8).*
* Access to screenshots (refer section 34.3)
* Access to game rules / help / pay-scale – The manufacturer may degrade[[429]](#footnote-105) / watermark this content (while keeping it human readable) if there are concerns about copyright.
* Machine operating and service manuals.
* Service contact information.
* License information

# Machine Software Upgrades

This component refers to the patching or upgrading of resident machine software for the purpose of addressing issues.

Rationale

Current generation gaming machines typically utilise commercially available operating systems, as well as a wide range of tools, libraries and APIs. QCOM 3 also requires machines utilise a range of commercially available libraries, APIs and protocols. In modern day computing, security related vulnerabilities are inevitable and thus machines will be required to be patched on occasion in order to maintain the machine’s security and integrity. In a wide area network of machines (such that exists in the State of Queensland), it is very costly to perform these upgrades physically venue-by-venue / machine-by-machine; **accordingly the implementation of remotely upgradeable machines to the requirements in this section is mandatory for machine manufacturers**. cp:[[430]](#endnote-325)

Scope

The machine software upgrade feature in this section typically applies to machine upgrades relating to software bug and security issue patching only. In regards to gaming machines, while the feature described in this section can also be used to add/remove games from a machine, it is not intended for this purpose. This type of functionality will be provided as a separate QCOM API class at a later time (demand ensuing) in order to allow the roles of machine patching and game swapping to be delivered by two separate service providers if need be. There are also special requirements in relation to a game delivery service that need to be catered for.

Related:

* Software Upgrade Authorities (SUA) refer s6.2.
* Peripheral Firmware Upgrades refer s21.2.

Methodologies

Two machine software upgrade methodologies must be supported by the machine to the requirements in this document. Namely:

* Ethernet based – instigated via the QCOM API
* USB attached storage device based – via an accessible USB port on the machine (internal to the machine but external to the a logic area seal – i.e. no seal break required)

The above methodologies must be only possible ways of upgrading a machine’s software without also having to access the sealed logic area of the machine and using a human physically present at the machine via manual upgrade procedure that allows the human to manually hash and verify the upgrade before instigating in a trusted manner. cp:[[431]](#endnote-326)

## General

It is preferred a machine does not clear NV memory upon an upgrade if possible. However, if the machine must perform a RAM clear in order to install an upgrade, then the RAM clear must be equivalent to a RAM clear via the QCOM API function **qcom\_machineRAMclear()** which does not require a physical tamper seal break and also preserves some settings in the machine. cp:[[432]](#endnote-327) Refer to this function’s description for more information.

## Upgrade Packages

In this section, the term ‘upgrade package’ refers to an effectively doubled-up file archive i.e. a file archive embedded within another file archive. The embedded file archive is a manufacturer specific file archive, which once successfully delivered to a machine and authenticated, is used by the machine to upgrade all, or part of a machine’s currently resident software.

The embedded archive of an upgrade package must be digitally signed and encrypted content; the encasing (overall) archive contains this embedded archive and the detached digital signatures of the embedded archive from the machine manufacturer and all installed QCOM 3 SUA certificates in the machine (refer s6.2) stored as unencrypted detached digital signature files.

The upgrade packages allow the machine to verify against, and subsequently applying package content when directed to (refer **qcom\_machineUpgradeQueue()**). cp:[[433]](#endnote-328)

A benefit of the above is that the storage and transport of upgrade packages does not need to be secure and allows anyone with the applicable public keys to verify all the digital sigatures.

The machine must ignore expiry dates on SUA certificates. cp:[[434]](#endnote-329)

At no time must code or functionality be executed by the machine which has not been previously signed and verified by the above parties by the machine. cp:[[435]](#endnote-330)

Related:

* Authentication of upgrades - requirements: Refer section 6.2.
* OLGR publication: “Principles for remotely upgradeable gaming machines”

It is expected there may be information, verification (shell) scripts and possibly other metadata attached outside the signed conponent of an upgrade package (such as digital signatures). However none of this information must never be considered trusted or reliable by a recipient machine unless it’s otherwise able to be authenticated.

All upgrade packages must have embedded in the signed portion of the upgrade package, not-before and not-after dates. The machine must not install a package if it’s current date is not within the date range specified.

Upgrade package design summary and additional requirements:

* A doubled-up file archive i.e. a file archive embedded within another file archive.
* **Detached signature files** must be utilised. With a file format able to be readily made and verified using the **openssl** command line.
* The ability to see and verify the signatures are in the package without requiring a decryption key is required.
* When the machine is processing an upgrade package, the machine **must** not consider any data as trusted at any time, that does not successfully authenticate against manufacturer public key/cert and all installed SUA certs.
* QCOM 3 SUA’s **must** also be able to evaluate what they are going to sign; able to trust and verify that exactly what they evaluated ends up in the signed component of an upgrade package.
* SUA’s (or a higher authority in charge of the SUA’s) **must** be able to verify and trust the tools and equipment that they use to sign with.
* **Compression of the embedded archive is mandatory**. Compression **must** be performed before any encryption.
* **Encryption of the embedded archive is mandatory.** As a result of the above, it’s easiest here if applicable SUA’s are simply provided with the necessary encryption keys. NB SUA’s are intended to be either regulators, or regulator approved testing facilities[[436]](#footnote-106); however there might be an arrangement where a SUA is operating (aka signing) at the direction of a higher authority that is not going to have decryption capability.
* Because the overall download package **must be single file,** the order of operations for creating a upgrade package it is expected to be something along the lines of: ***raw binaries / upgrade data and files -> compressed file archive -> encrypt -> signed using detached signatures -> file archive*** (store only; no compression). But what is “signed” exactly is the question…
* The ability to create a **single deterministic hash result** (for approvals as well as signing of the approved upgrade packages) from the raw binaries / upgrade data and files is a **requirement**. As mentioned above, it’s just a question of where prior the act of signing that this is done exactly. If the upgrade raw data is file based, then without any special action, hashing after compression will typically result in a non-deterministic hash result. A deterministic hash result is required. Examples of how to create a deterministic hash of upgrade data:
  + Use manifest files: A hash a deterministicallycompiled single **manifest file** of binaries / upgrade data and files. Each manifest entry would contain the following information for example: path and filename, filesize, hash of the file denoted by the path and filename.
  + Create or use an existing deterministic **custom file archive format** with all non-determistic data either stripped out, or forced. Include only path, filename, file size and file data for example. (Creating a custom file archive format is programtically very simple. Its just a matter of sorting entries on path and file name first, then for example, pipe it into the archive file: null terminated path and filename, filesize, file data. See following page for an example bash script that creates a deterministic tarball)
* As a result of the above, raw compression is recommended.
* The final outer file archive file format **must be the zip file format** (created using ‘store only’ compression).

#!/bin/sh

# Generates a Deterministic Tarball with the following default options

print\_usage**()** **{**

**echo** " -d <directory name>"

**echo** " -c <ouput archive name : default: archive.tar>"

**}**

ARCHIVE\_NAME**=**"archive.tar"

DIRECTORY**=**"null"

**while** **getopts** ":d:c?" opt**;** **do**

**case** **$opt** **in**

"d"**)** DIRECTORY**=$OPTARG** **;;**

"c"**)** ARCHIVE\_NAME**=$OPTARG** **;;**

"?"**)** print\_usage**;** **exit** 1**;;**

"\*"**)** **echo** "Invalid command line parameter" **;** **exit** 1 **;;**

**esac**

**done**

**[** **!** **-d** **$DIRECTORY** **]** **&&** print\_usage **&&** **exit** 1

# [ -d $DIRECTORY ] && find $DIRECTORY -print0

**[** **-d** **$DIRECTORY** **]** **&&** **find** **$DIRECTORY** **-**print0 **|** LC\_ALL**=**C sort -z **|** GZIP**=-**9n tar **--**null -T **-** **\**

**--**no-recursion **\**

**--**owner**=**root **--**group**=**root **--**numeric-owner **\**

**--**mode**=**go**=**rX**,**u+rw**,**a-s **\**

**-**zcf **$ARCHIVE\_NAME**

# MODE

# user = read & write

# group & others = r & execute directories (to view directories.

# all = clear set-user-ID and set-group-ID bits for deterministic tar balls

## Version and Compatibility Control

The machine must not allow downgrading cp:[[437]](#endnote-331) (as this would be a security issue given QCOM permits unauthenticated transport of upgrade packages) i.e. the machine must version check upgrade packages before applying. If a fall-back or downgrade is required then this would be implemented as another upgrade (from the perspective of the upgrade package version control).

The machine must automatically recognise and refuse to apply a package that is not applicable or compatible to the machine’s current configuration in any way. cp:[[438]](#endnote-332) This requirement infers that machines or packages have built-in checks, data and rules regarding compatibility information and checking.

A QCOM software upgrade approval authority (SUA) or a party delivering packages must not have to independently track compatibility information regarding software they approve. The machine manufacturer must ensure that QCOM 3 machines are **smart** and that the machine in isolation is automatically able to ensure that when presented with a validly signed upgrade package; that any dependencies are satisfied and that unworkable combinations of software will not be installed by the machine. cp:[[439]](#endnote-333)

It is permissible for a single upgrade package to contain upgrades for a range of machines makes and models from the same manufacturer and the machines intelligently extrude and apply only relevant patches.

If a machine is presented with a package which has no applicable upgrades in it, then the machine must report this as an error to the instigating user, via the machines display (if the upgrade was via USB) and via the **MACHINE\_UPGRADE\_DLC** state event. cp:[[440]](#endnote-334)

Example expected scenarios a machine must handle in relation to this sub-section:

* The upgrade package is not intended or compatible for that make and model machine.
* The upgrade package is compatible, but the upgrade version software received cannot be applied to the current version in the machine i.e. the machine is missing some pre-requisite upgrades. *In this regard it is preferred if upgrades can be applied to any version software currently in the machine where possible.*
* The upgrade package has expired or is presented to the machine before the not-before date.

If at any time during an installation of an upgrade package, if the machine’s integrity is significantly at risk if the machine was interrupted during the process, the machine must prominently display a warning message along the lines of “do not power off the machine”. cp:[[441]](#endnote-335)

An interruption during an upgrade package download at any stage must never compromise the integrity of the machine. cp:[[442]](#endnote-336)

## Ethernet Based Upgrades Requirements

Ethernet based upgrades must only be able to be instigated by a QCOM user with privilege to the QCOM API function:

machineUpgradeGetVerify(table | nil)

Refer to the QCOM 3 summary spreadsheet’s description for this function for more information about function arguments and return values.

The minimum set of protocols to be supported by the machine is **http** and **ftp**.

A machine must only support one download / verification operation at time. cp:[[443]](#endnote-337)

**Operation**

This function when successfully invoked, the QLE Lua software driver will send the *machineUpgradeGetVerify* message to the machine which requires the machine to attempt (in the background) a download and verification of a software upgrade package. cp:[[444]](#endnote-338)

The machine must throttle the rate of the download as per *mumaxrate* or more at its discretion in order to prevent the download form adversely affect the machine’s ability to perform other tasks (for example deliver smooth game play on an EGM). cp:[[445]](#endnote-339)

If the download is successful, the machine must verify the downloaded package. The *verification* of the package requires the machine to: cp:[[446]](#endnote-340)

* Calculate and verify the downloaded package’s SHA256 hash matches the SHA256 *hexstring* argument (if provided);
* Verify the upgrade package is signed by each resident SUA in the machine (s6.2);
* Verify that the upgrade package does not contain additional signatures for which the machine does not have corresponding SUA;
* Verify the not-before / not after dates;
* Verify the upgrade package is also actually applicable to the machine and current software set.

Once the download and verification is complete, regardless of success or fail, the machine must log the **MACHINE\_UPGRADE\_DLC** state event to report the result to QCOM users. cp:[[447]](#endnote-341)

If the verification was successful, the machine must also log the (ref s13.6.1) **MACHINE\_UPGRADE\_READY** event. cp:[[448]](#endnote-342) Installation/activation of the upgrade must only be possible via the QCOM API function **qcom\_machineUpgradeQueue()**.cp:[[449]](#endnote-343) The machine must be restarted and possibly RAM cleared in order for the upgrade to take effect with respect to the requirements in section 30.1. cp:[[450]](#endnote-344)

If an irrecoverable error occurs during upgrade activation that cannot be reapaired by automatic rollback, this must result in a critical error on the machine that can only be cleared by a full factory reset of the machine. cp:[[451]](#endnote-345)

Download / verify attempts must be conducted in the background by the machine with a priority which allows the machine to continue normal operation. cp:[[452]](#endnote-346)

The machine may abort/cancel any download / verify operations if the machine is restarted during the process at its discretion. Alternatively, the machine may attempt to automatically resume the download from where it left off. The machine indicates support for this via the ***autoresume*** property re the QCOM API function **qcom.machineUpgradeGetp()**.

**Other**

The machine must sanity check the ongoing download package size during the download. If it exceeds a reasonable threshold based on the machine’s available storage, then the download must be also aborted with an error. cp:[[453]](#endnote-347)

## USB Attached Storage Based Upgrades Requirements

Upgrades via Ethernet will be more common than USB attached storage based upgrades, however upgrades via USB must be supported in the event that conditions make an upgrade via Ethernet the less viable option. For example, there could be an issue that prevents the use of Ethernet as an upgrade transport medium.

A QCOM 3 machine must automatically detect and mount suitably formatted USB based storage devices only for the purpose of machine software upgrades at a **suitable time of the machine manufacture’s choosing**. cp:[[454]](#endnote-348) For example **during power up**, or while **in service mode**. Any constraints here must be stated in machine audit mode.

* The EGM may implement USB upgrade processing & readiness on boot up, in which case the upgrade is all readied-up before the QLE is even started and the EGM will never throw any **MACHINE\_UPGRADE\_\*** state events for it, only the machine **EVENT** events shown below.
* The other option is the EGM facilitates USB upgrades to be activated after boot (i.e. the QLE has started) from with a function in service mode for example. In this case the EGM must log the **MACHINE\_UPGRADE\_\*** state events to indicate progress.

The following machine events must always be logged (via the **EVENT** state event) in either case:

* **MACHINE\_UPGRADE\_READY**
* **MACHINE\_UPGRADED**

The file format of the USB attached storage device must be FAT32. cp:[[455]](#endnote-349)

SUAs must also be applied to USB based upgrades. cp:[[456]](#endnote-350) The machine must not under any circumstances, be able execute non-SUA signed / authenticated code, or ultimately trust non-SUA signed data from a USB storage device. cp:[[457]](#endnote-351)

Process

Upon detecting compatible a USB attached storage device, the machine must automatically scan[[458]](#footnote-107) it for an applicable upgrade package. cp:[[459]](#endnote-352) If one is found the machine must verify the SUA signatures and if all ok, ready the package for installation **on next machine restart** with respect to the requirements in section 30.1. cp:[[460]](#endnote-353)

To signal to QCOM users that an upgrade is about to occur, the machine must log (in order) the: cp:[[461]](#endnote-354)

* **MACHINE\_UPGRADE\_DLC** state event
* **MACHINE\_UPGRADE\_READY** event
* **SHUTDOWN\_PENDING** state event

Any issues concerning the USB attached storage delivered upgrade package must also be reported via the **MACHINE\_UPGRADE\_DLC** state event. cp:[[462]](#endnote-355) No other events would be logged in this case.

## Commissioning Caveats

A machine at factory defaults will require some form of initial BIOS / bootloader / OS software in order to commence initial operation with. What is present initially depends on the design of the machine, but it is expected that some cases the machine will factory default with a bare bones BIOS or bootloader and others may factory default with an OS pre-installed or similar. QCOM 3 allows for either arrangement provided the following requirements are met.

Hardware Requirements

Refer section 3.1, hardware requirements; in regards to the requirement to socket PSDs for verification at commissioning; particularly the machines factory default present software and firmware. cp:[[463]](#endnote-356)

Software Requirements

Before the machine’s logic seal has been “confirmed” via the “Logic Area Seal Confirmation Function” (s4.4), **regulator permitting**, the machine may allow software upgrades to occur via USB or similar provided the upgrade packages are verified by the machine as being signed by the machine manufacturer, and it is also still possible after the upgrade to be able to verify all the machine resident software/firmware at commissioning by an third party using a cheap, readily available off-the-shelf reader device (which is able to be verified/trusted by the third party) against expected image/s. cp:[[464]](#endnote-357)

After the machine’s logic seal has been “confirmed” via the “Logic Area Seal Confirmation Function”, the machine must disallow upgrades until specifically enabled via the QCOM API function qcom\_machineUpgradeSetp() aka the machine upgrade enabled flag (MUF global type).cp:[[465]](#endnote-358)

Software Upgrade Authority (SUA) Caveats

Refer section 6.2, Software Upgrade Authorities. QCOM has support for external SUAs which allow additional software signatories to be added in the process of machine software upgrades. The QCOM 3 SUA methodology is designed such that this does not encumber the machine manufacturer during development and testing because external SUAs do not come into play until installed via the QCOM 3 API.

Security

* The QCOM API function **qcom\_machineUpgradeGetVerify()** isnt available to any QCOM user until the QMA makes it available. This ensures that any required SUAs may be first installed by the QMA.
* The QMA isnt available until the machine's logic seal is confirmed via the “Logic Area Seal Confirmation Function” (s4.4).
* The QMA once set cannot be changed unless the machine is locally factory reset / fully RAM cleared. Otherwise a temporary QMA hijack during commissioning could occur. Refer section s6.1.4 for more information.
* All upgrades must be signed by the machine manufactuer at minimum in all scenerios.

## Upgrade process short summary

### Via Ethernet

* A QCOM user initiates an upgrade via a successful call to the **qcom\_machineUpgradeGetVerify()** QCOM 3 API function.
* QLE Lua software driver sends the *machineUpgradeGetVerify* message machine to download the file from the URL which instigates the download, hash & SUA digitial signature & applicability verification attempt of the upgrade package.
* The machine reports the overall result of the above via **MACHINE\_UPGRADE\_DLC** state event.
* If the result was a success then:
  + the machine also logs the **MACHINE\_UPGRADE\_READY** event.
  + A QCOM user may now queue the upgrade to occur on next machine restart via the QCOM API function machineUpgradeQueue(). Noted that this may also (at the discretion of the machine manufacturer) automatically queue a RAM clear of the machine as per the QCOM API function qcom\_machineRAMclear().

### Via USB

Refer section30.5. But it depends on the

# Electronic Seal Support

This section relates to OLGR electronic seal minimum requirements or the equivalent. Refer OLGR Electronic Seal Minimum Requirements publication for historical information.

Support for electronic seals in a QCOM 3 machine is **optional** unless specifically mandated for a given range of machines by a regulatory authority.

Related:

* Logic door access section 20.2.1

# Machine Discovery Protocol

This section defines requirements for a method which aids in the automatic discovery of QCOM 3 machines on the network via a simple UDP based QCOM 3 Machine Discovery Protocol (MDP).

The machine discovery protocol service must not be started until once the machine’s logic seal has been confirmed via the ”Logic Area Seal Confirmation Function” (s4.4). Once the seal is confirmed, then the MDP service must automatically start up each time the machine boots. Related: s20.2.1.

As of QCOM 3 v3.0.2, the MDP service is available as a part of the QCOM 3 SDK as an ordinary QCOM user. Its username is “**MDP**”. This gives the machine developer the option to either use the MDP QCOM user, or implement the MDP protocol from scratch.

If the machine elects to use the MDP QCOM user, then the machine is required to have a copy of the MDP user script hardcoded internally and to auto install the MDP QCOM user at every RAM clear. Refer section 5.10 for built-in QCOM user requirements.

## Features & requirements

* UDP based.
* ASCII / UTF8 / JSON formatted broadcast messages.
* IPv4 & IPv6 broacasts (ignore any routing errors).
* Low frequency broadcast messages (see below).
* On demand broadcast messages (on door open.
* Support for aiding in pairing of QCOM 3 machines with other devices.

Protocol: UDP (receive data must be disabled aka shutdown() )

The source port for the discovery protocol must be **23999.**

The destination port for the discovery protocol broadcasts must be **47416.**

Frequency of broadcasts must be:

* Once at power up, then
* then once every **3 minutes**.
* One whenever the machine exits audit mode.
* Every main door open event.

## Broadcast Message Format

Broadcast information JSON schema:

*<> in the schema below denotes the return value of a qcom API function, or other value as described.*

{

"interface": "QCOM3",

"version": "<qcom\_idInterfaceVersion()>",

"device": "<qcom\_idDeviceType()>",

"logicuid": "<qcom\_idLogicUID()>",

"manid": "<qcom\_idMfr3()>",

"commissionuid": "<qcom\_idCommissionUID()>", *-- indicates RAM clears*

"ready": <qcom\_machineReady() and true or false> -- boolean

"qusers": {

"username1": { -- there must be a record for each ordinary user.

"quarantined": true *–entry only present if user is quarantined*

}

}

}

The actual broadcast information must be transmitted by the machine with no whitespace characters.

Note that data broadcast in the MDP is not authenticated and easily spoofed. Systems observing this data should take this possibility into account.

## Audit mode

There must be a display in machine audit mode under a QCOM related page which displays the fields and values being sent by the machine in the broadcast message defined in the previous subsection. cp:[[466]](#endnote-359) The items do not have to be displayed or grouped together but should all appear under QCOM 3 related audit mode pages.

Related: Section 28 - Machine Audit Mode.

## QCOM 3 API Support

*None at this time; but the following under consideration:*

QCOM API classname: mdp

Possible features:

* Alter broadcast frequency, port.
* Customise fields (e.g. *pairid*) and protocol parameters.
* On demand broadcast (limited max frequency).

# Implementation, Development, Testing and Simulation

## Machine Implementation

Points concerning the implementation of QCOM 3 in a machine.

* Prerequisite knowledge:
  + Client / server socket programming.
  + SSL/TLS Client / Server socket connection programming using x509 certificates.
  + The Lua C API (may be avoided if the code examples in the QCOM 3 SDK relating to the Lua C API are utilised exactly)
  + UART serial port programming.
  + Threads, mutexes and event based programming techniques.
  + Also see sections 3.2 & 3.4.
  + QCOM 3 document (this document) and reference (summary spreadsheet) manual.
* **It is not possible for a gaming machine manufacturer to implement QCOM 3 without the QCOM 3 SDK**. The QCOM 3 SDK contains mandatory-to-use platform independent QLE Lua software driver source code.
* Endnotes throughout this document of the format: cp:x are used to designate QCOM 3 requirements that machines must implement.
* The QCOM 3 Summary Spreadsheet represents the next most important list of content for implementation in a machine.
* The use of the QLE Lua software driver denoted modules in the QCOM 3 SDK by the gaming machine is **mandatory**. This guarantees uniformity in the QLE across all brands of machines as seen by QCOM users.
  + The QLE Lua software driver source code modules provided in the SDK are intended to be able to be used directly with no modification, regardless of the machine’s platform.
  + The QLE Lua software driver source code modules implement most of the QCOM Lua Engine (QLE) and QCOM 3 minimum requirements. They will save a significant amount of resources in the implementation of QCOM 3 in machines.
  + The QLE Lua software driver source code implements a QCOM Lua engine that encompasses:
    - Full QCOM user support including the special users such as the QMA and anon users (s5 & 6).
    - The QCOM 3 API privilege system (s7).
    - QCOM core requirements and resource monitoring and control (s10).
    - The QCOM 3 API (s11).
    - The QCOM 3 event buffer (s13). Implementations provided in both C and Lua source.
    - The dispatching of state events to QCOM users (s14).
    - QCOM User Account Access (UAA) message dispatching to the QLE (s23).
    - The QCOM Command Interpreter (s24).
  + The starting pointin using the QCOM 3 SDK in a machine is the file/module called “qle.lua”.
* Seed and feed

The integration with the QLE driver mentioned above is very simple. The machine simply seed’s the EGM state in the QLE on EGM boot up and then keeps the QLE driver’s copy of the host machine’s state up to date by sending the QLE state event messages as required over time. Refer section 14.

* Areas of QCOM 3 that require implementation by the machine manufacturer are:
  + There are approximately **40** C-Lua functions that the host machine must provide to support the QLE. Refer to the QCOM 3 Summary spreadsheet: CLua worksheet, and the QCOM 3 SDK for the full functional description of each as well as example source code. The QCOM 3 SDK module: qlua.cpp contains example C source code.
  + The QCOM event buffer (s13.1). The QCOM 3 SDK contains a C (and Lua FYI) example implementation of the QCOM Event buffer that the machine developer can base their implementation on.
  + The state event FIFO. An important thing to note about state events sent to this buffer is the state event buffer’s primary key should not be the state event is, but what QLE Lua software driver receiver function is. Refer to the QCOM 3 SDK module: qcomluaengthread.cpp for more information.
  + The QLE Lua state message FIFO. This FIFO hold messages send out from the QLE Lua software driver. QCOM 3 Summary spreadsheet: Lua-API sheet, RHS columns.
  + Log QCOM 3 state events as defined to the state event FIFO. Refer to the QCOM 3 Summary spreadsheet: State-Events sheet.
  + Audit mode functionality (search on “audit” or “audit mode”).
  + The underlying TCP/UDP socket support for the QCOM 3 communications API (s12).
  + The System Lockup UI (s16).
  + PEF disable condition display in idle mode (s11.26.1).
  + The PAEL UI (s19).
  + QCOM User Account Access service (UAA service) (s23). (NB. The QCOM 3 QLE Lua software driver handles all connected message processing.)
  + Content Auditing (s27).
  + Machine Software Upgrade support (s30).
  + Machine Discovery Protocol (s32).

### Getting started

Recommendations:

* Understand the QCOM 3 block diagrams. Especially the threads and interfaces diagram. Refer to QSIM 3 docs directly / QCOM 3 Summary spreadsheet.
* The first few lines of code a machine developer may wish to consider writing may be found in the QCOM 3 SDK : file: cpp\gs1ex.cpp. This code coupled with the QLE Lua software driver source, gets a Lua interpreter up and running inside the machine, initialises the QLE Lua software driver and captures all print()’s that occur inside the Lua instance. As well as being a requirement, the capture of these print()’s is useful for diagnostics during development in addition to messages sent out by the QLE Lua software driver (ref: QCOM 3 Summary spreadsheet: Lua-API sheet, RHS column).
* Play with QSIM and the example QCOM users provided.

### Summary of QCOM 3 NV data

* QCOM user script storage 12,000 kB (s5.7). QCOM user scripts must be stored in their zip file archives as downloaded.
* QCOM 3 Persistent Variables PV 50kB (s10.10)
* Certificates: ~ 2kB per certificate and 2kB per private key if stored in PEM file format and uncompressed
  + QMA
  + SAA
  + SUAx4 max
  + local machine CA
  + UAA cert
  + [WWW cert]
* QCOM 3 Event buffer (s13):
  + 50-100kB if stored uncompressed as a Lua or json declaration.
* QCOM user settings:
  + If stored uncompressed as a Lua or json declaration it will be ~ 7kB per QCOM user. This estimate excludes the user's PV data which is dealt with separately above.
* Other QCOM 3 related host machine state (hms) data:
  + ~13kB if stored uncompressed as a Lua or json decl.
  + Note however, a good percentage of this data is pre-existing EGM NV data such meters and game data.

QCOM 3 is designed to be able to work from slower flash based memory and a machine may store all QLE specific NV data in flash based memory if desired. However if the machine has a spare 50kB fast-write NV memory store, then a performance increase will result if QCOM 3 Persistent Variables NV data can be stored in here. Even a small 500-1000 byte fast NV write cache for the flash memory will also help.

Excluding the PV data, there will be negligible performance increase or issues in storing any of the other of the above data in fast NV memory.

Currently, the main trade-off when using a slow-write NV data store for QCOM 3 in a machine is that the machine may need larger sized QCOM 3 related FIFOs in compensation.

At this time the following data is not required to be stored in a NV memory device:

* QCOM 3 state event FIFO buffer
* QLE generated (sendToHost) message FIFO buffer

Related: section 3.1 - Machine hardware requirements

## Machine Implementation Testing

As QCOM 3’s primary interface is an API, this allows QCOM 3 implementation testing on a machine to predominantly consist of a range of formal QCOM Lua Engine test scripts. These scripts will thoroughly test the QCOM Lua interpreter and QCOM API as implemented by the machine. This will largely automate QCOM 3 testing in machines.

In addition, because of the fact that most underlying network protocols in use by QCOM 3, are standard and well supported existing protocols, testers can utilise the wide range of existing, powerful extensive and freely available testing tools available.

QCOM 3 machine Lua test scripts will be made available to QCOM 3 licensees.

## Development tools for System Developers

A QCOM 3 machine simulator (QSIM 3) program is available to assist QCOM licensed system developers with the creation and implementation of QCOM 3 products and services.

The QCOM 3 machine simulator emulates a single QCOM machine. However, the simulator will have an interface designed to facilitate remote control over a network in order to allow an array of simulators to be readily controlled.

The simulator will be developed for Windows initially and then Linux (resources permitting). Meanwhile QSIM3 will run in Linux under Wine[[467]](#footnote-108).

## Software Development Kit (SDK)

The QCOM 3 SDK is **primarily** **intended for machine manufacturers** for the purpose of implementing QCOM 3 in their machines. (In contrast the QCOM 3 simulator is primarily intended for the purpose of implementing QCOM 3 in machine related systems.)

The QCOM 3 SDK is comprised of:

* Lua source code containing implementations of:
* QCOM Lua Engine.
* The QCOM API Command Interface processor.
* QCOM event buffer implementation.
* Lua test script code for the entire QCOM API (These will be the same test scripts used by OLGR to test QCOM 3 implementations in machines)
* Examples in “C” demonstrating how to interface the QCOM Lua Engine to a host machine for every applicable QCOM API function.
* Source code examples utilising the *openssl* API pertaining to the required secure sockets implementation and all crypto related functions required by the QCOM 3 specification.

The QCOM 3 SDK Lua code relating to the implementation of the QCOM Lua Engine (QLE) represents a platform independent QLE driver, it also ensures an identical Lua environment is seen by all QCOM users across all brands of machines, accordingly **it is mandatory for machines to utilise the QLE specific Lua source modules in the QCOM 3 SDK.** cp:[[468]](#endnote-360)(*Which specific QCOM 3 SDK Lua source code modules are mandatory are clearly identified in the SDK in the respective module’s header*)

The QCOM 3 SDK Lua source code modules that are specific to the implementation of the QLE must also be considered a set of supplementary requirements for the implementation of QCOM 3 in a machine.

If a machine manufacturer/developer needs to alter or exclude any of the mandatory QCOM 3 Lua source code, then please contact OLGR.

The QCOM 3 SDK is only available under license. Contact OLGR for more information.

# Possible Future Functionality

This section provides some information regarding possible future directions for networkable/QCOM related technologies.

None of the features proposed or outlined in this section are mandatory or should be implemented unless it is explicitly stated.

Feasibility and suitability studies for the features in this section have not been undertaken and no feedback is being requested on this section at this time unless feedback is explicitly requested.

*Design note: This section requires frequent updating as the information within can depreciate fairly quickly.*

## Wi-Fi Interface

In addition to a QCOM machine’s Ethernet port, it may be desirable for machines to have the option of a wireless network interface controller to the IEEE 802.11 or newer standard.

Rationale: In some circumstances Wi-Fi may also be considered as the primary interface for QCOM especially in cases were a hard-wired network is not present, or expensive to install.

## Built-in Player ID Card Readers

*This section is published to promote possible discussion concerning the concept of a generic machine manufacturer installed player ID card reader. Feedback is welcome.*

Card readers in gaming machines are common. Adding ID Card Readers to existing gaming machines post production / installation can be a costly exercise. A less expensive option may be to propose a standard for the inclusion of a machine manufacturer supplied card reader in all machines by default. The decision regarding which card technology to utilise would require negotiation but some options to consider would include magnetic-stripe, smart card, or proximity device. The resulting associated functionality gained via ID cards would then not only be broadly available at less expense but it would integrate more effectively in the machine being manufacturer installed.

It is proposed that the QCOM would allow third parties access to the card reader and data thereof via the QCOM API.

QCOM’s support for ID Card Readers would include allowing for multiple services and service providers to utilise the same physical card reader in the machine. There may need to be an agreed standard pertaining to the data stored / read from the card and possibly a vendor ID encoded to allow the machine to pass insertion/removal events to the correct associated QCOM user / service provider.

The card reader would operate as follows:

A QCOM API function would register the association between a vendor ID stored on the card with a specific QCOM user. This function would remain privileged only to the QMA or delegated authority to ensure that service providers only retrieve ID numbers pertaining to their vendor ID. The function template might be like:

qcom\_idcRegister(username, vendorID)

*(The classname prefix “idc” stands for ID Card)*

Upon any card insertion, the machine would throw a new state event such as

IDC\_CARD\_INSERTED

The only data attached to the above event would be the vendor ID read from the card.

Any QCOM user wishing to trap card insertions would hook onto this event and look for their vendor ID on each event callback. Upon a hit, the QCOM user would then use another new QCOM API function to retrieve the remaining card information if any. The function template might be like:

qcom\_idcGetInfo()

The return value from calling this function will contain the full card information as read off the card. The machine will ensure that it only returns card information to the registered QCOM user of the vendor ID of the inserted card.

The end result is that while all QCOM users can “see” card insertion / removals and the primary vendor ID, only the registered QCOM user for a given vendor ID can obtain any further information encoded onto the card.

*Higher levels of privacy can be attained by utilising another QCOM API function that queries “Is it mine” with respect to the vendor ID with a Boolean return value instead of including vendor ID in the QCOM state event, where any QCOM user can see it. This would effectively hide vendor IDs from all but the QCOM user registered with a particular vendor ID. Feedback is welcome.*

There will be a generic IDC\_CARD\_REMOVED state event permitting QCOM users to hook and take the desired action in response to a card removal.

If capture and hold card readers become the standard, then there will also be a QCOM API function of the form

qcom\_idcEjectCard()

*The machine must ensure that only the QCOM user associated with the vendor ID of the card in escrow may eject the card. The machine will also need a policy concerning a held card on restart or power up, or if the corresponding QCOM user gets deleted or ‘quarantined’ (refer 5.8).*

Card Data

Potential on card data highly negotiable and up for discussion; possibly a QCOM controlled vendorID, ID number and possibly another algorithm represented as data in order to prevent collisions with other card systems. It is envisaged that the cards will not store any personal information. An extensible card data format may be desirable.

## Screen Capture

The ability for a machine to be able to capture screenshots of its built-in displays on demand is potentially a useful tool in machine gaming. It could for example:

* Assist in authentication of large wins
* Be useful in support of last play recall functionality
* Assist in machine diagnostics and malfunctions
* Assist in providing evidence in player disputes

As many people now carry a camera with them at all times, this idea may have less practical value, however the feature being built into the machine means that the screen captures can be delivered with authentication.

Screen shots would be able to forward on demand to designated hosts by permitted QCOM users.

The primary Lua API function of interest is qcom\_machineTakeScreenshot(). Refer to section 11.7.23 for a full description.

Saved screenshots could also be made available to download from the machine’s www interface.

Attendant Instigated Screenshots

Any machine attendant with authorised access to machine audit mode could also be authorised to take screenshots of all displays on demand.

One method here for example, is that the attendant accesses audit mode, requests a screen shot and the machine responds and informs the attendant that a screen shot will be taken next time the attendant instigates a specific unique action (e.g. press a button combination or turn a key-switch). The attendant then navigates to the desired machine display and instigates the specific unique action to save a screenshot. In this case the screenshot could be saved locally on the machine and retrieved via the machine’s audit mode or www interface.

It would also be acceptable for the machine to have a persistent specific screen shot action provided only an authorised machine attendant can instigate the action (e.g. requires a key).

## Video Capture

In 2015, the ability for video game consoles[[469]](#footnote-109) to stream gameplay to a file or over a network in background has become a standard feature in the latest generation game consoles. This feature also exists in all PC video cards as well[[470]](#footnote-110).   
This feature (in order to be CPU efficient as possible) is typically built into the video cards driver and performed by the GPU.

The benefits of this technology in EGMs is obvious. A primary use envisaged would be for use with EGM last play recall. A driver level screen recorder would capture/save exactly what the player sees (last x minute’s worth). This inherently captures things not previous recorded by last play recall, such machine software issues/bugs in action. It would save on the coding effort currently put into implementing graphic visualisations of game replays. The recorder could also be setup so it sleeps while the machine is “idle” / zero credit, in order to keep recordings with activity for longer before overwriting.

Current and saved recording could be downloaded from the EGM’s www interface.

In relation to QCOM in support of this feature several new QCOM API commands would be created. Such as

qcom\_vcSaveToDisk(secs)

The above function would allow an authorised QCOM user to permanently save the last x secs of the current recording to EGM persistent storage for later download via the EGM’s www interface.

## Streaming Video onto Machine Displays

In this concept, the streaming video comes from a host external to the machine.

This concept has an application related to functionality in QCOM v1 known as the External Jackpot Information Poll (EXTJIP) which is an arbitrary jackpot display on the machine received as streaming video. However the concept is also generic enough for any number of arbitrary applications which could display a video media stream to a machine’s display. See list at the bottom of the section.

Currently QCOM 3 is facilitating what was the QCOM v1 EXTJIP functionality via the PAEL feature (refer section 19) and via the QCOM API function qcom\_egmSMS(), however, in the longer term, it is proposed to change this approach to one which uses a small opaque window overlaid on the machine’s primary display (somewhere suitable and/or movable on demand by the user) representing a streaming video/animation feed from an external source. The stream could for example represent any the relevant external jackpot current amounts related to that machine however myriad of possible applications are envisaged.

The stream would only be a very small area of the total display area on the machine. For example, something approximately 0.5% of the machine’s total display area (e.g. 200x50 pixels on a 1920x1080 display). This is in order minimise impact with respect to machine’s primary application. The window size is intended to be configurable and possibly even be user scalable and moveable.

The potential applications of this concept are not necessarily limited to jackpot displays, but in this scenario, it would allow the machine to host a miniature animated jackpot display on the machine via QCOM. This means an external jackpot system could show animated odometer type jackpot displays with custom graphics and win shows and other arbitrary graphically enhanced or animated messages. As per QCOM v1, it is not the intent to totally replace large jackpot displays.

There is a possibility that the prominence of the stream could be somewhat in the control of a QCOM user as well as what stream to display at any given time.

It is estimated the bandwidth required would be very small (e.g. only a few killobytes / sec). The video streaming protocol would be UDP broadcast based (connectionless) or some other multi-point streaming based protocol.

*Example protocols potentially of use for this application include:*

* *Real-time Transport Protocol (RTP)*
* *Secure Real-time Transport Protocol (SRTP)*

The actual display stream would be under the control of the QCOM API. It is envisaged that the machine could cycle between a number of video streams, also controllable via the QCOM API or human user at the machine.

It is proposed the video stream would only have to be visible while the machine is in idle mode (as per EXTJIP display as per QCOM v1).

If the machine utilises a touch screen, then it may be possible for the user to drag the window around the screen to any desired location.

Advantages of the concept:

* Compared with the existing EXTJIP / qcom\_egmSMS() functionality (which is text based), this feature facilitates customised colour animated video streams.
* The potential to eliminate (as a cost saving) additional hardware in gaming venues.
* The machine also has control of where and when it displays the video stream/s. This allows the machine to ensure the display does not overwrite important areas of the machine’s application display.

Other possible applications for the built in streaming video display (local regulations permitting)[[471]](#footnote-111):

* Related gaming results displays such as Keno
* Any gaming or sport related results
* Promotions / raffles and draws
* Racing
* Music and entertainment
* News
* Advertisements
* Clock / timer display
* Generic information display

# Appendix A – QCOM 3 Modes of Operation

This section is applicable reading for the authority in the role of the QMA.

QCOM 3 modes of operation require a good overall technical understanding of QCOM 3 to implement.

This section lists a range of possible QCOM 3 modes of operation. The list is by no means exhaustive. A QCOM mode of operation is a loose term which refers to how a QCOM machine could potentially be configured in different operating environments with respect to, for example, varying network bandwidths, network availability and the number and types of services and service providers. The intent is this section is to promote a wide range of ideas in regards to how QCOM 3 machine can be interfaced.

Some of the modes below have been mentioned previously - refer to section 1.7. All modes can be mixed with each other to any degree as well as switched between without ever having to factory default the machine.

1. Standalone (no ongoing / persistent network)

QCOM machines are setup at commissioning with any desired functionality delivered via autonomous QCOM user scripts. Refer 6.1.3 for more information on autonomous QCOM users.

1. Limited connectivity networks

QCOM 3 allows QCOM users to control how network demanding a QCOM 3 machine will be and this allows QCOM machines to function on networks that vary widely with respect to availability and bandwidth. Whether this is a hardware limitation of the physical LAN hardware or a bandwidth cost limitation (e.g. WANs) is irrelevant.

One thing emanating from the fact that QCOM 3 machines will work well in limited bandwidth networks, is that it is now theoretically possible for many possible types of QCOM machine related services operated over a WAN for example, to not require any local host machines. In Queensland gaming venues for example, it could in the longer term potentially even mean no “Site Controllers” would be required and in other cases, it can eliminate the “black box” type hosts and controllers per machine solutions which are notoriously costly. This is made possible in QCOM 3 when a service provider is permitted to be a QCOM user. Each QCOM 3 machine can essentially be a service host / resource via the QCOM 3 scripting engine.

1. Autonomous monitoring and control

One mode of operation of QCOM machines envisaged will be one in which control functionality is fully downloaded to the machine, executed autonomously and only updated on demand when changes to the control functionality or parameters are required (i.e. there is little network connectivity required for ongoing machine control). Monitoring is largely performed locally (via QCOM scripts) and the only ongoing network connectivity relates to forwarding any collated data directly onto a host database utilising existing standard database connectively protocols.

1. Remote Procedure Call Mode

It is easy to implement Remote Procedure Call functionality on a QCOM 3 machine via the QCOM Command Interpreter (24) or the QCOM API.

This mode of operation would be of interest to parties wanting quick-to-market network functionality and who do not wish to get into QCOM script development, or to minimise any implementation of network protocols.

1. Multi-protocol environments

QCOM 3 machines are multi-protocol capable as well as being able to patch or change network protocols remotely on demand. A QCOM user can utilise as many protocols as desired as long as they don’t exceed their QCOM user memory / CPU limits in the machine.

QCOM 3 machines can be configured to talk whatever protocol is necessary given the module scripts are available. (The intent is a market for QCOM 3 modules will evolve here). For example in order to permit a QCOM 3 machine to operate with a legacy monitoring system, the QCOM machine would be uploaded with the necessary module / script that speaks the protocol the monitoring system is using. If the monitoring system is changed then it is just a matter of changing the protocol the QCOM 3 currently talk, which can of course be done remotely.

It is envisaged that with QCOM 3, monitoring system developers would often design their own protocol modules for QCOM 3 machines to suit their own business needs and intended operating environments.

1. Multiple service providers

QCOM 3 was specifically designed to operate in environments where there are multiple service providers. Service providers can work in co-operation with each other if desired, but there is also no issue if they are in competition with each other. Independence and integrity with respect to potentially commercially hostile service providers is managed by the QMA (6.1).

At the other end of the scale, if the need arises, the QMA can also create forced dependencies if need be.

This is all dynamically programmable behaviour in a QCOM machine.

*A single service provider mode of operation is inherently supported.*

1. Zero trust environments.

In an environment where the regulator has a low or unknown level confidence in the service provider’s integrity, options exist.

For example, the QMA can issue a very limited, minimal set of low risk privileges to the service providers made QCOM users. Alternatively, QMA can configure the machine with any desired set of protocols or functionality to facilitate services and retain full control over them. In this second scenario, it means service providers are locked outside the machine entirely.

Both the above options manage risk equally well and other operational factors will decide which is the most suitable or cost effective approach to use.

1. Transitional Modes of Operation

This isn’t a mode of operation but a reiteration of a benefit mentioned earlier in this document regarding the way in which a QCOM 3 machine can dynamically change between one mode of operation (or protocol) to another without having to upgrade machine manufacturer software/firmware, or having to physically visit the machine.

1. Database Centric Modes of Operation

This is expected to be a popular approach to service delivery on QCOM 3 machine, especially those services which only rarely need to receive data from a host system (e.g. EGM monitoring and control services). This is a mode where potentially no custom protocols are required to be implemented in order to interface a QCOM 3 machine to a host system over a network to deliver a service. Ideally in the long term, the QCOM Lua Engine/ QCOM 3 API will provide a number of built-in network database protocol APIs.

The QCOM machine simply uses the database network protocol API (as provided by the QCOM Lua Engine) to connect and exchange information with a host system database. Any required control functionality is fully automated, implemented by user scripts within the QCOM Lua Engine.

It is predicted there will become an open market for QCOM 3 “applications”, where third parties develop license or share, all types of application modules intended for use with QCOM 3 machines.

# Appendix B - Designing a QCOM 3 Operating Environment

This section is applicable reading for the authority in the role of the QMA. This section also requires an overall technical understanding of QCOM 3.

Since QCOM 3 machines are highly versatile, in each jurisdiction the regulator (or the equivalent) will need to decide how exactly QCOM 3 machines will meet all their regulatory needs. This section discusses the process of determining a suitable QCOM operating environment for a given market or jurisdiction. This task only has to be undertaken once per jurisdiction and the process is not onerous. OLGR can provide all regulators that adopt QCOM with advice and templates to aid with designing their specific QCOM 3 operating environment.

The outcome of this process determines:

* The types of services that will be permitted / supported and how they will interact.
* Who will be the QMA and what (if any) autonomous users / functionality it will provide
* Which services will be implemented via QCOM users (refer section 5)
* Whether any network protocol interfaces will to be adopted (typically useful for low demand shared services)
* The set of QCOM API privileges for each QCOM user (service provider)
* A dependency review and check.

An example step by step approach would be to:

1. Summarise the regulatory and operating environment
   1. List applicable regulatory policies
   2. List all services intended to be supported. *Refer to term ‘service provider’ the glossary (section 2) for a list of potential services.*
   3. Classify each service as to which will adopt/share a network interface and which will be implemented via QCOM user account on the machines to QCOM machines.
   4. List all categories of service providers relating to the services if any
2. Based on the above information decide:
   1. on the role of the QMA and who it will be
   2. whether there be any autonomous users / functionality
   3. which services are to be **QCOM users** (section 5)
   4. if any standard protocols are to be implemented
3. For QCOM users, determine QCOM API privileges (there will eventually be a recommended set of privileges for each type of QCOM service published in this document)
   1. Perform a contention check across all privileges granted to QCOM users. (*Simply maintain a privilege-by-QCOM user spreadsheet, which can be based on the QCOM summary spreadsheet*)
4. Cross check the final operating environment design against previously listed regulatory polices
5. Seek industry feedback.
6. Future proof check

Notes:

* Existing QCOM 3 compatible products and designed operating environments should be considered first before going to the trouble of inventing what may be an existing QCOM operating environment.
* In designing a QCOM environment, avoid creating dependencies unless specifically intended, especially when parties are competing commercially.
* For services not permitted to be QCOM users, this means that a network protocol interface is required and thus it needs to be decided who will be the QCOM user relating to the implementation of that interface in the machine and what specific network protocol/s will facilitate the service.
* OLGR recommends any QCOM user based services which trigger prizes or jackpots be implemented as a QCOM user which implements no other functionality or service.
* Try to minimise costs and risk. Machine to system network protocol interfaces risk levels ,may include for example:
  + Low cost/risk:

Service provider A implementing network protocol X on a machine interfaces to service provider A on protocol X

* + High cost/risk:

Service provider A implementing network protocol X interfaces to service provider B on protocol X

Potentially high risk/cost scenarios would be typical in legacy system integration (i.e. a new QCOM 3 machine talking and old protocol in order to interface to an old monitoring system) or when multiple service providers are sharing a single network protocol interface (e.g. a jackpot display system or performance analysis management network protocol interface). If this arrangement occurs, then there should be a commercially neutral party (with respect to the delivery of gaming machine related services) acting as arbitrator / manager concerning protocol X implementation, support and gap monitoring and analysis between service providers. This is far less an issue when protocol X is already well established and supported, but is a far higher risk when protocol X is new.

The creation of a shared custom network interface is typically a result of multiple service providers having similar network functionality demands. This decision results in greater network efficiency because there is less duplication of information sent over the network.

Queensland Clubs and Hotels – proposed QCOM 3 operating environment

Regulatory Environment Summary:

* Multiple Licensed Monitoring Operators (LMO)
* Un-licensed service providers: permitted for some services, see unregulated services below
* Regulated services: Monitoring and control, Card based Gaming, TITO, jackpots (i.e. anything financial)
* Unregulated services: Player loyalty systems, Attended Paging Systems, Venue management and performance monitoring systems

Proposed operating environment:

* QMA: OLGR
* QMA retains critical (QMA) functions: yes
* SAA: OLGR
* SUA: OLGR
* QCOM users: LMOs. LMOs may request additional QCOM users if desired (e.g. segregation of services).
* Forced Dependencies (between service providers): none.

This proposal does not include jackpots.

Queensland Casinos – proposed QCOM 3 operating environment

Regulatory Environment Summary:

* Casinos are licensed entities
* Casinos own/operate the machines as well as monitoring
* Regulated services: Monitoring and control, Card based Gaming, TITO, jackpots (i.e. anything financial)
* Unregulated services: Attended Paging Systems, Venue management and performance monitoring systems

Proposed operating environment:

* QMA: OLGR
* QMA retains critical (QMA) functions: yes
* SAA: OLGR
* SUA: OLGR
* QCOM users: Casino licensees or their nominated service providers may request additional QCOM users.

This proposal does not include jackpots.

# Appendix C – ANZ Gaming Machine National Standards (GMNS) and QCOM

If a conflict is identified between GMNS and QCOM, then where it is possible to satisfy both requirements, then this must be done. Where conflicting requirements cannot both be satisfied, then generally QCOM takes priority unless specified otherwise in this document. Please notify OLGR of any conflicts that cannot be resolved by satisfying both requirements’ documents, or are not listed below. (Related: qcom v1.6.x appendix B.)

Known conflicts with GMNS: (GMNS version: conflict)

* tba

# Revision History

Date printed: 10 June 2022

Change marks / comments will appear as: Robert Larkin - RL

This document is published on the internet. Refer:

<https://www.business.qld.gov.au/industries/hospitality-tourism-sport/liquor-gaming>

*or:*

<https://publications.qld.gov.au>

*or direct link:*

<https://publications.qld.gov.au/dataset/qcom-3-interface-specification>

The revision history below covers changes to this document. To view a revision history of QCOM 3 overall; also read the QCOM 3 Summary spreadsheet’s revision history.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Version** | **Changes** | **Who** | **Release Date** | **Incept Date** |
| **3.0.3** | **See Below** | **RLL** | **10-Jun-2022** | **Contact OLGR** |
| * Added a full functional description for the qcom\_luaPublish() QCOM API function alieviating the need to refer to the associated SDK module for this information. * QCOM API function **qcom\_gameSetVar()**. Added information with respect to games with a different progr variation per game variation. * Section 14.4 on Sync events reviewed and made minor clarifications. * Section 25.2: Added more general information about QCOM 3 progressive support. * Section 23. Added more explanation concerning UAA related certificates. * Section 11.36 Added new requirement for a ‘send()’ function available in LPB UART scripts to allow them to send sideband messages to its UART object call-back function in the QLE. This enables reporting of low-level CRC errors, packet framing errors and UART LSR register events such as UART overrun errors.   Also refer to the revision history in the QCOM 3 summary spreadsheet. | | | | |
| **3.0.3 draft** | **See Below** | **RLL** | **10-Nov-2021** | **Contact OLGR** |
| * PAEL (section 19) status has been changed from mandatory to possible future functionality - do not implement PAEL until further notice. It might be better merged with custom feature in section 3.7. * IPV6 should also be able to be manually configured. This requirement was removed in error. Re-added this requirement. Refer section 4.3 * Added new sub-section 13.6.1 to help clarify event logging process. * Refer section 3.5.1 para. 10. Fixed an inadvertently deleted “no” word that reversed the polarity of the requirement. * Deleted requirement in section 4.3 re the enabling network interfaces only after logic seal confirmation. It was meant to be about when to enable qcom 3 related services like UAA, MDP and the optional WWW. The subject of when each of these services may be started is covered in each service’s section. * Refer section 10.2.2. Clarified handling of a QLE LSD runtime error. * QCOM API function **qcom.egmState()** section 11.19.20: removed substates: pid & rules. * Refer section 20.2.1 – Logic door access. Returned to QCOM v1 like behaviour. Rationale: No point in destroying secrets if this doesn't also happen when the logic door is opened while the machine is powered off. Most machines don’t require this level of security. +Allows the MDP service to continue during logic door open. * Refer section 25. Full review. * Added a bignum external Lua library to help with ticket authno numbers. Refer s10.3.2 * Added s3.3.2 ‘Machine RAM errors’ * Time related clarifications and changes   + s13.5 Added ‘bias’ field to event schema.   + s28 added requirement for timestamp to be displayed in a human readable format in local time.   + s11.5.4 Changes to timezone must take effect immediately (instead of machine restart) * Added new section 25.6 covering special progressive game arrangements. * Time management review.   + “Isdst” is gone. Daylight saving support is done via time zone bias.   + s9 deleted requirements for clock drift monitoring as modern operating systems take care of this and clocks in embedded systems are a lot better than they used to be.   + 10.3.1 Deleted the former requirement for the machine to overload/populate the os.date() function’s *isdst* return value.   + qcom\_timeSetTimezone (s11.5.4) changed polarity of time zone bias here so it’s the same everywhere. * Deleted section 14.2. ‘State Event SPAM / Runaway’. State event spam prevention is primarily hanlded by the QLE LSD by placing cooldowns on QCOM API functions that generate either sth messages or state events. * General   + Deleted some duplication of requirements across this document and the QCOM 3 summary spreadsheet. * (2020-Feb) section 30.2 reviewed this section again; made clarifications; mandated openssl compatable digital signatures. * “i” button functionality must no longer be available during a system lockup. * Added new section 4.8. * UART s11.36 API call formats and RV have changed. * Added new section 25.6 re progressives. * Clarified current QLE LSD QCOM API behaviour when an ‘empty string’ is provided for certain agruments. * Refer section 27.1 clarified the definition of the ‘platform’ record. Removed any restriction on how often it may be changed. * s10.7.1 removed requirement to display QLE print()’s during start-up in favour of a generic audit mode display of the last 20 QLE prints ongoing. Refer s28. * Changes related to new QCOM API function qcom\_progrModeChange() * “i” button s3.7. Relaxed requirements shape and colour. * Documented the (new) ability for qcom user script modules to return a function library (aka table of functions) and have it saved. This is in compensation for not having the ‘require’ Lua standard function available to qcom users. Refer s10.7.2. Also, clarified the term ‘scripts’ in this section for clarify. * Added appendix C (copied from qcom v1 appendix B) * Updated section 30.4 wrt the QCOM API function qcom.machineUpgradeGetVerify() updated behaviour stated in the xlsx and SDK. Avoid duplication requirements wrt the qcom 3 summary xlsx. * Added ‘idle mode’ term to glossary. * gameSetVar(). Deleted requirement to queue a var change in NV memory. QCOM users can do this if it is a requirement in a given protocol/jurisdiction. * Added section 3.8 on secrets. 28/8/2020 updated with an example. * Renamed QCOM API function egmOKex() to playOKex(). Updated function’s description accordingly. Refer to the QCOM 3 summary xlsx RH for more information. * Added section in hw requirements on the use of flash based memory * ‘userloadscripts’ review; made requirements clearer. * s13.9 real-time display of physical door status is now optional * qcom\_slRequest() s11.24.1 added new pre-condition for success. Affects SDK only. * s32 MDP: corrected error: broadcasts must be evey main door open (not logic door open) * s32 MDP: added requirement for both IPv4 & IPv6 broadcasts * Added section 5.3.2 re text file encoding   **Also refer to the QCOM 3 summary spreadsheet change history which contains more detail** | | | | |
| **3.0.2** | **See Below** | **RLL** | **28 June 2019** | **Contact OLGR** |
| **General review based on feedback received over the last year.**  Specific clarifications / changes of note:   * Updated audit mode display requirement re what must be display per QCOM user. Refer s28. * Removed QCOM API functions eventsSetHysteresisTime() & eventsHysteresisTime() in lieu of a hardcoded value. Refer s13.9. * Reviewed and clarified requirements relating to Electronic seals. * Removed the EGM’s game RNG from the scope of QCOM 3. Also provided additional information based on RNG requirement feedback to date. Refer section 3.3. * The MDP service s32 is now implemented by the QLE Lua software driver. * Added section 33.1.2 to help machine developers decide how to best manage QCOM 3 NV related data. * Added section 14.7 regarding state event buffer size and full requirements. * Support for Electronic Seals is now **optional** by default. Most EGMs operating risk is not high enough to justify making this mandatory for all EGMs. * IP and UART callback functions now receive the associated state event data as a table passed as a function argument. Refer sections 12.2 IP callbacks & 11.36.3 UART object member setState() function. * UDP: removed brdports parameter. * Mandated the use of 64 bit meters. * MDP (32.2): Added fields: commissionuid; ready; QCOM users list and quarantine status. * Reviewed & clarified section 14.4 on sync state events. * UDP API: increased the port range to 22000-65535 so that an ordinary QCOM user can implement the MDP service. * Section 30: Machine Software Upgrades: clarified process. | | | | |
| **3.0.2 draft** | **See Below** | **RLL** | **7 Dec 2018** | **Contact OLGR** |
| V3.0.2 draft for industry feedback   * General review and clarification as a result of feedback aligned with intent to implement. * Implementation of bnaFirmwareUpgrade() is no longer optional and is a **high priority** for implementation in EGMs. Refer s21.2. * qcom\_timeSetTimezone() : timezone changes must only be applied at machine boot. * Added new QCOM API function : qcom\_rcResetKey(). * Removed QCOM API function qcom.bnaStackerStatus(). * Documented new QCOM API uart object member functions: qwrite, qcancel, poke & free. Refer s11.36.3. * Serial ports (min. 2) are mandatory. (To aid with existing market penetration. Support to be phased out in the long term.) * IP API: now much more closely aligned to the Berkeley socket API. * Added new section 4.4.1 providing reationale for the Logic Seal confirmation function. * Added socket() function to the IP API. * Published methodology QCOM 3 uses for QCOM user Lua memory use. Refer section 10.2.1.   **The QCOM 3 summary spreadsheet may contain changes not listed here**. | | | | |
| **3.0.1** | **See Below** | **RLL** | **2 May 2018** | **Contact OLGR** |
| * There are many new edits as a result of the document’s language being adjusted in lieu of the fact that the mandatory-to-use QLE Lua software driver implements the majority of all QCOM 3 requirements in this document e.g. “the machine must” is typically changed to “the QLE Lua software driver will” throughout the document. This also included a review of the every use of the word “must” throughout the document. * Deleted the former section 8.2 entitled “QCOM User Certificates”. QCOM User certificates are not required as they have no application. Related: section 8. * Added a qcom\_userShutdown() QCOM API function. Refer s11.30.31. * Added a QCI shutdownuser command. Refer s24.2.6. * Added a close() member function to uart objects. Refer s11.36.3. * Reviewed s10’s introduction on Lua. * Added two new uart API setup functions uartSetp() & uartGetp(). Refer sections 11.36.1 & 11.36.2 respectively. * It was an error to delete qcom\_gameVarList() API function in the latest draft as it was the only way to discover a game’s variations. * Added QCOM API functions qcom\_timeSetStrfmt() & qcom\_timeStrfmt() * Recovered formally removed function gameVarList() * Added qcom\_userSetMyUAApublicKey() QCOM API function * Content auditing: Made more generic and added support for a range of hashing algorithms. Refer section 27 and section 11.13. * Added section 12.1 on Connection Management re the communications API. | | | | |
| **3.0.1 draft**  **edocs#tba** | **See below** | **RLL** | **22 Sep 2017** | **Contact OLGR** |
| * General   + General review, optimisations and clarifications throughout.   + Made certain areas easier to understand and avoid duplication were possible e.g. section 24.   + Fixed three orphaned doc. references *“Error!”…*   + Removed references that indicated the specification was still draft / industry consultation.   + Reviewed and standardised some QCOM API function names.   + Transposition of more applicable requirements from QCOM v1. * Machine Software Upgrades   + Clarified a number of sections that inferred that remote software upgrade support was optional.   + Added section 30.6 related to commissioning issues. * QCOM Users Meters are now a ‘concept only’, given that the QCOM Persistent Variable feature can do the same thing. Refer section 10.11. * Performed a review of QCOM 3 Communications API against QSIM’s and the QCOM 3 SDK’s implementation and addressed some discrepancies:   + The QCOM API function qcom\_tcpClientSecure() has been merged with qcom\_tcpClient() which now takes arguements. Refer s11.38.1.   + Refer verifyCert() (s12.3.4); error messages must be as per the openssl API.   + Renamed disconnect() function to close() (s12.2.9).   + Refer s12.4.2 udp.bind(). Clarified that only privileged source ports may be opened.   + Refer qcom\_udpSetp() s11.37.2. Added brdports parameters what permit what ports a user is allowed to send udp broadcasts on.   + Added missing free() function. Refer section 12.2.10.   + Renamed communications API UDP open() function to bind() (s12.4.2).   + Renamed communications API UDP shutdown() function to close() (s12.4.11)   + Added a UDP new shutdown() function that shuts down rx only - refer section 12.4.10   + Added a UDP set error call-back function (s12.4.6).   + Added sheets concerning the QCOM 3 Comms. API to the QCOM Summary Spreadsheet for testing and reference purposes.   + Deleted qcom\_gameVarList() API function. Was superfluous to qcom\_gameVarGetp() * QCOM 3 machine implementations **must** use QLE Lua source modules in the QCOM 3 SDK. (s33.4). * QCOM 3 Comms. API; added function useCert() (s12.3.2). * Changed qcom\_idMfrAlpha3() to qcom\_idMfr3() (s11.1.9). * Changed qcom\_idMfrName() to qcom\_idMfr() (s11.1.8). * Changed qcom\_machineSetMeterDenom() so that is argument is an integer (s11.7.5) * Added qcom\_x509decode() function (s11.12.1). * Added qcom\_gameCurrent() function (s11.20.1). * Added qcom\_egmOKex() function (s**Error! Reference source not found.**) and EGM\_OK\_EX state event. * QCOM User Account Access (UAA), section 23. Exchanged the SSHD solution in favour for a simpler SSL / TLS based solution. Rationale: SSHD was too linux specific and introduced a whole new service and key format. Whereas by using SSL / TLS it utilised an API and credential format already required for QCOM user communication support. The QCOM API has been updated accordingly. * Reviewed section 22.1 on credit redemption with multiple service providers and made improvements and simplified.   + Removed all qcom\_ectEOI…() functions as the generic qcom\_luaPushlish…() already provide the same functionality.   + Removed all qcom\_ect…UserMeters() functions as the CRM is in a suitable position to do this already via qcom\_userMeter…() class of functions. It also saves the machine implementing functions that may not be used in many operating environments. * QCOM Command Interpreter (s24).   + Removed the requirement for a command prompt. Rationale: Is simply more work for an automated remote logon control process and the humans can easily live without it.   + Added some parameters / requirements to limit machine resources.   + Simplified the *userloadscripts* QCI command (s24.2.4)   + Simplified the explanation of the *lua* QCI command. (s24.2.8) * Changes to anon QCOM users. Refer section 5.1.   + Anon users are now temporary and exist only during QCI anon logins.   + Anon users can now be quarantined (which just logs them out and deletes them. NB a quarantine event with details is still logged in this case). * New state events (refer to the QCOM Summary spreadsheet for more information):   + USER\_LOADSCRIPTS   + USER\_STARTUP   + GAME\_ADDED   + MACHINE\_UPGRADE   + GAME\_VAR\_CHANGED * Refer to the System Lockup qcom\_slRequest() function (s11.25.1) :   + Made button control more generic and did away with the *continue* and *question* flags and associated state event.   + Audit / test modes must be accessible during a system lockup.   + PAEL must be accessible during a system lockup   + ‘lamptest’ flag has been removed. * The concept of peripheral “quiet mode”, formerly defined in section 21 has been reworked and moved to a new section in a new chapter 20 entitled “Fault Conditions”. The QCOM API has been updated in relation to this. * Proposed a Machine Discovery Protocol. Refer chapter 32. * Added support and API for serial ports. Refer section 11.36. | | | | |
| **3.0.0**  **edocs#1605374** | **See below** | **RLL** | **5 Jul 2016** | **TBA** |
| * Updated copyright notice on page 2. *Edocs ref: #1597118* * New QCOM API functions:   + qcom\_luaPrintHistory() s11.15.3.   + qcom\_userQuarantine() s11.30.4.   + qcom\_userIsQuarantine() s11.30.5.   + qcom\_timerSetp() s11.6.2.   + qcom\_events API s11.22.   + qcom\_pv API s11.17. * NTP support is now mandatory. Refer section 9. * Optimised the event schema. Refer section 13.5. * Clarified intentions w.r.t. the machine discovery protocol. Section 4.3.1. * Fixed some issues with section numbering going backwards in places. | | | | |
| **3.0.0 draft 4** | **See below** | **RLL** | **31 May 2016** | **TBA** |
| * Deleted the term “*QCOM Scripting Engine*” in favour for a single term “*QCOM Lua Engine*”. * Reviewed / update the section on privacy. See section 10.2.4. * Added a requirement for an app level write queue to sockets. Refer sections 12.4.8 & 12.2.8. * Reviewed and hardened qcom\_ectAddCredit() s11.23.6 & qcom\_ectTicketInAddCredit() s11.23.10 * Section 18 - TITO, The requirements for ticket in / out logs has been removed as equivalent logs are inherently now available via the QCOM event buffer display in machine audit mode. * Added new QCOM API command: qcom\_userRestart() and equivalent QCI command. Refer section 11.30.30 & 24.2.5. * New events: USER\_RESTART & USER\_NEW\_SCRIPTS. Refer to the QCOM Summary spreadsheet. * Reviewed and hardened sections 10.2.5 & 10.2.6. * Removed QCOM API function qcom\_luaReadOnly() as qcom\_luaPublish() automatically internally write protects the table to be shared. * Added QCOM API function qcom\_luaPublishGetValue() s11.15.11 and reviewed section 10.2.6. * Refer section on Common Content Auditing (s27.1). Added *platform* field. * A state event ID (*seid*) field is now included in all state event data. Refer QCOM summary spreadsheet – State Events sheet. * State Event data is no longer automatically passed as an argument (of type table) to State Event hooked QCOM user scripts. The State Event data must be manually fetched each time via the new QCOM API function, qcom\_luaEventData() Refer section 11.15.7. * The ZIP file format is proposed for QCOM user script archives. Refer section 24.2.4. * QCOM 3 machines must now support both IPv4 and IPv6. Refer section 4.2. * Created a new section on the discovery of QCOM 3 machines on the network. Refer section 4.3.1. * Added section 10.2.7 on QCOM data persistence. * File transfer “protocols to be supported” now also includes **ftp**. * Added hashing and encryption classes to the QCOM API. Section 11.10 & 11.11. * Added a timer class to the QCOM API. Refer section 11.6. | | | | |
| **Version** | **Changes** | **Who** | **Release Date** | **Incept Date** |
| **3.0.0**  **draft 3** | **See below** | **RLL** | **15 Mar 2016** | **TBA** |
| The majority of changes and clarifications are primarily as a result of the implementation of QCOM 3 in QSIM 3 currently in progress.  Summary of the most significant changes:   * The QCOM Lua Engine process watchdog is no longer needed. *(There is now no scenario where a forced restart of the QCOM Lua Engine is required.)* A far better method for CPU use monitoring and control is built within the standard Lua interpreter. See next point below. * Added new section on User Lua Instruction Quota (s5.5). This replaces the formerly published QCOM watchdog. * Refer section 10.1. Deleted the requirement “*if the QCOM Lua interpreter’s process was stuck at 100%, this must have no adverse impact on the integrity of the machine*”. The requirement was benign, misleading and a source of undue concern. This scenario can only occur as a result of a machine software bug (and certainly not by QCOM users), like any other bug which could occur in any other process in the machine. Machines already manage this scenario adequately across all processes (typically they lock-up and require a machine restart which is fine). * Refer section 10.1.   + Added implementation notes to help with the concept and implementation of *QCOM User script seeing the machine in a frozen state*.   + All the formerly listed “*disadvantages (to be managed)*” have now been addressed, the list items have been transposed to the advantage list in the design notes discussion in the same section.   + Added a new core requirement at the bottom of the section. * Refer section 14.5. State Event dispatch latency. Clarified requirements. * Refer section 15.5. Clarified that this feature is a concept proposal only. * Reviewed and hardened the timekeeping class of functions (s11.5).   + Deleted time functions: qcom.time(), qcom.timeClock(), qcom.time\_uptime() and qcom.timeClockResolution(). They are no longer required now that the Lua library *os* time related functions are now available to QCOM users (s10.3.1).   + Moved time zone functions from location class to the timekeeping class in the QCOM API. Time zone is no longer a write-once only item. * Added requirements concerning floating point exceptions that can occur in the QCOM Lua Engine. Also clarified and simplified the requirements in this section. Refer s10.2.2. * Introduced global value HMACseed. Refer summary spreadsheet for definition. * Added getCiphers() & setCiphers() functions to the secure client socket API. Refer section 12.3. * Global type *gamehid* is gone, replaced with gameVer and created the gameName global type. * Deleted the QCOM API function qcom\_idManufID(). Not needed. Use qcom\_idMfr3() instead. * Lowered memory limits for QMA and Anon user. Refer s6.1.5. * Revised section 10.7.1 (QCOM start-up process) as a result of implementation in QSIM 3 and the QCOM 3 SDK. * Added more information to the diagnostics section. Refer section 10.2.12. * Finalised QCOM 3 audit mode requirements. Refer section 28. * Clarified section 10.12.7 a sub-section of Applications and Examples in order to resolve concerns in regards to QCOM 3 machines being used for purposes too far removed their primary function, or to use resources in the machine for purposes that do not contribute to the machines primary function. This is in-line with QCOM 3’s design mandate to be resource minimal. *This section was formerly titled ‘Service Hosting’. However with the removal of Lua co-routines in QCOM 3 draft 2, it is no longer possible for a QCOM user to be able to execute long running scripts in QCOM 3 machines. The reason for this was that after QCOM 3 draft 1 was published, the ability to do this was quickly recognised as overkill and not required in order to be able to accommodate any of the intended possible network based services or applications related to QCOM 3 gaming machines. With the removal of Lua co-routines, QCOM users are limited to the execution of short lived, must return event handler scripts. This, is in addition to the CPU, memory, storage and now Lua instruction quotas, effectively limits all forms of machine resources available to QCOM users.* * Revised section related to the privacy of QCOM user scripts. Section 10.2.4. * Deleted EGM meters: moneyin / moneyout. Reason; they were simply the sum of other meters and all they do is lengthen the implementation of QCOM 3. * Delete appendices on *script engine design notes* and *script review* as they are not required. | | | | |
| **3.0.0**  **draft 2** | **See below** | **RLL** | **18 Dec 2015** | **TBA** |
| Incorporated changes and clarifications as a result of the first round of feedback and the ongoing implementation of QSIM. Namely:   * Clarified context re all uses of the term *commercially neutral.* * Clarified that qcom\_functionName() API functions are actually invoked as **qcom.functionName(args)** in the QCOM Lua interpreter. Refer s11. * Added table showing differences between QMA, anon & regular qcom users. Refer s6.1.5. * Removed support for Lua co-routines in this version. * Clarified that the QCOM Lua Engine must be hosted in a dedicated process. * In the QCOM Lua Engine, during the execution of a set of scripts hooked to a state event, the state and meters of the machine’s main application as seen by QCOM users inside the QCOM Lua Engine must not change until those scripts have finished execution. Refer section 10.1 for more information. * Added qcom\_userCPUstats() & qcom\_userSetCPUquota() functions. Refer CPU Usage, Monitoring and Control s10.2.3. * Refer section 10.5.2. Clarified QCOM API function arguments general requirements. * Fine-tuned QCOM state event definitions.   + Created the TAKE\_WIN state event and adjusted the definition of some gamble related state events.   + Clarified state events and respective meter movements – refer QCOM Summary spreadsheet.   + Door state events are now thrown for each door event.   + FAULT\_CONDITION\_EXIT renamed to FAULT\_CLEARED and is now thrown for each FC cleared (was all).   + Removed STACKER\_REMOVED / STACKER\_RETURNED state events for now. * Machine meters must balance during script execution. Refer s14. * Added state transition diagrams to the QCOM summary spreadsheet. NB The state transitions shown in the diagrams are a requirement. Added event QCOM\_ENGINE\_EXCEPTION. Refer s14.2. * Added the concept of ‘sync’ state events. Refer s14.3. * egmState() review and hardening (11.19.20). * Added meter-too-large sanity check (s15.3) and updated overall section wrt Lua 5.3. * Mandated the use of Lua v5.3 (aka 64 bit integer support). Formerly was Lua v5.1. * Added new chapter aka RUGMs: Machine Software Upgrades (s30). * Communications API (s12)   + Finalised / hardened client sockets.   + Removed *connecting* call-back   + Added onWrite set call-back function   + Revised network quota limit management (s11.38.2) to not disconnect on a breach, but to apply back-pressure.   + Removed USER\_NETWORK\_QUOTA\_EXCEEDED event; no longer needed. * Upgraded from SHA-1 to SHA256 and updated all examples. * qcom\_egmGambleControl() QCOM API command replaced with qcom\_egmGamble() and qcom\_egmSetGamble() commands. Refer s11.19.37. * Added section under content auditing “Related Conventions” s27.4. Minor changes to content auditing schema return values. * Added new section 26, “Advanced Linked Progressive Prize Support ” * Reviewed global types relating to gameuid & gameID and added new global type gamehid | | | | |
| **3.0.0 draft** | **Initial release for the first round of industry comment and feedback.** | **RLL** | **5 May 2015** | **TBA** |

Notes regarding document editing in Word 2013:

* In the tracking pane, open the tracking options, go into ‘advanced options’ and turn off ‘track formatting’ as it prevents tracking/seeing format changes.
* Re code examples; to keep source formatting when copying from notepad++: See menu: plugins->nppExport->RTFtoClipboard; then paste into a Word text box.
* To turn off spell and grammar checking for selected text, select it, create a style, and then turn off spell and grammar checking under format->language. The style may be reused.
* To suppress spell and grammar checking for a single sentence (without creating a new style) omitting the full stop and other punctuation marks sometimes works.
* To view changes by Author via menu: Review->Show Markup->Specific People.

Before releasing check:

* Finalise all “subject to change” where applicable
* As above re ‘TBA’
* All www. Links are still valid
* Assign any new endnotes MRI numbers
* Comments removed
* Change marks all accepted
* Regenerate TOC and select all then F9

# Machine Requirements Index (MRI)

Each endnote in this document (listed below) denotes a QCOM 3 machine requirement aka a test for QCOM 3 machine compliance testing.

Related: QCOM 3 Summary spreadsheet: ‘**Test-Docx**’ sheet

Always reference the MRI number wrt testing and test results.

Do not ever reference the cp:<end-note numbers> as endnote numbers will change over time. Always reference the MRI number instead.

Note: if this document contains tracked changes, then ensure all changes are being shown to ensure endnote links all work properly.

***Next free mri: 369***

1. Refer to this term in the glossary for a list of some possible gaming machine related service providers [↑](#footnote-ref-1)
2. mri:001 [↑](#endnote-ref-1)
3. mri:002 [↑](#endnote-ref-2)
4. Excluding any software or firmware that cannot be modified in any way that could compromise the integrity of a subsequent software upgrade or any game outcome. Also excludes all peripheral device software provided it doesn’t participate in game outcome determination. [↑](#footnote-ref-2)
5. Refer to the related QCOM 3 API function’s ‘SendToHost’ message for acceptable required actions here. [↑](#footnote-ref-3)
6. mri:350 Confirm ping response on ipv4 and ipv6 [↑](#endnote-ref-3)
7. mri:003 [↑](#endnote-ref-4)
8. mri:004 [↑](#endnote-ref-5)
9. mri:005 [↑](#endnote-ref-6)
10. mri:006 [↑](#endnote-ref-7)
11. The 1st edition is freely available on-line at <http://www.lua.org/pil/>. [↑](#footnote-ref-4)
12. mri:007 [↑](#endnote-ref-8)
13. mri:008 [↑](#endnote-ref-9)
14. mri:009 [↑](#endnote-ref-10)
15. mri:010 [↑](#endnote-ref-11)
16. mri:011 [↑](#endnote-ref-12)
17. mri:012 [↑](#endnote-ref-13)
18. Alternatively, the machine may prominently and consistently display a message indicating that data and time configuration / confirmation is required until such time as this is done via the machine’s UI. [↑](#footnote-ref-5)
19. mri:013 [↑](#endnote-ref-14)
20. mri:014 [↑](#endnote-ref-15)
21. mri:015 [↑](#endnote-ref-16)
22. mri:016 [↑](#endnote-ref-17)
23. mri:017 [↑](#endnote-ref-18)
24. mri:018 [↑](#endnote-ref-19)
25. e.g. something in the shape of an Ethernet socket with a red slash/cross through it or equivalent. [↑](#footnote-ref-6)
26. mri:019 [↑](#endnote-ref-20)
27. mri:020 [↑](#endnote-ref-21)
28. mri:021 [↑](#endnote-ref-22)
29. mri:022 [↑](#endnote-ref-23)
30. mri:023 [↑](#endnote-ref-24)
31. mri:024 [↑](#endnote-ref-25)
32. *A machine might also display a picture/s to the operator to allow them to easily confirm the logic area door open detection circuitry is as “expected”. It may also show them where/how to install the physical tamper seal/s.* [↑](#footnote-ref-7)
33. A button linking to the date and time configuration display would be good here. [↑](#footnote-ref-8)
34. mri:025 [↑](#endnote-ref-26)
35. mri:026 [↑](#endnote-ref-27)
36. mri:027 [↑](#endnote-ref-28)
37. mri:028 [↑](#endnote-ref-29)
38. mri:029 [↑](#endnote-ref-30)
39. mri:030 [↑](#endnote-ref-31)
40. mri:031 [↑](#endnote-ref-32)
41. mri:032 [↑](#endnote-ref-33)
42. mri:033 [↑](#endnote-ref-34)
43. mri:034 [↑](#endnote-ref-35)
44. mri:035 [↑](#endnote-ref-36)
45. if implemented [↑](#footnote-ref-9)
46. mri:036 [↑](#endnote-ref-37)
47. The QLE Lua software driver will ensure ECT is disabled via the QCOM API. [↑](#footnote-ref-10)
48. mri:037 [↑](#endnote-ref-38)
49. mri:038 [↑](#endnote-ref-39)
50. mri:039 [↑](#endnote-ref-40)
51. mri:040 [↑](#endnote-ref-41)
52. mri:041 [↑](#endnote-ref-42)
53. This does not include the QMA or anon, or any internal-use-only QCOM users. [↑](#footnote-ref-11)
54. mri:042 [↑](#endnote-ref-43)
55. mri:349 [↑](#endnote-ref-44)
56. mri:043 [↑](#endnote-ref-45)
57. mri:044 [↑](#endnote-ref-46)
58. mri:045 [↑](#endnote-ref-47)
59. mri:046 [↑](#endnote-ref-48)
60. mri:351 [↑](#endnote-ref-49)
61. QCOM API function: **qcom\_userSetSAAcert()** (refer section 6.3) [↑](#footnote-ref-12)
62. mri:047 [↑](#endnote-ref-50)
63. mri:048 [↑](#endnote-ref-51)
64. mri:049 [↑](#endnote-ref-52)
65. mri:050 [↑](#endnote-ref-53)
66. FYI support for Unicode Lua strings in.lua files will be added for better localisation support in future updates QCOM 3. [↑](#footnote-ref-13)
67. mri:352 [↑](#endnote-ref-54)
68. Calling a host function is not to be considered returning control. [↑](#footnote-ref-14)
69. I.e. Lua functions implemented in the host program. [↑](#footnote-ref-15)
70. *Commercially neutral* with respect to the delivery of gaming machine related services. [↑](#footnote-ref-16)
71. Or any permitted user, but for the sake of convenience, this section will only refer to the QMA [↑](#footnote-ref-17)
72. Roles and operations which a QMA may not wish to delegate are those which could potentially create conflicts of interest. For example, delegating the role of broadcasting jackpot current amounts for a jurisdiction to a party/user which also is in the market of developing and selling jackpot display systems could create a conflict of interest. [↑](#footnote-ref-18)
73. mri:051 [↑](#endnote-ref-55)
74. mri:052 [↑](#endnote-ref-56)
75. Refer glossary. [↑](#footnote-ref-19)
76. mri:053 [↑](#endnote-ref-57)
77. mri:054 [↑](#endnote-ref-58)
78. mri:055 [↑](#endnote-ref-59)
79. mri:056 [↑](#endnote-ref-60)
80. mri:057 [↑](#endnote-ref-61)
81. mri:058 [↑](#endnote-ref-62)
82. mri:059 [↑](#endnote-ref-63)
83. mri:060 [↑](#endnote-ref-64)
84. mri:061 [↑](#endnote-ref-65)
85. mri:062 [↑](#endnote-ref-66)
86. mri:063 [↑](#endnote-ref-67)
87. mri:064 [↑](#endnote-ref-68)
88. mri:065 [↑](#endnote-ref-69)
89. mri:066 [↑](#endnote-ref-70)
90. mri:067 [↑](#endnote-ref-71)
91. mri:068 [↑](#endnote-ref-72)
92. mri:069 [↑](#endnote-ref-73)
93. mri:070 [↑](#endnote-ref-74)
94. mri:071 [↑](#endnote-ref-75)
95. mri:072 [↑](#endnote-ref-76)
96. mri:073 [↑](#endnote-ref-77)
97. mri:074 [↑](#endnote-ref-78)
98. mri:075 [↑](#endnote-ref-79)
99. A thread implementation is easier; only use a process if for some reason this is more convenient in the machine. [↑](#footnote-ref-20)
100. mri:076 [↑](#endnote-ref-80)
101. mri:077 [↑](#endnote-ref-81)
102. mri:078 [↑](#endnote-ref-82)
103. Exceptions: QCOM API functions relating to high precision time-stamping. Ref qcom\_timeClock(). [↑](#footnote-ref-21)
104. mri:079 [↑](#endnote-ref-83)
105. Some specific QCOM state events may be allowed to force the main machine application to wait under certain circumstances. Refer ‘sync events’. [↑](#footnote-ref-22)
106. mri:080 [↑](#endnote-ref-84)
107. mri:081 [↑](#endnote-ref-85)
108. Refer to Lua reference manual for a full explanation of the lua\_Alloc() function. [↑](#footnote-ref-23)
109. Note that the *lua\_Alloc* function is frequently called by the QLE. This requires the allocation function to be kept as efficient as possible. Note that the *lua\_Alloc* function in the QCOM 3 SDK does more work than it should for use with QSIM. Lines of code are labelled “QSIM use only” and may be removed from the equivalent function used by actual machines for efficiency. [↑](#footnote-ref-24)
110. This function in turn was based on the example in the Lua reference manual. [↑](#footnote-ref-25)
111. mri:082 [↑](#endnote-ref-86)
112. Refer Lua reference manual **lua\_newstate()** C API function. [↑](#footnote-ref-26)
113. which as of the time of writing, there are none. [↑](#footnote-ref-27)
114. mri:083 [↑](#endnote-ref-87)
115. mri:084 [↑](#endnote-ref-88)
116. mri:085 [↑](#endnote-ref-89)
117. Refer Lua C API function: **lua\_pcall()** [↑](#footnote-ref-28)
118. mri:086 [↑](#endnote-ref-90)
119. mri:353 [↑](#endnote-ref-91)
120. mri:087 [↑](#endnote-ref-92)
121. mri:088 [↑](#endnote-ref-93)
122. mri:089 [↑](#endnote-ref-94)
123. mri:090 [↑](#endnote-ref-95)
124. The key are the Lua v5.3 load() & loadfile() functions; refer to the *\_env* argument. [↑](#footnote-ref-29)
125. No issues if the string is a constant. [↑](#footnote-ref-30)
126. mri:091 [↑](#endnote-ref-96)
127. mri:092 [↑](#endnote-ref-97)
128. *Errors during compilation will be rare in production machines assuming that production scripts have been tested at least once.* [↑](#footnote-ref-31)
129. Which represents no less than 95% of the required Lua code required to implement QCOM 3. [↑](#footnote-ref-32)
130. mri:093 [↑](#endnote-ref-98)
131. mri:094 qle.lua has a built-in text for this. Ref pcall() @ line ~383 [↑](#endnote-ref-99)
132. https://en.wikipedia.org/wiki/Unix\_time [↑](#footnote-ref-33)
133. QCOM users note: A change was made in Lua v5.3.3 where the os.time() library function will now modify the table argument if supplied. (It didn’t in Lua v5.3.0 and earlier). The Lua v5.3.4 reference manual does not mention that the function will modify its argument. #undocumented change. Related: <http://lua-users.org/lists/lua-l/2017-01/msg00274.html> [↑](#footnote-ref-34)
134. mri:095 [↑](#endnote-ref-100)
135. Note: Lua documentation states that this function may not be thread safe on non-POSIX machine systems. [↑](#footnote-ref-35)
136. mri:096 [↑](#endnote-ref-101)
137. e.g. <http://stackoverflow.com/questions/1224708/how-can-i-create-a-secure-lua-sandbox>

     <http://lua-users.org/wiki/SandBoxes> [↑](#footnote-ref-36)
138. mri:100 [↑](#endnote-ref-102)
139. Note that during the QLE startup below, the QLE Lua software driver can log machine events which must be saved by the machine in NV memory. [↑](#footnote-ref-37)
140. mri:101 [↑](#endnote-ref-103)
141. mri:102 [↑](#endnote-ref-104)
142. mri:103 [↑](#endnote-ref-105)
143. mri:104 [↑](#endnote-ref-106)
144. Simply printing to a string is the easiest way to implement. An example of this approach for the hms->eventb key (qcom 3 event buffer serialisatron), may be found in the SDK example module egmevent.cpp : q3eb\_pushBuffer() function. [↑](#footnote-ref-38)
145. Alternatively, another approach is to use the Lua C API (built into the Lua interpreter) and build/push the hms table directly into the Lua state; then just execute qle\_dostring("qleHMSreceived()"). Examples: refer SDK cpp modules where lua\_newtable() is called. [↑](#footnote-ref-39)
146. mri:105 [↑](#endnote-ref-107)
147. mri:106 [↑](#endnote-ref-108)
148. mri:107 [↑](#endnote-ref-109)
149. NB the MDP service may be started earlier if it’s implemented via a QCOM user. [↑](#footnote-ref-40)
150. mri:108 [↑](#endnote-ref-110)
151. .txt files are only for display purposes within EGM audit mode. [↑](#footnote-ref-41)
152. mri:109 [↑](#endnote-ref-111)
153. it is common practise in Lua, for library modules to return a table of functions. [↑](#footnote-ref-42)
154. Under consideration: the ability to overload any other [library] function, or value in the QCOM Lua engine global environment. [↑](#footnote-ref-43)
155. mri:110 [↑](#endnote-ref-112)
156. mri:111 [↑](#endnote-ref-113)
157. Multiple physical copies are not mandatory for QCOM PV storage. [↑](#footnote-ref-44)
158. This value is subject to change until late in the implementation phase. [↑](#footnote-ref-45)
159. mri:112 [↑](#endnote-ref-114)
160. mri:113 [↑](#endnote-ref-115)
161. mri:114 [↑](#endnote-ref-116)
162. mri:115 [↑](#endnote-ref-117)
163. mri:116 [↑](#endnote-ref-118)
164. mri:117 [↑](#endnote-ref-119)
165. mri:118 [↑](#endnote-ref-120)
166. mri:119 [↑](#endnote-ref-121)
167. mri:120 [↑](#endnote-ref-122)
168. mri:121 [↑](#endnote-ref-123)
169. mri:122 [↑](#endnote-ref-124)
170. ‘External’ meaning that the jackpot is not triggered by the game being played on the machine, but by a controller device external to the game; i.e. the jackpot prize does not appear on the game’s pay-scale [↑](#footnote-ref-46)
171. This does not imply that OLGR is endorsing or considering the concept, but merely a statement outlining a possible theoretical mode of operation [↑](#footnote-ref-47)
172. mri:339 [↑](#endnote-ref-125)
173. mri:123 [↑](#endnote-ref-126)
174. mri:124 [↑](#endnote-ref-127)
175. mri:340 [↑](#endnote-ref-128)
176. mri:125 [↑](#endnote-ref-129)
177. mri:126 [↑](#endnote-ref-130)
178. mri:127 [↑](#endnote-ref-131)
179. mri:128 [↑](#endnote-ref-132)
180. mri:129 [↑](#endnote-ref-133)
181. mri:130 [↑](#endnote-ref-134)
182. An OS level broadcast may be required for all machine programs to pickup the time zone change. Also, machine devs should not assume that all the machine services will automatically pickup a timezone change on-the-fly and should verify that all applicable running programs do pickup the change. Older programs / services might need restarting before they pickup a time zone change. [↑](#footnote-ref-48)
183. mri:131 [↑](#endnote-ref-135)
184. Note that the time zone values here are different in both units and signage wrt the machine’s UI for human based time zone entry and audit mode display. Regardless this is the same time zone value and must be converted as required. Internally the machine may store it how it likes. [↑](#footnote-ref-49)
185. mri:132 [↑](#endnote-ref-136)
186. mri:133 [↑](#endnote-ref-137)
187. mri:134 [↑](#endnote-ref-138)
188. mri:135 [↑](#endnote-ref-139)
189. mri:136 [↑](#endnote-ref-140)
190. mri:137 [↑](#endnote-ref-141)
191. Hint: refer Lua function: string.dump(function) [↑](#footnote-ref-50)
192. mri:362 [↑](#endnote-ref-142)
193. mri:363 [↑](#endnote-ref-143)
194. mri:364 [↑](#endnote-ref-144)
195. mri:365 [↑](#endnote-ref-145)
196. mri:366 [↑](#endnote-ref-146)
197. mri:367 [↑](#endnote-ref-147)
198. mri:368 [↑](#endnote-ref-148)
199. mri:354 [↑](#endnote-ref-149)
200. mri:138 [↑](#endnote-ref-150)
201. mri:139 [↑](#endnote-ref-151)
202. This approach will result in one set of game meters for all denomninations of a game. Also a ‘den’ field would need to be added to the PLAY\_COMMENCED state event. [↑](#footnote-ref-51)
203. In Queensland, EGMs should continue to implement player selectable denomination games as per QCOM v1.6. Postfix gamename fields with the game’s denomination e.g. Bigbucks\_10; Bigbucks\_50; Bigbucks\_100 [↑](#footnote-ref-52)
204. mri:140 [↑](#endnote-ref-152)
205. mri:141 [↑](#endnote-ref-153)
206. mri:142 [↑](#endnote-ref-154)
207. mri:144 [↑](#endnote-ref-155)
208. Consult with OLGR before setting up a condition for denial not currently listed in the **GAME\_VAR\_CHANGED** state event. [↑](#footnote-ref-53)
209. mri:143 [↑](#endnote-ref-156)
210. mri:144 [↑](#endnote-ref-157)
211. mri:356 [↑](#endnote-ref-158)
212. As reported via the **qcom\_progrGetp()** QCOM API function. [↑](#footnote-ref-54)
213. In Queensland, a game’s Min/Max TPRTP are typically considered equal and there won’t be a maxrtp property reported. [↑](#footnote-ref-55)
214. In QCOM v1, a game could also have a progressive configuration per game variation, but if it did then the game was not permitted to have variation hot-switching. [↑](#footnote-ref-56)
215. Implements what was QCOM v1’s customSAP feature [↑](#footnote-ref-57)
216. mri:145 [↑](#endnote-ref-159)
217. mri:146 [↑](#endnote-ref-160)
218. mri:147 [↑](#endnote-ref-161)
219. mri:148 [↑](#endnote-ref-162)
220. mri:150 [↑](#endnote-ref-163)
221. refer definition of ‘ok’ refer section 11.18.28 titled qcom\_egmOK. [↑](#footnote-ref-58)
222. mri:151 [↑](#endnote-ref-164)
223. mri:152 [↑](#endnote-ref-165)
224. mri:355 [↑](#endnote-ref-166)
225. mri:153 [↑](#endnote-ref-167)
226. If there are any issues with the title, or it’s length is less than 4 characters, or greater than 16 characters in length, then the QLE LSD will substitute the user’s *username* as the title. [↑](#footnote-ref-59)
227. mri:154 [↑](#endnote-ref-168)
228. mri:155 [↑](#endnote-ref-169)
229. mri:156 [↑](#endnote-ref-170)
230. refer definition of ‘ok’ refer section 11.18.28 titled qcom\_egmOK. [↑](#footnote-ref-60)
231. mri:157 [↑](#endnote-ref-171)
232. mri:158 [↑](#endnote-ref-172)
233. mri:160 [↑](#endnote-ref-173)
234. mri:161 [↑](#endnote-ref-174)
235. mri:162 [↑](#endnote-ref-175)
236. mri:163 [↑](#endnote-ref-176)
237. mri:164 [↑](#endnote-ref-177)
238. mri:165 [↑](#endnote-ref-178)
239. mri:166 [↑](#endnote-ref-179)
240. mri:167 [↑](#endnote-ref-180)
241. ‘Active’ in the context of a sl means the SL has been entered into on the EGM and: all doors closed, no in audit/test, no faults. It doesn’t matter if the SL is not the currently displayed SL. [↑](#footnote-ref-61)
242. refer definition of ‘ok’ refer section 11.18.28 titled qcom\_egmOK. [↑](#footnote-ref-62)
243. mri:168 [↑](#endnote-ref-181)
244. Design rationale: keep it simple; if both slRequest and slReset could be queued, it creates a more complicated set of test cases. [↑](#footnote-ref-63)
245. mri:341 [↑](#endnote-ref-182)
246. mri:171 [↑](#endnote-ref-183)
247. mri:172 [↑](#endnote-ref-184)
248. mri:173 [↑](#endnote-ref-185)
249. mri:342 check while in SL, fault, door open & audit mode [↑](#endnote-ref-186)
250. mri:174 [↑](#endnote-ref-187)
251. mri:175 [↑](#endnote-ref-188)
252. mri:176 [↑](#endnote-ref-189)
253. mri:177 [↑](#endnote-ref-190)
254. mri:178 [↑](#endnote-ref-191)
255. mri:179 [↑](#endnote-ref-192)
256. mri:180 [↑](#endnote-ref-193)
257. mri:181 [↑](#endnote-ref-194)
258. mri:182 [↑](#endnote-ref-195)
259. mri:183 [↑](#endnote-ref-196)
260. mri:184 [↑](#endnote-ref-197)
261. mri:185 [↑](#endnote-ref-198)
262. mri:186 [↑](#endnote-ref-199)
263. It is up to the EGM manufacturer how fast their response times can be, although OLGR will encourage < 50 msec. However, if the EGM manufacturer for example wants to provide support for their QCOM 3 EGM running the qcom16 app outside\* of the State of QLD, then QCOM v1 protocol response times would be a requirement. (\*QLD clubs and hotels EGM monitoring systems now support up to a 50msec response time for this scenario). [↑](#footnote-ref-64)
264. mri:187 [↑](#endnote-ref-200)
265. mri:188 [↑](#endnote-ref-201)
266. mri:189 [↑](#endnote-ref-202)
267. The time to write a byte on the COM port following a given rx byte on the COM port. [↑](#footnote-ref-65)
268. mri:190 [↑](#endnote-ref-203)
269. mri:191 [↑](#endnote-ref-204)
270. mri:192 [↑](#endnote-ref-205)
271. mri:193 [↑](#endnote-ref-206)
272. mri:194 [↑](#endnote-ref-207)
273. The qcom16 app is supplied with QSIM 3 [↑](#footnote-ref-66)
274. mri:195 [↑](#endnote-ref-208)
275. mri:196 [↑](#endnote-ref-209)
276. mri:197 [↑](#endnote-ref-210)
277. mri:198 [↑](#endnote-ref-211)
278. mri:199 [↑](#endnote-ref-212)
279. mri:200 [↑](#endnote-ref-213)
280. mri:201 [↑](#endnote-ref-214)
281. mri:202 [↑](#endnote-ref-215)
282. *Commercially neutral* with respect to the delivery of gaming machine related services. [↑](#footnote-ref-67)
283. https://man7.org/linux/man-pages/man2/connect.2.html [↑](#footnote-ref-68)
284. mri:203 [↑](#endnote-ref-216)
285. mri:204 [↑](#endnote-ref-217)
286. https://en.wikipedia.org/wiki/Ephemeral\_port [↑](#footnote-ref-69)
287. As per QCOM v1, the QCOM 3 event buffer does not have to be stored as GMNS “critical memory” (i.e. one persistent copy in NV memory is sufficient) [↑](#footnote-ref-70)
288. mri:205 [↑](#endnote-ref-218)
289. mri:206 [↑](#endnote-ref-219)
290. mri:207 [↑](#endnote-ref-220)
291. mri:208 [↑](#endnote-ref-221)
292. Refer QCOM API function qcom\_machineOperatingTime() [↑](#footnote-ref-71)
293. mri:209 [↑](#endnote-ref-222)
294. mri:210 [↑](#endnote-ref-223)
295. mri:211 [↑](#endnote-ref-224)
296. mri:212 [↑](#endnote-ref-225)
297. *Commercially neutral* with respect to the delivery of gaming machine related services. [↑](#footnote-ref-72)
298. mri:213 [↑](#endnote-ref-226)
299. https://en.wikipedia.org/wiki/Unix\_time [↑](#footnote-ref-73)
300. mri:214 [↑](#endnote-ref-227)
301. os.date supports custom string time formats per the strftime() ISO C function. [↑](#footnote-ref-74)
302. mri:215 [↑](#endnote-ref-228)
303. Excluding stacker removed/returned events [↑](#footnote-ref-75)
304. mri:216 [↑](#endnote-ref-229)
305. mri:217 [↑](#endnote-ref-230)
306. mri:218 [↑](#endnote-ref-231)
307. mri:219 [↑](#endnote-ref-232)
308. the machine may also indicate which door is closing if desired. [↑](#footnote-ref-76)
309. mri:220 [↑](#endnote-ref-233)
310. mri:343 [↑](#endnote-ref-234)
311. mri:221 [↑](#endnote-ref-235)
312. mri:222 [↑](#endnote-ref-236)
313. mri:223 [↑](#endnote-ref-237)
314. mri:224 [↑](#endnote-ref-238)
315. refer definition of ‘ok’ refer section 11.18.28 titled qcom\_egmOK. [↑](#footnote-ref-77)
316. mri:225 [↑](#endnote-ref-239)
317. mri:226 [↑](#endnote-ref-240)
318. mri:227 [↑](#endnote-ref-241)
319. Refer to the QCOM 3 SDK: **syncevents.lua** module [↑](#footnote-ref-78)
320. mri:228 [↑](#endnote-ref-242)
321. It is also viable to simply ignore human input during the period, given how short the period will be and that a human would inherently try again if their input is ignored. This approach, if used, must not affect auto-play however. [↑](#footnote-ref-79)
322. Refer to the Lua API C pcall() library function and the QCOM 3 SDK for example code. [↑](#footnote-ref-80)
323. Take a reading while the machine is idle and in play [↑](#footnote-ref-81)
324. mri:229 [↑](#endnote-ref-243)
325. mri:230 [↑](#endnote-ref-244)
326. mri:231 [↑](#endnote-ref-245)
327. mri:232 [↑](#endnote-ref-246)
328. mri:233 [↑](#endnote-ref-247)
329. mri:234 [↑](#endnote-ref-248)
330. mri:357 [↑](#endnote-ref-249)
331. mri:344 [↑](#endnote-ref-250)
332. mri:345 [↑](#endnote-ref-251)
333. mri:346 [↑](#endnote-ref-252)
334. mri:347 [↑](#endnote-ref-253)
335. mri:348 #CEO [↑](#endnote-ref-254)
336. mri:235 [↑](#endnote-ref-255)
337. mri:236 [↑](#endnote-ref-256)
338. mri:237 [↑](#endnote-ref-257)
339. mri:238 [↑](#endnote-ref-258)
340. mri:239 [↑](#endnote-ref-259)
341. mri:240 [↑](#endnote-ref-260)
342. mri:241 [↑](#endnote-ref-261)
343. mri:242 [↑](#endnote-ref-262)
344. mri:243 [↑](#endnote-ref-263)
345. mri:244 [↑](#endnote-ref-264)
346. The difference to the machine is how they are reported to the QLE LSD via state events. Faults use the **FAULT\_CONDITION & FAULT\_CLEARED** state events and doors use the **DOOR\_OPENED & DOOR\_CLOSED** state events. [↑](#footnote-ref-82)
347. mri:245 [↑](#endnote-ref-265)
348. mri:246 [↑](#endnote-ref-266)
349. Machines using a key safe must use a master key strength equivalent to 256 bits or higher. [↑](#footnote-ref-83)
350. mri:247 [↑](#endnote-ref-267)
351. mri:248 [↑](#endnote-ref-268)
352. mri:249 [↑](#endnote-ref-269)
353. mri:250 [↑](#endnote-ref-270)
354. mri:251 [↑](#endnote-ref-271)
355. mri:252 [↑](#endnote-ref-272)
356. mri:253 [↑](#endnote-ref-273)
357. These upgrades are expected to be quite small; e.g. 5…20MB in size [↑](#footnote-ref-84)
358. mri:254 [↑](#endnote-ref-274)
359. mri:255 [↑](#endnote-ref-275)
360. e.g. an account/card play, cashless only machine where collects are instigated from a device external to the machine, such as the player removing their card. [↑](#footnote-ref-85)
361. mri:256 [↑](#endnote-ref-276)
362. mri:257 [↑](#endnote-ref-277)
363. mri:258 [↑](#endnote-ref-278)
364. mri:259 [↑](#endnote-ref-279)
365. mri:260 [↑](#endnote-ref-280)
366. mri:261 [↑](#endnote-ref-281)
367. mri:262 [↑](#endnote-ref-282)
368. mri:263 [↑](#endnote-ref-283)
369. mri:264 refer SDK example: qcithreadunit.cpp: verify\_callback() [↑](#endnote-ref-284)
370. mri:265 [↑](#endnote-ref-285)
371. mri:266 [↑](#endnote-ref-286)
372. mri:360 refer SDK example: qcithreadunit.cpp: verify\_callback() [↑](#endnote-ref-287)
373. mri:267 [↑](#endnote-ref-288)
374. mri:268 [↑](#endnote-ref-289)
375. mri:269 [↑](#endnote-ref-290)
376. mri:270 [↑](#endnote-ref-291)
377. mri:361 refer SDK example: qcithreadunit.cpp: verify\_callback() [↑](#endnote-ref-292)
378. It is ok for it to be optional because client certificate public keys are always verified by the QLE software driver either way. A QCOM user’s connection will be terminated by the QLE LSD unless their public key is known to it (re: **qcom\_userSetUAApublicKey()**). Also refer to the **QCIE\_USER\_REFUSED** message type in the QCOM 3 summary spreadsheet. [↑](#footnote-ref-86)
379. mri:271 [↑](#endnote-ref-293)
380. mri:272 [↑](#endnote-ref-294)
381. mri:273 [↑](#endnote-ref-295)
382. mri:274 [↑](#endnote-ref-296)
383. mri:275 [↑](#endnote-ref-297)
384. mri:276 [↑](#endnote-ref-298)
385. mri:277 [↑](#endnote-ref-299)
386. mri:278 [↑](#endnote-ref-300)
387. mri:279 [↑](#endnote-ref-301)
388. mri:280 [↑](#endnote-ref-302)
389. mri:281 [↑](#endnote-ref-303)
390. For this requirement, treat printable characters as ASCII codes 33-126; also include spaces (32). [↑](#footnote-ref-87)
391. mri:282 [↑](#endnote-ref-304)
392. *Devnote: the QMA cert must not be able to be changed once set in order to prevent unauthorised machine upgrades from occurring just after logic seal confirmation.* [↑](#footnote-ref-88)
393. mri:283 [↑](#endnote-ref-305)
394. ‘external’ as in externally triggered with respect to the machine’s resident games [↑](#footnote-ref-89)
395. It is optional for the EGM manufacturer to support multiple progressive modes for any given level. [↑](#footnote-ref-90)
396. mri:358 [↑](#endnote-ref-306)
397. mri:284 [↑](#endnote-ref-307)
398. Implements what was QCOM v1’s customSAP feature [↑](#footnote-ref-91)
399. mri:286 [↑](#endnote-ref-308)
400. mri:287 [↑](#endnote-ref-309)
401. These events must be queued the instant each specific progressive level won is revealed to the winning player (either directly, or by inference). [↑](#footnote-ref-92)
402. mri:288 [↑](#endnote-ref-310)
403. mri:289 [↑](#endnote-ref-311)
404. mri:290 [↑](#endnote-ref-312)
405. FYI QCOM v1 also had a ‘Verifying please wait’ stage associated with LP lockups. This is removed from QCOM 3 as it was rarely visible and arguably unnecessary; it’s removal will simplify LP implementation in QCOM 3 machines. [↑](#footnote-ref-93)
406. mri:291 [↑](#endnote-ref-313)
407. covers all types of progressive awards (e.g. sap & lp) [↑](#footnote-ref-94)
408. QCOM v1 required a LP award lockup entry exit per LP award. [↑](#footnote-ref-95)
409. mri:359 [↑](#endnote-ref-314)
410. **qcom.progrSetp()** implements what was QCOM v1’s customSAP feature [↑](#footnote-ref-96)
411. Win events, unreasonable contributions, configuration changes, etc.; [↑](#footnote-ref-97)
412. In Queensland this would be “Site Controllers” (refer QCOM v1). [↑](#footnote-ref-98)
413. The totaliser VM scripts and EGMs may communicate via a network protocol of the EGM manufacturer’s choosing. (Must meet minimum requirements and be evaluated and approved.) [↑](#footnote-ref-99)
414. mri:292 [↑](#endnote-ref-315)
415. mri:293 [↑](#endnote-ref-316)
416. mri:294 [↑](#endnote-ref-317)
417. mri:295 [↑](#endnote-ref-318)
418. Contact OLGR for the current time-out value. [↑](#footnote-ref-100)
419. Individual *uid* fields can be easily extruded via a Lua script such as:

     **name,ver,year,loc = string.match(uid,"([^;]+);([^;]+);([^;]+);([^;]+)")** [↑](#footnote-ref-101)
420. For game content auditing, the EGM is not required to be able to run more than one game content audit type (“cd” or “media”), across all games, at a time. In other words, only one process/thread in the EGM should ever be doing a game content at a time in the whole EGM. [↑](#footnote-ref-102)
421. mri:296 [↑](#endnote-ref-319)
422. This is an error checking application w.r.t. this hash, so a SHA-1 is considered adequate. [↑](#footnote-ref-103)
423. mri:298 [↑](#endnote-ref-320)
424. Windows OS machines can convert this from the current Windows value for FILETIME if its easier than accessing C RTL time() function here. [↑](#footnote-ref-104)
425. mri:299 [↑](#endnote-ref-321)
426. mri:300 [↑](#endnote-ref-322)
427. mri:301 [↑](#endnote-ref-323)
428. mri:302…This is not mandatory functionality [↑](#endnote-ref-324)
429. e.g. any combination of image distortion, colour removal, blurring, or lossy filter [↑](#footnote-ref-105)
430. mri:303 [↑](#endnote-ref-325)
431. mri:304 [↑](#endnote-ref-326)
432. mri:305 [↑](#endnote-ref-327)
433. mri:306 [↑](#endnote-ref-328)
434. mri:307 [↑](#endnote-ref-329)
435. mri:308 [↑](#endnote-ref-330)
436. In Queensland, the abbreviated term is LTFO’s [↑](#footnote-ref-106)
437. mri:309 [↑](#endnote-ref-331)
438. mri:310 [↑](#endnote-ref-332)
439. mri:311 [↑](#endnote-ref-333)
440. mri:312 [↑](#endnote-ref-334)
441. mri:313 [↑](#endnote-ref-335)
442. mri:314 [↑](#endnote-ref-336)
443. mri:315 [↑](#endnote-ref-337)
444. mri:316 [↑](#endnote-ref-338)
445. mri:317 [↑](#endnote-ref-339)
446. mri:318 [↑](#endnote-ref-340)
447. mri:319 [↑](#endnote-ref-341)
448. mri:320 [↑](#endnote-ref-342)
449. mri:321 [↑](#endnote-ref-343)
450. mri:322 [↑](#endnote-ref-344)
451. mri:323 [↑](#endnote-ref-345)
452. mri:324 [↑](#endnote-ref-346)
453. mri:325 [↑](#endnote-ref-347)
454. mri:326 [↑](#endnote-ref-348)
455. mri:327 [↑](#endnote-ref-349)
456. mri:328 [↑](#endnote-ref-350)
457. mri:329 [↑](#endnote-ref-351)
458. The machine may opt to simply look for a specific file or within a specific directory on the device if desired. [↑](#footnote-ref-107)
459. mri:330 [↑](#endnote-ref-352)
460. mri:331 [↑](#endnote-ref-353)
461. mri:332 [↑](#endnote-ref-354)
462. mri:333 [↑](#endnote-ref-355)
463. mri:334 [↑](#endnote-ref-356)
464. mri:335 [↑](#endnote-ref-357)
465. mri:336 [↑](#endnote-ref-358)
466. mri:337 [↑](#endnote-ref-359)
467. https://en.wikipedia.org/wiki/Wine\_(software) [↑](#footnote-ref-108)
468. mri:338 [↑](#endnote-ref-360)
469. https://en.wikipedia.org/wiki/Video\_game\_console [↑](#footnote-ref-109)
470. For example https://en.wikipedia.org/wiki/Nvidia\_Shadowplay [↑](#footnote-ref-110)
471. this list does not imply or infer that the application listed here would be acceptable for use in Queensland, or any jurisdiction for that matter [↑](#footnote-ref-111)