

# User Manual

For QP/C 8.0.3

---

Document: DOC\_MAN\_QPC

© Quantum Leaps, LLC  
<https://www.state-machine.com>  
info@state-machine.com



---

<b>1 Overview</b>	<b>1</b>
1.1 What is it? . . . . .	1
1.2 What does it do? . . . . .	2
1.3 What's special about it? . . . . .	2
1.3.1 Functional Safety . . . . .	2
1.3.2 QP Framework Editions . . . . .	2
1.3.3 SafeQP Editions . . . . .	3
1.3.4 SafeQP Certification Kits . . . . .	3
1.3.5 Object Orientation . . . . .	3
1.3.6 Hierarchical State Machines . . . . .	4
1.3.7 Built-in Kernels . . . . .	4
1.3.8 3rd-Party Kernels . . . . .	4
1.3.9 Size and Efficiency . . . . .	4
1.3.10 Software Tracing . . . . .	5
1.3.11 Documentation Traceability . . . . .	5
1.3.12 Popularity & Maturity . . . . .	5
1.3.13 Books . . . . .	5
1.4 How is it licensed? . . . . .	6
1.4.1 Open Source Projects . . . . .	6
1.4.2 Closed Source Projects . . . . .	6
1.5 How to get help? . . . . .	6
1.6 Contact Information . . . . .	6
<b>2 Getting Started</b>	<b>7</b>
2.1 Downloading & Installing QP/C . . . . .	8
2.1.1 Downloading QP/C in QP-Bundle . . . . .	8
2.1.2 Downloading QP/C from GitHub . . . . .	8
2.1.3 Downloading QP/C as a CMSIS-Pack . . . . .	9
2.1.4 QP/C Installation Folder . . . . .	10
2.1.5 QP License Certificate File . . . . .	11
2.2 QP/C Tutorial . . . . .	12
2.2.1 Simple Blinky Application . . . . .	13
2.2.2 Dining Philosophers Problem (DPP) . . . . .	15
2.2.3 "Fly 'n' Shoot" Game . . . . .	17
2.2.4 Low-Power Example . . . . .	20
2.2.4.1 The Tickless Mode . . . . .	20
2.2.4.2 Multiple Tick Rates . . . . .	21
2.2.4.3 The Low-Power Code . . . . .	21
2.2.4.4 State Machines . . . . .	23

---

2.2.4.5 The Idle Callback (QK/QXK) . . . . .	25
<b>3 Ports</b>	<b>27</b>
3.1 General Comments . . . . .	27
3.1.1 Code Structure . . . . .	27
3.2 PC-Lint-Plus . . . . .	28
3.2.1 Linting the QP/C Source Code . . . . .	29
3.2.2 Linting QP/C Application Code . . . . .	29
3.2.3 Structure of PC-Lint-Plus Options for QP/C . . . . .	30
3.2.3.1 The std.int option file . . . . .	30
3.2.3.2 The qpc.int option file . . . . .	31
3.3 Native Ports (Built-in Kernels) . . . . .	31
3.3.1 ARM Cortex-M . . . . .	31
3.3.1.1 Directories and Files . . . . .	32
3.3.1.2 Interrupts in the QP/C Ports to ARM Cortex-M . . . . .	32
3.3.1.3 References . . . . .	36
3.3.1.4 Non-preemptive QV Kernel . . . . .	36
3.3.1.5 Preemptive Non-Blocking QK Kernel . . . . .	43
3.3.1.6 Preemptive "Dual-Mode" QXK Kernel . . . . .	58
3.3.2 ARM Cortex-R . . . . .	60
3.3.3 MSP430 . . . . .	61
3.4 Ports to Third-Party RTOS . . . . .	61
3.4.1 embOS . . . . .	62
3.4.1.1 QP/C Source Files Needed in this QP/C Port . . . . .	63
3.4.2 FreeRTOS . . . . .	63
3.4.2.1 About the QP/C Port to FreeRTOS . . . . .	64
3.4.2.2 Example Code . . . . .	65
3.4.3 ThreadX . . . . .	66
3.4.3.1 QP/C Source Files Needed in this QP/C Port . . . . .	66
3.4.4 uC-OS2 . . . . .	67
3.4.4.1 About the QP/C Port to uC-OS2 . . . . .	67
3.4.4.2 uC-OS2 Source and ARM Cortex-M3/M4 Ports . . . . .	68
3.4.4.3 Examples for the uC-OS2 Port . . . . .	68
3.4.5 Zephyr . . . . .	69
3.4.5.1 About the QP/C Port to Zephyr . . . . .	69
3.4.5.2 Examples for the Zephyr port . . . . .	70
3.5 Ports to General-Purpose OSes . . . . .	70
3.5.1 POSIX-QV (Single Threaded) . . . . .	71
3.5.2 POSIX (Multithreaded) . . . . .	71

---

3.5.3 Win32-QV (Single Threaded) . . . . .	71
3.5.4 Win32 API (Multithreaded) . . . . .	72
<b>4 Examples</b>	<b>73</b>
4.1 General Comments . . . . .	73
4.1.1 Example Applications . . . . .	73
4.1.2 Development Boards . . . . .	74
4.1.3 Development Tools . . . . .	74
4.1.4 Build Configurations . . . . .	74
4.1.5 QM Models . . . . .	74
4.1.6 Third-Party Code . . . . .	75
4.1.7 Creating your Own QP/C Projects . . . . .	75
4.1.8 Next Steps and Further Reading About QP and QM . . . . .	76
4.2 Example Code Organization . . . . .	76
4.2.1 Native Examples . . . . .	77
4.2.2 Examples for 3rd-party RTOS . . . . .	77
4.2.3 Examples for GPOS . . . . .	78
4.2.4 Examples for 3rd-party Middleware . . . . .	78
4.2.5 Examples for QUTest . . . . .	78
4.2.6 Examples for QWin-GUI . . . . .	78
<b>5 Software Requirements Specification</b>	<b>79</b>
5.1 About this Document . . . . .	79
5.1.1 DOC_SRS_QP . . . . .	79
5.2 Document Conventions . . . . .	81
5.2.1 General Requirement UIDs . . . . .	81
5.2.2 Use of "shall" . . . . .	81
5.2.3 Use of "shall not" . . . . .	81
5.2.4 Use of "should" . . . . .	81
5.2.5 Use of "may" . . . . .	81
5.3 References . . . . .	81
5.4 Overview . . . . .	83
5.4.1 Goals . . . . .	83
5.4.2 Functional Safety . . . . .	83
5.4.3 Context of Use . . . . .	84
5.4.3.1 Inversion of Control . . . . .	85
5.4.3.2 Framework, NOT a Library . . . . .	85
5.4.3.3 Deriving Application from a Framework . . . . .	85
5.4.4 Main Use Cases . . . . .	85
5.4.5 Portability & Configurability . . . . .	85

---

5.4.5.1 Compile-Time Configurability . . . . .	85
5.4.5.2 Run-Time Configurability . . . . .	86
5.5 Active Objects . . . . .	87
5.5.1 Concepts & Definitions . . . . .	87
5.5.1.1 Active Object Model of Computation . . . . .	87
5.5.1.2 Active Objects . . . . .	87
5.5.1.3 Encapsulation for Concurrency . . . . .	87
5.5.1.4 Shared-Nothing Principle . . . . .	88
5.5.1.5 Execution Context . . . . .	89
5.5.1.6 Priority . . . . .	89
5.5.1.7 Event Queue . . . . .	89
5.5.1.8 Asynchronous Communication . . . . .	89
5.5.1.9 Run-to-Completion (RTC) . . . . .	89
5.5.1.10 Current Event . . . . .	90
5.5.1.11 No Blocking . . . . .	90
5.5.1.12 Support For State Machines . . . . .	90
5.5.1.13 Inversion of Control . . . . .	90
5.5.1.14 Framework vs. Toolkit . . . . .	90
5.5.1.15 Low Power Architecture . . . . .	91
5.5.2 Requirements . . . . .	91
5.5.2.1 SRS_QP_AO_00 . . . . .	91
5.5.2.2 SRS_QP_AO_01 . . . . .	91
5.5.2.3 SRS_QP_AO_10 . . . . .	92
5.5.2.4 SRS_QP_AO_11 . . . . .	92
5.5.2.5 SRS_QP_AO_20 . . . . .	93
5.5.2.6 SRS_QP_AO_21 . . . . .	93
5.5.2.7 SRS_QP_AO_22 . . . . .	94
5.5.2.8 SRS_QP_AO_23 . . . . .	94
5.5.2.9 SRS_QP_AO_30 . . . . .	94
5.5.2.10 SRS_QP_AO_31 . . . . .	95
5.5.2.11 SRS_QP_AO_32 . . . . .	95
5.5.2.12 SRS_QP_AO_40 . . . . .	95
5.5.2.13 SRS_QP_AO_50 . . . . .	96
5.5.2.14 SRS_QP_AO_51 . . . . .	96
5.5.2.15 SRS_QP_AO_60 . . . . .	97
5.5.2.16 SRS_QP_AO_70 . . . . .	97
5.6 Events . . . . .	98
5.6.1 Concepts & Definitions . . . . .	98
5.6.1.1 Event Signal . . . . .	98

---

5.6.1.2 Event Parameters . . . . .	98
5.6.2 Requirements . . . . .	98
5.6.2.1 SRS_QP_EVT_00 . . . . .	98
5.6.2.2 SRS_QP_EVT_20 . . . . .	99
5.6.2.3 SRS_QP_EVT_21 . . . . .	99
5.6.2.4 SRS_QP_EVT_22 . . . . .	99
5.6.2.5 SRS_QP_EVT_23 . . . . .	100
5.6.2.6 SRS_QP_EVT_30 . . . . .	100
5.6.2.7 SRS_QP_EVT_31 . . . . .	100
5.6.2.8 SRS_QP_EVT_40 . . . . .	101
5.6.2.9 SRS_QP_EVT_41 . . . . .	101
5.7 State Machines . . . . .	102
5.7.1 Concepts & Definitions . . . . .	102
5.7.1.1 State . . . . .	102
5.7.1.2 Transition . . . . .	102
5.7.1.3 State Machine . . . . .	102
5.7.1.4 Hierarchical State Machine . . . . .	102
5.7.1.5 State Machine Implementation Strategy . . . . .	102
5.7.1.6 Initializing a State Machine . . . . .	103
5.7.1.7 Dispatching Events to a State Machine . . . . .	103
5.7.2 Requirements . . . . .	104
5.7.2.1 SRS_QP_SM_00 . . . . .	104
5.7.2.2 SRS_QP_SM_01 . . . . .	105
5.7.2.3 SRS_QP_SM_10 . . . . .	105
5.7.2.4 SRS_QP_SM_20 . . . . .	106
5.7.2.5 SRS_QP_SM_21 . . . . .	106
5.7.2.6 SRS_QP_SM_22 . . . . .	107
5.7.2.7 SRS_QP_SM_23 . . . . .	107
5.7.2.8 SRS_QP_SM_24 . . . . .	108
5.7.2.9 SRS_QP_SM_25 . . . . .	108
5.7.2.10 SRS_QP_SM_30 . . . . .	109
5.7.2.11 SRS_QP_SM_31 . . . . .	110
5.7.2.12 SRS_QP_SM_32 . . . . .	110
5.7.2.13 SRS_QP_SM_33 . . . . .	111
5.7.2.14 SRS_QP_SM_34 . . . . .	111
5.7.2.15 SRS_QP_SM_35 . . . . .	112
5.7.2.16 SRS_QP_SM_36 . . . . .	114
5.7.2.17 SRS_QP_SM_37 . . . . .	114
5.7.2.18 SRS_QP_SM_38 . . . . .	115

---

5.7.2.19 SRS_QP_SM_39 . . . . .	116
5.7.2.20 SRS_QP_SM_40 . . . . .	116
5.8 Event Delivery Mechanisms . . . . .	117
5.8.1 Concepts & Definitions . . . . .	117
5.8.1.1 Direct Event Posting . . . . .	117
5.8.1.2 Publish-Subscribe . . . . .	118
5.8.1.3 Event Delivery Guarantee . . . . .	118
5.8.1.4 Best-Effort Event Delivery . . . . .	118
5.8.2 Requirements . . . . .	119
5.8.2.1 SRS_QP_EDM_00 . . . . .	119
5.8.2.2 SRS_QP_EDM_01 . . . . .	119
5.8.2.3 SRS_QP_EDG_10 . . . . .	120
5.8.2.4 SRS_QP_EDM_50 . . . . .	120
5.8.2.5 SRS_QP_EDM_51 . . . . .	120
5.8.2.6 SRS_QP_EDM_52 . . . . .	121
5.8.2.7 SRS_QP_EDM_53 . . . . .	121
5.8.2.8 SRS_QP_EDM_54 . . . . .	122
5.8.2.9 SRS_QP_EDM_55 . . . . .	122
5.8.2.10 SRS_QA_EDM_60 . . . . .	122
5.8.2.11 SRS_QP_EDM_61 . . . . .	123
5.8.2.12 SRS_QP_EDM_62 . . . . .	123
5.8.2.13 SRS_QP_EDM_64 . . . . .	124
5.8.2.14 SRS_QP_EDM_65 . . . . .	124
5.8.2.15 SRS_QP_EDM_66 . . . . .	124
5.8.2.16 SRS_QP_EDM_80 . . . . .	125
5.8.2.17 SRS_QP_EDM_81 . . . . .	125
5.9 Event Memory Management . . . . .	126
5.9.1 Concepts & Definitions . . . . .	126
5.9.1.1 Immutable Events . . . . .	126
5.9.1.2 Mutable Events . . . . .	126
5.9.1.3 Automatic Event Recycling . . . . .	126
5.9.1.4 Zero-Copy Event Management . . . . .	126
5.9.1.5 Event Pools . . . . .	127
5.9.2 Requirements . . . . .	127
5.9.2.1 SRS_QP_EMM_00 . . . . .	127
5.9.2.2 SRS_QP_EMM_10 . . . . .	127
5.9.2.3 SRS_QP_EMM_11 . . . . .	128
5.9.2.4 SRS_QP_EMM_20 . . . . .	128
5.9.2.5 SRS_QP_EMM_30 . . . . .	128

---

5.9.2.6 SRS_QP_EMM_40 . . . . .	129
5.10 Time Management . . . . .	130
5.10.1 Concepts & Definitions . . . . .	130
5.10.1.1 Time Events . . . . .	130
5.10.1.2 System Clock Tick . . . . .	130
5.10.1.3 Power Efficiency . . . . .	131
5.10.1.4 "Tickless Mode" . . . . .	131
5.10.1.5 Multiple Tick Rates . . . . .	131
5.10.2 Requirements . . . . .	132
5.10.2.1 SRS_QP_TM_00 . . . . .	132
5.10.2.2 SRS_QP_TM_10 . . . . .	132
5.10.2.3 SRS_QP_TM_11 . . . . .	133
5.10.2.4 SRS_QP_TM_20 . . . . .	133
5.10.2.5 SRS_QP_TM_21 . . . . .	133
5.10.2.6 SRS_QP_TM_22 . . . . .	134
5.10.2.7 SRS_QP_TM_23 . . . . .	134
5.10.2.8 SRS_QP_TM_30 . . . . .	134
5.10.2.9 SRS_QP_TM_40 . . . . .	135
5.11 Software Tracing . . . . .	136
5.11.1 Concepts & Definitions . . . . .	136
5.11.1.1 What is Software Tracing? . . . . .	136
5.11.1.2 Software Tracing & Active Objects . . . . .	137
5.11.1.3 QP/Spy Software Tracing System . . . . .	137
5.11.1.4 Data Protocol . . . . .	138
5.11.1.5 Run-time Filtering . . . . .	138
5.11.1.6 Predefined Trace Records . . . . .	138
5.11.1.7 Application-Specific Trace Records . . . . .	138
5.11.1.8 QS Dictionaries . . . . .	139
5.11.1.9 QS-RX Receive Channel . . . . .	139
5.11.1.10 Reentrancy . . . . .	139
5.11.2 Requirements . . . . .	139
5.11.2.1 SRS_QP_QS_00 . . . . .	139
5.11.2.2 SRS_QP_QS_01 . . . . .	139
5.11.2.3 SRS_QP_QS_10 . . . . .	140
5.11.2.4 SRS_QP_QS_11 . . . . .	140
5.11.2.5 SRS_QP_QS_20 . . . . .	141
5.11.2.6 SRS_QP_QS_21 . . . . .	141
5.11.2.7 SRS_QP_QS_30 . . . . .	142
5.11.2.8 SRS_QP_QS_31 . . . . .	142

---

5.11.2.9 SRS_QP_QS_40 . . . . .	142
5.11.2.10 SRS_QP_QS_50 . . . . .	143
5.12 Non-Preemptive Kernel . . . . .	144
5.12.1 Concepts & Definitions . . . . .	144
5.12.1.1 QV Non-Preemptive Kernel . . . . .	144
5.12.1.2 Sharing Resources in QV . . . . .	144
5.12.1.3 Idle Processing in QV . . . . .	145
5.12.1.4 Task-Level Response in QV . . . . .	145
5.12.2 Requirements . . . . .	146
5.12.2.1 SRS_QP_QV_00 . . . . .	146
5.12.2.2 SRS_QP_QV_10 . . . . .	146
5.12.2.3 SRS_QP_QV_11 . . . . .	146
5.12.2.4 SRS_QP_QV_12 . . . . .	147
5.12.2.5 SRS_QP_QV_20 . . . . .	147
5.12.2.6 SRS_QP_QV_21 . . . . .	147
5.13 Preemptive Non-Blocking Kernel . . . . .	149
5.13.1 Concepts & Definitions . . . . .	149
5.13.1.1 QK Preemptive Non-Blocking Kernel . . . . .	149
5.13.1.2 Preemptions in QK . . . . .	149
5.13.1.3 Single-Stack Kernel . . . . .	151
5.13.1.4 Idle Processing in QK . . . . .	151
5.13.1.5 Selective Scheduler Locking . . . . .	151
5.13.1.6 Preemption-Threshold Scheduling . . . . .	152
5.13.1.7 Task-Level Response . . . . .	152
5.13.2 Requirements . . . . .	152
5.13.2.1 SRS_QP_QK_00 . . . . .	152
5.13.2.2 SRS_QP_QK_10 . . . . .	153
5.13.2.3 SRS_QP_QK_20 . . . . .	153
5.13.2.4 SRS_QP_QK_21 . . . . .	153
5.13.2.5 SRS_QP_QK_30 . . . . .	154
5.13.2.6 SRS_QP_QK_31 . . . . .	154
5.14 Preemptive Dual-Mode Kernel . . . . .	155
5.14.1 Concepts & Definitions . . . . .	155
5.14.1.1 QXK Dual-Mode Kernel . . . . .	155
5.14.1.2 Basic Tasks . . . . .	155
5.14.1.3 Extended tasks . . . . .	155
5.14.1.4 QXK Blocking Mechanisms . . . . .	155
5.14.1.5 Selective Scheduler Locking . . . . .	156
5.14.1.6 Task-Level Response . . . . .	156

---

5.14.2 Requirements . . . . .	156
5.14.2.1 SRS_QP_QXK_00 . . . . .	156
5.14.2.2 SRS_QP_QXK_10 . . . . .	157
5.14.2.3 SRS_QP_QXK_11 . . . . .	157
5.14.2.4 SRS_QP_QXK_12 . . . . .	157
5.14.2.5 SRS_QP_QXK_13 . . . . .	158
5.14.2.6 SRS_QP_QXK_20 . . . . .	158
5.14.2.7 SRS_QP_QXK_21 . . . . .	158
5.14.2.8 SRS_QP_QXK_22 . . . . .	159
5.15 Non-Functional Requirements . . . . .	160
5.15.1 Standards Compliance . . . . .	160
5.15.1.1 SRS_QP_NF_01 . . . . .	160
5.15.1.2 SRS_QP_NF_02 . . . . .	160
5.15.1.3 SRS_QP_NF_03 . . . . .	160
5.15.1.4 SRS_QP_NF_04 . . . . .	161
5.15.2 Determinism . . . . .	161
5.15.2.1 SRS_QP_NF_20 . . . . .	161
5.15.2.2 SRS_QP_NF_21 . . . . .	162
5.15.3 Portability . . . . .	162
5.15.3.1 SRS_QP_NF_40 . . . . .	162
5.15.3.2 SRS_QP_NF_41 . . . . .	162
5.15.4 Ease of Development . . . . .	163
5.15.4.1 SRS_QP_NF_50 . . . . .	163
5.15.5 Constraints . . . . .	163
5.15.5.1 SRS_QP_NF_10 . . . . .	163
5.15.5.2 SRS_QP_NF_11 . . . . .	163
5.15.5.3 SRS_QP_NF_12 . . . . .	164
5.15.5.4 SRS_QP_NF_13 . . . . .	164
5.15.5.5 SRS_QP_NF_14 . . . . .	164
<b>6 Software Architecture Specification</b> . . . . .	<b>165</b>
6.1 About this Document . . . . .	165
6.1.1 DOC_SAS_QP . . . . .	165
6.2 Document Conventions . . . . .	166
6.2.1 Architecture Specification UIDs . . . . .	166
6.2.2 UML Semantics . . . . .	167
6.3 References . . . . .	167
6.4 Technology Viewpoint . . . . .	168
6.4.1 Views . . . . .	168

---

---

6.4.1.1 SAS_QP_OO . . . . .	168
6.4.1.2 SAS_QP_EDA . . . . .	168
6.4.1.3 SAS_QP_AF . . . . .	169
6.5 Context Viewpoint . . . . .	170
6.5.1 Layer View . . . . .	170
6.5.1.1 SAS_QP_APP . . . . .	171
6.5.1.2 SAS_QP_FRM . . . . .	171
6.5.1.3 SAS_QP_OSAL . . . . .	172
6.5.1.4 SAS_QP_OS . . . . .	173
6.5.2 Interface View . . . . .	173
6.5.2.1 SAS_QP_CLS . . . . .	173
6.5.2.2 SAS_QP_API . . . . .	174
6.5.2.3 SAS_OSAL_API . . . . .	175
6.5.2.4 SAS_OS_API . . . . .	175
6.6 Resource Viewpoint . . . . .	175
6.6.1 Views . . . . .	175
6.6.1.1 SAS_QP_MEM . . . . .	175
6.6.1.2 SAS_QP_EMM . . . . .	177
<b>7 Software Design Specification</b>	<b>179</b>
7.1 About This Document . . . . .	179
7.1.1 DOC_SDS_QP . . . . .	179
7.2 Document Conventions . . . . .	180
7.2.1 Software-Design-Specification UIDs . . . . .	180
7.3 References . . . . .	181
7.4 Structure Viewpoint . . . . .	182
7.4.1 Sub-Layer View . . . . .	182
7.4.1.1 SDS_QP_QEP . . . . .	183
7.4.1.2 SDS_QP_QF . . . . .	183
7.4.2 Class View . . . . .	184
7.4.2.1 SDS_QP_QEvt . . . . .	185
7.4.2.2 SDS_QP_QAsm . . . . .	186
7.4.2.3 SDS_QP_QHsm . . . . .	186
7.4.2.4 SDS_QP_QMsm . . . . .	188
7.4.2.5 SDS_QP_QActive . . . . .	189
7.4.2.6 SDS_QP_QMctive . . . . .	191
7.4.2.7 SDS_QP_QTimeEvt . . . . .	191
7.5 Interaction Viewpoint . . . . .	193
7.5.1 Behavior View . . . . .	193

---

7.5.1.1 SDS_QA_START . . . . .	193
7.5.2 Event Exchange View . . . . .	195
7.5.2.1 SDS_QP_POST . . . . .	195
7.5.2.2 SDS_QP_PUB . . . . .	197
7.5.3 Mutable Event Life Cycle . . . . .	199
7.5.3.1 SDS_QP_MELC . . . . .	200
7.6 State Dynamics Viewpoint . . . . .	202
7.6.1 QHsm Design View . . . . .	203
7.6.1.1 SDS_QA_QHsm_decl . . . . .	203
7.6.1.2 SDS_QA_QHsm_top_init . . . . .	204
7.6.1.3 SDS_QA_QHsm_state . . . . .	205
7.6.1.4 SDS_QA_QHsm_entry . . . . .	206
7.6.1.5 SDS_QA_QHsm_exit . . . . .	207
7.6.1.6 SDS_QA_QHsm_nest_init . . . . .	208
7.6.1.7 SDS_QA_QHsm_tran . . . . .	209
7.6.1.8 SDS_QA_QHsm_intern . . . . .	209
7.6.1.9 SDS_QA_QHsm_choice . . . . .	210
7.6.1.10 SDS_QA_QHsm_hist . . . . .	211
7.6.1.11 SDS_QA_QHsm_hist_tran . . . . .	212
7.6.2 QMsm Design View . . . . .	212
7.6.2.1 SDS_QA_QMsm_decl . . . . .	212
7.6.2.2 SDS_QA_QMsm_top_init . . . . .	214
7.6.2.3 SDS_QA_QMsm_state . . . . .	216
7.6.2.4 SDS_QA_QMsm_entry . . . . .	217
7.6.2.5 SDS_QA_QMsm_exit . . . . .	218
7.6.2.6 SDS_QA_QMsm_nest_init . . . . .	218
7.6.2.7 SDS_QA_QMsm_tran . . . . .	219
7.6.2.8 SDS_QA_QMsm_intern . . . . .	220
7.6.2.9 SDS_QA_QMsm_choice . . . . .	220
7.6.2.10 SDS_QA_QMsm_hist . . . . .	221
7.6.2.11 SDS_QA_QMsm_hist_tran . . . . .	222
7.7 Time Viewpoint . . . . .	224
7.7.1 Time Event Life Cycle . . . . .	224
7.7.1.1 SDS_QP_TELC . . . . .	224
7.8 Algorithm Viewpoint . . . . .	226
7.8.1 QHsm Implementation View . . . . .	226
7.8.1.1 SDS_QP_QHsm_ctor . . . . .	226
7.8.1.2 SDS_QP_QHsm_init . . . . .	226
7.8.1.3 SDS_QP_QHsm_dispatch . . . . .	227

---

7.8.1.4 SDS_QP_QHsm_tran-simple . . . . .	227
7.8.1.5 SDS_QP_QHsm_tran-complex . . . . .	227
7.8.2 QMsm Implementation View . . . . .	227
7.8.2.1 SDS_QP_QMsm_ctor . . . . .	227
7.8.2.2 SDS_QP_QMsm_init . . . . .	228
7.8.2.3 SDS_QP_QMsm_disp . . . . .	228
7.8.2.4 SDS_QP_QMsm_tran . . . . .	228
7.8.2.5 SDS_QP_QMsm_tat . . . . .	228
7.9 Interface Viewpoint . . . . .	229
7.9.1 Critical Section . . . . .	229
7.9.1.1 SDS_QP_CRIT . . . . .	229
7.9.2 Active Object OSAL . . . . .	230
<b>8 API Reference</b>	<b>231</b>
8.1 Compile-Time Configuration . . . . .	231
8.2 Active Objects . . . . .	231
8.3 Events . . . . .	232
8.4 State Machines . . . . .	232
8.5 Mutable Event Management . . . . .	232
8.6 Time Management . . . . .	233
8.7 Event Queues (raw thread-safe) . . . . .	233
8.8 Software Tracing . . . . .	233
8.8.1 QS-Transmit (QS-TX) . . . . .	233
8.8.2 QS Receive-Channel (QS-RX) . . . . .	234
8.9 QV (Non-preemptive Kernel) . . . . .	234
8.9.1 Kernel Initialization and Control . . . . .	235
8.10 QK (Preemptive RTC Kernel) . . . . .	235
8.10.1 Kernel Initialization and Control . . . . .	235
8.11 QXK (Dual-Mode Kernel) . . . . .	235
8.11.1 Kernel Initialization and Control . . . . .	235
<b>9 Deprecated APIs</b>	<b>237</b>
<b>10 Revision History</b>	<b>239</b>
10.1 Version 8.0.3, 2025-04-07 . . . . .	239
10.2 Version 8.0.2, 2025-01-20 . . . . .	239
10.3 Version 8.0.1, 2024-12-17 . . . . .	240
10.4 Version 8.0.0, 2024-10-31 . . . . .	241
10.5 Version 7.3.4, 2024-03-21 . . . . .	243
10.6 Version 7.3.3, 2024-03-01 . . . . .	243

---

10.7 Version 7.3.2, 2023-12-13 . . . . .	244
10.8 Version 7.3.1, 2023-12-05 . . . . .	245
10.9 Version 7.3.0, 2023-09-12 . . . . .	246
10.10 Version 7.2.1, 2023-01-15 . . . . .	248
10.11 Version 7.2.0, 2023-01-06 . . . . .	248
10.12 Version 7.1.3, 2022-11-18 . . . . .	250
10.13 Version 7.1.2, 2022-10-07 . . . . .	251
10.14 Version 7.1.0, 2022-08-30 . . . . .	251
10.15 Version 7.0.2, 2022-08-12 . . . . .	252
10.16 Version 7.0.1, 2022-07-31 . . . . .	253
10.17 Version 7.0.0, 2022-04-30 . . . . .	254
10.18 Version 6.9.3, 2021-04-12 . . . . .	255
10.19 Version 6.9.2, 2021-01-18 . . . . .	256
10.20 Version 6.9.1, 2020-09-28 . . . . .	257
10.21 Version 6.9.0, 2020-08-21 . . . . .	259
10.22 Version 6.8.2, 2020-07-17 . . . . .	260
10.23 Version 6.8.1, 2020-04-04 . . . . .	261
10.24 Version 6.8.0, 2020-03-21 . . . . .	262
10.25 Version 6.7.0, 2019-12-30 . . . . .	262
10.26 Version 6.6.0, 2019-10-31 . . . . .	263
10.27 Version 6.5.1, 2019-05-24 . . . . .	263
10.28 Version 6.5.0, 2019-03-30 . . . . .	264
10.29 Version 6.4.0, 2019-02-10 . . . . .	264
10.30 Version 6.3.8, 2018-12-31 . . . . .	264
10.31 Version 6.3.7, 2018-11-20 . . . . .	265
10.32 Version 6.3.6, 2018-10-20 . . . . .	265
10.33 Version 6.3.4, 2018-08-10 . . . . .	266
10.34 Version 6.3.3a, 2018-07-16 . . . . .	266
10.35 Version 6.3.3, 2018-06-22 . . . . .	266
10.36 Version 6.3.2, 2018-06-20 . . . . .	266
10.37 Version 6.3.1, 2018-05-24 . . . . .	267
10.38 Version 6.3.0, 2018-05-10 . . . . .	267
10.39 Version 6.2.0, 2018-03-16 . . . . .	267
10.40 Version 6.1.1, 2018-02-18 . . . . .	268
10.41 Version 6.1.0, 2018-02-04 . . . . .	268
10.42 Version 6.0.4, 2018-01-10 . . . . .	269
10.43 Version 6.0.3, 2017-12-12 . . . . .	269
10.44 Version 6.0.1, 2017-11-10 . . . . .	270
10.45 Version 6.0.0, 2017-10-13 . . . . .	270

---

10.46 Version 5.9.9, 2017-09-29 . . . . .	270
10.47 Version 5.9.8, 2017-09-15 . . . . .	271
10.48 Version 5.9.7, 2017-08-18 . . . . .	271
10.49 Version 5.9.6, 2017-08-04 . . . . .	272
10.50 Version 5.9.5, 2017-07-20 . . . . .	272
10.51 Version 5.9.4, 2017-07-07 . . . . .	273
10.52 Version 5.9.3, 2017-06-19 . . . . .	273
10.53 Version 5.9.2, 2017-06-05 . . . . .	273
10.54 Version 5.9.1, 2017-05-26 . . . . .	273
10.55 Version 5.9.0, 2017-05-19 . . . . .	274
10.56 Version 5.8.2, 2017-02-08 . . . . .	275
10.57 Version 5.8.1, 2016-12-16 . . . . .	275
10.58 Version 5.8.0, 2016-11-30 . . . . .	276
10.59 Version 5.7.4, 2016-11-04 . . . . .	276
10.60 Version 5.7.3, 2016-10-07 . . . . .	276
10.61 Version 5.7.2, 2016-09-30 . . . . .	277
10.62 Version 5.7.0, 2016-08-31 . . . . .	277
10.63 Version 5.6.5, 2016-06-06 . . . . .	277
10.64 Version 5.6.4, 2016-04-25 . . . . .	278
10.65 Version 5.6.3, 2016-04-12 . . . . .	278
10.66 Version 5.6.2, 2016-03-31 . . . . .	278
10.67 Version 5.6.1, 2016-01-01 . . . . .	279
10.68 Version 5.6.0-beta, 2015-12-24 . . . . .	279
10.69 Version 5.5.1, 2015-10-05 . . . . .	280
10.70 Version 5.5.0, 2015-09-04 . . . . .	281
10.71 Version 5.4.2, 2015-06-04 . . . . .	282
10.72 Version 5.4.1, 2015-05-14 . . . . .	282
10.73 Version 5.4.0, 2015-04-26 . . . . .	283
10.74 Version 5.3.1, 2014-09-19 . . . . .	285
10.75 Version 5.3.0, 2014-03-31 . . . . .	285
10.76 Version 5.2.1, 2014-01-06 . . . . .	286
10.77 Version 5.2.0, 2013-12-26 . . . . .	287
10.78 Version 5.1.1, 2013-10-10 . . . . .	288
10.79 Version 5.1.0, 2013-09-23 . . . . .	288
10.80 Version 5.0.0, 2013-09-10 . . . . .	289
10.81 Version 4.5.04, Feb 08, 2013 . . . . .	292
10.82 Version 4.5.03, Nov 27, 2012 . . . . .	292
10.83 Version 4.5.02, Aug 04, 2012 . . . . .	292
10.84 Version 4.5.01, Jun 14, 2012 . . . . .	293

---

10.85 Version 4.5.00, May 29, 2012 . . . . .	294
10.86 Version 4.4.01, Mar 23, 2012 . . . . .	295
10.87 Version 4.4.00, Jan 30, 2012 . . . . .	295
10.88 Version 4.3.00, Nov 01, 2011 . . . . .	296
10.89 Version 4.2.04, Sep 24, 2011 . . . . .	297
10.90 Version 4.2.02, Sep 08, 2011 . . . . .	297
10.91 Version 4.2.01, Aug 13, 2011 . . . . .	297
10.92 Version 4.2.00, Jul 14, 2011 . . . . .	297
10.93 Version 4.1.07, Feb 28, 2011 . . . . .	298
10.94 Version 4.1.06, Jan 07, 2011 . . . . .	299
10.95 Version 4.1.05, Nov 01, 2010 . . . . .	299
10.96 Version 4.1.04, Mar 16, 2010 . . . . .	299
10.97 Version 4.1.03, Jan 21, 2010 . . . . .	299
10.98 Version 4.1.02, Jan 14, 2010 . . . . .	300
10.99 Version 4.1.01, Nov 05, 2009 . . . . .	300
10.100 Version 4.1.00, Oct 09, 2009 . . . . .	300
10.101 Version 4.0.04, Apr 09, 2009 . . . . .	301
10.102 Version 4.0.03, Dec 27, 2008 . . . . .	301
10.103 Version 4.0.02, Nov 15, 2008 . . . . .	302
10.104 Version 4.0.01, June 09, 2008 . . . . .	302
10.105 Version 4.0.00, Apr 07, 2008 . . . . .	302
<b>11 Hierarchical Index</b>	<b>305</b>
11.1 Class Hierarchy . . . . .	305
<b>12 Class Index</b>	<b>307</b>
12.1 Class List . . . . .	307
<b>13 File Index</b>	<b>309</b>
13.1 File List . . . . .	309
<b>14 Class Documentation</b>	<b>311</b>
14.1 QActive Class Reference . . . . .	311
14.1.1 Detailed Description . . . . .	313
14.1.2 Member Function Documentation . . . . .	314
14.1.2.1 QActive_ctor() . . . . .	314
14.1.2.2 QActive_setAttr() . . . . .	314
14.1.2.3 QActive_start() . . . . .	314
14.1.2.4 QActive_stop() . . . . .	315
14.1.2.5 QActive_register_() . . . . .	316
14.1.2.6 QActive_unregister_() . . . . .	316

---

---

14.1.2.7 QActive_post_()	317
14.1.2.8 QActive_postLIFO_()	318
14.1.2.9 QActive_get_()	319
14.1.2.10 QActive_psInit()	319
14.1.2.11 QActive_publish_()	320
14.1.2.12 QActive_getQueueMin()	320
14.1.2.13 QActive_subscribe()	321
14.1.2.14 QActive_unsubscribe()	322
14.1.2.15 QActive_unsubscribeAll()	323
14.1.2.16 QActive_defer()	323
14.1.2.17 QActive_recall()	324
14.1.2.18 QActive_flushDeferred()	324
14.1.2.19 QActive_evtLoop_()	325
14.1.2.20 QActive_postFIFO_()	325
14.1.3 Member Data Documentation	325
14.1.3.1 super	325
14.1.3.2 prio	325
14.1.3.3 pthre	325
14.1.3.4 thread	326
14.1.3.5 osObject	326
14.1.3.6 eQueue	326
14.1.3.7 QActive_registry_	326
14.1.3.8 QActive_subscrList_	326
14.1.3.9 QActive_maxPubSignal	327
14.2 QActiveDummy Class Reference	327
14.2.1 Detailed Description	327
14.3 QAsm Class Reference	327
14.3.1 Detailed Description	328
14.3.2 Member Function Documentation	328
14.3.2.1 QAsm_ctor()	328
14.3.3 Member Data Documentation	328
14.3.3.1 vptr	328
14.3.3.2 state	328
14.3.3.3 temp	329
14.4 QAsmAttr Union Reference	329
14.4.1 Detailed Description	329
14.4.2 Member Data Documentation	329
14.4.2.1 fun	329
14.4.2.2 act	329

---

14.4.2.3 thr . . . . .	329
14.4.2.4 tatbl . . . . .	330
14.4.2.5 obj . . . . .	330
14.5 QAsmVtable Struct Reference . . . . .	330
14.5.1 Detailed Description . . . . .	330
14.5.2 Member Data Documentation . . . . .	330
14.5.2.1 init . . . . .	330
14.5.2.2 dispatch . . . . .	330
14.5.2.3 isIn . . . . .	330
14.5.2.4 getStateHandler . . . . .	331
14.6 QEQueue Class Reference . . . . .	331
14.6.1 Detailed Description . . . . .	332
14.6.2 Member Function Documentation . . . . .	332
14.6.2.1 QEQueue_init() . . . . .	332
14.6.2.2 QEQueue_post() . . . . .	333
14.6.2.3 QEQueue_postLIFO() . . . . .	333
14.6.2.4 QEQueue_get() . . . . .	334
14.6.2.5 QEQueue_getNFree() . . . . .	335
14.6.2.6 QEQueue_getNMin() . . . . .	335
14.6.2.7 QEQueue_isEmpty() . . . . .	335
14.6.3 Member Data Documentation . . . . .	336
14.6.3.1 frontEvt . . . . .	336
14.6.3.2 ring . . . . .	336
14.6.3.3 end . . . . .	336
14.6.3.4 head . . . . .	336
14.6.3.5 tail . . . . .	336
14.6.3.6 nFree . . . . .	336
14.6.3.7 nMin . . . . .	336
14.7 QEvt Class Reference . . . . .	337
14.7.1 Detailed Description . . . . .	337
14.7.2 Member Function Documentation . . . . .	338
14.7.2.1 QEvt_ctor() . . . . .	338
14.7.2.2 QEvt_init() . . . . .	339
14.7.2.3 QEvt_refCtr_inc_() . . . . .	340
14.7.2.4 QEvt_refCtr_dec_() . . . . .	340
14.7.3 Member Data Documentation . . . . .	340
14.7.3.1 sig . . . . .	340
14.7.3.2 poolNum_ . . . . .	340
14.7.3.3 refCtr_ . . . . .	341

---

---

14.8 QF Class Reference . . . . .	341
14.8.1 Detailed Description . . . . .	342
14.8.2 Member Function Documentation . . . . .	342
14.8.2.1 QF_init() . . . . .	342
14.8.2.2 QF_stop() . . . . .	342
14.8.2.3 QF_run() . . . . .	343
14.8.2.4 QF_onStartup() . . . . .	343
14.8.2.5 QF_onCleanup() . . . . .	343
14.8.2.6 QF_onContextSw() . . . . .	344
14.8.2.7 QF_poolInit() . . . . .	345
14.8.2.8 QF_poolGetMaxBlockSize() . . . . .	345
14.8.2.9 QF_getPoolMin() . . . . .	346
14.8.2.10 QF_newX_() . . . . .	346
14.8.2.11 QF_gc() . . . . .	347
14.8.2.12 QF_newRef_() . . . . .	347
14.8.2.13 QF_deleteRef_() . . . . .	348
14.8.2.14 QF_gcFromISR() . . . . .	348
14.8.2.15 QF_bzero_() . . . . .	349
14.8.2.16 QF_pslnit() . . . . .	349
14.8.3 Member Data Documentation . . . . .	349
14.8.3.1 QF_priv_ . . . . .	349
14.9 QF_Attr Class Reference . . . . .	349
14.9.1 Detailed Description . . . . .	349
14.9.2 Member Data Documentation . . . . .	349
14.9.2.1 ePool_ . . . . .	349
14.9.2.2 maxPool_ . . . . .	350
14.10 QFreeBlock Struct Reference . . . . .	350
14.10.1 Detailed Description . . . . .	350
14.11 QHsm Class Reference . . . . .	350
14.11.1 Detailed Description . . . . .	351
14.11.2 Member Function Documentation . . . . .	352
14.11.2.1 QHsm_ctor() . . . . .	352
14.11.2.2 QHsm_init_() . . . . .	352
14.11.2.3 QHsm_dispatch_() . . . . .	353
14.11.2.4 QHsm_isIn_() . . . . .	354
14.11.2.5 QHsm_getStateHandler_() . . . . .	354
14.11.2.6 QHsm_top() . . . . .	355
14.11.2.7 QHsm_state() . . . . .	355
14.11.2.8 QHsm_childState() . . . . .	355

---

14.11.2.9 QHsm_tran_simple_()	356
14.11.2.10 QHsm_tran_complex_()	356
14.11.2.11 QHsm_enter_target_()	356
14.11.3 Member Data Documentation	356
14.11.3.1 super	356
14.12 QHsmDummy Class Reference	356
14.12.1 Detailed Description	356
14.13 QK Class Reference	357
14.13.1 Detailed Description	357
14.13.2 Member Function Documentation	357
14.13.2.1 QK_sched_()	357
14.13.2.2 QK_sched_act_()	357
14.13.2.3 QK_activate_()	358
14.13.2.4 QK_schedLock()	358
14.13.2.5 QK_schedUnlock()	358
14.13.2.6 QK_onIdle()	359
14.13.3 Member Data Documentation	359
14.13.3.1 QK_priv_	359
14.14 QK_Attr Class Reference	359
14.14.1 Detailed Description	360
14.14.2 Member Data Documentation	360
14.14.2.1 readySet	360
14.14.2.2 actPrio	360
14.14.2.3 nextPrio	360
14.14.2.4 actThre	360
14.14.2.5 lockCeil	360
14.14.2.6 intNest	360
14.15 QMActive Class Reference	360
14.15.1 Detailed Description	362
14.15.2 Member Function Documentation	363
14.15.2.1 QMActive_ctor()	363
14.15.3 Member Data Documentation	363
14.15.3.1 super	363
14.16 QMPool Class Reference	363
14.16.1 Detailed Description	364
14.16.2 Member Function Documentation	364
14.16.2.1 QMPool_init()	364
14.16.2.2 QMPool_get()	366
14.16.2.3 QMPool_put()	367

---

14.16.3 Member Data Documentation . . . . .	368
14.16.3.1 start . . . . .	368
14.16.3.2 end . . . . .	368
14.16.3.3 freeHead . . . . .	368
14.16.3.4 blockSize . . . . .	368
14.16.3.5 nTot . . . . .	368
14.16.3.6 nFree . . . . .	368
14.16.3.7 nMin . . . . .	368
14.17 QMsm Class Reference . . . . .	368
14.17.1 Detailed Description . . . . .	370
14.17.2 Member Function Documentation . . . . .	370
14.17.2.1 QMsm_ctor() . . . . .	370
14.17.2.2 QMsm_init_() . . . . .	371
14.17.2.3 QMsm_dispatch_() . . . . .	372
14.17.2.4 QMsm_isIn_() . . . . .	372
14.17.2.5 QMsm_getStateHandler_() . . . . .	373
14.17.2.6 QMsm_stateObj() . . . . .	373
14.17.2.7 QMsm_childStateObj() . . . . .	374
14.17.2.8 QMsm_execTatbl_() . . . . .	374
14.17.2.9 QMsm_exitToTranSource_() . . . . .	375
14.17.2.10 QMsm_enterHistory_() . . . . .	376
14.17.3 Member Data Documentation . . . . .	376
14.17.3.1 super . . . . .	376
14.18 QMState Struct Reference . . . . .	376
14.18.1 Detailed Description . . . . .	377
14.18.2 Member Data Documentation . . . . .	377
14.18.2.1 superstate . . . . .	377
14.18.2.2 stateHandler . . . . .	377
14.18.2.3 entryAction . . . . .	377
14.18.2.4 exitAction . . . . .	377
14.18.2.5 initAction . . . . .	377
14.19 QMTranActTable Struct Reference . . . . .	377
14.19.1 Detailed Description . . . . .	377
14.19.2 Member Data Documentation . . . . .	378
14.19.2.1 target . . . . .	378
14.19.2.2 act . . . . .	378
14.20 QPSet Class Reference . . . . .	378
14.20.1 Detailed Description . . . . .	378
14.20.2 Member Function Documentation . . . . .	379

---

14.20.2.1 QPSet_setEmpty() . . . . .	379
14.20.2.2 QPSet_isEmpty() . . . . .	379
14.20.2.3 QPSet_notEmpty() . . . . .	379
14.20.2.4 QPSet_hasElement() . . . . .	379
14.20.2.5 QPSet_insert() . . . . .	379
14.20.2.6 QPSet_remove() . . . . .	379
14.20.2.7 QPSet_findMax() . . . . .	379
14.20.3 Member Data Documentation . . . . .	380
14.20.3.1 bits . . . . .	380
14.21 QS Class Reference . . . . .	380
14.21.1 Detailed Description . . . . .	380
14.22 QS_Filter Struct Reference . . . . .	380
14.22.1 Detailed Description . . . . .	380
14.23 QS_TProbe Struct Reference . . . . .	380
14.23.1 Detailed Description . . . . .	381
14.23.2 Member Data Documentation . . . . .	381
14.23.2.1 addr . . . . .	381
14.23.2.2 data . . . . .	381
14.23.2.3 idx . . . . .	381
14.24 QSpyId Struct Reference . . . . .	381
14.24.1 Detailed Description . . . . .	381
14.25 QSubscrList Struct Reference . . . . .	381
14.25.1 Detailed Description . . . . .	381
14.25.2 Member Data Documentation . . . . .	382
14.25.2.1 set . . . . .	382
14.26 QTicker Class Reference . . . . .	382
14.26.1 Detailed Description . . . . .	384
14.26.2 Member Function Documentation . . . . .	385
14.26.2.1 QTicker_ctor() . . . . .	385
14.26.2.2 QTicker_init_() . . . . .	385
14.26.2.3 QTicker_dispatch_() . . . . .	385
14.26.2.4 QTicker_trig_() . . . . .	385
14.26.3 Member Data Documentation . . . . .	385
14.26.3.1 super . . . . .	385
14.27 QTimeEvt Class Reference . . . . .	385
14.27.1 Detailed Description . . . . .	387
14.27.2 Member Function Documentation . . . . .	388
14.27.2.1 QTimeEvt_ctorX() . . . . .	388
14.27.2.2 QTimeEvt_armX() . . . . .	388

---

14.27.2.3 QTimeEvt_disarm()	389
14.27.2.4 QTimeEvt_rearm()	390
14.27.2.5 QTimeEvt_wasDisarmed()	391
14.27.2.6 QTimeEvt_currCtr()	391
14.27.2.7 QTimeEvt_init()	392
14.27.2.8 QTimeEvt_tick_()	392
14.27.2.9 QTimeEvt_expire_()	392
14.27.2.10 QTimeEvt_tick1_()	392
14.27.2.11 QTimeEvt_noActive()	392
14.27.3 Member Data Documentation	393
14.27.3.1 super	393
14.27.3.2 next	393
14.27.3.3 act	393
14.27.3.4 ctr	393
14.27.3.5 interval	394
14.27.3.6 tickRate	394
14.27.3.7 flags	394
14.27.3.8 QTimeEvt_timeEvtHead_	394
14.28 QV Class Reference	394
14.28.1 Detailed Description	394
14.28.2 Member Function Documentation	395
14.28.2.1 QV_schedDisable()	395
14.28.2.2 QV_schedEnable()	395
14.28.2.3 QV_onIdle()	396
14.28.3 Member Data Documentation	396
14.28.3.1 QV_priv_	396
14.29 QV_Attr Class Reference	396
14.29.1 Detailed Description	396
14.29.2 Member Data Documentation	396
14.29.2.1 readySet	396
14.29.2.2 schedCeil	397
14.30 QXK Class Reference	397
14.30.1 Detailed Description	397
14.31 QXK_Attr Class Reference	397
14.31.1 Detailed Description	397
14.32 QXMutex Class Reference	397
14.32.1 Detailed Description	397
14.33 QXSemaphore Class Reference	398
14.33.1 Detailed Description	398

14.34 QXThread Class Reference . . . . .	399
14.34.1 Detailed Description . . . . .	399
<b>15 File Documentation</b>	<b>401</b>
15.1 history.dox File Reference . . . . .	401
15.2 cmd_options.dox File Reference . . . . .	401
15.2.1 Detailed Description . . . . .	401
15.2.2 Macro Definition Documentation . . . . .	401
15.2.2.1 QP_CONFIG . . . . .	401
15.2.2.2 Q_SPY . . . . .	401
15.2.2.3 Q_UTEST . . . . .	402
15.3 qqueue.dox File Reference . . . . .	402
15.3.1 Macro Definition Documentation . . . . .	402
15.3.1.1 QF_EQUEUE_CTR_SIZE . . . . .	402
15.3.2 Typedef Documentation . . . . .	402
15.3.2.1 QEQueueCtr . . . . .	402
15.4 qk.dox File Reference . . . . .	403
15.5 qmpool.dox File Reference . . . . .	403
15.6 qp.dox File Reference . . . . .	403
15.7 qp_config.h File Reference . . . . .	403
15.7.1 Detailed Description . . . . .	404
15.7.2 Macro Definition Documentation . . . . .	405
15.7.2.1 QP_API_VERSION . . . . .	405
15.7.2.2 Q_UNSAFE . . . . .	406
15.7.2.3 Q_SIGNAL_SIZE . . . . .	406
15.7.2.4 QF_MAX_ACTIVE . . . . .	406
15.7.2.5 QF_MAX_EPOOL . . . . .	407
15.7.2.6 QF_MAX_TICK_RATE . . . . .	407
15.7.2.7 QEVT_PAR_INIT . . . . .	407
15.7.2.8 QACTIVE_CAN_STOP . . . . .	407
15.7.2.9 QF_EVENT_SIZ_SIZE . . . . .	408
15.7.2.10 QF_TIMEEVT_CTR_SIZE . . . . .	408
15.7.2.11 QF_EQUEUE_CTR_SIZE . . . . .	408
15.7.2.12 QF_MPOOL_CTR_SIZE . . . . .	408
15.7.2.13 QF_MPOOL_SIZ_SIZE . . . . .	409
15.7.2.14 QS_TIME_SIZE . . . . .	409
15.7.2.15 QS_CTR_SIZE . . . . .	409
15.7.2.16 QF_ON_CONTEXT_SW . . . . .	409
15.7.2.17 QF_MEM_ISOLATE . . . . .	410

---

15.7.2.18 QK_USE_IRQ_NUM . . . . .	410
15.7.2.19 QK_USE_IRQ_HANDLER . . . . .	410
15.7.2.20 QXK_USE_IRQ_NUM . . . . .	410
15.7.2.21 QXK_USE_IRQ_HANDLER . . . . .	411
15.8 qp_pkg.dox File Reference . . . . .	411
15.9 qp_port.h File Reference . . . . .	411
15.9.1 Detailed Description . . . . .	412
15.9.2 Macro Definition Documentation . . . . .	412
15.9.2.1 Q_NORETURN . . . . .	412
15.9.2.2 QACTIVE_EQUEUE_TYPE . . . . .	412
15.9.2.3 QACTIVE_OS_OBJ_TYPE . . . . .	412
15.9.2.4 QACTIVE_THREAD_TYPE . . . . .	412
15.9.2.5 QF_INT_DISABLE . . . . .	413
15.9.2.6 QF_INT_ENABLE . . . . .	413
15.9.2.7 QF_CRIT_STAT . . . . .	413
15.9.2.8 QF_CRIT_ENTRY . . . . .	413
15.9.2.9 QF_CRIT_EXIT . . . . .	413
15.9.2.10 QV_CPU_SLEEP . . . . .	413
15.9.2.11 QK_ISR_CONTEXT_ . . . . .	414
15.9.2.12 QK_ISR_ENTRY . . . . .	414
15.9.2.13 QK_ISR_EXIT . . . . .	414
15.9.2.14 QXK_ISR_CONTEXT_ . . . . .	414
15.9.2.15 QXK_CONTEXT_SWITCH_ . . . . .	414
15.9.2.16 QXK_ISR_ENTRY . . . . .	414
15.9.2.17 QXK_ISR_EXIT . . . . .	414
15.9.3 Typedef Documentation . . . . .	415
15.9.3.1 crit_stat_t . . . . .	415
15.9.4 Function Documentation . . . . .	415
15.9.4.1 critEntry() . . . . .	415
15.9.4.2 critExit() . . . . .	415
15.10 qs.dox File Reference . . . . .	415
15.11 qs_pkg.dox File Reference . . . . .	415
15.12 qs_port.h File Reference . . . . .	415
15.12.1 Detailed Description . . . . .	415
15.12.2 Macro Definition Documentation . . . . .	416
15.12.2.1 QS_OBJ_PTR_SIZE . . . . .	416
15.12.2.2 QS_FUN_PTR_SIZE . . . . .	416
15.13 qsafe.dox File Reference . . . . .	417
15.14 qv.dox File Reference . . . . .	417

---

15.15 qxk.dox File Reference . . . . .	417
15.16 sas-qp.dox File Reference . . . . .	417
15.17 sds-qp.dox File Reference . . . . .	417
15.18 srs-qp.dox File Reference . . . . .	417
15.19 api.dox File Reference . . . . .	417
15.20 dir.dox File Reference . . . . .	417
15.21 help.dox File Reference . . . . .	417
15.22 qp-exa.dox File Reference . . . . .	417
15.23 qp-gs.dox File Reference . . . . .	417
15.24 qp-main.dox File Reference . . . . .	417
15.25 qp-ports.dox File Reference . . . . .	417
15.26 queue.h File Reference . . . . .	417
15.26.1 Detailed Description . . . . .	417
15.26.2 Typedef Documentation . . . . .	418
15.26.2.1 QEQueueCtr . . . . .	418
15.27 qk.h File Reference . . . . .	418
15.27.1 Detailed Description . . . . .	418
15.27.2 Macro Definition Documentation . . . . .	418
15.27.2.1 QF_SCHED_STAT_ . . . . .	418
15.27.2.2 QF_SCHED_LOCK_ . . . . .	419
15.27.2.3 QF_SCHED_UNLOCK_ . . . . .	419
15.27.2.4 QACTIVE_EQUEUE_WAIT_ . . . . .	419
15.27.2.5 QACTIVE_EQUEUE_SIGNAL_ . . . . .	419
15.27.2.6 QF_EPOOL_TYPE_ . . . . .	419
15.27.2.7 QF_EPOOL_INIT_ . . . . .	419
15.27.2.8 QF_EPOOL_EVENT_SIZE_ . . . . .	419
15.27.2.9 QF_EPOOL_GET_ . . . . .	420
15.27.2.10 QF_EPOOL_PUT_ . . . . .	420
15.27.3 Typedef Documentation . . . . .	420
15.27.3.1 QSchedStatus . . . . .	420
15.28 qmpool.h File Reference . . . . .	420
15.28.1 Detailed Description . . . . .	420
15.28.2 Macro Definition Documentation . . . . .	421
15.28.2.1 QF_MPOOL_EL . . . . .	421
15.28.3 Typedef Documentation . . . . .	421
15.28.3.1 QMPoolSize . . . . .	421
15.28.3.2 QMPoolCtr . . . . .	422
15.29 qp.h File Reference . . . . .	422
15.29.1 Detailed Description . . . . .	426

---

15.29.2 Macro Definition Documentation . . . . .	426
15.29.2.1 QP_VERSION_STR . . . . .	426
15.29.2.2 QP_VERSION . . . . .	426
15.29.2.3 QP_RELEASE . . . . .	426
15.29.2.4 Q_UNUSED_PAR . . . . .	426
15.29.2.5 Q_DIM . . . . .	427
15.29.2.6 Q_UINT2PTR_CAST . . . . .	427
15.29.2.7 QEVT_INITIALIZER . . . . .	427
15.29.2.8 QEVT_DYNAMIC . . . . .	427
15.29.2.9 Q_EVT_CAST . . . . .	428
15.29.2.10 Q_STATE_CAST . . . . .	428
15.29.2.11 Q_ACTION_CAST . . . . .	429
15.29.2.12 Q_ACTION_NULL . . . . .	429
15.29.2.13 Q_RET_SUPER . . . . .	429
15.29.2.14 Q_RET_UNHANDLED . . . . .	429
15.29.2.15 Q_RET_HANDLED . . . . .	429
15.29.2.16 Q_RET_IGNORED . . . . .	429
15.29.2.17 Q_RET_ENTRY . . . . .	429
15.29.2.18 Q_RET_EXIT . . . . .	429
15.29.2.19 Q_RET_NULL . . . . .	429
15.29.2.20 Q_RET_TRAN . . . . .	429
15.29.2.21 Q_RET_TRAN_INIT . . . . .	430
15.29.2.22 Q_RET_TRAN_HIST . . . . .	430
15.29.2.23 Q_EMPTY_SIG . . . . .	430
15.29.2.24 Q_ENTRY_SIG . . . . .	430
15.29.2.25 Q_EXIT_SIG . . . . .	430
15.29.2.26 Q_INIT_SIG . . . . .	430
15.29.2.27 Q_USER_SIG . . . . .	430
15.29.2.28 QASM_INIT . . . . .	430
15.29.2.29 QASM_DISPATCH . . . . .	431
15.29.2.30 QASM_IS_IN . . . . .	432
15.29.2.31 Q_ASM_UPCAST . . . . .	433
15.29.2.32 Q_HSM_UPCAST . . . . .	433
15.29.2.33 Q_TRAN . . . . .	434
15.29.2.34 Q_TRAN_HIST . . . . .	434
15.29.2.35 Q_SUPER . . . . .	435
15.29.2.36 Q_HANDLED . . . . .	436
15.29.2.37 Q_UNHANDLED . . . . .	436
15.29.2.38 Q_MSM_UPCAST . . . . .	436

---

15.29.2.39 QM_ENTRY . . . . .	437
15.29.2.40 QM_EXIT . . . . .	437
15.29.2.41 QM_TRAN . . . . .	437
15.29.2.42 QM_TRAN_INIT . . . . .	437
15.29.2.43 QM_TRAN_HIST . . . . .	438
15.29.2.44 QM_HANDLED . . . . .	438
15.29.2.45 QM_UNHANDLED . . . . .	438
15.29.2.46 QM_SUPER . . . . .	438
15.29.2.47 QM_STATE_NULL . . . . .	439
15.29.2.48 Q_PRIO . . . . .	439
15.29.2.49 QF_NO_MARGIN . . . . .	439
15.29.2.50 Q_NEW . . . . .	439
15.29.2.51 Q_NEW_X . . . . .	440
15.29.2.52 Q_NEW_REF . . . . .	442
15.29.2.53 Q_DELETE_REF . . . . .	442
15.29.2.54 QACTIVE_POST . . . . .	443
15.29.2.55 QACTIVE_POST_X . . . . .	444
15.29.2.56 QACTIVE_PUBLISH . . . . .	445
15.29.2.57 QTIMEEV_TICK_X . . . . .	445
15.29.2.58 QTICKER_TRIG . . . . .	446
15.29.2.59 QACTIVE_POST_LIFO . . . . .	446
15.29.2.60 QTIMEEV_TICK . . . . .	447
15.29.2.61 QF_CRIT_EXIT_NOP . . . . .	447
15.29.3 Typedef Documentation . . . . .	447
15.29.3.1 int_t . . . . .	447
15.29.3.2 enum_t . . . . .	447
15.29.3.3 QSignal . . . . .	447
15.29.3.4 QEvtPtr . . . . .	448
15.29.3.5 QState . . . . .	448
15.29.3.6 QStateHandler . . . . .	448
15.29.3.7 QActionHandler . . . . .	448
15.29.3.8 QXThreadHandler . . . . .	448
15.29.3.9 QPrioSpec . . . . .	448
15.29.3.10 QTimeEvtCtr . . . . .	449
15.29.3.11 QPSetBits . . . . .	449
15.29.4 Function Documentation . . . . .	449
15.29.4.1 QF_LOG2() . . . . .	449
15.29.5 Variable Documentation . . . . .	449
15.29.5.1 QP_versionStr . . . . .	449

---

---

15.30 qp_pkg.h File Reference . . . . .	449
15.30.1 Detailed Description . . . . .	450
15.30.2 Macro Definition Documentation . . . . .	450
15.30.2.1 QACTIVE_CAST_ . . . . .	450
15.30.2.2 Q_PTR2UINT_CAST_ . . . . .	450
15.30.2.3 QTE_FLAG_IS_LINKED . . . . .	450
15.30.2.4 QTE_FLAG_WAS_DISARMED . . . . .	450
15.31 qpc.h File Reference . . . . .	451
15.31.1 Detailed Description . . . . .	451
15.31.2 Macro Definition Documentation . . . . .	451
15.31.2.1 QP_API_VERSION . . . . .	451
15.31.2.2 QM_SUPER_SUB . . . . .	451
15.31.2.3 QM_TRAN_EP . . . . .	452
15.31.2.4 QM_TRAN_XP . . . . .	452
15.31.2.5 QACTIVE_START . . . . .	452
15.31.2.6 QXTHREAD_START . . . . .	452
15.31.2.7 Q_onAssert . . . . .	452
15.31.2.8 Q_ALLEGEE_ID . . . . .	453
15.31.2.9 Q_ALLEGEE . . . . .	453
15.31.2.10 Q_ASSERT_COMPILE . . . . .	453
15.31.2.11 QHSM_INIT . . . . .	453
15.31.2.12 QHSM_DISPATCH . . . . .	454
15.31.2.13 QHsm_isIn . . . . .	454
15.31.2.14 QF_PUBLISH . . . . .	454
15.31.2.15 QF_TICK_X . . . . .	454
15.31.2.16 QF_TICK . . . . .	454
15.31.2.17 QF_getQueueMin . . . . .	454
15.31.3 Typedef Documentation . . . . .	455
15.31.3.1 char_t . . . . .	455
15.32 qs_dummy.h File Reference . . . . .	455
15.32.1 Detailed Description . . . . .	457
15.32.2 Macro Definition Documentation . . . . .	457
15.32.2.1 QS_INIT . . . . .	457
15.32.2.2 QS_EXIT . . . . .	457
15.32.2.3 QS_DUMP . . . . .	457
15.32.2.4 QS_GLB_FILTER . . . . .	457
15.32.2.5 QS_LOC_FILTER . . . . .	457
15.32.2.6 QS_BEGIN_ID . . . . .	457
15.32.2.7 QS_END . . . . .	457

---

15.32.2.8 QS_BEGIN_INCRIT . . . . .	458
15.32.2.9 QS_END_INCRIT . . . . .	458
15.32.2.10 QS_I8 . . . . .	458
15.32.2.11 QS_U8 . . . . .	458
15.32.2.12 QS_I16 . . . . .	458
15.32.2.13 QS_U16 . . . . .	458
15.32.2.14 QS_I32 . . . . .	458
15.32.2.15 QS_U32 . . . . .	458
15.32.2.16 QS_F32 . . . . .	459
15.32.2.17 QS_F64 . . . . .	459
15.32.2.18 QS_I64 . . . . .	459
15.32.2.19 QS_U64 . . . . .	459
15.32.2.20 QS_ENUM . . . . .	459
15.32.2.21 QS_STR . . . . .	459
15.32.2.22 QS_MEM . . . . .	459
15.32.2.23 QS_SIG . . . . .	460
15.32.2.24 QS_OBJ . . . . .	460
15.32.2.25 QS_FUN . . . . .	460
15.32.2.26 QS_SIG_DICTIONARY . . . . .	460
15.32.2.27 QS_OBJ_DICTIONARY . . . . .	460
15.32.2.28 QS_OBJ_ARR_DICTIONARY . . . . .	460
15.32.2.29 QS_FUN_DICTIONARY . . . . .	460
15.32.2.30 QS_USR_DICTIONARY . . . . .	460
15.32.2.31 QS_ENUM_DICTIONARY . . . . .	461
15.32.2.32 QS_ASSERTION . . . . .	461
15.32.2.33 QS_FLUSH . . . . .	461
15.32.2.34 QS_TEST_PROBE_DEF . . . . .	461
15.32.2.35 QS_TEST_PROBE . . . . .	461
15.32.2.36 QS_TEST_PROBE_ID . . . . .	461
15.32.2.37 QS_TEST_PAUSE . . . . .	461
15.32.2.38 QS_OUTPUT . . . . .	462
15.32.2.39 QS_RX_INPUT . . . . .	462
15.32.2.40 QS_RX_PUT . . . . .	462
15.32.2.41 QS_ONLY . . . . .	462
15.32.2.42 QS_BEGIN_PRE . . . . .	462
15.32.2.43 QS_END_PRE . . . . .	462
15.32.2.44 QS_U8_PRE . . . . .	462
15.32.2.45 QS_2U8_PRE . . . . .	462
15.32.2.46 QS_U16_PRE . . . . .	463

15.32.2.47 QS_U32_PRE . . . . .	463
15.32.2.48 QS_TIME_PRE . . . . .	463
15.32.2.49 QS_SIG_PRE . . . . .	463
15.32.2.50 QS_EVS_PRE . . . . .	463
15.32.2.51 QS_OBJ_PRE . . . . .	463
15.32.2.52 QS_FUN_PRE . . . . .	463
15.32.2.53 QS_EQC_PRE . . . . .	463
15.32.2.54 QS_MPC_PRE . . . . .	464
15.32.2.55 QS MPS_PRE . . . . .	464
15.32.2.56 QS_TEC_PRE . . . . .	464
15.32.2.57 QS_CRIT_STAT . . . . .	464
15.32.2.58 QS_CRIT_ENTRY . . . . .	464
15.32.2.59 QS_CRIT_EXIT . . . . .	464
15.32.2.60 QS_TR_CRIT_ENTRY . . . . .	464
15.32.2.61 QS_TR_CRIT_EXIT . . . . .	464
15.32.2.62 QS_TR_ISR_ENTRY . . . . .	464
15.32.2.63 QS_TR_ISR_EXIT . . . . .	465
15.32.3 Typedef Documentation . . . . .	465
15.32.3.1 QSTimeCtr . . . . .	465
15.32.4 Function Documentation . . . . .	465
15.32.4.1 QS_initBuf() . . . . .	465
15.32.4.2 QS_getByte() . . . . .	466
15.32.4.3 QS_getBlock() . . . . .	466
15.32.4.4 QS_doOutput() . . . . .	467
15.32.4.5 QS_onStartup() . . . . .	467
15.32.4.6 QS_onCleanup() . . . . .	467
15.32.4.7 QS_onFlush() . . . . .	467
15.32.4.8 QS_onGetTime() . . . . .	467
15.32.4.9 QS_rxInitBuf() . . . . .	468
15.32.4.10 QS_rxParse() . . . . .	469
15.32.4.11 QS_onTestSetup() . . . . .	469
15.32.4.12 QS_onTestTeardown() . . . . .	469
15.32.4.13 QS_onTestEvt() . . . . .	469
15.32.4.14 QS_onTestPost() . . . . .	469
15.32.4.15 QS_onTestLoop() . . . . .	469
15.33 qsafe.h File Reference . . . . .	469
15.33.1 Detailed Description . . . . .	470
15.33.2 Macro Definition Documentation . . . . .	471
15.33.2.1 QF_CRIT_STAT . . . . .	471

15.33.2.2 QF_CRIT_ENTRY . . . . .	471
15.33.2.3 QF_CRIT_EXIT . . . . .	471
15.33.2.4 Q_ASSERT_INCRIT . . . . .	471
15.33.2.5 Q_ERROR_INCRIT . . . . .	472
15.33.2.6 Q_ASSERT_ID . . . . .	472
15.33.2.7 Q_ERROR_ID . . . . .	473
15.33.2.8 Q_ASSERT . . . . .	474
15.33.2.9 Q_ERROR . . . . .	474
15.33.2.10 Q_REQUIRE_ID . . . . .	474
15.33.2.11 Q_REQUIRE . . . . .	475
15.33.2.12 Q_REQUIRE_INCRIT . . . . .	475
15.33.2.13 Q_ENSURE_ID . . . . .	476
15.33.2.14 Q_ENSURE . . . . .	476
15.33.2.15 Q_ENSURE_INCRIT . . . . .	477
15.33.2.16 Q_INVARIANT_ID . . . . .	477
15.33.2.17 Q_INVARIANT . . . . .	478
15.33.2.18 Q_INVARIANT_INCRIT . . . . .	478
15.33.2.19 Q_ASSERT_STATIC . . . . .	478
15.33.2.20 Q_NORETURN . . . . .	479
15.33.2.21 Q_DIM . . . . .	479
15.33.3 Function Documentation . . . . .	479
15.33.3.1 Q_onError() . . . . .	479
15.34 qstamp.h File Reference . . . . .	480
15.34.1 Detailed Description . . . . .	480
15.34.2 Variable Documentation . . . . .	480
15.34.2.1 Q_BUILD_DATE . . . . .	480
15.34.2.2 Q_BUILD_TIME . . . . .	480
15.35 qv.h File Reference . . . . .	480
15.35.1 Detailed Description . . . . .	481
15.35.2 Macro Definition Documentation . . . . .	481
15.35.2.1 QF_SCHED_STAT_ . . . . .	481
15.35.2.2 QF_SCHED_LOCK_ . . . . .	481
15.35.2.3 QF_SCHED_UNLOCK_ . . . . .	481
15.35.2.4 QACTIVE_EQUEUE_WAIT_ . . . . .	481
15.35.2.5 QACTIVE_EQUEUE_SIGNAL_ . . . . .	481
15.35.2.6 QF_EPOOL_TYPE_ . . . . .	481
15.35.2.7 QF_EPOOL_INIT_ . . . . .	482
15.35.2.8 QF_EPOOL_EVENT_SIZE_ . . . . .	482
15.35.2.9 QF_EPOOL_GET_ . . . . .	482

---

15.35.2.10 QF_EPOOL_PUT_ . . . . .	482
15.36 qep_hsm.c File Reference . . . . .	482
15.36.1 Detailed Description . . . . .	482
15.36.2 Macro Definition Documentation . . . . .	482
15.36.2.1 QHSM_MAX_NEST_DEPTH_ . . . . .	482
15.37 qep msm.c File Reference . . . . .	483
15.37.1 Detailed Description . . . . .	483
15.37.2 Macro Definition Documentation . . . . .	483
15.37.2.1 QMSM_MAX_ENTRY_DEPTH_ . . . . .	483
15.38 qf_act.c File Reference . . . . .	483
15.38.1 Detailed Description . . . . .	483
15.38.2 Function Documentation . . . . .	484
15.38.2.1 QF_LOG2() . . . . .	484
15.38.3 Variable Documentation . . . . .	484
15.38.3.1 QP_versionStr . . . . .	484
15.38.3.2 QF_priv_ . . . . .	484
15.39 qf_actq.c File Reference . . . . .	484
15.39.1 Detailed Description . . . . .	484
15.40 qf_defer.c File Reference . . . . .	484
15.40.1 Detailed Description . . . . .	485
15.41 qf_dyn.c File Reference . . . . .	485
15.41.1 Detailed Description . . . . .	485
15.42 qf_mem.c File Reference . . . . .	485
15.42.1 Detailed Description . . . . .	485
15.43 qf_ps.c File Reference . . . . .	486
15.43.1 Detailed Description . . . . .	486
15.44 qf_qact.c File Reference . . . . .	486
15.44.1 Detailed Description . . . . .	486
15.45 qf_qeq.c File Reference . . . . .	486
15.45.1 Detailed Description . . . . .	487
15.46 qf_qmact.c File Reference . . . . .	487
15.46.1 Detailed Description . . . . .	487
15.47 qf_time.c File Reference . . . . .	487
15.47.1 Detailed Description . . . . .	487
15.48 qk.c File Reference . . . . .	487
15.48.1 Detailed Description . . . . .	487
15.49 qstamp.c File Reference . . . . .	487
15.49.1 Detailed Description . . . . .	488
15.49.2 Variable Documentation . . . . .	488

---

15.49.2.1 Q_BUILD_DATE . . . . .	488
15.49.2.2 Q_BUILD_TIME . . . . .	488
15.50 qv.c File Reference . . . . .	488
15.50.1 Detailed Description . . . . .	488
<b>Index</b>	<b>489</b>



# Chapter 1

## Overview



- To check what's new in QP/C, please see [QP/C Revision History](#).
- The recommended way of obtaining QP/C is by downloading the official [QP-bundle↑](#).
- The latest QP/C code, with the most recent enhancements and bug fixes, is available in the [GitHub QP/C repository↑](#).

### 1.1 What is it?

QP/C real-time event framework (RTEF) is a lightweight implementation of the asynchronous, event-driven Active Object (a.k.a. Actor) model of computation specifically designed for real-time embedded systems, such as microcontrollers (MCUs). QP/C is both a *software infrastructure* for building applications consisting of Active Objects (Actors) and a *runtime environment* for executing the Active Objects in a deterministic, real-time fashion. Additionally, QP/C Framework supports **Hierarchical State Machines** with which to specify the behavior of Active Objects [UML 2.5], [Sutter:10], [ROOM:94]. The QP/C Framework can be viewed as a modern, asynchronous, and truly event driven real-time operating system.

## 1.2 What does it do?

The QP/C RTEF provides a reusable, event-driven software architecture, which combines the model of concurrency, known as [Active Objects](#) (Actors) with [Hierarchical State Machines](#). The QP/C RTEF provides the following benefits:

- Modern, event-driven, asynchronous, and non-blocking architecture based on the best practices of concurrent programming collectively known as the [Active Object](#) (a.k.a. Actor) model of computation;
- *Extensible and responsive* architecture (easier to analyze for real-time) by avoiding hard-coded blocking calls;
- *Inherently safer* concurrency architecture than the traditional "shared-state concurrency" approach of a traditional RTOS by replacing direct resource sharing with event exchanges.
- *Higher-level of abstraction* closer to the problem domain than the "naked" RTOS threads
- The *right abstractions* for applying modern techniques like hierarchical state machines, visual modeling, and automatic code generation (see [QM Graphical Modeling Tool](#)<sup>↑</sup>);
- Efficient, readable implementation of [Hierarchical State Machines](#)<sup>↑</sup> for specifying the internal behavior of Active Objects;
- Built-in, configurable and flexible [Software Tracing](#) for troubleshooting (debugging), testing, monitoring, and optimizing embedded applications with minimal impact on the real-time performance.

## 1.3 What's special about it?

QP/C offers numerous advantages over the traditional "shared state concurrency" based on a conventional RTOS.

### 1.3.1 Functional Safety

All QP editions are a natural fit for safety-related applications because they implement a number of best practices highly recommended by the functional safety standards, such as strictly modular design (Active Objects) or hierarchical state machines (semi-formal methods). Indeed, for decades the QP/C and QP/C++ Frameworks have been [widely used in safety-related applications](#)<sup>↑</sup>, such as medical, aerospace, and industrial.

### 1.3.2 QP Framework Editions

QP real-time event frameworks form a family consisting of the following QP editions:

QP Edition	Programming Language	API Compatibility	Safety Functions	Certification Artifacts	Licensing
<i>Standard QP editions</i>					
QP/C	C (C11)	Same as SafeQP/C	Assertions	Requirements, Architecture & Design Specifications	Open-source & Commercial ( <a href="#">dual licensing</a> ) <sup>↑</sup>

QP Edition	Programming Language	API Compatibility	Safety Functions	Certification Artifacts	Licensing
QP/C++	C++ (C++17)	Same as SafeQP/C++	Assertions	Requirements, Architecture & Design Specifications	Open-source & Commercial (dual licensing)↑
<i>SafeQP editions engineered for functional safety</i>					
SafeQP/C	C (C11)	Same as QP/C	All identified Safety Functions	Complete Certification Kit	Commercial only↑
SafeQP/C++	C++ (C++17)	Same as QP/C++	All identified Safety Functions	Complete Certification Kit	Commercial only↑

#### Remarks

All QP editions are accompanied by the [Requirements Specification](#), [Architecture Specification](#), and [Design Specification](#), which are the *best source of information* about the underlying concepts, functionality, architecture, and design of the QP Frameworks and the QP Applications based on the frameworks.

#### 1.3.3 SafeQP Editions



The **SafeQP/C** and **SafeQP/C++** frameworks were originally derived from QP/C and QP/C++, respectively, but were extensively reengineered for the safety market using compliant Software Safety Lifecycle (SSL). In this process, the QP framework functional model has been subjected to a full Hazard and Risk Analysis, which identified all areas of weakness within the functional model and API. These findings led to creation of Safety Requirements and risk mitigation by *Safety Functions*, which were subsequently implemented, verified, and validated in the SafeQP editions.

#### Note

The **SafeQP editions** remain fully *API- and functionally compatible* with the corresponding standard QP frameworks. This ensures existing QP Applications can transition seamlessly to SafeQP without requiring any modifications. SafeQP retains QP Frameworks' hallmark features, including a small memory footprint, excellent efficiency, and hard real-time functionality.

#### 1.3.4 SafeQP Certification Kits

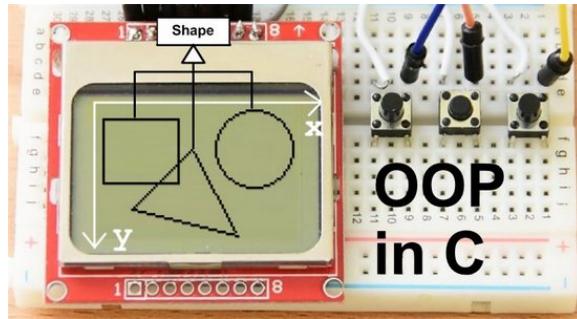
The SafeQP frameworks are accompanied by the **SafeQP Certification Kits**, which provide developers with ready-to-use artifacts, enabling them to *save time, mitigate risk, and reduce costs* during application certification for safety-critical devices in the industrial, medical, aerospace, and automotive industries.

#### 1.3.5 Object Orientation

Even though it is written in MISRA-compliant ISO-C11, QP/C is fundamentally an **object-oriented** framework, which means that the framework itself and your applications derived from the framework are fundamentally composed of [classes](#) and only classes can have [state machines](#) associated with them.

### Remarks

If you program in C and object-oriented programming is new to you, please refer to the article and set of videos "[Object-Oriented Programming](#)"↑, which among others describes how you can implement the concepts of *classes*, *inheritance*, and *polymorphism* to portable, standard C.



*Application Note: Object-Oriented Programming in C*

### 1.3.6 Hierarchical State Machines

The behavior of active objects is specified in QP/C by means of [hierarchical state machines \(UML statecharts\)](#)↑. The framework supports manual coding of UML state machines in C as well as fully **automatic code generation** by means of the free graphical [QM model-based design \(MBD\) tool](#)↑.

### 1.3.7 Built-in Kernels

The QP/C framework can run on [bare-metal single-chip microcontrollers](#), completely replacing a traditional RTOS. The framework contains a selection of built-in real-time kernels, such as the non-preemptive [QV kernel](#), the preemptive non-blocking [QK kernel](#), and the preemptive, dual-mode, blocking [QXK kernel](#). The [QXK kernel](#) provides all the features you might expect from a traditional [RTOS kernel](#) and has been specifically designed for mixing event-driven active objects with traditional blocking code, such as commercial middleware (TCP/IP stacks, UDP stacks, embedded file systems, etc.) or legacy software. [Native QP/C ports](#) and ready-to-use [examples](#) are provided for major embedded CPU families, such as ARM Cortex-M, ARM Cortex-R, and MSP430.

### 1.3.8 3rd-Party Kernels

QP/C can also work with many traditional [Real-Time Operating Systems \(RTOSes\)](#) and [General-Purpose OSes \(GPOSes\)](#) (such as Linux (POSIX) and Windows).

### 1.3.9 Size and Efficiency

Even though QP/C offers higher level of abstraction than a traditional RTOS, when combined with the native built-in kernels it typically outperforms equivalent traditional RTOS applications both in RAM/ROM footprint and in CPU efficiency. The specific measurements and results are reported in the [Application Note: "QP/C Performance Tests and Results"](#)↑:

### 1.3.10 Software Tracing

**Software tracing** is a method of capturing and recording information about the execution of a software program. Software tracing is particularly effective and powerful in combination with the event-driven Active Object model of computation. Due to the inversion of control, a running application built of Active Objects is a highly structured affair where all important system interactions funnel through the underlying event-driven framework. This arrangement offers a unique opportunity for applying Software Tracing in a framework like QP.

### 1.3.11 Documentation Traceability

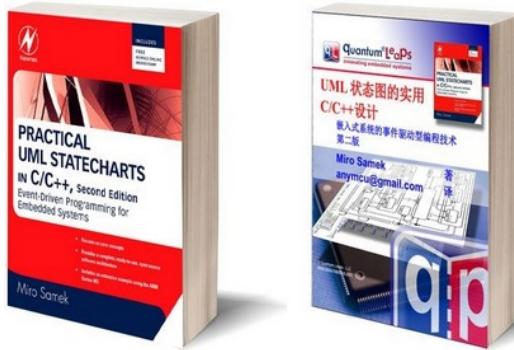
QP/C offers unprecedented, bidirectional traceability among all work artifacts, which gives teams full visibility from requirements through architecture, design, source code, tests, and back again.

### 1.3.12 Popularity & Maturity

With 20 years of continuous development, over [400 commercial licensees↑](#), and many times more open source users worldwide, QP Frameworks are the most popular such offering on the market. They power countless electronic products across a [wide variety of markets↑](#), such as medical, consumer, IoT, defense, robotics, industrial, communication, transportation, semiconductor IP, and many others.

### 1.3.13 Books

The two editions of the book, [Practical Statecharts in C/C++↑](#) provide a detailed design study of the QP/C and QP/C++ frameworks and explain the related concepts.



*Practical UML Statecharts in C/C++, 2nd Edition*



*Practical Statecharts in C/C++, 1st Edition*

## 1.4 How is it licensed?

The QP/C and QP/C++ frameworks are licensed under the [dual licensing model](#)↑, in which both the open source software distribution mechanism and traditional closed source software distribution models are combined.

### Note

If your company has a policy forbidding open source in your product, the QP/C and QP/C++ frameworks can be [licensed commercially](#)↑, in which case you don't use any open source license and you do not violate your policy.

### 1.4.1 Open Source Projects

If you are developing and distributing **open source** applications under the GNU General Public License (GPL), as published by the Free Software Foundation, then you are free to use the Quantum Leaps software under the [GPL version 3](#)↑ of the License, or (at your option) any later version.

### Note

Please note that GPL requires that all modifications to the original code as well as your application code (Derivative Works as defined in the Copyright Law) must also be released under the terms of the GPL open source license.

### 1.4.2 Closed Source Projects

If you are developing and distributing traditional **closed source** applications, you can purchase one of [Quantum Leaps commercial licenses](#)↑, which are specifically designed for users interested in retaining the proprietary status of their code. All Quantum Leaps commercial licenses expressly supersede the GPL open source license. This means that when you license Quantum Leaps software under a commercial license, you specifically do not use the software under the open source license and therefore you are not subject to any of its terms.

## 1.5 How to get help?

Please post any **technical questions** to the [Free Support Forum](#)↑ hosted on SourceForge.net. Posts to this forum benefit the whole community and are typically answered the same day.

Direct **Commercial Support** is available to the commercial licensees. Every commercial license includes one year of Technical Support for the licensed software. The support term can be extended annually.

Training and consulting services are also available from Quantum Leaps. Please refer to the [Contact web-page](#)↑ for more information.

### Note

The features of this online help and tips for using it are described in Section help.

## 1.6 Contact Information

- Quantum Leaps Web site: [state-machine.com](http://state-machine.com)↑
- Quantum Leaps licensing: [state-machine.com/licensing](http://state-machine.com/licensing)↑
- Quantum Leaps on GitHub: [github.com/QuantumLeaps](http://github.com/QuantumLeaps)↑
- e-mail: [info@state-machine.com](mailto:info@state-machine.com)↑

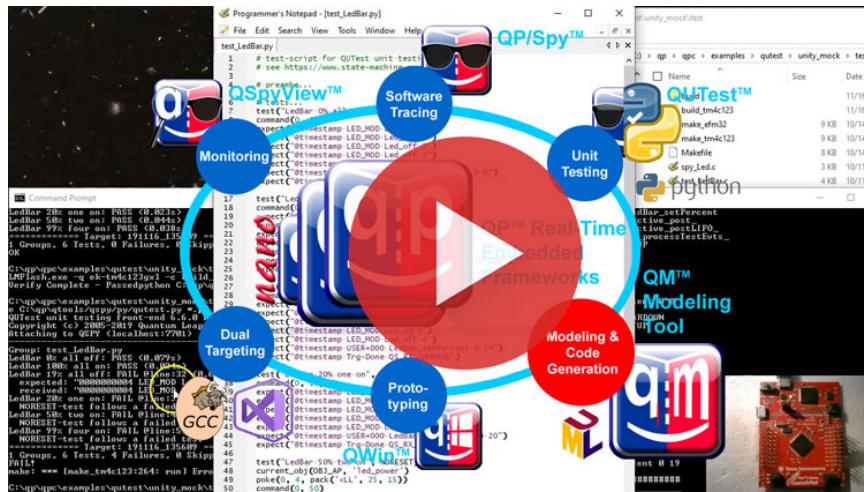
# Chapter 2

## Getting Started

The following sections describe how to get started with QP/C quickly:

- Downloading & Installing QP/C
- QP/C Tutorial

The YouTube Video [Getting Started with QP Frameworks](#)↑ provides instructions on how to download, install and get started with QP quickly.



Getting Started with QP Real-Time Event Framework



The QP Certification Kit is the best source of information about:

- Software Requirements Specification — QP/C **functionality**
- Software Architecture Specification — QP/C **architecture**
- Software Design Specification — QP/C **design**

## 2.1 Downloading & Installing QP/C

### 2.1.1 Downloading QP/C in QP-Bundle

The most recommended way of obtaining QP/C is by downloading the [QP-bundle↑](#), which includes QP/C as well as other QP frameworks and also the [QM modeling tool↑](#) and the [QTools collection↑](#). The main advantage of obtaining QP/C bundled together like that is that you get all components, tools and examples ready to go.



*QP-bundle downloads*

#### Note

If you are allergic to installers and GUIs or don't have administrator privileges you can also **download and install QP/C separately** from the [QP/C GitHub repository↑](#), as described in the next section.

### 2.1.2 Downloading QP/C from GitHub

Go to the [QP/C release page on GitHub↑](#), and choose the QP/C version number you wish to download. You should select the latest QP/C version, unless you have a very specific reason to go with an older release.

Asset	Description	Size	Last Updated
qpc_7.3.0.zip	[1]	21.7 MB	Sep 27
QuantumLeaps.qpc.7.3.0.pack	[2]	288 KB	6 hours ago
Source code (zip)			Sep 27
Source code (tar.gz)			Sep 27

*QP/C downloads from GitHub*

[1] QP/C framework ZIP archive (contains HTML documentation and 3rd-party code for the examples)

[2] QP/C framework CMSIS-Pack (see the [next section](#))

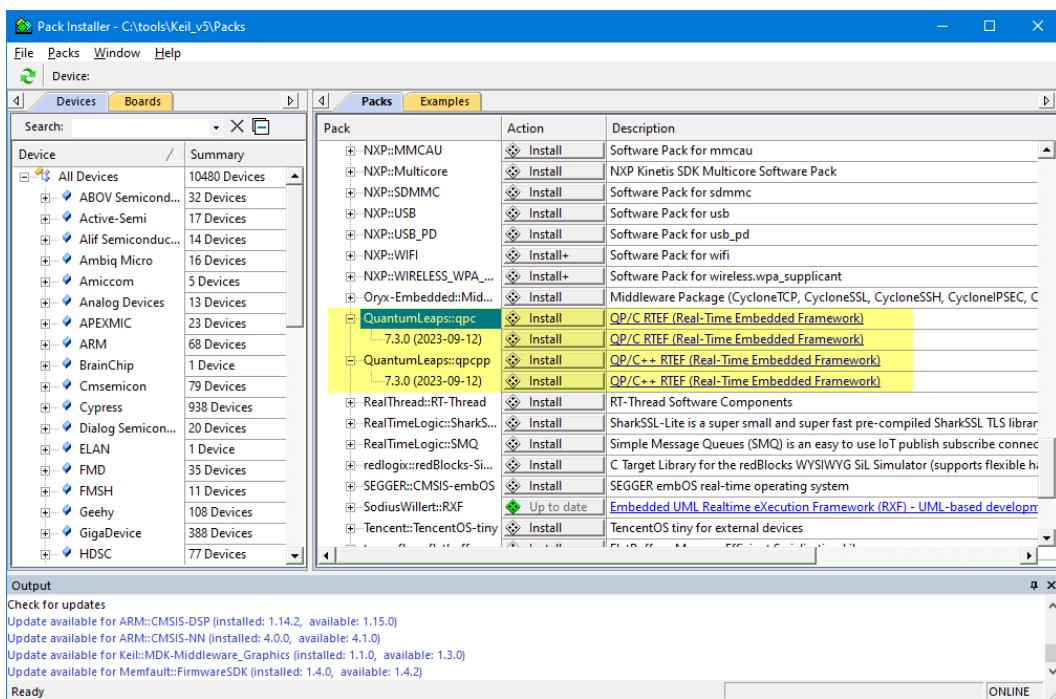
**Remarks**

The QP/C GitHub repository contains the `3rd_party` and `examples` folders as **git submodules**. Therefore, while cloning the `qpc` repository directly, it is recommended to do this as follows:

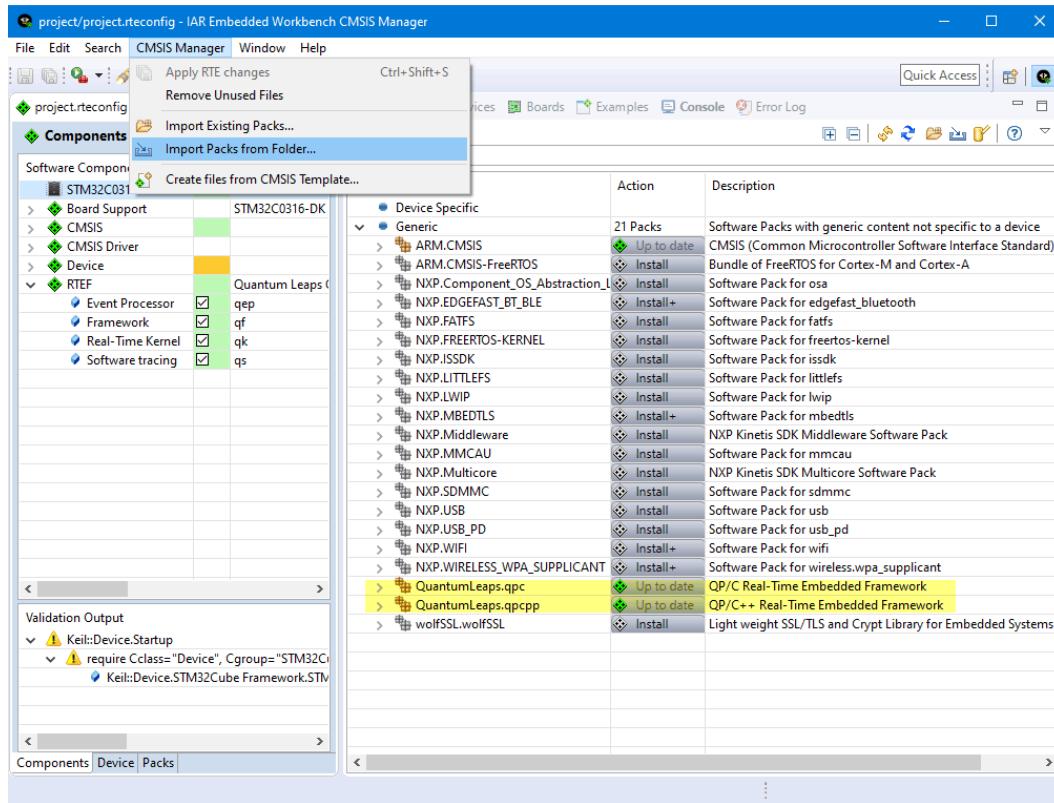
```
git clone https://github.com/QuantumLeaps/qpc --recurse-submodules --depth 1
```

### 2.1.3 Downloading QP/C as a CMSIS-Pack

The QP/C framework is available as a [CMSIS-Pack](#), which you can obtain either in the [KEIL Pack Installer](#) or directly from the QP/C release directory on GitHub (see [previous section](#)).



*QuantumLeaps qpc and qcpp packs in KEIL Pack Installer*



*QuantumLeaps qpc and qpcpp packs in IAR Pack Installer*

## 2.1.4 QP/C Installation Folder

The following annotated directory tree lists the top-level directories provided in the standard QP/C distribution.

```

qpC
+---include    // platform-independent QP/C header files
|
+---src        // platform-independent QP/C source files
|   +---qf      // QF active object framework source files
|   +---qv      // QV non-preemptive kernel files
|   +---qk      // QK preemptive, non-blocking kernel files
|   +---qxk     // QXK preemptive, non-blocking/blocking kernel files
|   +---qs      // QS software tracing (target-resident component)
|
+---ports      // platform-specific QP/C ports
|
..... the following folders can be deleted if not needed .....
|
+---examples   // QP/C examples
+---3rd_party  // 3rd-party code used in the examples

```

### Remarks

The [examples/](#) and [3rd\\_party](#) folders contain many [Example Projects](#), which are specifically designed to help you learn to use QP/C and to serve you as starting points for your own projects.

## 2.1.5 QP License Certificate File

Every QP/C commercial license comes with the **QP License Certificate file**, which stores the commercial license information (for the [QM modeling tool↑](#) and the [QSPY software tracing host utility↑](#)). The QP License Certificate file is always named exactly the same as the corresponding commercial license and has the extension `.qlc`. For example, a commercial license with the license number QPC-SP-230416A will be accompanied by the license file named `QPC-SP-230416A.qlc`.

### Note

If you are a commercial licensee and for some reason you did not receive your QP License Certificate file, please send email with your license number to [support@state-machine.com↑](mailto:support@state-machine.com) to request your QP License Certificate.

The QP License Certificate file is a plain-text file, which you can open in any editor and inspect (but you cannot change without corrupting it). Here is an example of a QP License Certificate file:

**File name:** QPC-SP-230416A.qlc

```
Bamboozle Technologies, Inc.  
qpc  
2025-08-17
```

```
Copyright (c) 2024-2025 Bamboozle Technologies, Inc. All rights reserved.
```

```
SPDX-License-Identifier: LicenseRef-QL-single-product
```

```
THIS SINGLE PRODUCT LICENSE IS VALID SOLELY FOR THE FOLLOWING PRODUCT:  
"eGizmo Shuffle 2nd Generation"  
#20EB458E46F910356133D2A0B24E3F14A2236969
```

The QP License Certificate file contains the following information:

- The license number (the file name of the certificate file)
- The name of the commercial Licensee (company or individual)
- The licensed QP Framework type(s) (qpc/qpcpp)
- The expiration date of the [Support Term↑](#) associated with the license
- The licensed [Single Product / Product Line/etc.↑](#) of the Licensee
- The cryptographic hash over the contents of the file and the file name, which prevents tampering with it.

## 2.2 QP/C Tutorial

This Tutorial describes how to use the QP/C real-time event framework in a series of progressively advancing examples. The first example ("Blinky") uses only one Active Object with a simple non-hierarchical state machine. The following example ("DPP") demonstrates multiple, communicating Active Objects. Finally, the last example ("Fly'n'Shoot" game) demonstrates all features the QP framework. It is highly recommended to study the simpler examples before the more advanced ones, as the basic information won't be repeated in the later examples.

This Tutorial consists of the following lessons:

- [Simple Blinky Application](#)
- [Dining Philosophers Problem \(DPP\)](#)
- ["Fly 'n' Shoot" Game](#)
- [Low-Power Example](#)

### Remarks

Perhaps the most important fact of life to remember is that in embedded systems nothing works until everything works. This means that you should always start with a *working* system and *gradually* evolve it, changing one thing at a time and making sure that it keeps *working* every step of the way.

Keeping this in mind, the provided [QP/C application examples](#), such as the super-simple Blinky, or a bit more advanced DPP or "Fly 'n' Shoot" game, allow you to get started with a working project rather than starting from scratch. You should also always try one of the provided example projects on the same evaluation board that it was designed for, before making any changes.

Only after convincing yourself that the example project works "as is", you can think about creating your own projects. At this point, the easiest and recommended way is to copy the existing working example project folder (such as the Blinky example) and rename it.

After copying the project folder, you still need to change the name of the project/workspace. The easiest and safest way to do this is to open the project/workspace in the corresponding IDE and use the "Save As" option to save the project under a different name. (You can do this also with the [QM model file↑](#), which you can open in QM and "Save As" a different model.)

### Note

By copying and re-naming an existing, working project, as opposed to creating a new one from scratch, you inherit the correct compiler and linker options and other project settings, which will help you get started much faster.

## 2.2.1 Simple Blinky Application

The ultra-simple Blinky example is the embedded systems' equivalent of the venerable "*Hello World!*" program, that is, the simplest possible working QP application that does "something". In the case of Blinky, this "something" is blinking an LED at the rate of 1Hz, where an LED turns on and remains on for 0.5 seconds on then turns off and remains off for 0.5 seconds.

### Note

The Blinky examples is a bit too simplistic as a starting point for real-life projects. A better, more complete starting point is the [DPP example](#).



Blinky on EK-TM4C123GLX (TivaC LaunchPad)

The Blinky example is provided for other supported boards as well. Please look for examples named `blinky_<board-name>` in the [qpc/examples/examples](#) directory (e.g., `qpc/examples/arm-cm/blinky_ek-tm4c123gxl` for the EK-TM4C123GXL board (TivaC LaunchPad)).

The ultra-simple Blinky application, which consists of just one active object named `Blinky`, is intentionally kept small and illustrates only the most basic QP features, such as:

- a simple Blinky active object (AO);
- hand-coding the simple state machine of the Blinky AO;
- using a periodic time event;
- initializing the QP framework;
- starting an active object; and
- transferring control to QP to run the application.

Built-in kernels:

- non-preemptive `QV` kernel;

- preemptive, non-blocking QK kernel;

Build configurations:

- debug configuration (default)
- release configuration
- spy (software tracing) configuration is NOT demonstrated

The details of the Blinky application are describe in the Quantum Leaps Application Note [Getting Started with QP Real-Time Event Frameworks](#)↑.



*Getting Started with QP Real-Time Event Frameworks*

## 2.2.2 Dining Philosophers Problem (DPP)

The Dining Philosophers Problem (DPP) example is an intermediate-level application with *multiple* active objects. It illustrates the following QP features, such as:

- multiple active objects, including an array of active objects;
- designing the simple state machines in the QM tool and generating the code automatically (can be also done manually);
- using multiple periodic time events;
- using mutable events with parameters;
- direct event posting to active objects;
- publish-subscribe event delivery;
- initializing the QP framework;
- starting multiple active objects; and
- transferring control to QP to run the application.

Built-in kernels:

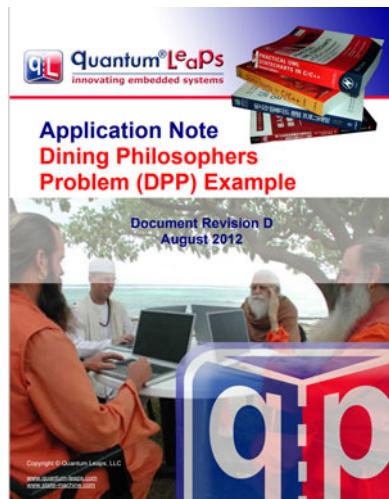
- non-preemptive [QV](#) kernel;
- preemptive, non-blocking [QK](#) kernel;
- dual-mode, blocking/non-blocking [QXK](#) kernel (for ARM Cortex-M);

Build configurations:

- debug configuration (default)
- release configuration
- spy (software tracing) configuration

The DPP example is provided for most supported boards as well as in the command-line version (on the host). Please look for examples named `dpp_<board-name>` in the [qpc/examples/examples](#) directory (e.g., `qpc/examples/arm-cm/dpp_ek-tm4c123gxl` for the EK-TM4C123GXL board (TivaC LaunchPad)).

The Dining Philosophers Problem (DPP) example is described in the [Application Note: Dining Philosophers Problem \(DPP\) Example↑](#).



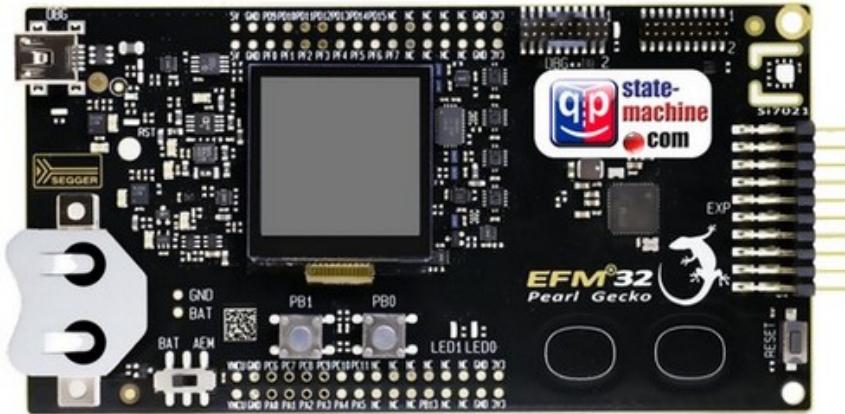
*Application Note: Dining Philosophers Problem (DPP) Example*

**Note**

There is also a DPP variant that implements the "Philosophers" as passive "Orthogonal Components" of the "Table" active object. That DPP version is located in  
`qpc/examples/examples/workstation/dpp-comp/`

### 2.2.3 "Fly 'n' Shoot" Game

The "Fly 'n' shoot" game example is a moderately advanced application of a vintage computer game. It requires a LCD screen and push-buttons on the board.



*EFM32 Pearl-Gecko*

"Fly 'n' shoot" game and illustrates the following QP features, such as:

- multiple active objects;
- multiple passive state machines ("Orthogonal Components");
- multiple periodic time events;
- mutable events with parameters;
- direct event posting to active objects;
- publish-subscribe event delivery;
- developing of embedded software on Windows (see also [QWin GUI Prototyping Toolkit↑](#))

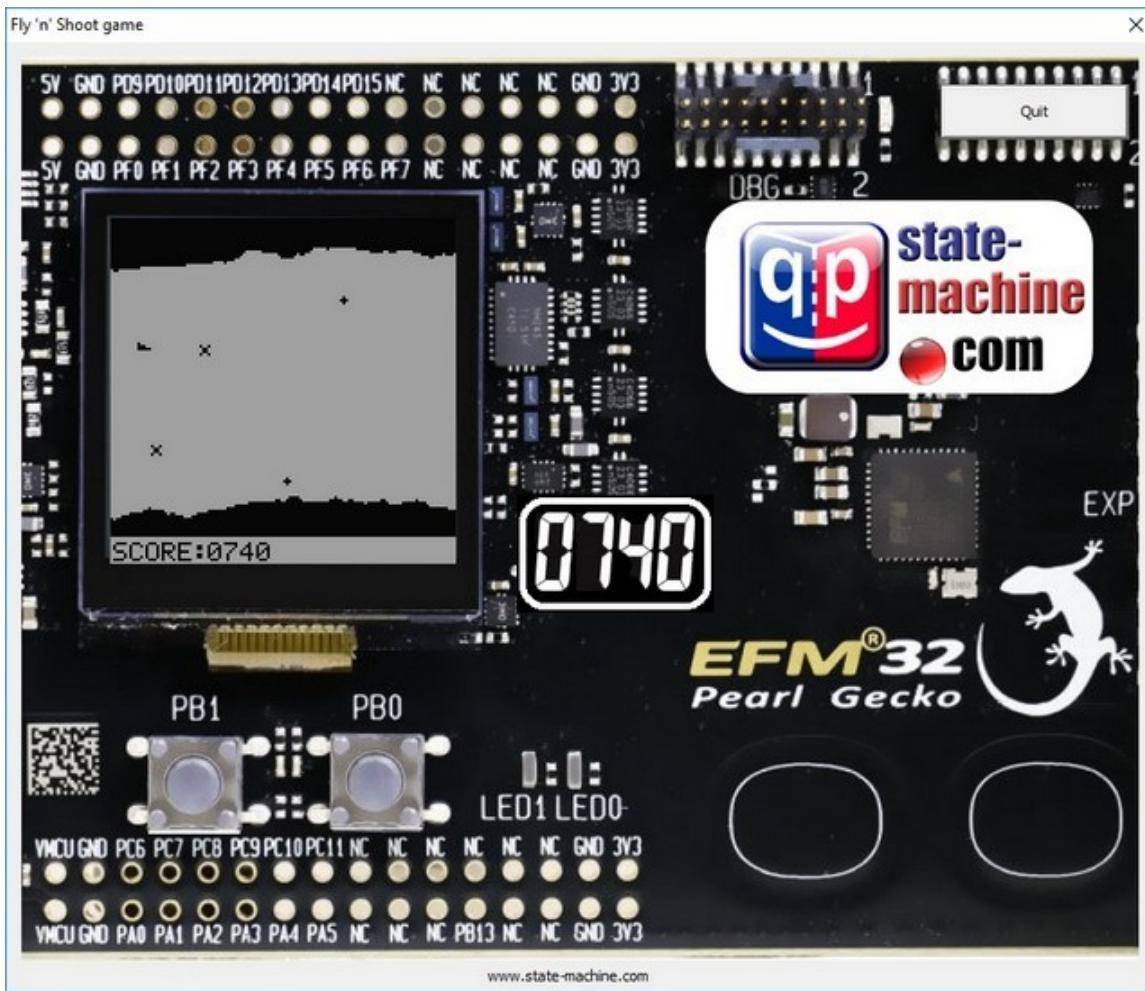
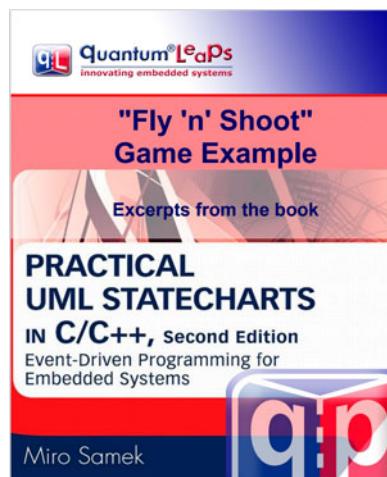
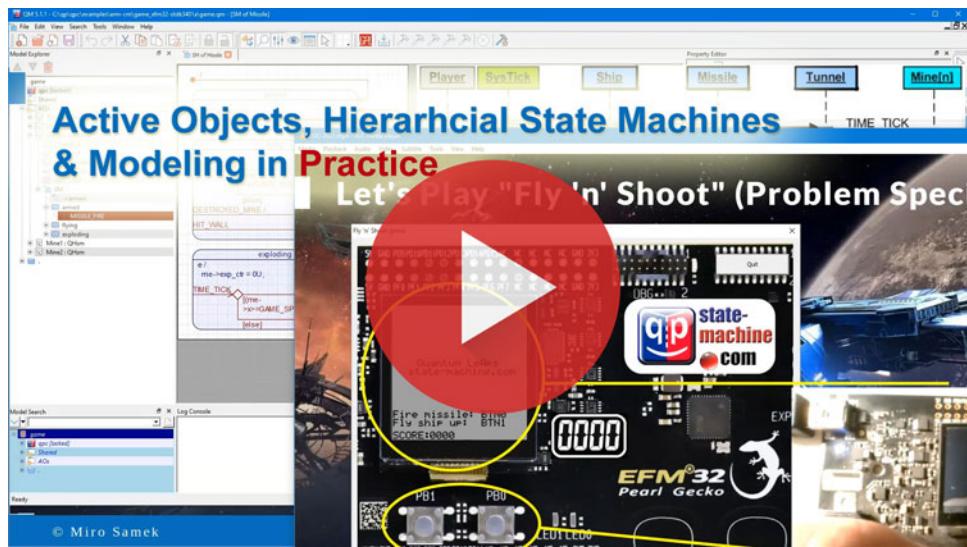


Figure 2.1 Fly 'n' Shoot game running on Windows

The "Fly 'n' Shoot" game example is described in the [Application Note: Fly 'n' Shoot Game Example↑](#).



*Application Note: Fly 'n' Shoot Game Example**Fly'n'Shoot game tutorial video*

## 2.2.4 Low-Power Example

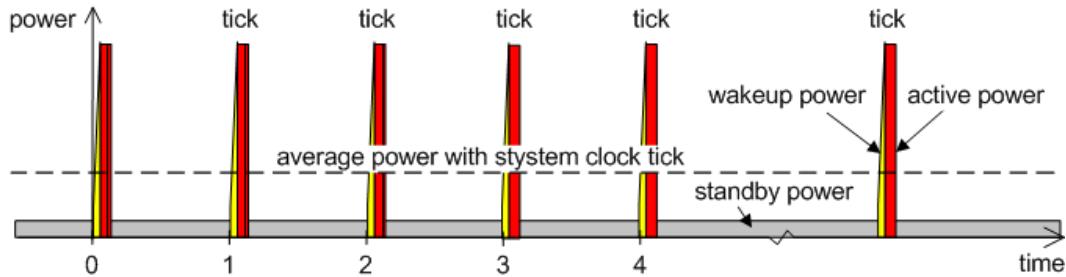
The main principle of low-power design for software is to keep the hardware in the most appropriate low-power sleep mode for as long as possible. Most commonly, the software enters a low-power sleep mode from the **idle callback** (a.k.a. "idle hook"), which is called when the software has nothing left to do and is waiting for an interrupt to deliver more work. The QP/C and QP/C++ Real-Time Event Frameworks (RTEFs) support the *idle callback* in all of the built-in real-time kernels, such as the non-preemptive **QV kernel**, the preemptive non-blocking **QK kernel** and the preemptive blocking **QXK kernel**. Also, such an *idle callback* is provided in all 3rd-party traditional RTOS kernels that QP/C/C++ have been [ported to](#).

### Remarks

Design for low-power is a broad subject that requires a holistic system approach to achieve a really long battery-powered operation. This example covers only certain *software-related* aspects of the problem.

Most real-time systems, including traditional RTOSes and RTEFs, require a periodic time source called the **system clock tick** to keep track of time delays, timeouts, and **QTimeEvt** "time events" in case of the event-driven QP Frameworks. The system clock tick is typically a periodic interrupt that occurs at a predetermined rate, typically between 10Hz and 1000Hz.

While the system clock tick is very useful, it also has the unfortunate side effect of taking the processor out of a low power state as many as 1000 times per second regardless if real work needs to be done or not. This effect can have a significant negative impact on the power efficiency of the system.



*Additional power dissipation caused by the system clock tick*

### 2.2.4.1 The Tickless Mode

Some real-time kernels use the low-power optimization called the "tickless mode" (a.k.a. "tick suppression" or "dynamic tick"). In this mode, instead of indiscriminately making the clock tick fire with a fixed period, the kernel adjusts the clock ticks *dynamically*, as needed. Specifically, after each clock tick the kernel re-calculates the time for the next clock tick and then sets the clock tick interrupt for the earliest timeout it has to wait for. So for example, if the shortest wait the kernel has to attend to is 300 milliseconds into the future, then the clock interrupt will be set for 300 milliseconds.

This approach maximizes the amount of time the processor can remain asleep, but requires the kernel to perform the additional work to calculate the dynamic tick intervals and to program them into the hardware. This additional bookkeeping adds complexity to the kernel, is often non-deterministic and, most importantly, extends the time CPU spends in the high-power active mode and thus eliminates some of the power gains the "tickless mode" was supposed to bring.

Also, the "tickless mode" requires a more capable hardware timer that must be able of being reprogrammed for every interrupt in a wide dynamic range and also must accurately keep track of the elapsed time to correct for the irregular (dynamic) tick intervals. Still, "tickless mode" often causes a drift in the timing of the clock tick.

### 2.2.4.2 Multiple Tick Rates

For the reasons just mentioned, the QP Real-Time Event Frameworks don't provide the "tickless mode". Instead, the QP frameworks support **multiple clock tick rates**, which can be turned on and off, as needed. The QP frameworks also provide methods for finding out *when* a given clock tick rate is not used, which allows the idle callback inside the application to shut down the given clock tick rate and also to decide which sleep mode to use for the CPU and the peripherals.

The support for multiple static clock tick rates is much *simpler* than the "dynamic tick", and essentially does not increase the complexity of the kernel (because the same code for the single tick rate can handle other tick rates the same way). Also, multiple static tick rates require much simpler hardware timers, which can be clocked specifically to the desired frequency and don't need particularly wide dynamic range. For example, 16-bit timers or even 8-bit timers are completely adequate.

Yet the *multiple clock rates* can deliver similar low-power operation for the system, while keeping the QP Framework much simpler and easier to certify than "tickless" kernels. The rest of this example explains how to use the multiple static clock rates in QP/C/C++ and shows how to leverage this feature to achieve low-power software design.

### 2.2.4.3 The Low-Power Code

The low-power example is located in QP/C and QP/C++ distributions, in the directory with the following structure:

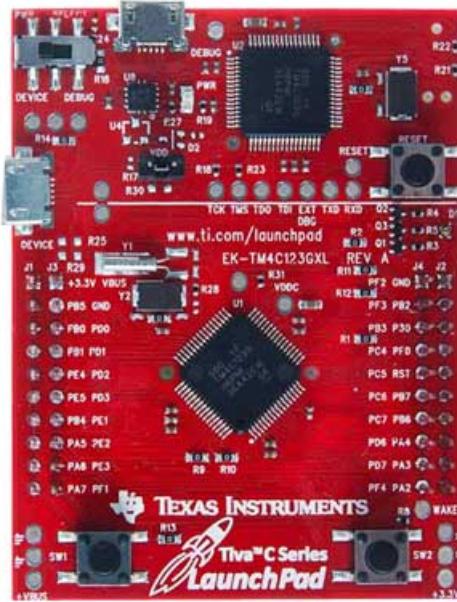
```

qpclqpcpp/                                // QP/C/C++ installation directory
+-examples/                               // QP/C/C++ examples directory (application)
| +-arm-cm/                               // QP/C/C++ examples for ARM Cortex-M
| | +-low-power_ek-tm4c123gx1/           // Low-Power example on the EK-TM4C123GLX board
| | | +-qk/                                //----- Projects for the preemptive QK kernel
| | | | +-arm/                            // ARM-KEIL toolchain
| | | | | low-power-qk.uvprojx // uVision project
| | | | | +-gnu/                           // GNU-ARM toolchain
| | | | | Makefile                         // Makefile for building the project
| | | | | +-iar/                           // IAR-ARM toolchain
| | | | | low-power-qk.eww // IAR EW-ARM workspace
| | | | | bsp.c                            // BSP for the QK kernel
| | | +-qv/                                //----- Projects for the non-preemptive QV kernel
| | | | +-arm/                            // ARM-KEIL toolchain
| | | | | low-power-qv.uvprojx // uVision project
| | | | | +-gnu/                           // GNU-ARM toolchain
| | | | | Makefile                         // Makefile for building the project with GNU-ARM
| | | | | +-iar/                           // IAR-ARM toolchain
| | | | | low-power-qv.eww // IAR EW-ARM workspace
| | | | | bsp.c|.cpp                      // BSP for the QV kernel
| | | +-qxk/                                //----- Projects for the dual-mode QXK kernel
| | | | +-arm/                            // ARM-KEIL toolchain
| | | | | low-power-qxk.uvprojx // uVision project
| | | | | +-gnu/                           // GNU-ARM toolchain
| | | | | Makefile                         // Makefile for building the project
| | | | | +-iar/                           // IAR-ARM toolchain
| | | | | low-power-qxk.eww // IAR EW-ARM workspace
| | | | | bsp.c|.cpp                      // BSP for the QxK kernel
| | | | | xblinky1.c|.cpp                 // eXtended thread for the QXK kernel

```

#### Note

To focus the discussion, this example uses the **EK-TM4C123GXL** evaluation board (a.k.a. TivaC LaunchPad) with Cortex-M4 MCU. However, the general demonstrated principles apply to most modern MCUs.



*EK-TM4C123GXL (TivaC LaunchPad)*

### Behavior

The low-power example illustrates the use of two clock tick rates to toggle the LEDs available on the EK-TM4C123GXL board. After the application code is loaded to the board, the **Green-LED** starts blinking once per two seconds (a second on and a second off), while the **Red-LED** lights up and stays on. If no buttons on the board are pressed, the **Green-LED** stops blinking after 4 times. The **Red-LED** shows the **idle** condition, where the system is in a sleep mode.

When you press the **SW1-Button**, the **Green-LED** starts blinking as before, but additionally, the **Blue-LED** starts blinking rapidly for 13 times (1/10 of a second on and 1/10 off).

So, depending when the SW1 switch is pressed, you can have only **Green-LED** blinking, or both green and blue blinking at different rates. The **Red-LED** appears to be on all the time.

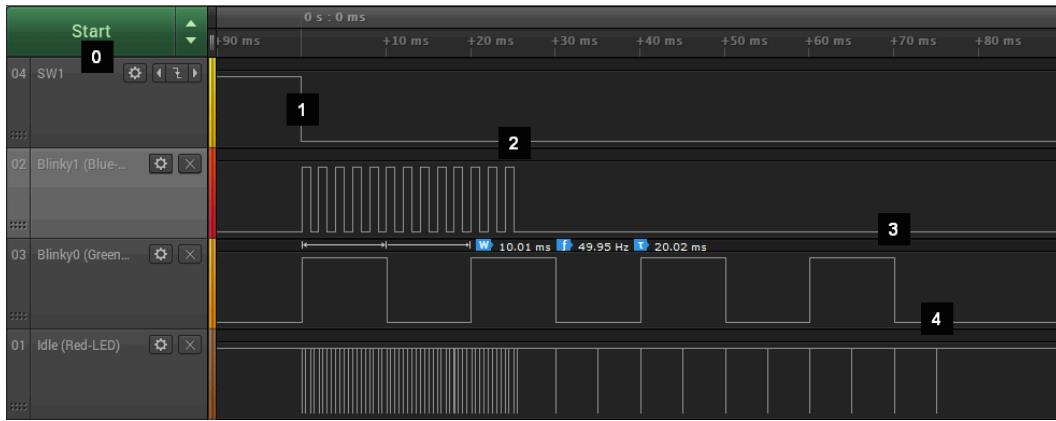
### Note

Actually, the **Red-LED** is also turned off for very brief moments, but this is imperceptible to the human eye. Instead, the **Red-LED** appears to be on all the time, which corresponds to the application being mostly idle.

The behavior just described is designed for the slow human interaction with the application. However, for more precise measurements with a logic analyzer, it is more convenient to speed up the application by factor of 100. This speed up can be achieved by editing the `bsp.h` header file:

```
// The following ticks-per-second constants determine the speed of the app.
// The default (#if 1) is the SLOW speed for humans to see the blinking.
// Change the #if 1 into #if 0 for FAST speed appropriate for logic analyzers.
//
#if 0
    #define BSP_TICKS0_PER_SEC    2U
    #define BSP_TICKS1_PER_SEC    20U
#else
    #define BSP_TICKS0_PER_SEC    2000U
    #define BSP_TICKS1_PER_SEC    20000U
#endif
...
```

The following logic analyzer trace shows the behavior of the low-power application at the faster time scale. The explanation section immediately following the trace explains the most interesting points:



*low-power application after pressing the SW1 button*

[ 0 ] The plot shows the logic-analyzer traces of the following signals (from the top): **SW1** button (active low), **Blinky1** (Blue-LED), **Blinky0** (Green-LED) and **Idle** (Red-LED). The **Idle** callback turns the **Red-LED** on when the system enters the low-power sleep mode and it turns the **Red-LED** off when the system is active.

[ 1 ] At this time the **SW1** button is depressed, which triggers an interrupt that activates both the slow and the fast clock tick rates. The clock tick interrupts trigger toggling of the **Blinky0** (Green-LED) at the slower tick rate-0 and **Blinky1** (Blue-LED) at the faster tick rate-1.

[ 2 ] The **Blinky1** (Blue-LED) stops toggling after 13 cycles. At this point also the **Idle** callback turns the **fast tick rate-1** off.

[ 3 ] The **Blinky0** (Green-LED) stops toggling after 4 cycles.

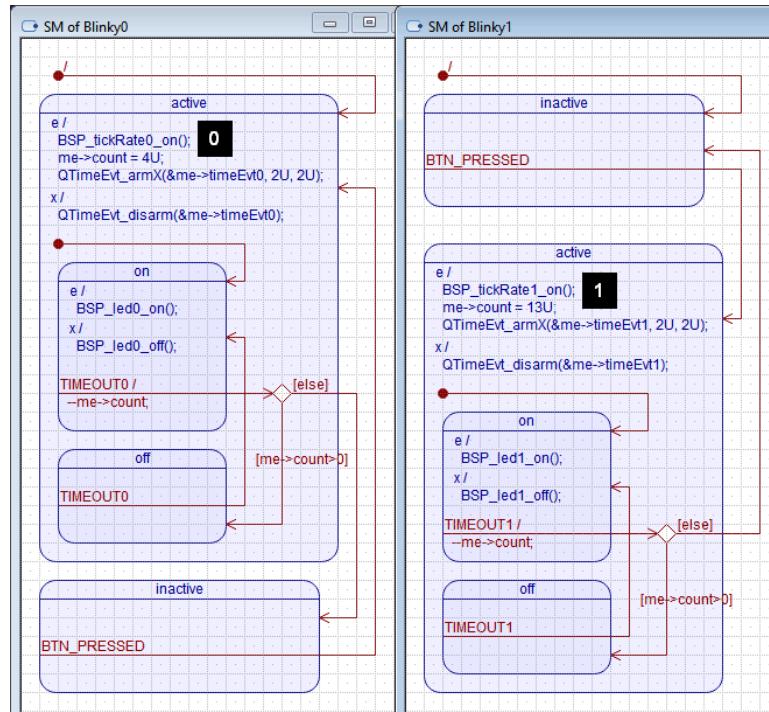
[ 4 ] The **Idle** callback turns the **slow tick rate-0** off. The system enters the low-power sleep mode without any clock ticks running and remains in this state until the **SW1** button is pressed again.

#### Remarks

The **Idle** line (Red-LED) goes down twice as fast as the changes in the state of the **Green-LED** or the **Blue-LED**. This is because the application uses timeouts of **2 clock ticks** for toggling these lines instead of just one clock tick, which then could be slower. This is not quite optimal for the energy dissipation (as the CPU is woken up twice as often as it needs to be), but it illustrates more accurately how the fixed clock rates work as well as the one-tick delay to enter the low-power sleep mode from the idle callback.

#### 2.2.4.4 State Machines

The versions of this low-power example for the **QK** and **QV** kernels use two active objects **Blinky0** and **Blinky1**, which toggle the **Green-LED** at the slow tick rate 0, and the **Blue-LED** at the fast tick rate 1, respectively. The state machines of the Blinky0 and Blinky1 active objects are shown below:



*state machines Blinky0 and Blinky1 active objects*

[0] The **Blinky0** state machine, in the entry action to the "active" state, calls `BSP_tickRate0_on()` to turn the tick rate-0 on. This is done *before* arming the time event `me->timeEvt0` that uses the clock rate-0.

[1] Similarly, the **Blinky1** state machine, in the entry action to the "active" state, calls `BSP_tickRate1_on()` to turn the tick rate-1 on. This is done *before* arming the time event `me->timeEvt1` that uses the clock rate-1.

## Extended Thread (QXK)

The version of this low-power example for the **QXK** kernel uses one active object **Blinky0** (with the state machine shown above), but instead of the Blinky1 active object, the **QXK** version uses an eXtended thread (**QXThread**) called **XBlinky1**, with the code shown below:

```
#include "qpc.h"
#include "low_power.h"
#include "bsp.h"

// local objects -----
static void XBlinky1_run(QXThread * const me);

// global objects -----
QXThread XT_Blinky1;
QXSemaphore XSEM_sw1;

//.....
void XBlinky1_ctor(void) {
    QXThread_ctor(&XT_Blinky1,
                  &XBlinky1_run, // thread routine
                  1U); // associate the thread with tick rate-1
}

//.....
static void XBlinky1_run(QXThread * const me) {
    bool isActive = false;

    (void)me; // unused parameter
    for (;;) {
        if (XSEM_sw1.wait() == 0) {
            if (!isActive) {
                // turn LED on
                bsp_setPinDigitalOutputLevel(BSP_LED_PINK, 1);
                isActive = true;
            }
        } else {
            if (isActive) {
                // turn LED off
                bsp_setPinDigitalOutputLevel(BSP_LED_PINK, 0);
                isActive = false;
            }
        }
    }
}
```

```

[0]     if (!isActive) {
[1]         QXSemaphore_wait(&XSEM_sw1, QXTTHREAD_NO_TIMEOUT);
[2]         isActive = true;
[3]     }
[4]     else {
[5]         uint8_t count;
[6]         BSP_tickRate1_on();
[7]         for (count = 13U; count > 0U; --count) {
[8]             BSP_led1_on();
[9]             QXThread_delay(1U);
[10]            BSP_led1_off();
[11]            QXThread_delay(1U);
[12]        }
[13]        isActive = false;
[14]    }
[15] }

```

[0] The **XBlinky1** extended thread emulates the states with the `isActive` flag. When the flag is not set (meaning that the system is not active), the thread waits (and blocks) on the global semaphore `XSEM_sw1`.

[1] After the semaphore is signalled (from the GPIO interrupt in the BSP), the **XBlinky1** extended thread calls `BSP_tickRate1_on()` to turn the tick rate-1 on. This is done *before* later calling `QXThread_delay()` function, which in case uses the clock rate-1.

#### 2.2.4.5 The Idle Callback (QK/QXK)

The most important functionality in this low-power example is implemented in the **idle callback** located in the BSP (Board Support Package). The idle callback `QK_onIdle()` for the preemptive **QK** kernel and the idle callback `QXK_onIdle()` for the **QXK** kernel are almost identical and are explained in this section.

```

[0] void QXK_onIdle(void) {
[1]     QF_INT_DISABLE();
[2]     if (((l_activeSet & (1U << SYSTICK_ACTIVE)) != 0U) // rate-0 enabled?
[3]         && QF_noTimeEvtsActiveX(0U)) // no time events at rate-0?
[4]     {
[5]         // safe to disable SysTick and interrupt
[6]         SysTick->CTRL &= ~(SysTick_CTRL_TICKINT_Msk | SysTick_CTRL_ENABLE_Msk);
[7]         l_activeSet &= ~(1U << SYSTICK_ACTIVE); // mark rate-0 as disabled
[8]     }
[9]     if (((l_activeSet & (1U << TIMER0_ACTIVE)) != 0U) // rate-1 enabled?
[10]        && QF_noTimeEvtsActiveX(1U)) // no time events at rate-1?
[11]    {
[12]        // safe to disable Timer0 and interrupt
[13]        TIMER0->CTL &= ~(1U << 0); // disable Timer0
[14]        TIMER0->IMR &= ~(1U << 0); // disable timer interrupt
[15]        l_activeSet &= ~(1U << TIMER0_ACTIVE); // mark rate-1 as disabled
[16]    }
[17]    QF_INT_ENABLE();
[18]
[19]    GPIOF->DATA_Bits[LED_RED] = 0xFFU; // turn LED on, see NOTE2
[20]    __WFI(); // wait for interrupt
[21]    GPIOF->DATA_Bits[LED_RED] = 0x0U; // turn LED off, see NOTE2
}

```

[0] The idle callback for **QK** and **QXK** are called from the idle loop with interrupts enabled.

[1] Interrupts are disabled to access the shared bitmask `l_activeSet`, which stores the information about active clock rates and peripherals. This bitmask is shared between the idle callback and the application-level threads.

[2] If the SYSTICK timer is active (source of the tick-0 rate)

[3] The `QF_noTimeEvtsActiveX(0)` function is called to check whether no time events are active at the clock rate-0.

### Remarks

The QF\_noTimeEvtsActiveX() function is designed to be called from a critical section, which is the case here.

- [ 4 ] If both of these conditions hold, it is safe to turn the clock rate-0 off, which is done here.
- [ 5 ] The bit indicating that SYSTICK timer is active is cleared in the l\_activeSet bitmask.
- [ 6 ] Similarly, the bit corresponding to TIMER0 is checked in the l\_activeSet bitmask.
- [ 7 ] The QF\_noTimeEvtsActiveX(1) function is called to check whether no time events are active at the clock rate-1.
- [ 8–9 ] If both of these conditions hold, it is safe to turn the clock rate-1 off, which is done here.
- [ 10 ] The bit indicating that TIMER0 timer is active is cleared in the l\_activeSet bitmask.
- [ 11 ] Interrupts are enabled, so the following code is no longer inside critical section
- [ 12 ] The **Red-LED** is turned ON right before entering the low-power sleep mode
- [ 13 ] The `__WFI()` instruction stops the CPU and enters the **low-power sleep mode**
- [ 14 ] The **Red-LED** is turned off after waking up from the sleep mode

### The Idle Callback ([QV](#))

The idle callback QV\_onIdle() for the non-preemptive [QV](#) kernel is slightly different, because it is called with interrupts **disabled**. The following listing shows the complete QV\_onIdle() callback, with the most important points explained in the section below:

```
[0] void QV_onIdle(void) { // CAUTION: called with interrupts DISABLED
[1]     if (((l_activeSet & (1U << SYSTICK_ACTIVE)) != 0U) // rate-0 enabled?
[2]         && QF_noTimeEvtsActiveX(0U)) // no time events at rate-0?
{
    // safe to disable SysTick and interrupt
    SysTick->CTRL &= ~(SysTick_CTRL_TICKINT_Msk | SysTick_CTRL_ENABLE_Msk);
    l_activeSet &= ~(1U << SYSTICK_ACTIVE); // mark rate-0 as disabled
}
if (((l_activeSet & (1U << TIMER0_ACTIVE)) != 0U) // rate-1 enabled?
    && QF_noTimeEvtsActiveX(1U)) // no time events at rate-1?
{
    // safe to disable Timer0 and interrupt
    TIMER0->CTL &= ~(1U << 0); // disable Timer0
    TIMER0->IMR &= ~(1U << 0); // disable timer interrupt
    l_activeSet &= ~(1U << TIMER0_ACTIVE); // mark rate-1 as disabled
}

GPIOF->DATA_Bits[LED_RED] = 0xFFU; // turn LED on, see NOTE2
[3]     QV_CPU_SLEEP(); // atomically go to sleep and enable interrupts
    GPIOF->DATA_Bits[LED_RED] = 0x00U; // turn LED off, see NOTE2
}
```

- [ 0 ] The idle callback for [QV](#) is called from the [QV](#) event-loop with interrupts **disabled**.
- [ 1 ] The l\_activeSet bitmask is tested right away, because interrupts are already disabled
- [ 2 ] The QF\_noTimeEvtsActiveX(0) function is called to check whether no time events are active at the clock rate-0.

### Remarks

The QF\_noTimeEvtsActiveX() function is designed to be called from a critical section, which is the case here.

- [ 3 ] The `QV_CPU_SLEEP()` macro enters **low-power sleep mode** with interrupts still disabled. This port-specific macro is designed to re-enable interrupts **atomically** with entering the sleep mode.

# Chapter 3

## Ports

### 3.1 General Comments

The QP/C framework can be easily adapted to various operating systems, processor architectures, and compilers. Adapting the QP/C software is called porting and the QP/C framework has been designed from the ground up to make porting easy.

The QP/C distribution contains many QP/C ports, which are organized into the three categories:

- [PC-Lint-Plus](#) (generic C compiler) QP/C "port" to the PC-Lint Plus static analysis tool (a "compiler")
- [Native Ports](#) adapt QP/C to run on bare-metal processors "natively", using one of the built-in kernels ([QV](#), [QK](#), or [QXK](#))
- [3rd-Party RTOS Ports](#) adapt QP/C to run on top of a 3rd-Party Real-Time Operating System (RTOS)
- [General-Purpose OS Ports](#) adapt QP/C to run on top of a General Purpose Operating System (GPOS), such as [Linux](#) or [Windows](#).

#### 3.1.1 Code Structure

All QP/C ports are located in sub-directories of the `ports/` top-level directory, with the hierarchical organization outlined below:

```
qpc|qpcpp/          // QP/C or QP/C++ installation directory
+---ports/          // ports directory
|   |
|   [1a]-arm-cm/    // ports for ARM Cortex-M
|   |   +---qk/      // ports for preemptive QK kernel
|   |   |   +---armclang/ // port with ARM-Clang (Compiler Version 6) toolchain
|   |   |   +---gnu/   // port with GNU-ARM toolchain
|   |   |   +---iar/   // port with IAR toolchain
|   |   +---qv/       // ports for non-preemptive QV kernel
|   |   \---qxk/     // ports for dual-mode QXK kernel
|   |
|   [1b]-arm-cr/    // ports for ARM Cortex-R
|   . . .
```

```

|   |
| [2a]-embos/           // ports for embOS (3rd-party RTOS)
|
| [2b]-freertos/       // ports for FreeRTOS (3rd-party RTOS)
|
| . .
|
| [3a]-posix/          // port to POSIX (multithreaded)
|
| [3b]-posix-qv/        // port to POSIX (single-threaded)
|
| [4a]-win32/           // port to Win32 (multithreaded)
|
| [4b]-win32-qv/         // port to Win32 (single-threaded)

```

[1a-b] **Native Ports** are located in sub-directories named after the CPU architecture, such as `arm-cm` for ARM Cortex-M or `arm-cr` for ARM Cortex-R. Under that directory, the sub-directories `qk` and `qv` contain ports for the **QK** and **QV** kernels, respectively.

[2a-b] **Ports for 3rd-party RTOS** are located in sub-directories named after the RTOS, such as `embos` for embOS RTOS or `freertos` for FreeRTOS.

[3a-b] **Ports for the POSIX OS Standard** are located in sub-directories named `posix` for the POSIX API (multi-threaded variant) and `posix-qv` for the POSIX API (single-threaded) variant.

[4a-b] **Ports for the Win32 API** are located in sub-directories named `win32` for the Win32 API (multi-threaded variant) and `win32-qv` for the Win32 API (single-threaded) variant.

#### Note

Because the QP distribution contains *all* ports, the number of sub-directories and files in the `ports` folder may seem daunting. However, knowing the structure of the `ports` folder, you can simply **delete** the sub-directories that are not relevant to you.

## 3.2 PC-Lint-Plus

The QP/C distribution contains a "port" to [PC-Lint-Plus](#) static analysis tool from [Vector](#), which is a static analysis tool for C and C++ with one of the longest track records and best value of the money in the industry. The "PC-Lint-Plus port" allows you to statically analyze the QP/C source code and facilitates static analysis of your **application code** based on QP/C.

The QP/C "port" to PC-Lint-Plus is located in the directory `qpc/ports/lint-plus` and includes also lint configuration files, as well as an example of "linting" application code in the directory `qpc/examples/arm-cm/dpp_ek-tm4c123gxl/lint-plus`. The following listing describes the most important files in these three directories.

```

qpc\qpcpp/                  // QP/C or QP/C++ installation directory
+---ports/                   // ports directory
|   +---lint-plus/           // "port" to PC-Lint-Plus
|   |   +---16bit/            // "port" to 16-bit CPUs
|   |   |   +cpu.lnt          // Lint options for a 16-bit CPU
|   |   |   \stdint.h          // Standard exact-width integers for a 16-bit CPU
|   |   +---32bit/            // "port" to 32-bit CPUs
|   |   |   +cpu.lnt          // Lint options for a 32-bit CPU
|   |   |   \stdint.h          // Standard exact-width integers for a 32-bit CPU
|   |   +---qk/                // port with the QK kernel
|   |   +---qv/                // port with the QV kernel
|   |   +---qxk/               // port with the QXK kernel
|
|   |

```

```

|   |   au-misra4.lnt // MISRA-C:2023 compliance checks
|   |   au-ds.lnt   // Dan Saks recommendations
|   |   qpc.lnt    // PC-Lint-Plus options for applications
|   |   std.lnt    // Standard PC-Lint-Plus settings recommended by Quantum Leaps
|   |   make.bat   // Batch file to invoke PC-Lint-Plus to run analysis of @QPX code
|   |   options.lnt // PC/Lint-Plus options for "linting" @QPX source code
|   |   lint_qf.log // PC/Lint-Plus output for the QEP/QF components of @QPX
|   |   lint_qs.log // PC/Lint-Plus output for the QS component of @QPX
|   |   lint_qv.log // PC/Lint-Plus output for the QV component of @QPX
|   |   lint_qk.log // PC/Lint-Plus output for the QK component of @QPX
|   |   lint_qxk.log // PC/Lint-Plus output for the QXK component of @QPX
|   |   stdbool.h  // Standard Boolean type and constants for a "generic C compiler"
|
+---examples/          // examples directory (application)
| +---arm-cm/          // examples for ARM Cortex-M
| | +---dpp_ek-tm4c123gxl/ // DPP example on the EK-TM4C123GLX board
| | | +---lint-plus/   // directory for linting the application
| | |
| | |   make.bat // Batch to run PC-Lint-Plus analysis of application code
| | |   options.lnt // PC-Lint-Plus options for "linting" of application code

```

### 3.2.1 Linting the QP/C Source Code

The directory `qpc/ports/lint-plus` (see listing above) contains also the `make.bat` batch file for "linting" the QP/C source code. The `make.bat` batch file invokes PC-Lint-Plus and generates the lint output files. As shown in the listing above, the lint output is collected into four text files `lint_qf.log`, `lint_qs.log`, `lint_qk.log`, `lint_qv.log`, and `lint_qs.log`, for QEP/QF, **QK**, **QV**, **QXK** and **QS** components of the QP/C framework, respectively.

#### Note

In order to execute the `make.bat` file on your system, you might need to adjust the symbol `PCLP_DIR` at the top of the batch file, to the PC-Lint-Plus installation directory on your computer.

#### Remarks

The `make.bat` batch file invoked without any command-line options checks the QP/C code in the **Q\_SPY** build configuration with software tracing enabled. However, by the nature of software tracing, the **Q\_SPY** configuration transgresses many more MISRA-C rules than the standard configuration. However, the **Q\_SPY** configuration is never used for production code, so the MISRA-C compliance of the QP/C framework should not be judged by the deviations that happen only in the **Q\_SPY** configuration.

According to the PC-Lint-Plus guidelines, the `make.bat` uses two option files: the `qpc.lnt` configuration file discussed before and the `options.lnt` configuration file that covers all deviations from the MISRA-C rules **within the QP/C source code**. These deviations are intentionally localized to QP/C code and are independent from your **application-level** code. In other words, a MISRA-C deviation present in the QP/C code does **not** mean that such deviation is somehow allowed or its detection is somehow suppressed in the **application-level** code. This is because the the `options.lnt` configuration file for internals of QP/C is **not** used to "lint" the application-level code.

### 3.2.2 Linting QP/C Application Code

The QP/C baseline code contains an example of MISRA-C compliance checking with PC-Lint-Plus: the DPP example for the EK-TM4C123GLX Cortex-M4F board, located in the directory `qpc/examples/arm-cm/dpp_ek-tm4c123gxl/lint-plus`. The PC-Lint-Plus analysis is very simple and requires invoking the `make.bat` file.

#### Note

In order to execute the **make.bat** file on your system, you might need to adjust the symbol `PCLP_DIR` at the top of the batch file, to the PC-Lint-Plus installation directory on your computer. You

The `lint-plus` subdirectory contains also the local version of the `options.lnt` configuration file with the PC-Lint-Plus options specific to linting the application. Here, you might include linting options for your specific compiler, as described in the "PC-Lint-Plus Manual", Chapter 2 "Installation and Configuration".

### 3.2.3 Structure of PC-Lint-Plus Options for QP/C

PC-Lint-Plus has several places where it reads its currently valid options:

- From special PC-Lint-Plus option files (usually called `*.lnt`)
- From the command line
- From within the special lint-comments in the source code modules (not recommended)

The QP/C source code and example application code has been "linted" only by means of the first alternative (option files) with possibility of adding options via command line. The third alternative—lint comments—is not used and Quantum Leaps does not recommend this alternative.

#### Note

The QP/C source code is completely free of lint comments, which are viewed as a contamination of the source code.

The structure of the PC-Lint-Plus option files used for "linting" QP/C follows the [PC-Lint-Plus \(PCLP\)↑](#) guidelines for configuring PC-Lint-Plus (See Section 2 "Configuration" in the *PC-Lint-Plus Manual*). The design and grouping of the lint options also reflects the fact that static code analysis of a software framework, such as QP/C, has really two major aspects. First, the source code of the framework itself has to be analyzed. But even more important and helpful to the users of the framework is providing the infrastructure to effectively analyze the application-level code based on the framework. With this in mind, the PC-Lint-Plus options for static analysis of QP/C are divided into two groups, located in directories `qpc/include` and `qpc/ports/lint`. These two groups are for analyzing QP/C **applications** and QP/C **source code**, respectively.

As shown in the PC-Lint-Plus "port" files description, the directory `qpc/include`, contains the PC-Lint-Plus options for "linting" the application code along with all platform-independent QP/C header files required by the applications. This collocation of lint options with header files simplifies "linting", because specifying just `-iqpc/include` includes the `qpc/include` directory to PC-Lint-Plus accomplishes both inclusion of QP/C header files and PC-Lint-Plus options. Note that the `qpc/include` directory contains all PC-Lint-Plus option files used in "linting" the code, including the standard MISRA-C:2023 `au-misr4.lnt` option file as well as Dan Saks' recommendations `au-ds.lnt`, which are copied from the PC-Lint-Plus distribution. This design freezes the lint options for which the compliance has been checked.

#### 3.2.3.1 The std.int option file

According to the *PC-Lint-Plus Configuration Guidelines*, the file **std.int** file, contains the top-level options, which Quantum Leaps recommends for all projects. These options include the formatting of the PC-Lint-Plus messages and making two passes to perform better cross-module analysis. However, the most important option is `-restore_at_end`, which has the effect of surrounding each source file with options `-save` and `-restore`. This precaution prevents options from "bleeding" from one file to another.

### 3.2.3.2 The qpc.Int option file

The most important file for "linting" QP/C applications is the **qpc.Int** option file. This file handles all deviations from the MISRA-C:2023 rules, which might arise at the application-level code from the use of the QP/C framework. In other words, the **qpc.Int** option file allows completely clean "linting" of the application-level code, as long as the application code does not violate any of the MISRA-C:2023 rules.

At the same time, the **qpc.Int** option file has been very carefully designed not to suppress any MISRA-C:2023 rule checking outside the very specific context of the QP/C API. In other words, the **qpc.Int** option file still supports 100% of the MISRA-C:2023 rule checks that PC-Lint-Plus is capable of performing.

#### Remarks

For example, for reasons explained in Section 5.10 of the "[QP/C MISRA Compliance Matrix](#)"↑, QP/C extensively uses function-like macros, which deviates from the MISRA-C:2023 advisory Rule 4.9 and which PC-Lint-Plus checks with the warning 9026. However, instead of suppressing this warning globally (with the -e9096 directive), the **qpc.Int** option file suppresses warning 9096 only for the specific QP function-like macros that are visible to the application level. So specifically, the **qpc.Int** file contains directives `-esym(9026, Q_TRAN, Q_SUPER, ...)`, which suppresses the warning only for the specified macros, but does not disable checking of any other macros in the application-level code.

## 3.3 Native Ports (Built-in Kernels)

- [ARM Cortex-M](#) (Cortex-M0/M0+/M3/M4/M4/M7/M23/M33/M85)
  - Non-preemptive [QV Kernel](#) (ARM-CLANG, ARM-KEIL, GNU-ARM, IAR-ARM)
  - Preemptive Non-Blocking [QK Kernel](#) (ARM-CLANG, ARM-KEIL, GNU-ARM, IAR-ARM)
  - Preemptive "Dual-Mode" [QXK Kernel](#) (ARM-CLANG, ARM-KEIL, GNU-ARM, IAR-ARM)
- [ARM Cortex-R](#) (Cortex-R)
  - Coopertaive [QV kernel](#)
  - Preemptive Non-Blocking [QK kernel](#)
- [MSP430](#) ("classic" MSP430 and "extended" MSP430X)
  - Coopertaive [QV kernel](#)
  - Preemptive Non-Blocking [QK kernel](#)

### 3.3.1 ARM Cortex-M

This section describes the QP/C ports to the ARM Cortex-M processor family (Cortex M0/M0+/M3/M4/M7/M33). Three main implementation options are covered: the [non-preemptive QV kernel](#), the [preemptive, run-to-completion QK kernel](#), and the [preemptive, dual-mode blocking QXK kernel](#). Additionally, the use of the VFP (floating point coprocessor) in the M4F/M7/M33 CPUs is explained as well. This document assumes QP/C version 7.x or higher.

#### Note

To focus the discussion, this section references the **GNU-ARM toolchain**, the [EK-TM4C123GXL](#) (ARM Cortex-M4F) and the Eclipse-based IDE (CCS from Texas Instruments). However, the general implementation strategy applies equally to all toolchains for ARM Cortex-M, such as **ARM-KEIL**, **IAR EWARM**, **GNU-ARM** and **TI-ARM**, which are all supported as well. The QP code downloads contain also examples for other boards, such as STM32 Nucleo, NXP mbed-1768, SiLabs Gecko and others.

### 3.3.1.1 Directories and Files

The QP ports to ARM Cortex-M are available in the standard QP/C distribution. Specifically, the ARM Cortex-M ports are placed in the following directories:

```

qp/ports/arm-cm      // QP ports to ARM Cortex-M
+---qk               // ports to QK preemptive kernel
|   +---armclang     // ports for ARM-CLANG (LLVM)
|   |     qp_port.h  // QP port
|   |     qs_port.h  // QS port
|   |     qk_port.c   // QK port implementation
|   +---gnu            // ports for GNU-ARM
|   |     ~ ~ ~
|   +---iar             // ports for IAR EWARM
|   |     ~ ~ ~
+---qv               // ports to QV non-preemptive kernel
|   +---armclang     // ports for ARM-CLANG (LLVM)
|   |     qp_port.h  // QP port
|   |     qs_port.h  // QS port
|   |     qv_port.c   // QV port implementation
|   +---gnu            // ports for GNU-ARM
|   |     ~ ~ ~
|   +---iar             // ports for IAR EWARM
|   |     ~ ~ ~
+---qxk              // ports to QXK dual-mode kernel
|   +---armclang     // ports for ARM-CLANG (LLVM)
|   |     qp_port.h  // QP port
|   |     qs_port.h  // QS port
|   |     qxk_port.c  // QXK port implementation
|   +---gnu            // ports for GNU-ARM
|   |     ~ ~ ~
|   +---iar             // ports for IAR EWARM
|   |     ~ ~ ~

```

### 3.3.1.2 Interrupts in the QP/C Ports to ARM Cortex-M

The QP/C real-time event framework, like any real-time kernel, needs to disable interrupts in order to access critical sections of code and re-enable interrupts when done. This section describes the general policy used in the ARM Cortex-M ports of all built-in real time kernels in QP/C, such as [QV](#), [QK](#), and [QXK](#).

#### "Kernel-Aware" and "Kernel-Unaware" Interrupts

The presence of critical sections in QP/C delays interrupt processing, which the CPU might not service fast enough. However, critical sections are necessary only for **kernel aware** ISRs that interact with the QP/C framework or the underlying real-time kernel. However, critical sections aren't necessary for ISRs that *don't interact* with the QP/C Framework or the underlying real-time kernel in any way, which are called **kernel unaware** ISRs.

Many ARM Cortex-M CPUs allow disabling interrupts selectively, so it is possible to disable only kernel aware ISRs, while leaving kernel unaware ISRs always enabled. Such ISRs are sometimes called "zero latency ISRs". Specifically, on the ARM Cortex-M that support the BASEPRI register, the QP/C port can disable interrupts **selectively**. This policy is enabled by defining the macro `QF_USE_BASEPRI`.

#### Note

The BASEPRI register is not implemented in all ARM Cortex-M, such as Cortex-M0/M0+ as well as Cortex-M23 CPUs. The CPUs without BASEPRI need to use the PRIMASK register to disable interrupts globally. In other words, in the QP/C ports to Cortex-M0/M0+, all interrupts are "kernel-aware".

## Attention

Only "kernel-aware" interrupts are allowed to call QP/C services. "Kernel-unaware" interrupts are **not** allowed to call any QP/C services and they can communicate with QP/C only by triggering a "kernel-aware" interrupt (which can post or publish events).

As illustrated in the figures below, the number of interrupt priority bits actually available is implementation dependent, meaning that the various ARM Cortex-M silicon vendors can provide different number of priority bits, varying from just 3 bits (which is the minimum for ARMv7-M architecture) up to 8 bits. For example, the TI Tiva-C microcontrollers implement only 3 priority bits (see figure below).

Interrupt type	NVIC priority bits	Priority for CMSIS NVIC_SetPriority()
Kernel-unaware interrupt	0 0 0 0 0 0 0 0	0 Never disabled
Kernel-aware interrupt	0 0 1 0 0 0 0 0	1 = QF_AWARE_ISR_CMSIS_PRI
Kernel-aware interrupt	0 1 0 0 0 0 0 0	2
Kernel-aware interrupt	0 1 1 0 0 0 0 0	3 Disabled in critical sections
Kernel-aware interrupt	1 0 0 0 0 0 0 0	4
Kernel-aware interrupt	1 0 1 0 0 0 0 0	5
Kernel-aware interrupt	1 1 0 0 0 0 0 0	6
PendSV interrupt for QK	1 1 1 0 0 0 0 0	7 Should not be used for regular interrupts

*(Kernel-aware and Kernel-unaware interrupts with 3 priority bits)*

On the other hand, the STM32 MCUs implement 4 priority bits (see figure below). The CMSIS standard provides the macro **NVIC\_PRIO\_BITS**, which specifies the number of NVIC priority bits defined in a given ARM Cortex-M implementation.

Interrupt type	NVIC priority bits	Priority for CMSIS NVIC_SetPriority()
Kernel-unaware interrupt	0 0 0 0 0 0 0 0	0
Kernel-unaware interrupt	0 0 0 1 0 0 0 0	1 Never disabled
Kernel-unaware interrupt	0 0 1 0 0 0 0 0	2
Kernel-aware interrupt	0 0 1 1 0 0 0 0	3 = QF_AWARE_ISR_CMSIS_PRI
Kernel-aware interrupt	0 1 0 0 0 0 0 0	4
Kernel-aware interrupt	0 1 0 1 0 0 0 0	5
Kernel-aware interrupt	0 1 1 0 0 0 0 0	6
Kernel-aware interrupt	0 1 1 1 0 0 0 0	7
Kernel-aware interrupt	... ... ... ...	...
Kernel-aware interrupt	1 1 1 0 0 0 0 0	14
Kernel-aware interrupt	1 1 0 1 0 0 0 0	12
PendSV interrupt for QK	1 1 1 1 0 0 0 0	15 Should not be used for regular interrupts

*Kernel-aware and Kernel-unaware interrupts with 4 priority bits](arm-cm\_int4bit.png)*

Another important fact to note is that the ARM Cortex-M core stores the interrupt priority values in the *most significant bits* of its eight bit interrupt priority registers inside the NVIC (Nested Vectored Interrupt Controller). For example, if an implementation of a ARM Cortex-M microcontroller only implements three priority bits, then these three bits are shifted to occupy bits five, six and seven respectively. The unimplemented bits can be written as zero or one and always read as zero.

And finally, the NVIC uses an inverted priority numbering scheme for interrupts, in which priority zero (0) is the highest possible priority (highest urgency) and larger priority numbers denote actually lower-priority interrupts. So for example, interrupt of priority 2 can preempt an interrupt with priority 3, but interrupt of priority 3 cannot preempt interrupt of priority 3. The default value of priority of all interrupts out of reset is zero (0).

#### Note

Starting with QP/C 5.9.x, the QF\_init() call sets interrupt priority of all IRQs to the "kernel aware" value **QF\_BASEPRI**. Still, it is highly recommended to set the priority of all interrupts used by an application **explicitly**, preferably in the QF\_onStartup().

#### Attention

Some 3rd-party libraries (e.g., STM32Cube) change the interrupt priorities and sometimes priority grouping internally and unexpectedly, so care must be taken to change the priorities back to the appropriate values right before running the application.

The CMSIS provides the function `NVIC_SetPriority()` which you should use to set priority of every interrupt.

#### Note

The priority scheme passed to `NVIC_SetPriority()` is different again than the values stored in the NVIC registers, as shown in the figures above as "CMSIS priorities"

#### Assigning Interrupt Priorities

The [example projects](#) included in the QP/C distribution the recommended way of assigning interrupt priorities in your applications. The initialization consist of two steps: (1) you enumerate the "kernel-unaware" and "kernel-aware" interrupt priorities, and (2) you assign the priorities by calling the `NVIC_SetPriority()` CMSIS function. The following snippet of code illustrates these steps with the explanation section following immediately after the code.

*Listing: Assigning the interrupt priorities (see file bsp.c in the example projects)*

```
//!!!!!!!!!!!!!! CAUTION !!!!!!!!!!!!!!!
// Assign a priority to EVERY ISR explicitly by calling NVIC_SetPriority().
// DO NOT LEAVE THE ISR PRIORITIES AT THE DEFAULT VALUE!
//
[1] enum KernelUnawareISRs { // see NOTE0
    // ...
[2]     MAX_KERNEL_UNAWARE_CMSIS_PRI // keep always last
};
// "kernel-unaware" interrupts can't overlap "kernel-aware" interrupts
[3] Q_ASSERT_COMPILE(MAX_KERNEL_UNAWARE_CMSIS_PRI <= QF_AWARE_ISR_CMSIS_PRI);

[4] enum KernelAwareISRs {
[5]     GPIOPORTA_PRI = QF_AWARE_ISR_CMSIS_PRI, // see NOTE00
        SYSTICK_PRI,
    // ...
[6]     MAX_KERNEL_AWARE_CMSIS_PRI // keep always last
```

```

};

// "kernel-aware" interrupts should not overlap the PendSV priority
[7] Q_ASSERT_COMPILE(MAX_KERNEL_AWARE_CMSIS_PRI <= (0xFF»(8-__NVIC_PRIO_BITS)));

~ ~ ~

[8] void QF_onStartup(void) {
    // set up the SysTick timer to fire at BSP_TICKS_PER_SEC rate
    SysTick_Config(ROM_SysCtlClockGet() / BSP_TICKS_PER_SEC);

    // assign all priority bits for preemption-prio. and none to sub-prio.
[9]   NVIC_SetPriorityGrouping(OU);

    // set priorities of ALL ISRs used in the system, see NOTE00
    //
    // !!!!!!! CAUTION !!!!!!!
    // Assign a priority to EVERY ISR explicitly by calling NVIC_SetPriority().
    // DO NOT LEAVE THE ISR PRIORITIES AT THE DEFAULT VALUE!
    //
[10]  NVIC_SetPriority(SysTick_IRQn,     SYSTICK_PRIO);
[11]  NVIC_SetPriority(GPIOPortA_IRQn,  GPIOPORTA_PRIO);
    ~ ~ ~
    // enable IRQs...
[12]  NVIC_EnableIRQ(GPIOPortA_IRQn);
}

```

- [1] The enumeration KernelUnawareISRs lists the priority numbers for the "kernel-unaware" interrupts. These priorities start with zero (highest possible). The priorities are suitable as the argument for the NVIC\_SetPriority() CMSIS function.

#### Note

The NVIC allows you to assign the same priority level to multiple interrupts, so you can have more ISRs than priority levels running as "kernel-unaware" or "kernel-aware" interrupts.

- [2] The last value in the enumeration MAX\_KERNEL\_UNAWARE\_CMSIS\_PRI keeps track of the maximum priority used for a "kernel-unaware" interrupt.

- [3] The compile-time assertion ensures that the "kernel-unaware" interrupt priorities do not overlap the "kernel-aware" interrupts, which start at QF\_AWARE\_ISR\_CMSIS\_PRI.

- [4] The enumeration KernelAwareISRs lists the priority numbers for the "kernel-aware" interrupts.

- [5] The "kernel-aware" interrupt priorities start with the QF\_AWARE\_ISR\_CMSIS\_PRI offset, which is provided in the [qp\\_port.h](#) header file.

- [6] The last value in the enumeration MAX\_KERNEL\_AWARE\_CMSIS\_PRI keeps track of the maximum priority used for a "kernel-aware" interrupt.

- [7] The compile-time assertion ensures that the "kernel-aware" interrupt priorities do not overlap the lowest priority level reserved for the PendSV exception.

- [8] The QF\_onStartup() callback function is where you set up the interrupts.

- [9] This call to the CMSIS function NVIC\_SetPriorityGrouping() assigns all the priority bits to be preempt priority bits, leaving no priority bits as subpriority bits to preserve the direct relationship between the interrupt priorities and the ISR preemption rules. This is the default configuration out of reset for the ARM Cortex-M3/M4 cores, but it can be changed by some vendor-supplied startup code. To avoid any surprises, the call to NVIC\_SetPriorityGrouping(OU) is recommended.

- [10] The interrupt priorities fall all interrupts ("kernel-unaware" and "kernel-aware" alike) are set explicitly by calls to the CMSIS function NVIC\_SetPriority().

[11] All used IRQ interrupts need to be explicitly enabled by calling the CMSIS function.

### Interrupts and the FPU (ARMv7M or higher architectures)

The QP/C ports described in this section support also the ARMv7M or higher architectures. Compared to all other members of the Cortex-M family, these cores includes the single precision variant of the ARMv7-M Floating-Point Unit (Fpv4-SP). The hardware FPU implementation adds an extra floating-point register bank consisting of S0-S31 and some other FPU registers. This FPU register set represents additional context that need to be preserved across interrupts and thread switching (e.g., in the preemptive **QK** kernel).

The ARM VFP has a very interesting feature called **lazy stacking** [ARM-AN298]. This feature avoids an increase of interrupt latency by skipping the stacking of floating-point registers, if not required, that is:

- if the interrupt handler does not use the FPU, or
- if the interrupted program does not use the FPU.

If the interrupt handler has to use the FPU and the interrupted context has also previously used by the FPU, then the stacking of floating-point registers takes place at the point in the program where the interrupt handler first uses the FPU. The lazy stacking feature is programmable and by default it is turned ON.

#### Note

All built-in kernels in QP/C are designed to take advantage of the lazy stacking feature [ARM-AN298].

#### 3.3.1.3 References

- [ARM AN298] ARM Application Note 298 "Cortex-M4 (F) Lazy Stacking and Context Switching"; ARM 2012↑
- [ARM-EPM-064408] ARM Processor Cortex-M7 (AT610) and Cortex-M7 with FPU (AT611) Software Developers Errata Notice↑
- [Reminder] Reminder State Pattern↑

#### 3.3.1.4 Non-preemptive QV Kernel

The non-preemptive **QV** kernel executes active objects one at a time, with priority-based scheduling performed after run-to-completion (RTC) processing of each event. Due to naturally short duration of event processing in state machines, the simple **QV** kernel is often adequate for many real-time systems. (NOTE: Long RTC steps can be often broken into shorter pieces by means of the "Reminder" state pattern [Reminder])

### Remarks

In the **QV** port, the only components requiring platform-specific porting are **QF** and **QV** itself. The other two components: QEP and **QS** require merely recompilation and will not be discussed here. With the **QV** port you're not using the **QK** or **QXK** kernels. The **QV** port to ARM Cortex-M is located in the folder `/ports/arm-cm/qv/`.

### Synopsis of the **QV** Port on ARM Cortex-M

The non-preemptive **QV** kernel works essentially as the traditional foreground-background system (a.k.a. "superloop") in that all active objects are executed in the main loop and interrupts always return back to the point of preemption. To avoid race conditions between the main loop and the interrupts, **QV** briefly disables interrupts.

1. The ARM Cortex-M processor executes application code (the main loop) in the Privileged Thread mode, which is exactly the mode entered out of reset.
2. The exceptions (including all interrupts) are always processed in the Privileged Handler mode.
3. **QV** uses only the Main Stack Pointer. The Process Stack Pointer is not used and is not initialized.
4. ARM Cortex-M enters interrupt context without disabling interrupts (without setting the PRIMASK bit or the BASEPRI register). Generally, you should not disable interrupts inside your ISRs. In particular, the QP services **QF\_PUBLISH()**, **QF\_TICK\_X()**, and **QACTIVE\_POST()** should be called with **interrupts enabled**, to avoid nesting of critical sections.

### Note

If you don't wish an interrupt to be preempted by another interrupt, you can always prioritize that interrupt in the NVIC to a higher level (use a lower numerical value of priority).

1. The `QF_init()` function calls the function `QV_init()` to set the interrupt priority of all IRQs available in the MCU to the safe value of `QF_BASEPRI` (for ARM-v7 architecture).

### The `qp_port.h` Header File

The **QF** header file for the ARM Cortex-M port is located in `/ports/arm-cm/qv/gnu/qp_port.h`. This file specifies the interrupt disabling policy (**QF** critical section) as well as the configuration constants for **QF** (see Chapter 8 in [PSICC2]).

### Note

The ARM Cortex-M allows you to use the simplest "unconditional interrupt disabling" policy (see Section 7.3.2 in [PSICC2]), because ARM Cortex-M is equipped with the standard nested vectored interrupt controller (NVIC) and generally runs ISRs with interrupts enabled (so the body of an ISR is not a critical section).

The following listing shows the `qp_port.h` header file for ARM Cortex-M with the GNU-ARM toolchain. Other toolchains use slightly different conditional compilation macros to select the Cortex-M variants, but implement the same policies.

*Listing: The `qp_port.h` header file for ARM Cortex-M*

```

#include <stdint.h> /* Exact-width types. WG14/N843 C99 Standard */
#include <stdbool.h> /* Boolean type. WG14/N843 C99 Standard */

/* The maximum number of active objects in the application, see NOTE1 */
[1] #define QF_MAX_ACTIVE 32

/* The maximum number of system clock tick rates */
[2] #define QF_MAX_TICK_RATE 2

/* QF interrupt disable/enable and log2()... */
[3] #if (__ARM_ARCH == 6) /* Cortex-M0/M0+/M1(v6-M, v6S-M)? */

    /* Cortex-M0/M0+/M1(v6-M, v6S-M) interrupt disabling policy, see NOTE2 */
[4] #define QF_INT_DISABLE() __asm volatile ("cpsid i")
[5] #define QF_INT_ENABLE() __asm volatile ("cpsie i")

    /* QF critical section (unconditional interrupt disabling) */
[6] #define QF_CRIT_STAT
[7] #define QF_CRIT_ENTRY() QF_INT_DISABLE()
[8] #define QF_CRIT_EXIT() QF_INT_ENABLE()

    /* CMSIS threshold for "QF-aware" interrupts, see NOTE2 and NOTE5 */
[9] #define QF_AWARE_ISR_CMSIS_PRI 0

    /* hand-optimized LOG2 in assembly for Cortex-M0/M0+/M1(v6-M, v6S-M) */
[10] #define QF_LOG2(n_) QF_qlog2((n_))

[11] #else /* Cortex-M3/M4/M7 */

    /* Cortex-M3/M4/M7 alternative interrupt disabling with PRIMASK */
[12] #define QF_PRIMASK_DISABLE() __asm volatile ("cpsid i")
[13] #define QF_PRIMASK_ENABLE() __asm volatile ("cpsie i")

    /* Cortex-M3/M4/M7 interrupt disabling policy, see NOTE3 and NOTE4 */
[14] #define QF_INT_DISABLE() __asm volatile (\n
        "cpsid i\n" "msr BASEPRI,%0\n" "cpsie i" :: "r" (QF_BASEPRI) : )
[15] #define QF_INT_ENABLE() __asm volatile (\n
        "msr BASEPRI,%0" :: "r" (0) : )

    /* QF critical section (unconditional interrupt disabling) */
[16] #define QF_CRIT_STAT
[17] #define QF_CRIT_ENTRY() QF_INT_DISABLE()
[18] #define QF_CRIT_EXIT() QF_INT_ENABLE()

    /* BASEPRI threshold for "QF-aware" interrupts, see NOTE3 */
[19] #define QF_BASEPRI 0x3F

    /* CMSIS threshold for "QF-aware" interrupts, see NOTE5 */
[20] #define QF_AWARE_ISR_CMSIS_PRI (QF_BASEPRI >> (8 - __NVIC_PRIO_BITS))

    /* Cortex-M3/M4/M7 provide the CLZ instruction for fast LOG2 */
[21] #define QF_LOG2(n_) ((uint_fast8_t)(32U - __builtin_clz(n_)))

#endif

[22] #define QF_CRIT_EXIT_NOP() __asm volatile ("isb")

#if (__ARM_ARCH == 6) /* Cortex-M0/M0+/M1(v6-M, v6S-M)? */
    /* hand-optimized quick LOG2 in assembly */
[23] uint_fast8_t QF_qlog2(uint32_t x);
#endif /* Cortex-M0/M0+/M1(v6-M, v6S-M) */

// include files -----
#include "qqueue.h" // QV kernel uses the native QP event queue
#include "qmpool.h" // QV kernel uses the native QP memory pool
#include "qp.h" // %QP Framework
#include "qv.h" // QV kernel

```

[1] The `QF_MAX_ACTIVE` specifies the maximum number of active object priorities in the application. You always need to provide this constant. Here, `QF_MAX_ACTIVE` is set to 32, but it can be increased up to the maximum limit of 63 active object priorities in the system.

### Note

The `qp_port.h` header file does not change the default settings for all the rest of various object sizes inside `QF`. Please refer to Chapter 8 of [PSiCC2] for discussion of all configurable `QF` parameters.

[2] The macro `QF_MAX_TICK_RATE` specifies the maximum number of clock tick rates for QP/C time events. If you do not need to specify this limit, in which case the default of a single clock rate will be chosen.

[3] As described in the previous [Section](#), the interrupt disabling policy for the ARMv6-M architecture (Cortex-M0/M0+) is different than the policy for the ARMv7-M. In GNU-ARM, the macro `__ARM_ARCH` is defined as 6 for the ARMv6-M architecture (Cortex-M0/M0+), and 7 for ARMv7-M (Cortex-M3/M4/M4F).

#### Note

The `__ARM_ARCH` macro is specific to the GNU-ARM compiler. Other compilers for ARM Cortex-M provide different macros to detect the CPU type.

[4–5] For the ARMv6-M architecture, the interrupt disabling policy uses the PRIMASK register to disable interrupts globally. The `QF_INT_DISABLE()` macro resolves in this case to the inline assembly instruction "CPSD i", which sets the PRIMASK. The `QF_INT_ENABLE()` macro resolves to the inline assembly instruction "CPSE i", which clears the PRIMASK.

[6] The `QF_CRIT_STAT` is empty, meaning that the critical section uses the simple policy of "unconditional interrupt disabling".

#### Note

The "unconditional interrupt disabling" policy precludes nesting of critical sections, but this is not needed for ARM Cortex-M, because this CPU never disables interrupts, even when handling exceptions/interrupts.

[7] The `QF_CRIT_ENTRY()` enters a critical section. Interrupts are disabled by setting the PRIMASK register.

[8] The `QF_CRIT_EXIT()` macro leaves the critical section. Interrupts are unconditionally re-enabled by clearing the PRIMASK register.

[9] For the ARMv6-M architecture, the `QF_AWARE_ISR_CMSIS_PRI` priority level is defined as zero, meaning that all interrupts are "kernel-aware", because all interrupt priorities are disabled by the kernel.

[10] The `QF_LOG2()` macro is defined as a call to the function `QF_qlog2()` ("quick log-base-2 logarithm"). This function is coded in hand-optimized assembly, which always takes only 14 CPU cycles to execute (see also label [23]).

#### Note

ARM Cortex-M0/M0+ does NOT implement the `CLZ` instruction. Therefore the log-base-2 calculation cannot be accelerated in hardware, as it is for ARM Cortex-M3 and higher.

[11] For the ARMv7-M (Cortex-M3/M4/M4F) architecture...

[12] The `QF_PRIMASK_DISABLE()` macro resolves to the inline assembly instruction `CPSD i`, which sets the PRIMASK.

[13] The `QF_PRIMASK_ENABLE()` macro resolves to the inline assembly instruction `CPSE i`, which clears the PRIMASK.

[14] Interrupts are disabled by setting the BASEPRI register to the value defined in the `QF_BASEPRI` macro (see label [19]). This setting of the BASEPRI instruction `msr BASEPRI, ...` is surrounded by setting and clearing the PRIMASK register, as a workaround a hardware problem in ARMv7M or higher architectures core r0p1:

**Note**

The selective disabling of "QF-aware" interrupts with the BASEPRI register has a problem on ARMv7M or higher architectures core r0p1 (see [ARM-EPM-064408], Erratum 837070). The workaround recommended by ARM is to surround MSR BASEPRI, . . . with the CPSID i/CPSIE i pair, which is implemented in the [QF\\_INT\\_DISABLE\(\)](#) macro. This workaround works also for Cortex-M3/M4 cores.

[15] The [QF\\_INT\\_ENABLE\(\)](#) macro sets the BASEPRI register to zero, which disables BASEPRI interrupt masking.

**Note**

This method can never disable interrupt of priority 0 (highest).

[16] The [QF\\_CRIT\\_STAT](#) is empty, meaning that the critical section uses the simple policy of "unconditional interrupt disabling".

**Note**

The "unconditional interrupt disabling" policy precludes nesting of critical sections, but this is not needed for ARM Cortex-M, because this CPU never disables interrupts, even when handling exceptions/interrupts.

[17] The [QF\\_CRIT\\_ENTRY\(\)](#) enters a critical section. Interrupts are disabled with the macro [QF\\_INT\\_DISABLE\(\)](#) defined at label [12].

[18] The [QF\\_CRIT\\_EXIT\(\)](#) macro leaves the critical section. Interrupts are unconditionally re-enabled with the macro [QF\\_INT\\_ENABLE\(\)](#) defined at label [13].

[19] The [QF\\_BASEPRI](#) value is defined such that it is the lowest priority for the minimum number of 3 priority-bits that the ARMv7M or higher architectures architecture must provide. This partitions the interrupts as "kernel-unaware" and "kernel-aware" interrupts, as shown in section arm-cm\_int-assign.

[20] For the ARMv7-M architecture, the [QF\\_AWARE\\_ISR\\_CMSIS\\_PRI](#) priority level suitable for the CMSIS function [NVIC\\_SetPriority\(\)](#) is determined by the [QF\\_BASEPRI](#) value.

[21] The macro [QF\\_LOG2\(\)](#) is defined to take advantage of the CLZ instruction (Count Leading Zeroes), which is available in the ARMv7-M architecture.

**Note**

The [\\_\\_builtin\\_cls\(\)](#) intrinsic function is specific to the GNU-ARM compiler. Other compilers for ARM Cortex-M use different function names for this intrinsic function.

[22] The macro [QF\\_CRIT\\_EXIT\\_NOP\(\)](#) provides the protection against merging two critical sections occurring back-to-back in the QP/C code.

[23] For ARMv6 architecture, the prototype of the quick, hand-optimized log-base-2 function is provided (see also label [10]).

**The qv\_port.h Header File**

The [QV](#) header file for the ARM Cortex-M port is located in `/ports/arm-cm/qv/gnu/qv_port.h`. This file provides the macro [QV\\_CPU\\_SLEEP\(\)](#), which specifies how to enter the CPU sleep mode safely in the non-preemptive [QV](#) kernel (see also Section 4.7) and [Samek 07]).

**Note**

To avoid race conditions between interrupts waking up active objects and going to sleep, the non-preemptive **QV** kernel calls the [QV\\_CPU\\_SLEEP\(\)](#) callback with interrupts disabled.

[1] For the ARMv6-M architecture, the macro [QV\\_CPU\\_SLEEP\(\)](#) stops the CPU with the WFI instruction (Wait For Interrupt). After the CPU is woken up by an interrupt, interrupts are re-enabled with the PRIMASK.

[2] For the ARMv7-M architecture, the macro [QV\\_CPU\\_SLEEP\(\)](#) first disables interrupts by setting the PRIMASK, then clears the BASEPRI to enable all "kernel-aware" interrupts and only then stops the CPU with the WFI instruction (Wait For Interrupt). After the CPU is woken up by an interrupt, interrupts are re-enabled with the PRIMASK. This sequence is necessary, because the ARMv7M or higher architectures cannot be woken up by any interrupt blocked by the BASEPRI register.

[3] The macro [QV\\_INIT\(\)](#) is defined as a call to the [QV\\_init\(\)](#) function, which means that this function will be called from [QF\\_init\(\)](#). The [QV\\_init\(\)](#) function initializes all available IRQ priorities in the MCU to the safe value of [QF\\_BASEPRI](#).

**The qv\_port.c Implementation File**

The **QV** implementation file for the ARM Cortex-M port is located in `/ports/arm-cm/qv/gnu/qf_port.c`. This file defines the function [QV\\_init\(\)](#), which for the ARMv7-M architecture sets the interrupt priorities of all IRQs to the safe value [QF\\_BASEPRI](#).

*Listing: The qv\_port.c header file for ARM Cortex-M*

```
#include "qp_port.h"

[1] #if (__ARM_ARCH != 6) /* NOT Cortex-M0/M0+/M1 ? */

#define SCnSCB_ICTR ((uint32_t volatile *)0xE000E004)
#define SCB_SYSPRI ((uint32_t volatile *)0xE000ED14)
#define NVIC_IP ((uint32_t volatile *)0xE000E400)

void QV_init(void) {
    uint32_t n;

    /* set exception priorities to QF_BASEPRI...
     * SCB_SYSPRI1: Usage-fault, Bus-fault, Memory-fault
     */
[2]    SCB_SYSPRI[1] |= (QF_BASEPRI << 16) | (QF_BASEPRI << 8) | QF_BASEPRI;

    /* SCB_SYSPRI2: SVCall */
[3]    SCB_SYSPRI[2] |= (QF_BASEPRI << 24);

    /* SCB_SYSPRI3: SysTick, PendSV, Debug */
[4]    SCB_SYSPRI[3] |= (QF_BASEPRI << 24) | (QF_BASEPRI << 16) | QF_BASEPRI;

    /* set all implemented IRQ priorities to QF_BASEPRI... */
[5]    n = 8 + (*SCnSCB_ICTR << 3); /* # interrupt priority registers */
    do {
        --n;
[6]        NVIC_IP[n] = (QF_BASEPRI << 24) | (QF_BASEPRI << 16)
                     | (QF_BASEPRI << 8) | QF_BASEPRI;
    } while (n != 0);
}

#endif /* NOT Cortex-M0/M0+/M1 */
```

[1] For the ARMv7-M architecture (Cortex-M3/M4/M7)...

[2] The exception priorities for User-Fault, Bus-Fault, and Mem-Fault are set to the value [QF\\_BASEPRI](#).

[3] The exception priority for SVCCall is set to the value [QF\\_BASEPRI](#).

[4] The exception priority for SysTick, PendSV, and Debug is set to the value [QF\\_BASEPRI](#).

[5] The number of implemented IRQs is read from the SCnSCB\_ICTR register

[6] The interrupt priority of all implemented IRQs is set to the safe value [QF\\_BASEPRI](#) in a loop.

**Writing ISRs for QV**

The ARM Cortex-M CPU is designed to use regular C functions as exception and interrupt service routines (ISRs).

### Note

The ARM EABI (Embedded Application Binary Interface) requires the stack be 8-byte aligned, whereas some compilers guarantee only 4-byte alignment. For that reason, some compilers (e.g., GNU-ARM) provide a way to designate ISR functions as interrupts. For example, the GNU-ARM compiler provides the `attribute((interrupt))` designation that will guarantee the 8-byte stack alignment.

Typically, ISRs are application-specific (with the main purpose to produce events for active objects). Therefore, ISRs are not part of the generic QP/C port, but rather part of the BSP (Board Support Package).

The following listing shows an example of the SysTick\_Handler() ISR (from the DPP example application). This ISR calls the `QF_TICK_X()` macro to perform QF time-event management.

*Listing: An ISR header for QV*

```
void SysTick_Handler(void) __attribute__((__interrupt__));
void SysTick_Handler(void) {
    ~ ~ ~
    // process time events at rate 0
    QTIMEEVNT_TICK_X(0U, &l_SysTick_Handler);
}
```

### Note

The QP/C port to ARM Cortex-M complies with the CMSIS standard, which dictates the names of all exception handlers and IRQ handlers.

### Using the FPU in the QV Port

If you use ARMv7M or higher CPU and your application uses the hardware FPU, it should be enabled because it is turned off out of reset. The CMSIS-compliant way of turning the FPU on looks as follows:

```
SCB->CPACR |= (0xFU << 20);
```

### Note

The FPU must be enabled before executing any floating point instruction. An attempt to execute a floating point instruction will fault if the FPU is not enabled.

If the FPU is configured in the project, the `QV` kernel initializes the FPU to use the automatic state preservation and the lazy stacking feature as follows:

```
FPU->FPCCR |= (1U << FPU_FPCCR_ASSEN_Pos) | (1U << FPU_FPCCR_LSPEN_Pos);
```

### FPU NOT used in the ISRs

If you use the FPU only at the thread-level (inside active objects) and none of your ISRs use the FPU, you can setup the FPU not to use the automatic state preservation and not to use the lazy stacking feature as follows:

```
FPU->FPCCR &= ~((1U << FPU_FPCCR_ASSEN_Pos) | (1U << FPU_FPCCR_LSPEN_Pos));
```

With this setting, the processor uses only the standard 8-register interrupt stack frame with R0-R3,R12,LR,PC,xPSR. This scheme is the fastest and incurs no additional CPU cycles to save and restore the FPU registers.

**Attention**

This FPU setting will lead to FPU errors, if any of the ISRs indeed starts to use the FPU

**QV Idle Processing Customization in QV\_onIdle()**

When no events are available, the non-preemptive **QV** kernel invokes the platform-specific callback function **QV\_onIdle()**, which you can use to save CPU power, or perform any other "idle" processing (such as Quantum Spy software trace output).

**Note**

The idle callback **QV\_onIdle()** must be invoked with interrupts disabled, because the idle condition can be changed by any interrupt that posts events to event queues. **QV\_onIdle()** must internally enable interrupts, ideally atomically with putting the CPU to the power-saving mode (see also [Samek 07] and Chapter 7 in [PSiCC2]).

Because **QV\_onIdle()** must enable interrupts internally, the signature of the function depends on the interrupt locking policy. In case of the simple "unconditional interrupt locking and unlocking" policy, which is used in this ARM Cortex-M port, the **QV\_onIdle()** takes no parameters. Listing 6 shows an example implementation of **QV\_onIdle()** for the TM4C MCU. Other ARM Cortex-M embedded microcontrollers (e.g., NXP's LPC1114/1343) handle the power-saving mode very similarly.

*Listing: QV\_onIdle() for ARM Cortex-M*

```
[1] void QV_onIdle(void) { /* entered with interrupts DISABLED, see NOTE01 */
    ~ ~ ~
[2] #if defined NDEBUG
    /* Put the CPU and peripherals to the low-power mode */
[3]     QV_CPU_SLEEP(); /* atomically go to sleep and enable interrupts */
[4] #else
    QF_INT_ENABLE(); /* just enable interrupts */
#endif
}
```

[1] The non-preemptive **QV** kernel calls the **QV\_onIdle()** callback with interrupts disabled, to avoid race condition with interrupts that can post events to active objects and thus invalidate the idle condition.

[2] The sleep mode is used only in the non-debug configuration, because sleep mode stops CPU clock, which can interfere with debugging.

[3] The macro **QV\_CPU\_SLEEP()** is used to put the CPU to the low-power sleep mode safely. The macro **QV\_CPU\_SLEEP()** is defined in the **qv\_port.h** header file for the **QV** kernel and depends on the interrupt disabling policy used.

[4] When a sleep mode is not used, the **QV\_onIdle()** callback simply re-enables interrupts.

### 3.3.1.5 Preemptive Non-Blocking QK Kernel

The **preemptive, non-blocking QK kernel** is specifically designed to execute non-blocking active objects. **QK** runs active objects in the same way as prioritized interrupt controller (such as NVIC in ARM Cortex-M) runs interrupts using the **single stack** (MSP on Cortex-M). This section explains how the **preemptive non-blocking QK kernel** works on ARM Cortex-M.

## Remarks

In a **QK** port, the only components requiring platform-specific porting are **QF** and **QV** itself. The other two components: **QEP** and **QS** require merely recompilation and will not be discussed here. With the **QV** port you're not using the **QV** or **QXK** kernels. The **QK** port to ARM Cortex-M is located in the folder `/ports/arm-cm/qk/`.

## Synopsis of the **QK** Port on ARM Cortex-M

The ARM Cortex-M architecture is designed primarily for the traditional real-time kernels that use multiple per-thread stacks. Therefore, implementation of the non-blocking, single-stack kernel like **QK** is a bit more involved on Cortex-M than other CPUs and works as follows:

- The ARM Cortex-M processor executes the **QK** application code (active objects) in the Privileged Thread mode, which is exactly the mode entered out of reset. The exceptions (including all interrupts) are always processed in the Privileged Handler mode.
- **QK** uses only the Main Stack Pointer (**QK** is a single stack kernel). The Process Stack Pointer is not used and is not initialized.
- ARM Cortex-M enters interrupt context without disabling interrupts (without setting the PRIMASK bit or the BASEPRI register). Generally, you should not disable interrupts inside your ISRs. In particular, the QP/C services **QF\_PUBLISH()**, **QF\_TICK\_X()**, and **QACTIVE\_POST()** should be called with interrupts enabled, to avoid nesting of critical sections. (NOTE: If you don't wish an interrupt to be preempted by another interrupt, you can always prioritize that interrupt in the NVIC to a higher level – use a lower numerical value of priority).
- The **QK** port uses the PendSV exception (number 14) to perform asynchronous preemption (see Chapter 10 in [PSiCC2↑](#)). The startup code must initialize the Interrupt Vector Table with the addresses of `PendSV_Handler()` exception handler.

## Note

**QK** uses only the CMSIS-compliant exception and interrupt names, such as `PendSV_Handler`

- The **QK** port uses the NMI exception (number 2) or any unused IRQ interrupt to *return* to the preempted thread (see Chapter 10 in [PSiCC2↑](#)). The startup code must initialize the Interrupt Vector Table with the addresses of `NMI_Handler()` and `<IRQ-name>_IRQHandler()` exception handlers.

## Note

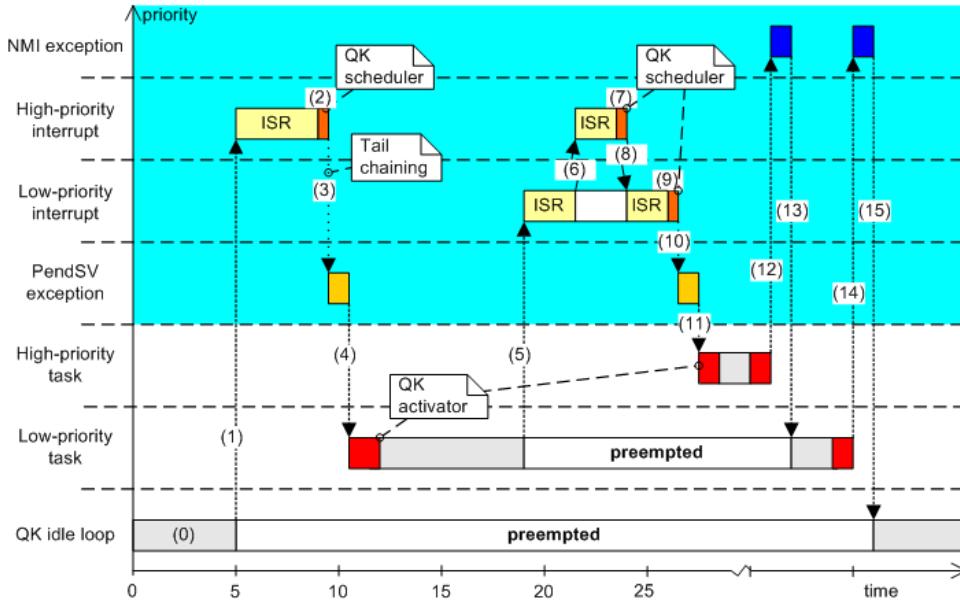
In case the NMI exception is needed for something else, the **QK** allows the developers to configure any otherwise unused IRQ to be used instead. This is accomplished by the pair of macros: `QK_USE_IRQ_NUM` and `QK_USE_IRQ_HANDLER`.

The **QK** port specifically does **not** use the SVC exception (Supervisor Call). This makes the **QK** ports compatible with various "hypervisors" (such as mbed uVisor or Nordic SoftDevice), which use the SVC exception.

- The `QF_init()` function calls the function `QK_init()` to set the priority of the PendSV exception to the lowest level in the whole system (0xFF). The function `QK_init()` additionally sets the interrupt priority of all IRQs available in the MCU to the safe value of `QF_BASEPRI` (for ARM-v7 architecture).
- It is strongly recommended that you do not assign the lowest priority (0xFF) to any interrupt in your application. With 3 MSB-bits of priority, this leaves the following 7 priority levels for you (listed from the lowest to the highest urgency): 0xC0, 0xA0, 0x80, 0x60, 0x40, 0x20, and 0x00 (the highest priority).

- Before returning, every "kernel aware" ISR must check whether an active object has been activated that has a higher priority than the currently running active object. If this is the case, the ISR must set the PendSV pending flag in the NVIC. All this is accomplished in the macro `QK_ISR_EXIT()`, which must be called just before exiting every ISRs.
  - In ARM Cortex-M the whole prioritization of interrupts, including the PendSV exception, is performed entirely by the NVIC. Because the PendSV has the lowest priority in the system, the NVIC tail-chains to the PendSV exception only after exiting the last nested interrupt.
  - The pushing of the 8 registers comprising the ARM Cortex-M interrupt stack frame upon entry to the preempted thread (NMI or IRQ) is wasteful in a single-stack kernel, but is necessary to perform full interrupt return to the preempted context through the exception return.

## Preemption Scenarios in QK on ARM Cortex-M



## *Several preemption scenarios in QK*

[ 0 ] The timeline begins with the QK executing the idle loop.

[1] At some point an interrupt occurs and the CPU immediately suspends the idle loop, pushes the interrupt stack frame to the Main Stack and starts executing the ISR.

[2] The ISR performs its work, and in QK always must call the `QK_ISR_EXIT()` macro, which calls the QK scheduler (`QK_sched()`) to determine if there is a higher-priority AO to run. If so, the macro sets the pending flag for the PendSV exception in the NVIC. The priority of the PendSV exception is configured to be the lowest of all exceptions (0xFF), so the ISR continues executing and PendSV exception remains pending. At the ISR return, the ARM Cortex-M CPU performs tail-chaining to the pending PendSV exception.

[3] The PendSV exception synthesize an exception stack frame to return to the QK "activator" (QK\_activate\_()) to run this new thread.

### Note

The [QK](#) activator must run in the thread context, while PendSV executes in the exception context. The change of the context is accomplished by returning from the PendSV exception directly to the [QK](#) "activator".

To return directly to the [QK](#) activator, PendSV synthesizes an exception stack frame, which contains the exception return address set to `QK_activate_()`. The [QK](#) activator activates the Low-priority thread (discovered by the [QK](#) scheduler `QK_sched()`). The [QK](#) activator enables interrupts and launches the Low-priority thread, which is simply a C-function call in [QK](#). The Low-priority thread (active object) starts running.

[ 4 ] Some time later a low-priority interrupt occurs. The Low-priority thread is suspended and the CPU pushes the interrupt stack frame to the Main Stack and starts executing the ISR.

[ 5 ] Before the Low-priority ISR completes, it too gets preempted by a High-priority ISR. The CPU pushes another interrupt stack frame and starts executing the High-priority ISR.

[ 6 ] The High-priority ISR sets the pending flag for the PendSV exception by means of the [QK\\_ISR\\_EXIT\(\)](#) macro. When the High-priority ISR returns, the NVIC does not tail-chain to the PendSV exception, because a higher-priority ISR than PendSV is still active. The NVIC performs an exception return to the preempted Low-priority interrupt, which finally completes.

[ 7 ] Upon the exit from the Low-priority ISR, it too sets the pending flag for the PendSV exception by means of the [QK\\_ISR\\_EXIT\(\)](#) macro. The PendSV is already pended from the High-priority interrupt, so pending is again redundant, but it is not an error. At the ISR return, the ARM Cortex-M CPU performs tail-chaining to the pending PendSV exception.

[ 8 ] The PendSV exception synthesizes an interrupt stack frame to return to the [QK](#) activator. The [QK](#) activator detects that the High-priority thread is ready to run and launches the High-priority thread (normal C-function call). The High-priority thread runs to completion and returns to the activator.

[ 9 ] The [QK](#) activator does not find any more higher-priority threads to execute and needs to return to the preempted thread. The only way to restore the interrupted context in ARM Cortex-M is through the interrupt return, but the thread is executing outside of the interrupt context (in fact, threads are executing in the Privileged Thread mode). The thread enters the Handler mode by pending the NMI or IRQ exception.

[ 10 ] The only job of the NMI or IRQ exception is to discard its own interrupt stack frame, re-enable interrupts, and return using the interrupt stack frame that has been on the stack from the moment of thread preemption.

[ 11 ] The Low-priority thread, which has been preempted all that time, resumes and finally runs to completion and returns to the [QK](#) activator. The [QK](#) activator does not find any more threads to launch and causes the NMI or IRQ exception to return to the preempted thread.

[ 12 ] The NMI or IRQ exception discards its own interrupt stack frame and returns using the interrupt stack frame from the preempted thread context

### The [qp\\_port.h](#) Header File

The [QF](#) header file for the ARM Cortex-M port is located in `/ports/arm-cm/qk/gnu/qp_port.h`. This file is almost identical to the QV port, except the header file in the [QK](#) port includes `qk_port.h` header file instead of `qv_porth`. The most important function of `qk_port.h` is specifying interrupt entry and exit.

**Note**

As any preemptive kernel, **QK** needs to be notified about entering the interrupt context and about exiting an interrupt context in order to perform a context switch, if necessary.

*Listing: qk\_port.h header file for ARM Cortex-M*

```
// determination if the code executes in the ISR context
[1] #define QK_ISR_CONTEXT_() (QK_get_IPSR() != (uint32_t)0)

__attribute__((always_inline))
[2] static inline uint32_t QK_get_IPSR(void) {
    uint32_t regIPSR;
    __asm volatile ("mrs %0,ipsr" : "=r" (regIPSR));
    return regIPSR;
}

// QK interrupt entry and exit
[3] #define QK_ISR_ENTRY() ((void)0)

[4] #define QK_ISR_EXIT() do { \
[5]     QF_INT_DISABLE(); \
[6]     if (QK_sched_() != (uint_fast8_t)0) { \
[7]         (*Q_UINT2PTR_CAST(uint32_t, 0xE000ED04U) = (uint32_t)(1U << 28)); \
[8]     } \
[9]     QF_INT_ENABLE(); \
} while (0)

// initialization of the QK kernel
[9] #define QK_INIT() QK_init()
void QK_init(void);

#include "qk.h" // QK platform-independent public interface
```

- [1] The macro `QK_ISR_CONTEXT()` returns true when the code executes in the ISR context and false otherwise. The macro takes advantage of the ARM Cortex-M register IPSR, which is non-zero when the CPU executes an exception (or interrupt) and is zero when the CPU is executing thread code.

**Note**

**QK** needs to distinguish between ISR and thread contexts, because threads need to perform synchronous context switch (when a higher-priority thread becomes ready to run), while ISRs should not do that.

- [2] The inline function `QK_get_IPSR()` obtains the IPSR register and returns it to the caller. This function is defined explicitly for the GNU-ARM toolchain, but many other toolchains provide this function as an intrinsic, built-in facility.

- [3] The `QK_ISR_ENTRY()` macro notifies **QK** about entering an ISR. The macro is empty, because the determination of the ISR vs thread context is performed independently in the `QK_ISR_CONTEXT()` macro (see above).

- [4] The `QK_ISR_EXIT()` macro notifies **QK** about exiting an ISR.

- [5] Interrupts are disabled before calling **QK** scheduler.

- [6] The **QK** scheduler is called to find out whether an active object of a higher priority than the current one needs activation. The `QK_sched_()` function returns non zero value if this is the case.

- [7] If asynchronous preemption becomes necessary, the code sets the PENDSV Pend bit(28) in the ICSR register (Interrupt Control and State Register). The register is mapped at address 0xE000ED04 in all ARM Cortex-M cores.

- [8] The interrupts are re-enabled after they have been disabled in step [5].

### Note

Because the priority of the PendSV exception is the lowest of all interrupts, it is actually triggered only after all nested interrupts exit. The PendSV exception is then entered through the efficient **tail-chaining** process, which eliminates the restoring and re-entering the interrupt context.

### QK Port Implementation for ARM Cortex-M

The QK port to ARM Cortex-M requires coding the PendSV and NMI or IRQ exceptions in assembly. This ARM Cortex-M-specific code, as well as QK initialization (`QK_init()`) is located in the file `ports/arm-cm/qk/gnu/qk_port.c`

### Note

The single assembly module `qk_port.s` contains common code for all Cortex-M variants (Architecture v6M and v7M) as well as options with and without the VFP. The CPU variants are distinguished by conditional compilation, when necessary.

### `QK_init()` Implementation

*Listing: QK\_init() function in qk\_port.c file*

```
[1] void QK_init(void) {
[2] #if __ARM_ARCH != 6 // NOT Cortex-M0/M0+/M1 (v6-M, v6S-M)?
    // set exception priorities to QF_BASEPRI...
    // SCB_SYSPRI1: Usage-fault, Bus-fault, Memory-fault
[3]    SCB_SYSPRI[1] |= (QF_BASEPRI << 16) | (QF_BASEPRI << 8) | QF_BASEPRI;
    // SCB_SYSPRI2: SVCall
[4]    SCB_SYSPRI[2] |= (QF_BASEPRI << 24);
    // SCB_SYSPRI3: SysTick, PendSV, Debug
[5]    SCB_SYSPRI[3] |= (QF_BASEPRI << 24) | (QF_BASEPRI << 16) | QF_BASEPRI;
    // set all implemented IRQ priorities to QF_BASEPRI...
[6]    uint8_t nprio = (8U + ((SCnSCB_ICTR & 0x7U) << 3U))*4;
    for (uint8_t n = 0U; n < nprio; ++n) {
[7]        NVIC_IP[n] = QF_BASEPRI;
    }
#endif // NOT Cortex-M0/M0+/M1 (v6-M, v6S-M)

    // SCB_SYSPRI3: PendSV set to the lowest priority 0xFF
[8]    SCB_SYSPRI[3] |= (0xFFU << 16);

    #ifdef QK_USE_IRQ_NUM
        // The QK port is configured to use a given ARM Cortex-M IRQ #
        // to return to thread mode (default is to use the NMI exception)
[9]        NVIC_IP[QK_USE_IRQ_NUM] = 0U; // priority 0 (highest)
[10]        NVIC_EN[QK_USE_IRQ_NUM / 32U] = (1U << (QK_USE_IRQ_NUM % 32U));
    #endif
}
```

[1] The `QK_init()` function is called from `QF_init()` to perform initialization specific to the QK kernel.

[2] If the ARM Architecture is NOT v6 (Cortex-M0/M0+), that is for ARMv7M or higher architectures, the function initializes the exception priorities of PendSV and NMI as well as interrupt priorities of all IRQs available in a given MCU. (NOTE: for Cortex-M0/M0+, this initialization is not needed, as the CPU does not support the BASEPRI register and the only way to disable interrupts is via the PRIMASK register. In this case, all interrupts are "kernel-aware" and there is no need to initialize interrupt priorities to a safe value.)

[3] Exception priorities of Usage-fault, Bus-fault, and Memory-fault are set to `QF_BASEPRI`.

[4] Exception priorities of SVCall is set to `QF_BASEPRI`.

[5] Exception priorities of SysTick, PendSV and Debug are set to `QF_BASEPRI`.

## Note

The exception priority of PedSV is later changed to 0xFF in step [8]

- [ 6 ] The number of implemented interrupts is extracted from SCnSCB\_ICTR register.
  - [ 7 ] Exception priorities of all implemented interrupts are set to [QF\\_BASEPRI](#).
  - [ 8 ] Exception priority of PendSV is set to 0xFF, which is the lowest interrupt priority in the system.
  - [ 9 ] In case a regular IRQ is configured for returning to the thread mode, the priority of the IRQ is set to zero (highest).
  - [ 10 ] In case a regular IRQ is configured for returning to the thread mode, the IRQ is enabled in the NVIC.

## PendSV\_Handler() Implementation

*Listing: PendSV\_Handler() and Thread\_ret() functions in qk\_port.c file*

```

[1] __attribute__ ((naked))
[2] void PendsV_Handler(void) {
[3] __asm volatile (
[4]     /* Prepare constants in registers before entering critical section */
[5]     " LDR    r3,=" STRINGIFY(NVIC_ICSR) "\n" /* Interrupt Control and State */
[6]     " MOV    r1,#1          \n"                /* M3/M4/M7 */
[7]     " LSL    r1,r1,#27      \n"                /* r0 := (1 << 27) (UNPENDSVSET bit) */
[8] 
[9]     /*<<<<<<<<<< CRITICAL SECTION BEGIN <<<<<<<<</ */
[10]    #if (_ARM_ARCH_ == 6)           /* Cortex-M0/M0+/M1 (v6-M, v6S-M)? */
[11]        " CPSID   i             \n" /* disable interrupts (set PRIMASK) */
[12]    #else
[13]        /* M3/M4/M7 */
[14]        #if (_ARM_FP != 0)         /* if VFP available... */
[15]            " PUSH    {r0,lr}       \n" /* ... push lr plus stack-aligner */
[16]        #endif
[17]        /* VFP available */
[18]        " MOV     r0,#" STRINGIFY(QF_BASEPRI) "\n"
[19]        " CPSID   i             \n" /* disable interrupts with BASEPRI */
[20]        " MSR     BASEPRI,r0      \n" /* apply the Cortex-M7 erratum */
[21]        " CPSIE   i             \n" /* 837070, see ARM-EPM-064408. */
[22]        /* M3/M4/M7 */
[23]    #endif
[24] 
[25]    /* The PendSV exception handler can be preempted by an interrupt,
[26]     * which might pend PendSV exception again. The following write to
[27]     * ICSR[27] un-pends any such spurious instance of PendSV.
[28]     */
[29]    " STR     r1,[r3]          \n" /* ICSR[27] := 1 (unpend PendSV) */
[30] 
[31]    /* The QK activator must be called in a Thread mode, while this code
[32]     * executes in the Handler mode of the PendSV exception. The switch
[33]     * to the Thread mode is accomplished by returning from PendSV using
[34]     * a fabricated exception stack frame, where the return address is
[35]     * QK_activate_().
[36]     */
[37]     /* returns with interrupts DISABLED.
[38]      * <b>NOTE:</b> the QK activator is called with interrupts DISABLED and also
[39]      */
[40]    " LSR     r3,r1,#3          \n" /* r3 := (r1 >> 3), set the T bit (new xpsr) */
[41]    " LDR     r2,=QK_activate_ \n" /* address of QK_activate_ */
[42]    " SUB    r2,r2,#1          \n" /* align Thumb-address at halfword (new pc) */
[43] 
[44]    " LDR     r1,=Thread_ret  \n" /* return address after the call (new lr) */
[45]    " SUB    sp,sp,#8*4        \n" /* reserve space for exception stack frame */
[46]    " ADD    r0,sp,#5*4        \n" /* r0 := 5 registers below the SP */
[47]    " STM    r0!,{r1-r3}       \n" /* save xpsr,pc,lr */
[48] 
[49]    " MOV     r0,#6          \n"
[50]    " MVN     r0,r0          \n" /* r0 := ~6 == 0xFFFFFFFF9 */
[51]    " BX     r0             \n" /* exception-return to the QK activator */
[52] );
}

```

```

[24] __attribute__ ((naked))
void Thread_ret(void) {
__asm volatile (
    /* After the QK activator returns, we need to resume the preempted
     * thread. However, this must be accomplished by a return-from-exception,
     * while we are still in the thread context. The switch to the exception
     * context is accomplished by triggering the NMI exception.
     * <b>NOTE:</b> The NMI exception is triggered with ninterrupts DISABLED,
     * because QK activator disables interrupts before return.
    */

    /* before triggering the NMI exception, make sure that the
     * VFP stack frame will NOT be used...
    */
#if (__ARM_FP != 0)           /* if VFP available... */
[25]   " MRS      r0,CONTROL    \n" /* r0 := CONTROL */
[26]   " BICS     r0,r0,#4      \n" /* r0 := r0 & ~4 (FPCA bit) */
[27]   " MSR      CONTROL,r0    \n" /* CONTROL := r0 (clear CONTROL[2] FPCA bit) */
[28]   " ISB      .              \n" /* ISB after MSR CONTROL (ARM AN321,Sect.4.16) */
#endif
/* trigger NMI to return to preempted task...
 * <b>NOTE:</b> The NMI exception is triggered with ninterrupts DISABLED
 */
[29]   " LDR      r0,=0xE000ED04  \n" /* Interrupt Control and State Register */
[30]   " MOV      r1,#1          \n"
[31]   " LSL      r1,r1,#31      \n" /* r1 := (1 < 31) (NMI bit) */
[32]   " STR      r1,[r0]        \n" /* ICSR[31] := 1 (pend NMI) */
[33]   " B       .              \n" /* wait for preemption by NMI */
);
}

```

[1] Attribute naked means that the GNU-ARM compiler won't generate any entry/exit code for this function.

[2] PendSV\_Handler is a CMSIS-compliant name of the PendSV exception handler. The PendSV\_Handler exception is always entered via tail-chaining from the last nested interrupt.

[3] Entire body of this function will be defined in this one inline-assembly instruction.

[4–5] Before interrupts are disabled, the following constants are loaded into registers: address of ICSR into r3 and  $(1 << 27)$  into r1.

For the ARMv6-M architecture (Cortex-M0/M0+)...

[7] Interrupts are globally disabled by setting PRIMASK (see Section 3)

Otherwise, for the ARMv7-M architecture (Cortex-M3/4/7) and when the \_\_ARM\_FP macro is defined...

#### Note

The symbol \_\_ARM\_FP is defined by the GNU-ARM compiler when the compile options indicate that the ARM FPU is used.

[8] The lr register (EXC\_RETURN) is pushed to the stack along with r0, to keep the stack aligned at 8-byte boundary.

#### Note

In the presence of the FPU (ARMv7M or higher architectures), the EXC\_RETURN[4] bit carries the information about the stack frame format used, whereas EXC\_RETURN[4] == 0 means that the stack contains room for the S0-S15 and FPSCR registers in addition to the usual R0-R3,R12,LR,PC,xPSR registers. This information must be preserved, in order to properly return from the exception at the end.

[9] For the ARMv7-M architecture (Cortex-M3/M4), interrupts are selectively disabled by setting the BASEPRI register.

**Note**

The value moved to BASEPRI must be identical to the [QF\\_BASEPRI](#) macro defined in [qp\\_port.h](#).

[10] Before setting the BASEPRI register, interrupts are disabled with the PRIMASK register, which is the recommended workaround for the Cortex-M7 r0p1 hardware bug, as described in the ARM Ltd. [[ARM-EPM-064408](#)], Erratum 837070.

[11] The BASEPRI register is set to the [QF\\_BASEPRI](#) value.

[12] After setting the BASEPRI register, interrupts are re-enabled with the PRIMASK register, which is the recommended workaround for the Cortex-M7 r0p1 hardware bug, as described in the ARM Ltd. [[ARM-EPM-064408](#)], Erratum 837070.

[13] The PendSV exception is **explicitly** un-pended.

**Note**

The PendSV exception handler can be preempted by an interrupt, which might pend PendSV exception again. This would trigger PendSV incorrectly again immediately after calling [QK](#) activator without destroying the original exception stack frame of the PendSV exception. This is necessary to preserve the context of the preempted code.

[14] The value ( $1 \ll 24$ ) is synthesized in r3 from the value ( $1 \ll 27$ ) already available in r1. This value is going to be stacked and later restored to xPSR register (only the T bit set).

[15] The address of the [QK](#) activator function `QK_activate_()` is loaded into r2. This will be pushed to the stack as the PC register value.

[16] The address of the [QK](#) activator function `QK_activate_()` in r2 is adjusted to be half-word aligned instead of being an odd THUMB address.

**Note**

This is necessary, because the value will be loaded directly to the PC, which cannot accept odd values.

[17] The address of the `Thread_ret()` function is loaded into r1. This will be pushed to the stack as the lr register value.

**Note**

The address of the `Thread_ret` label must be a THUMB address, that is, the least-significant bit of this address must be set (this address must be odd number). This is essential for the correct return of the [QK](#) activator with setting the THUMB bit in the PSR. Without the LS-bit set, the ARM Cortex-M CPU will clear the T bit in the PSR and cause the Hard Fault. The GNU-ARM assembler/linker will synthesize the correct THUMB address of the `svc_ret` label only if this label is declared with the `.type Thread_ret, function` attribute (see step [23]).

[18] The stack pointer is adjusted to leave room for 8 registers.

[19] The top of stack, adjusted by 5 registers, (r0, r1, r2, r3, and r12) is stored to r0.

[20] The values of xpsr, pc, and lr prepared in r3, r2, and r1, respectively, are pushed on the top of stack (now in r0). This operation completes the synthesis of the exception stack frame. After this step the stack looks as follows:

```

Hi memory
    (optionally S0-S15, FPSCR), if EXC_RETURN[4]==0
    xPSR
    pc (interrupt return address)
    lr
    r12
    r3
    r2
    r1
    r0
    EXC_RETURN (pushed in step [7] if FPU is present)
old SP --> "aligner" (pushed in step [7] if FPU is present)
    xPSR == 0x01000000
    PC == QK_activate_
    lr == Thread_ret
    r12 don't care
    r3 don't care
    r2 don't care
    r1 don't care
    SP --> r0 don't care
Low memory

```

[21] The special exception-return value 0xFFFFFFFF9 is synthesized in r0 (two instructions are used to make the code compatible with Cortex-M0, which has no barrel shifter).

#### Note

the r0 register is used instead of lr because the Cortex-M0 instruction set cannot manipulate the higher-registers (r9-r15).

The exception-return value is consistent with the synthesized stack-frame with the lr[4] bit set to 1, which means that the FPU registers are not included in this stack frame.

[23] PendSV exception returns using the special value of the r0 register of 0xFFFFFFFF9 (return to Privileged Thread mode using the Main Stack pointer). The synthesized stack frame causes actually a function call to QK\_sched\_function in C.

#### Note

The return from the PendSV exception just executed switches the ARM Cortex-M core to the Privileged Thread mode. The QK\_sched\_function internally re-enables interrupts before launching any thread, so the threads always run in the Thread mode with interrupts enabled and can be preempted by interrupts of any priority.

In the presence of the FPU, the exception-return to the **QK** activator does not change any of the FPU status bit, such as CONTROL.FPCA or LSPACT.

[24] The Thread\_ret function is the place, where the **QK** activator **QK\_activate\_()** returns to, because this return address is pushed to the stack in step [16]. Please note that the address of the Thread\_ret label must be a THUMB address.

[25] If the FPU is present, the read-modify-write code clears the CONTROL[2] bit [2]. This bit, called CONTROL.FPCA (Floating Point Active), would cause generating the FPU-type stack frame, which you want to avoid in this case (because the NMI exception will certainly not use the FPU).

**Note**

Clearing the CONTROL.FPCA bit occurs with interrupts disabled, so it is protected from a context switch.

[28–32] The asynchronous NMI exception is triggered by setting ICSR[31]. The job of this exception is to put the CPU into the exception mode and correctly return to the thread level.

[33] This endless loop should not be reached, because the NMI exception should preempt the code immediately after step [31]

**NMI\_Handler() Implementation**

*Listing: NMI\_Handler() function in qk\_port.c file*

```

__attribute__((naked))
[1] void NMI_Handler(void) {
    __asm volatile (
[2]     " ADD     sp,sp,#(8*4)          \n" /* remove one 8-register exception frame */
        /* Cortex-M0/M0+/M1 (v6-M, v6S-M)? */
[3]     " CPSIE   i                  \n" /* enable interrupts (clear PRIMASK) */
[4]     " BX      lr                 \n" /* return to the preempted task */
        /* M3/M4/M7 */
[5]     " MOV     r0,#0              \n"
[6]     " MSR     BASEPRI,r0         \n" /* enable interrupts (clear BASEPRI) */
        /* if VFP available... */
[7]     " POP     {r0,pc}           \n" /* pop stack aligner and EXC_RETURN to PC */
        /* no VFP */
[8]     " BX      lr                 \n" /* return to the preempted task */
        /* no VFP */
        /* M3/M4/M7 */
);
}

```

[1] The `NMI_Handler` is the CMSIS-compliant name of the NMI exception handler. This exception is triggered after returning from the `QK` activator in step [31] of the previous listing. The job of NMI is to discard its own stack frame and cause the exception-return to the original preempted thread context. The stack contents just after entering NMI is shown below:

```

Hi memory
    (optionally S0-S15, FPSCR), if EXC_RETURN[4]==0
    xPSR
    pc (interrupt return address)
    lr
    r12
    r3
    r2
    r1
    r0
old SP --> EXC_RETURN (pushed in PendSV [7] if FPU is present)
    "aligner" (pushed in PendSV [7] if FPU is present)
    xPSR don't care
    PC  don't care
    lr  don't care
    r12 don't care
    r3  don't care
    r2  don't care
    r1  don't care
SP --> r0  don't care
Low memory

```

[2] The stack pointer is adjusted to un-stack the 8 registers of the interrupt stack frame corresponding to the NMI exception itself. This moves the stack pointer from the "old SP" to "SP" in the picture above, which "uncovers" the original exception stack frame left by the PendSV exception.

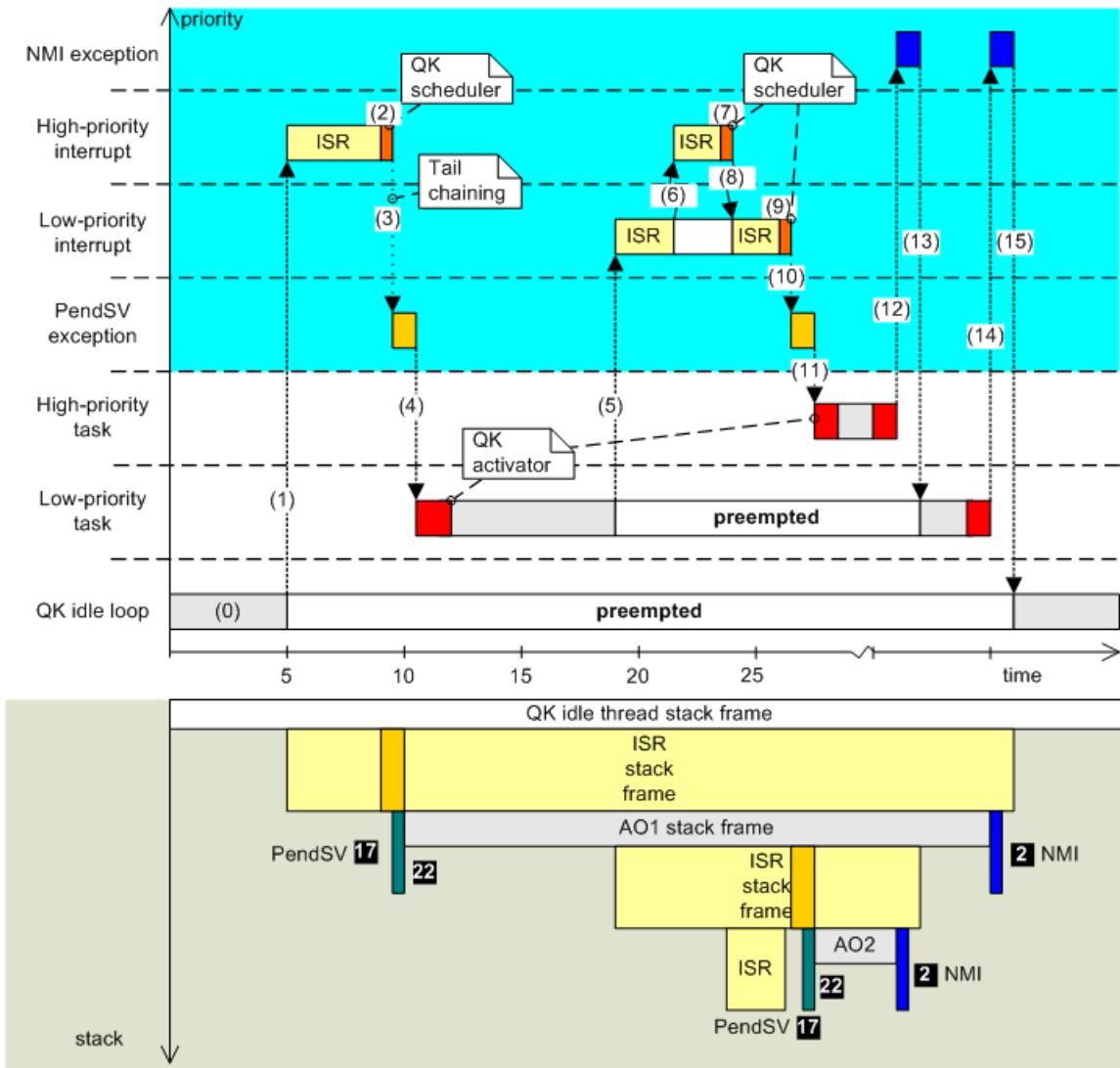
[3] For ARMv6-M, interrupts are enabled by clearing the PRIMASK.

[4] For ARMv6-M, The NMI exception returns to the preempted thread using the standard EXC\_RETURN, which is in lr.

[5–6] For the ARMv7-M, interrupts are enabled by writing 0 into the BASEPRI register.

[7] If the FPU is used, the EXC\_RETURN and the "stack aligner" saved in PendSV step [7] are popped from the stack into r0 and pc, respectively. Updating the pc causes the return from PendSV.

[8] Otherwise, NMI returns to the preempted thread using the standard EXC\_RETURN, which is in lr.



Detailed stack allocation in QK for ARM Cortex-M

### Writing ISRs for QK

The ARM Cortex-M CPU is designed to use regular C functions as exception and interrupt service routines (ISRs).

**Note**

The ARM EABI (Embedded Application Binary Interface) requires the stack be 8-byte aligned, whereas some compilers guarantee only 4-byte alignment. For that reason, some compilers (e.g., GNU-ARM) provide a way to designate ISR functions as interrupts. For example, the GNU-ARM compiler provides the `__attribute__((__interrupt__))` designation that will guarantee the 8-byte stack alignment.

Typically, ISRs are application-specific (with the main purpose to produce events for active objects). Therefore, ISRs are not part of the generic QP/C port, but rather part of the BSP (Board Support Package).

The following listing shows an example of the `SysTick_Handler()` ISR (from the DPP example application). This ISR calls the `QF_TICK_X()` macro to perform QF time-event management.

*Listing: An ISR header for QK*

```
void SysTick_Handler(void) __attribute__((__interrupt__));
void SysTick_Handler(void) {
    ~ ~ ~
[1]    QK_ISR_ENTRY(); /* inform QK about entering an ISR */
    ~ ~ ~
    QF_TICK_X(0U, &_SysTick_Handler); /* process all armed time events */
    ~ ~ ~
[2]    QK_ISR_EXIT(); /* inform QK about exiting an ISR */
}
```

[1] Every ISR for **QK** must call `QK_ISR_ENTRY()` before calling any QP/C API

[2] Every ISR for **QK** must call `QK_ISR_EXIT()` right before exiting to let the **QK** kernel schedule an asynchronous preemption, if necessary.

**Note**

The **QK** port to ARM Cortex-M complies with the requirement of the ARM-EABI to preserve stack pointer alignment at **8-byte boundary**. Also, all QP/C examples for ARM Cortex-M comply with the CMSIS naming convention for all exception handlers and IRQ handlers.

### Using the FPU in the **QK** Port (ARMv7M or higher architectures)

If you have the Cortex-M4F CPU and your application uses the hardware FPU, it should be enabled because it is turned off out of reset. The CMSIS-compliant way of turning the FPU on looks as follows:

```
SCB->CPACR |= (0xFU << 20);
```

**Note**

If the FPU is configured in the project, the **QK** kernel initializes the FPU to use the automatic state preservation and the lazy stacking feature as follows:

```
FPU->FPCCR |= (1U << FPU_FPCCR_ASSEN_Pos) | (1U << FPU_FPCCR_LSPEN_Pos);
```

As described in the ARM Application Note "Cortex-M4(F) Lazy Stacking and Context Switching" [[ARM-AN298](#)], the FPU automatic state saving requires more stack plus additional CPU time to save the FPU registers, but only when the FPU is actually used.

### **QK Idle Processing Customization in QK\_onIdle()**

QK can very easily detect the situation when no events are available, in which case QK calls the `QK_onIdle()` callback. You can use `QK_onIdle()` to suspended the CPU to save power, if your CPU supports such a power-saving mode. Please note that `QK_onIdle()` is called repetitively from an endless loop, which is the QK idle-thread. The `QK_onIdle()` callback is called with interrupts **enabled** (which is in contrast to the `QV_onIdle()` callback used in the non-preemptive configuration).

The THUMB-2 instruction set used exclusively in ARM Cortex-M provides a special instruction WFI (Wait-for-Interrupt) for stopping the CPU clock, as described in the "ARMv7-M Reference Manual" [ARM 06a]. The following listing shows the `QK_onIdle()` callback that puts ARM Cortex-M into a low-power mode.

*Listing: QV\_onIdle() for ARM Cortex-M*

```
[1] void QK_onIdle(void) {
    ~ ~ ~
[2] #if defined NDEBUG
    /* Put the CPU and peripherals to the low-power mode.
     * you might need to customize the clock management for your application,
     * see the datasheet for your particular Cortex-M3 MCU.
     */
[3]     __WFI(); /* Wait-For-Interrupt */
#endif
}
```

- [1] The preemptive QK kernel calls the `QK_onIdle()` callback with interrupts enabled.
- [2] The sleep mode is used only in the non-debug configuration, because sleep mode stops CPU clock, which can interfere with debugging.
- [3] The WFI instruction is generated using inline assembly.

### **Testing QK Preemption Scenarios**

The `bsp.c` file included in the `examples/arm-cm/dpp_ek-tm4c123gx1/qk` directory contains special instrumentation (an ISR designed for testing) for convenient testing of [various preemption scenarios in QK](#).

The technique described in this section will allow you to trigger an interrupt at any machine instruction and observe the preemption it causes. The interrupt used for the testing purposes is the GPIOA interrupt (INTID == 0). The ISR for this interrupt is shown below:

```
void GPIOPortA_IRQHandler(void) {
    QK_ISR_ENTRY(); /* inform QK about entering an ISR */
    QACTIVE_POST(AO_Table, Q_NEW(QEvt, MAX_PUB_SIG), /* for testing... */
                 &l_GPIOPortA_IRQHandler);
    QK_ISR_EXIT(); /* inform QK about exiting an ISR */
}
```

`GPIOPortA_IRQHandler()`, as all interrupts in the system, invokes the macros `QK_ISR_ENTRY()` and `QK_ISR_EXIT()`, and also posts an event to the Table active object, which has higher priority than any of the Philo active object.

The figure below shows how to trigger the GPIOA interrupt from the CCS debugger. From the debugger you need to first open the register window and select NVIC registers from the drop-down list (see right-bottom corner of Figure 6). You scroll to the `NVIC_SW_TRIG` register, which denotes the Software Trigger Interrupt Register in the NVIC. This write-only register is useful for software-triggering various interrupts by writing various masks to it. To trigger the GPIOA interrupt you need to write 0x00 to the `NVIC_SW_TRIG` by clicking on this field, entering the value, and pressing the Enter key.

The general testing strategy is to break into the application at an interesting place for preemption, set breakpoints to verify which path through the code is taken, and trigger the GPIO interrupt. Next, you need to free-run the code (don't use single stepping) so that the NVIC can perform prioritization. You observe the order in which the breakpoints are hit. This procedure will become clearer after a few examples.

### Interrupt Nesting Test

The first interesting test is verifying the correct tail-chaining to the PendSV exception after the interrupt nesting occurs, as shown in [Synchronous Preemption in QK](#). To test this scenario, you place a breakpoint inside the `GPIOPortA_IRQHandler()` and also inside the `SysTick_Handler()` ISR. When the breakpoint is hit, you remove the original breakpoint and place another breakpoint at the very next machine instruction (use the Disassembly window) and also another breakpoint on the first instruction of the `QK_PendSV` handler. Next you trigger the PIOINT0 interrupt per the instructions given in the previous section. You hit the Run button.

The pass criteria of this test are as follows:

1. The first breakpoint hit is the one inside the `GPIOPortA_IRQHandler()` function, which means that GPIO ISR preempted the SysTick ISR.
2. The second breakpoint hit is the one in the `SysTick_Handler()`, which means that the SysTick ISR continues after the PIOINT0 ISR completes.
3. The last breakpoint hit is the one in `PendSV_Handler()` exception handler, which means that the PendSV exception is tail-chained only after all interrupts are processed. You need to remove all breakpoints before proceeding to the next test.

### Thread Preemption Test

The next interesting test is verifying that threads can preempt each other. You set a breakpoint anywhere in the Philosopher state machine code. You run the application until the breakpoint is hit. After this happens, you remove the original breakpoint and place another breakpoint at the very next machine instruction (use the Disassembly window). You also place a breakpoint inside the `GPIOPortA_IRQHandler()` interrupt handler and on the first instruction of the `PendSV_Handler()` handler. Next you trigger the GPIOA interrupt per the instructions given in the previous section. You hit the Run button.

The pass criteria of this test are as follows:

1. The first breakpoint hit is the one inside the `GPIOPortA_IRQHandler()` function, which means that GPIO ISR preempted the Philo thread.
2. The second breakpoint hit is the one in `PendSV_Handler()` exception handler, which means that the PendSV exception is activated before the control returns to the preempted Philosopher thread.
3. After hitting the breakpoint in `PendSV_Handler()`, you single step into `QK_activate_()`. You verify that the activator invokes a state handler from the Table state machine. This proves that the Table thread preempts the Philo thread.
4. After this you free-run the application and verify that the next breakpoint hit is the one inside the Philosopher state machine. This validates that the preempted thread continues executing only after the preempting thread (the Table state machine) completes.

### Testing the FPU

In order to test the FPU (ARMv7M or higher architectures), the Board Support Package (BSP) for the Cortex-M4F EK-TM4C123GXL board uses the FPU in the following contexts:

- In the idle loop via the `QK_onIdle()` callback (QP priority 0)
- In the thread level via the `BSP_random()` function called from all five Philo active objects (QP priorities 1-5).
- In the thread level via the `BSP_displayPhiloStat()` function called from the Table active object (QP priority 6)
- In the ISR level via the `SysTick_Handler()` ISR (priority above all threads)

To test the FPU, you could step through the code in the debugger and verify that the expected FPU-type exception stack frame is used and that the FPU registers are saved and restored by the "lazy stacking feature" when the FPU is actually used.

Next, you can selectively comment out the FPU code at various levels of priority and verify that the `QK` context switching works as expected with both types of exception stack frames (with and without the FPU).

### Other Tests

Other interesting tests that you can perform include changing priority of the GPIOA interrupt to be lower than the priority of SysTick to verify that the PendSV is still activated only after all interrupts complete.

In yet another test you could post an event to Philosopher active object rather than Table active object from the `GPIOPortA_IRQHandler()` function to verify that the `QK` activator will not preempt the Philosopher thread by itself. Rather the next event will be queued and the Philosopher thread will process the queued event only after completing the current event processing.

#### 3.3.1.6 Preemptive "Dual-Mode" QXK Kernel

This section describes how to use QP/C on ARM Cortex-M with the **preemptive, dual-mode QXK real-time kernel**, which combines the lightweight non-blocking **basic threads** of **QK** with traditional blocking **extended threads** found in conventional RTOS kernels. **QXK** provides all typical services of a conventional blocking RTOS, such as blocking time-delays, semaphores, mutexes, and message queues.

**QXK** has been designed specifically for mixing event-driven active objects with traditional blocking code, such as commercial middleware (TCP/IP stacks, UDP stacks, embedded file systems, etc.) or legacy software.

#### Note

If you are currently using QP/C on top of a conventional 3rd-party RTOS, consider moving your application to the **QXK** kernel. **QXK** is not only more efficient than running QP/C on top of a **traditional 3rd-party RTOS** (because non-blocking **basic threads** take far less stack space and CPU cycles for context switch than the much heavier **extended threads**). But the biggest advantage of **QXK** is that it **protects** the application-level code from inadvertent mixing of blocking calls inside the event-driven active objects. Specifically, **QXK** "knows" the type of the thread context (extended/basic) and asserts internally if a blocking call (e.g., semaphore-wait or a time-delay) is attempted in a basic thread (active object). This is something that a QP/C port to a **conventional 3rd-party RTOS** cannot do, because such an RTOS runs all code (including active objects) in the context of heavyweight extended threads.

#### Synopsis of the **QXK** Port on ARM Cortex-M

The preemptive, blocking **QXK** kernel works on ARM Cortex-M as follows:

- The ARM Cortex-M processor executes application code in the Privileged Thread mode, which is exactly the mode entered out of reset. The exceptions (including all interrupts) are always processed in the Privileged Handler mode.

- **QXK** uses the Main Stack Pointer (MSP) for [basic threads](#), interrupts and exceptions (such as the PendSV exception). The MSP is also used for the **QXK** idle thread (which is a non-blocking basic thread).
- **QXK** uses the Process Stack Pointer (PSP) for handling [extended threads](#). Each extended thread must provide a private stack space to be associated with the PSP.
- The **QXK** port uses the PendSV (exception number 14) and the NMI or the IRQ exception (number 2) to perform context switch. The application code (your code) must initialize the Interrupt Vector Table with the addresses of the `PendSV_Handler` and `NMI_Handler` exception handlers.

#### Note

**QXK** uses only the CMSIS-compliant exception and interrupt names, such as `PendSV_Handler`, `NMI_Handler`, etc.

The **QXK** port specifically does **not** use the SVC exception (Supervisor Call). This makes the **QXK** ports compatible with various "hypervisors" (such as mbed uVisor or Nordic SoftDevice), which use the SVC exception.

- You need to explicitly **assign priorities of the all interrupts** used in your application, as described in [Interrupts in the QP/C Ports to ARM Cortex-M](#).

#### Note

For ARMv7M or higher architectures (M3/M4/M7/M33...), the **QXK** initialization code (executed from the **QF** initialization) initializes all interrupt priorities to the safe value maskable with the BASEPRI register. However, this is just a safety precaution not to leave the interrupts kernel-unaware, which they are out of reset. It is highly recommended to set the priorities of all interrupts explicitly in the application-level code.

- It is strongly recommended that you do not assign the lowest NVIC priority (0xFF) to any interrupt in your application, because it is used by the PendSV handler. For example, with 3 bits of priority implemented in the NVIC, this leaves the following 7 priority levels for you (listed from the lowest to the highest urgency): 0xC0, 0xA0, 0x80, 0x60, 0x40, 0x20, and 0x00 (the highest priority).

#### Note

The prioritization of interrupts, including the PendSV exception, is performed entirely by the NVIC. Because the PendSV has the lowest priority in the system, the NVIC tail-chains to the PendSV exception only after exiting the last nested interrupt.

- ISRs are written as regular C functions, but they need to call `QXK_ISR_ENTRY()` before using any **QF** services, and they must call `QXK_ISR_EXIT()` after using any of the **QF** services.
- ARM Cortex-M enters interrupt context without disabling interrupts. Generally, you should not disable interrupts inside your ISRs. In particular, the **QF** services (such as `QF_PUBLISH()`, `QF_TICK_X()`, and `QACTIVE_POST()`) should be called with interrupts enabled, to avoid nesting of critical sections.

**Note**

If you don't wish an interrupt to be preempted by another interrupt, you can always prioritize that interrupt in the NVIC to a higher or equal level as other interrupts (use a lower numerical value of priority).

- In compliance with the ARM Application Procedure Call Standard (AAPCS), the **QXK** kernel always preserves the 8-byte alignment of the stack (both MSP and PSP).

**Using the VFP**

If you have the ARMv7M or higher architectures (ARMv7M or higher architectures) and your application is compiled with the VFP present, the **QXK** kernel will enable the VFP along with the VFP automatic state preservation and lazy stacking features. This will cause the NVIC to automatically use the VFP-exception stack frame (with additional 18 VFP registers S0-S15 plus VFP status and stack "aligner"). The **QXK** context switch will add to this the rest of the VFP registers (S16-S31) on context switches to and from extended threads.

**Note**

With VFP enabled, any **QXK** thread (both a basic and an extended thread) will use 136 more bytes of its stack space, regardless if VFP is actually used by this thread. However, due to the "lazy-stacking" hardware feature, only a thread that actually uses the VFP will save and restore the VFP registers on the stack (which will cost some additional CPU cycles to perform a context switch).

**3.3.2 ARM Cortex-R**

The QP/C ports and examples for ARM Cortex-R are described in the Quantum Leaps Application Note [QP/C](#) and [ARM Cortex-R](#). The following built-in kernels are supported:

- Coopertaive **QV** kernel
- Preemptive Non-Blocking **QK** kernel



*Application Note: "QP and ARM Cortex-R"*

### 3.3.3 MSP430

The QP/C ports and examples for MSP430 are described in the Quantum Leaps Development Kit [QP](#) and [MSP430 with CCS↑](#) and [QP and MSP430 with IAR↑](#). The following built-in kernels are supported:

- Cooperative [QV](#) kernel
- Preemptive Non-Blocking [QK](#) kernel



*Application Note: "QP and MSP430-CCS"*



*Application Note: "QP and MSP430-IAR"*

## 3.4 Ports to Third-Party RTOS

The most important reason why you might consider using a traditional RTOS kernel for executing event-driven QP/C applications is compatibility with the existing software. For example, most communication stacks (TCP/IP, USB, CAN, etc.) are designed for a traditional **blocking** kernel. In addition, a lot of legacy code requires blocking mechanisms, such as semaphores or time-delays. A conventional RTOS allows you to run the existing software components as regular "blocking" tasks in parallel to the event-driven QP/C application.

Another reason you might be interested in running QP/C on top of a conventional RTOS is **safety certification**, which your RTOS kernel might have but the built-in QP kernels currently don't provide.

**Note**

You do **not** need to use a traditional RTOS just to achieve preemptive multitasking with QP. The [preemptive QK kernel](#), available as part of the QP package, supports preemptive priority-based multitasking and is fully compatible with Rate Monotonic Scheduling to achieve guaranteed, hard real-time performance. The preemptive, run-to-completion [QK kernel](#) perfectly matches the run-to-completion execution semantics of active objects, yet it is simpler, faster, and more efficient than any traditional blocking kernel.

**Attention**

QP/C 6.x includes a small, preemptive, priority-based, [dual-mode blocking QXK kernel](#) that executes active objects like the [QK kernel](#) ([basic threads](#)), but can also execute traditional blocking threads ([extended threads](#)). In this respect, [QXK](#) behaves exactly like a conventional RTOS. The [QXK](#) kernel is recommended as the preferred RTOS kernel for applications that need to mix active objects with traditional blocking code. Due to the tight and optimal integration between [QXK](#) and the rest of QP, [QXK](#) offers better performance and smaller memory footprint than any [QP port to a 3rd-party RTOS](#). Additionally, [QXK](#) is already included in QP, so you avoid additional licensing costs of 3rd-party kernels.

The QP/C framework can work with virtually any traditional real-time operating system (RTOS). The currently supported 3rd-party RTOS kernels are:

- [embOS](#)
- [FreeRTOS](#)
- [ThreadX](#)
- [uC-OS2](#)
- [Zephyr](#)
- [OSEK/VDX RTOS ERIKA Enterprise↑](#)

Combined with a conventional RTOS, QP/C takes full advantage of the multitasking capabilities of the RTOS by executing each active object in a separate RTOS task. The QP/C Platform Abstraction Layer (PAL) includes an abstract RTOS interface to enable integration between QP/C and the underlying RTOS. Specifically, the PAL allows adapting most message queue variants as event queues of active objects as well as most memory partitions as QP/C event pools.

### 3.4.1 embOS



*SEGGER embOS*

The QP/C ports and examples for embOS are described in the Quantum Leaps Application Note [QP and embOS↑](#).

### 3.4.1.1 QP/C Source Files Needed in this QP/C Port

It is important to note that not all QP/C source files should be included in the build process. Specifically, the QP/C source file `qf_actq.c` must **NOT** be included in the build, because this functionality is taken from embOS. Here is the list of QP/C source files needed:

```
qp/c/
+---src/
|   +---qf/
|   |   qep_hsm.c
|   |   qep_msm.c
|   |   qf_act.c
|   |   qf_actq.c // NOT included (implemented in embOS)
|   |   qf_defer.c
|   |   qf_dyn.c
|   |   qf_mem.c
|   |   qf_ps.c
|   |   qf_qeq.c
|   |   qf_qmact.c
|   |   qf_time.c
|   |
|   +---qs/
|   |   qs.c      // included only in the Spy build configuration
|   |   qs_fp.c    // included only in the Spy build configuration
|
+---ports/
|   +---embos/           // QP/C port to embOS
|   |   qp_port.h
|   |   qf_port.c
```



*Application Note: "QP and embOS"*

### 3.4.2 FreeRTOS



*FreeRTOS*

### 3.4.2.1 About the QP/C Port to FreeRTOS

The `ports/freertos/` directory contains a generic platform-independent QP/C port to [FreeRTOS kernel](#)<sup>↑</sup> (version 10). The provided QP port to FreeRTOS has been designed *generically* to rely exclusively on the existing FreeRTOS API. This means that the port should run without changes on any CPU/compiler platform supported by FreeRTOS.

The QP/C-FreeRTOS port works as follows:

- The QP/C port uses the `static memory allocation of FreeRTOS`. This requires the FreeRTOS configuration to define the `configSUPPORT_STATIC_ALLOCATION`
- Each QP/C active object executes in a separate FreeRTOS task (`StaticTask_t`) and requires a private stack space.
- The task-level critical section used in `QF` and `QS` is based on the FreeRTOS APIs `taskENTER_CRITICAL()`/`taskEXIT_CRITICAL()`.
- The ISR-level critical section used in `QF` and `QS` is based on the FreeRTOS APIs `taskENTER_CRITICAL_FROM_ISR()`/`taskEXIT_CRITICAL_FROM_ISR()`.
- The QP/C port to FreeRTOS provides new "FromISR" APIs, which must be used in the ISRs (but cannot be used at the task-level)

#### Attention

The design of FreeRTOS requires the use of special "FromISR" API inside ISRs, which imposes the requirement to also provide the "FromISR" variants of the QP/C APIs, such as `QACTIVE_POST_FROM_ISR()`, `QF_PUBLISH_FROM_ISR()`, etc. These "FromISR" QP/C APIs must be used inside ISRs instead of the task-level APIs (`QACTIVE_POST()`, `QF_PUBLISH()`, etc.) and conversely, they cannot be used inside tasks and active objects. Unfortunately, FreeRTOS provides no generic way to enforce the proper API use via assertions.

- The QP/C port uses the FreeRTOS message queue (`StaticQueue_t`) for active object event queues.
- The QP/C port uses the native `QF` memory pool (`QMPool`) to implement event pools.
- The QP/C port does not mandate any specific method to manage the QP/C time events, but the provided examples use the FreeRTOS "hook" `vApplicationTickHook()` to periodically invoke the QP/C clock tick `QF_TICK_X_FROM_ISR()`. (NOTE: the `vApplicationTickHook()` executes in the ISR context and therefore mandates the use of the "FromISR" APIs).

#### QP/C Source Files Needed in this QP/C Port

It is important to note that not all QP/C source files should be included in the build process. Specifically, the QP/C source file `qf_actq.c` must **NOT** be included in the build, because this functionality is taken from FreeRTOS. Here is the list of QP/C source files needed:

```
qpc/
+--src/
|   +--qf/
|   |   qep_hsm.c
|   |   qep_msm.c
|   |   qf_act.c
|   |   qf_actq.c // NOT included (implemented in FreeRTOS)
|   |   qf_defer.c
|   |   qf_dyn.c
|   |   qf_mem.c
```

```

|   |       qf_ps.c
|   |       qf_qeq.c
|   |       qf_qmact.c
|   |       qf_time.c
|
|   +---qs/
|   |       qs.c      // included only in the Spy build configuration
|   |       qs_fp.c   // included only in the Spy build configuration
|
+---ports/
|   +---freertos/    // QP/C port to FreeRTOS
|   |       qp_port.h
|   |       qf_port.c

```

### 3.4.2.2 Example Code

The QP/C port to FreeRTOS comes with examples located in the directory `qpc/examples/freertos`. Currently, the examples are provided for the following boards and development toolchains:

- EK-TM4C123GXL (ARM Cortex-M4F), ARM-KEIL, GNU-ARM, IAR-ARM
- STM32F746G-Discovery (ARM Cortex-M7), ARM-KEIL, GNU-ARM, IAR-ARM

#### Writing ISRs for QP/C-FreeRTOS

The provided examples show how to write regular "kernel-aware" ISRs as well as "kernel-unaware" ISRs for QP/C-FreeRTOS. (See also the FreeRTOS documentation for `configMAX_SYSCALL_INTERRUPT_PRIORITY`.

Here is an example of a regular "kernel-aware" ISR (note the use of the `FromISR` suffix in the QP/C APIs):

```

// NOTE: only the "FromISR" API variants are allowed in the ISRs!
void GPIOPortA_IRQHandler(void) {
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    // for testing
    QACTIVE_POST_FROM_ISR(AO_Table,
        _Q_NEW_FROM_ISR(QEvt, MAX_PUB_SIG),
        &xHigherPriorityTaskWoken,
        &l_GPIOPortA_IRQHandler);

    // the usual end of FreeRTOS ISR
    portEND_SWITCHING_ISR(xHigherPriorityTaskWoken);
}

```

Here is an example of a "kernel-unaware" ISR (See also the FreeRTOS documentation for `configMAX_SYSCALL_INTERRUPT_PRIORITY`):

```

//
// ISR for receiving bytes from the QSPY Back-End
// NOTE: This ISR is "kernel-unaware" meaning that it does not interact with
// the FreeRTOS or QP and is not disabled. Such ISRs don't need to call
// portEND_SWITCHING_ISR() at the end, but they also cannot call any
// FreeRTOS or QP APIs.
//
void UART0_IRQHandler(void) {
    uint32_t status = UART0->RIS; // get the raw interrupt status
    UART0->ICR = status;         // clear the asserted interrupts

    while (((UART0->FR & UART_FR_RXFE) == 0) { // while RX FIFO NOT empty
        uint32_t b = UART0->DR;
        QS_RX_PUT(b);
    }
}

```

#### Writing FreeRTOS Hooks Running in ISR Context

FreeRTOS provides "hooks" that are user functions that execute in the ISR context (e.g.,

`vApplicationTickHook()`). Such ISR-level functions are closely related to ISRs and should also use exclusively only the "FromISR" APIs. Here is an example of the `vApplicationTickHook()`:

```
// NOTE: only the "FromISR" API variants are allowed in vApplicationTickHook
void vApplicationTickHook(void) {
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    .
    .
    // process time events for rate 0
    QF_TICK_X_FROM_ISR(0U, &xHigherPriorityTaskWoken, &l_TickHook);
    .
    // notify FreeRTOS to perform context switch from ISR, if needed
    portEND_SWITCHING_ISR(xHigherPriorityTaskWoken);
}
```

### Starting Active Objects in QP/C-FreeRTOS

As mentioned in the [FreeRTOS port summary](#), the QP/C port to FreeRTOS uses the `static memory allocation of FreeRTOS`. This means that all memory for an active object, including the private queue buffer and the private **stack** for the the associated FreeRTOS task must be allocated by the user. Here is an example code that starts an active object:

```
int main() {
    .
    .
    static QEvt const *tableQueueSto[N_PHILO];
    static StackType_t tableStack[configMINIMAL_STACK_SIZE];
    .
    Table_ctor(); // instantiate the Table active object
    .
    QActive_SetAttr(AO_Table, TASK_NAME_ATTR, "Table");
    QACTIVE_START(AO_Table, // AO to start
        N_PHILO + 1U, // QP priority of the AO
        tableQueueSto, // event queue storage
        Q_DIM(tableQueueSto), // queue length [events]
        tableStack, // stack storage
        sizeof(tableStack), // stack size [bytes]
        (void *)0); // initialization param (not used)
    .
    return QF_run(); // run the QF application
}
```

### 3.4.3 ThreadX



*ThreadX (Eclipse ThreadX)*

The QP/C ports and examples for ThreadX (now Azure RTOS) are described in the Quantum Leaps Application Note [QP and ThreadX↑](#).

#### 3.4.3.1 QP/C Source Files Needed in this QP/C Port

It is important to note that not all QP/C source files should be included in the build process. Specifically, the QP/C source file `qf_actq.c` must **NOT** be included in the build, because this functionality is taken from ThreadX. Here is the list of QP/C source files needed:

```
qpc/
+--src/
|  +--qf/
|  |  qep_hsm.c
|  |  qep_msm.c
```

```

|   |       qf_act.c
|   |       qf_actq.c // NOT included (implemented in ThreadX)
|   |       qf_defer.c
|   |       qf_dyn.c
|   |       qf_mem.c
|   |       qf_ps.c
|   |       qf_qeq.c
|   |       qf_qmact.c
|   |       qf_time.c
|
|   +---qs/
|       qs.c      // included only in the Spy build configuration
|       qs_fp.c    // included only in the Spy build configuration
|
+---ports/
  +---threadx/           // QP/C port to ThreadX
  |   qp_port.h
  |   qf_port.c

```



*Application Note: "QP and ThreadX"*

### 3.4.4 uC-OS2



*uC-OS2*

#### 3.4.4.1 About the QP/C Port to uC-OS2

This directory contains a generic platform-independent QP/C port to uC-OS2 V2.92.

Typically, you should not need to change the files in this directory to adapt the QP/C-uC-OS2 port on any CPU/Compiler to which uC-OS2 has been ported, because all the CPU and compiler specifics are handled by the uC-OS2 RTOS.

### 3.4.4.2 uC-OS2 Source and ARM Cortex-M3/M4 Ports

The uC-OS2 V2.92 source code and ports are located in `3rd_party/uc-os2`. Please make sure to read about uC-OS2 licensing in the README file found in this directory.

#### Note

The official Micrium ARM-Cortex-M ports have been modified by Quantum Leaps to remove the dependencies on the Micrium's uC-CPU and uC-LIB components, and instead to inter-operate with the Cortex Microcontroller Software Interface Standard (CMSIS).

### 3.4.4.3 Examples for the uC-OS2 Port

The example projects for this port are located in the `examples/uc-os2/arm-cm/dpp_ek-tm4c123gx1` folder. Currently, ARM-KEIL and IAR-ARM toolsets are supported (in the `arm/` and `iar/` sub-directories within this example project).

The example projects use this port by directly including the QP/C source code (and this port) in the application projects. (There is no QP/C library to build.)

#### QP/C Source Files Needed in this QP/C Port

It is important to note that not all QP/C source files should be included in the build process. Specifically, the QP/C source files `qf_actq.c` and `qf_mem.c` must **NOT** be included in the build, because this functionality is taken from uC-OS2. Here is the list of QP/C source files needed:

```
qpcl/
+---src/
|   +---qf/
|   |   qep_hsm.c
|   |   qep_msm.c
|   |   qf_act.c
|   |   qf_actq.c // NOT included (implemented in uC-OS2)
|   |   qf_defer.c
|   |   qf_dyn.c
|   |   qf_mem.c // NOT included (implemented in uC-OS2)
|   |   qf_ps.c
|   |   qf_qeq.c
|   |   qf_qmact.c
|   |   qf_time.c
|   |
|   +---qs/
|   |   qs.c      // included only in the Spy build configuration
|   |   qs_fp.c    // included only in the Spy build configuration
|
+---ports/
|   +---uc-os2/           // QP/C port to uC/OS2
|   |   qp_port.h
|   |   qf_port.c
```



*Application Note: QP/C and uC-OS2*

### 3.4.5 Zephyr



*QP port to Zephyr RTOS*

#### 3.4.5.1 About the QP/C Port to Zephyr

This directory contains a generic platform-independent QP/C port to the [Zephyr RTOS](#).

Typically, you should not need to change the files in this directory to adapt the QP/C-Zephyr port on any CPU/Compiler to which Zephyr has been ported, because all the CPU and compiler specifics are handled by the Zephyr RTOS.

The QP/C-Zephyr port works as follows:

- The critical section used in this port is based on `k_spin_lock()`/`k_spin_unlock()` Zephyr API. This is the modern Zephyr critical section API, which is ready for SMP (symmetric multiprocessing).
- Each QP/C active object executes in a separate Zephyr thread (`struct k_thread`) and requires a private stack space.

#### Note

As demonstrated in the `examples/zephyr` folder, the private stacks for active objects in this port must be allocated with the Zephyr `K_THREAD_STACK_DEFINE()` macro. Also, the stack size passed to `QACTIVE_START()` must be calculated with the Zephyr `K_THREAD_STACK_SIZEOF()` macro. Failure to use these macros can lead to memory corruption in the application.

- The active object event queue is implemented with the Zephyr message queue (`struct k_msgq`).

#### Note

The Zephyr message queue currently supports only the FIFO policy and does NOT support the LIFO policy. For that reason, the `QActive_postLIFO()` implementation in this port uses the FIFO policy. A feature request has been filed in the Zephyr project for adding the LIFO policy, so perhaps this can be improved, if the feature is added.

- The QP/C port uses Zephyr scheduler locking (`k_sched_lock()`/`k_sched_unlock()`), which locks all threads indiscriminately. Currently Zephyr does not provide a selective scheduler locking (with priority-ceiling).
- The QP/C port uses the native QF memory pool ([QMPool](#)) to implement event pools.

- The QP/C port does not mandate any specific method to manage the QP/C time events, but the provided examples use the Zephyr timer (`struct k_timer`) to tick periodically and invoke `QF_TICK_X()`.

### QP/C Source Files Needed in the Zephyr port

It is important to note that not all QP/C source files should be included in the build process. Specifically, the QP/C source files `qf_actq.c` must **NOT** be included in the build, because this functionality is taken from Zephyr. Here is the list of QP/C source files needed:

```
qpc/
+---src/
|   +---qf/
|   |   qep_hsm.c
|   |   qep msm.c
|   |   qf_act.c
|   |   qf_actq.c // NOT included (implemented with Zephyr message queues)
|   |   qf_defer.c
|   |   qf_dyn.c
|   |   qf_mem.c
|   |   qf_ps.c
|   |   qf_qeq.c
|   |   qf_qmact.c
|   |   qf_time.c
|
|   +---qs/
|   |   qs.c    // included only in the Spy build configuration
|   |   qs_fp.c // included only in the Spy build configuration
|
+---zephyr/      // QP/C port to Zephyr (packaged as a "Zephyr module")
|   CMakeLists.txt
|   Kconfig
|   module.yml
|   qp_port.h
|   qs_port.h
|   qf_port.c // implementation of the Zephyr port
```

#### Note

The QP/C port to Zephyr is NOT located in the `qpc/ports` folder (as all other 3rd-party RTOS ports) because the port is packaged according to the rules of a **"Zephyr Module"**. These rules require a **"Zephyr Module"** to be located directly in the `qpc` folder.

#### 3.4.5.2 Examples for the Zephyr port

The example projects for this port are located in the `examples/zephyr` folder.

## 3.5 Ports to General-Purpose OSes

- [POSIX-QV \(Single Threaded\)](#) (single-threaded Linux, macOS, embedded-Linux, QNX, etc.)
- [POSIX \(Multithreaded\)](#) (multi-threaded Linux, macOS, embedded-Linux, QNX, etc.)
- [Win32-QV \(Single Threaded\)](#) (single-threaded Windows)
- [Win32 API \(Multithreaded\)](#) API (multi-threaded Windows)

### 3.5.1 POSIX-QV (Single Threaded)

The QP/C ports and examples for POSIX-QV (e.g., embedded-Linux, QNX, etc.) are described in the Quantum Leaps Application Note [QP and POSIX↑](#).



The standard QP/C distribution contains the POSIX-QV port and [Ports to General-Purpose OSes](#).

### 3.5.2 POSIX (Multithreaded)

The QP/C ports and examples for POSIX (e.g., embedded-Linux, QNX, etc.) are described in the Quantum Leaps Application Note [QP and POSIX↑](#).



The standard QP/C distribution contains the POSIX port and [Ports to General-Purpose OSes](#).

### 3.5.3 Win32-QV (Single Threaded)

The QP/C ports and examples for Windows with single-thread (like the [QV non-preemptive kernel](#)) are described in the Quantum Leaps Application Note [QP and Win32 \(Windows\)↑](#).



The standard QP/C distribution contains the Win32-QV port and [Ports to General-Purpose OSes](#).

### 3.5.4 Win32 API (Multithreaded)

The QP/C ports and examples for Windows (e.g., desktop Windows, Windows Embedded) are described in the Quantum Leaps Application Note [QP and Win32 \(Windows\)](#)↑.



The standard QP/C distribution contains the Win32 (Windows) port and [Ports to General-Purpose OSes](#).

# Chapter 4

## Examples

### 4.1 General Comments

The QP/C distribution contains many **example projects** with the following main goals:

- **to help you learn how to use the QP Framework** — the examples show the intended way of using QP/C features and structuring your QP/C applications.
- **to provide unit testing support** — the `QUTest` examples illustrate unit testing techniques both for the development host computers and for embedded targets.
- **to provide a starting point for your own projects** — the examples are complete working projects, with correctly pre-configured tools, such as compiler options, linker script, debugger setup, etc.

#### Note

It is highly recommended that you create your own projects by **copying and modifying** existing example projects rather than starting your QP/C projects from scratch.

#### 4.1.1 Example Applications

To demonstrate QP/C features on an embedded board, you need to create an application that does "something interesting". Instead of inventing this "something interesting" for each and every example, the example projects implement one of the three "example applications", which are described on the [QP/C Tutorial](#):

- [Simple Blinky Application](#)
- [Dining Philosophers Problem \(DPP\)](#)
- ["Fly 'n' Shoot" Game](#)
- examples described in the book [Practical UML Statecharts in C/C++; 2nd Edition↑](#)

With the exception of the game application, all other example applications can be implemented on a board with just a couple of LEDs. The ["Fly 'n' Shoot" Game](#) application is a bit more involved and requires a small graphic display on the board.

The QP/C examples/posix-win32 directory contains all examples described in the book [Practical UML Statecharts in C/C++; 2nd Edition↑](#). These examples work on the host computer (e.g., Windows, Linux, macOS), so you don't need any special embedded hardware to build and run these examples.

### 4.1.2 Development Boards

The embedded example projects require special hardware in form of various evaluation boards, which you need to acquire to be able to run the examples. The boards chosen for the examples are generally inexpensive and self-contained with no need for external hardware (such as external JTAG debuggers or power supplies).

### 4.1.3 Development Tools

Most provided examples require special embedded cross-development tools, such as embedded compilers, linkers, debuggers and IDEs, which you need to acquire independently from the QP/C distribution. Generally, the examples work with the free evaluation versions of the commercial tools.

### 4.1.4 Build Configurations

The QP/C examples are typically provided in the following three **build configurations**:

- **Debug** — this configuration is built with full debugging information and minimal optimization. The [QS trace instrumentation](#) is **disabled**.
- **Release** — this configuration is built with high optimization. Debugging at the source-code level is severely impaired due to the highly optimized code, but the debugger can be used to download and start the executable. The [QS trace instrumentation](#) is **disabled**.
- **Spy** — like the debug variant, this variant is built with full debugging information and minimal optimization. Additionally, it is build with the [QS trace instrumentation](#) **enabled**. The on-board serial port and the Q-Spy host application are used for sending and viewing trace data.

#### Remarks

##### Why do you need multiple build configurations?

The different phases of embedded software life cycle pose different challenges. During the development and maintenance phase, for example, the emphasis is on the ease of debugging and verifying the correctness of the code, which require lower levels of optimization and special instrumentation code. In contrast, for releasing the code in the final product, the emphasis is on small memory footprint, CPU efficiency, and low power consumption, which require high-level of optimization and removal of any code instrumentation. To address these conflicting objectives, the same source code is compiled into multiple **build configurations** that differ in the use of compiler options and activation of the code instrumentation.

### 4.1.5 QM Models

Many example projects contain code auto-generated by the [QM modeling tool](#)<sup>↑</sup>. Such projects always contain the corresponding **QM model** file (extension `.qm`), which you can open in QM, modify, and re-generate the code.

#### Remarks

The auto-generated files are saved as **read-only**. This protects them from inadvertent modifications, which will get lost when the files are re-generated by QM (or QMC). All modifications to the auto-generated code should be done in the QM model, not in the code.

#### Note

If you don't like automatic code generation, you **can** create all your QP/C application code manually. In that case, the generated files can serve as starting points for your implementation. To edit the files manually, you need to change the file attributes to remove the read-only protection.

### 4.1.6 Third-Party Code

The QP/C example projects often need to use various additional code, such as MCU register definition files, startup code, device drivers, etc., which are provided by Third-Party vendors. All such code is located in the `3rd_party top-level folder`.

#### Note

As far as possible, the code in the `3rd_party` folder has been left unchanged from the original source. (Any modified code is clearly identified by top-level comments that detail the applied changes.) For that reason, the Third-Party code might produce **compilation warnings** in your builds.

The code in the `3rd_party` folder comes from various sources, and Quantum Leaps, LLC expressly makes **no claims of ownership** to any of this code, even though some of the code might be customized or modified by Quantum Leaps.

#### Attention

The Third-Party software components included in the `3rd_party` folder are licensed under a variety of different licensing terms that are defined by the respective owners of this software and are spelled out in the `README.txt` or `LICENSE.txt` files included in the respective sub-folders.

### 4.1.7 Creating your Own QP/C Projects

Perhaps the most important fact of life to remember is that in embedded systems nothing works until everything works. This means that you should always start with a **working system** and gradually evolve it, changing one thing at a time and making sure that it keeps working every step of the way.

Keeping this in mind, the provided QP/C application examples, such as the super-simple Blinky, or a bit more advanced [Dining Philosophers Problem \(DPP\)](#) or ["Fly 'n' Shoot" Game](#), allow you to get started with a working project rather than starting from scratch. You should also always try one of the provided example projects on the same evaluation board that it was designed for, before making any changes.

#### Note

The evaluation boards used in the examples are all very **inexpensive** and available from major electronic distributors (e.g., [Arrow](#), [DigiKey](#), [Mouser](#), [Avnet](#)), as well as directly from the silicon vendors (e.g., a TI board is also available from [TI.com](#)).

#### Remarks

Even before you acquire a specific evaluation board, you can always at least **build** a project that interests you on your host computer.

Only after convincing yourself that the example project works "as is", you can think about creating your own projects. At this point, the easiest and recommended way is to copy the existing working example project folder (such as the Blinky example) and rename it.

After copying the project folder, you still need to change the name of the project/workspace. The easiest and safest way to do this is to open the project/workspace in the corresponding IDE and use the `Save As~~~` option to save the project under a different name. You can do this also with the QM model file, which you can open in QM and "Save As" a different model.

#### Note

By copying and re-naming an existing, working project, as opposed to creating a new one from scratch, you inherit the correct compiler and linker options and other project settings, which will help you get started much faster.

#### 4.1.8 Next Steps and Further Reading About QP and QM

To work with QP/C effectively, you need to learn a bit more about active objects and state machines. Below is a list of links to enable you to further your knowledge:

1. The book "Practical UML Statecharts in C/C++, 2nd Edition" [PSiCC2] and the companion web-page to the book (<https://www.state-machine.com/psicc2/>)
2. Free Support Forum for QP/QM (<https://sourceforge.net/p/qpc/discussion/668726/>)
3. QP Code Downloads summary (<https://www.state-machine.com/#Downloads>)
4. QP Application Notes (<https://www.state-machine.com/an/>)
5. "State Space" Blog (<https://www.state-machine.com/category/blog/>)

## 4.2 Example Code Organization

QP/C examples are located in sub-directories of the examples `top-level folder`, with the hierarchical organization outlined below:

### Note

Each example project is described on its own dedicated `README.md` file located in the example folder.

```

qpcc/                                // QP/C installation directory
+---examples/                         // examples directory (applications)
| |
| [1]--arm-cm/                      // native examples for ARM Cortex-M
| | +---dpp_ek-tm4c123gx1/          // DPP example on the EK-TM4C123GLX board
| | | +---qk/                       // version for preemptive QK kernel
| | | | +---armclang/              // build with ARM-Clang (Compiler Version 6) toolchain
| | | | | +---dbg/                 // debug build configuration
| | | | | +---rel/                 // release build configuration
| | | | | +---spy/                 // spy build configuration (QP/Spy tracing enabled)
| | | | | +---gnu/                 // build with GNU-ARM toolchain
| | | | | +---iar/                 // build with IAR toolchain
| | | | | ~ ~ ~                   // source files independent from the toolchain
| | | | |
| | | | +---qv/                   // version for non-preemptive QV kernel
| | | +---qzk/                   // version for dual-mode QXK kernel
| | | |
| | | | | ~ ~ ~                   // source files independent on the built-in kernel
| | | | README.md                // documentation for this example
| |
| [2]--freertos/                     // examples for FreeRTOS (3rd-party RTOS)
| +---arm-cm/                      // examples for ARM Cortex-M
| | +---dpp_ek-tm4c123gx1/          // DPP example on the EK-TM4C123GLX board
| | | +---gnu/                       // build with GNU-ARM toolchain
| | | | ~ ~ ~                       // source files independent from the toolchain
| | | | README.md                  // documentation for this example
| |
| [3]--posix-win32/                 // examples for GPOS (Linux, macOS, Windows)
| | +---blinky/                    // simple Blinky example
| | +---calc/                      // calculator example
| | +---comp/                      // "Orthogonal Component" example
| | +--- ~ ~ ~                     // other examples from the PSiCC2 book
| |
| [4]--lwip/                        // examples for lwIP (3rd-party middleware)
| | +---arm-cm/                    // examples for ARM Cortex-M
| | | +---qk/                       // version for preemptive QK kernel
| | | +---qv/                       // version for non-preemptive QV kernel
| | | \---qzk/                      // version for dual-mode QXK kernel
| |
| [5]--qutest/                      // examples for QUTest unit testing harness

```

```

|   |   +---blinky/      // simple Blinky example
|   |   |   +---src/      // source code under test
|   |   |   |           ~ ~ ~ // CUT (Code Under Test) header file
|   |   |   |           ~ ~ ~ // CUT (Code Under Test) source file
|   |   |   +---test/     // code for unit testing
|   |   |   |           Makefile      // makefile for testing on the host
|   |   |   |           make_nucleo-1053r8 // makefile for testing on NUCLEO board
|   |   |   |           make_tm4c123  // makefile for testing on NUCLEO board
|   |   |   |           test_blinky.c|.cpp // test fixture
|   |   |   |           test_blinky.py   // test script
|   |   |   |           README.md      // documentation for this example
|
| [6]--qwin-gui/        // examples for QWin-GUI prototyping system (dual-targeting)
|   +---dpp-gui/        // DPP GUI prototype
|   |   dpp-gui.sln      // Visual Studio solution
|   |
|   +---game-gui/       // "Fly 'n' Shoot" game prototype
|   |   game-gui.sln      // Visual Studio solution

```

The examples provided in the QP/C distribution fall into the following categories:

[1] [Native Examples](#)

[2] [Examples for 3rd-party RTOS](#)

[3] [Examples for GPOS](#)

[4] [Examples for 3rd-party Middleware](#)

[5] [Examples for QUTest](#)

[6] [Examples for QWin-GUI](#)

#### Remarks

Because the QP distribution contains *all* examples, the number of sub-directories and files in the `examples` folder may seem daunting. However, knowing the structure of the `examples` folder, you can simply ignore or even delete the sub-directories that are not interesting to you.

### 4.2.1 Native Examples

Native examples are located in sub-directories named after the CPU architecture (see [1] in [Example Code Organization](#)), such as `arm-cm` for ARM Cortex-M. Under that directory, the sub-directories `blinky_ek-tm4c123gxl` contain the specific example on the specified board, such as "Blinky" on the EK-TM4C123GXL board here. In the specific example folder, you find sub-folders for the `QV`, `QK` and `QXK` kernels, respectively.

### 4.2.2 Examples for 3rd-party RTOS

Examples for 3rd-party RTOS are located in sub-directories named after the RTOS/OS, such as `freertos` for FreeRTOS (see [2] in [Example Code Organization](#)). Under that directory, the sub-directories, such as `arm-cm`, contain examples for the specified CPU architecture, such as ARM Cortex-M here.

### 4.2.3 Examples for GPOS

Examples for GPOS (POSIX/Win32) General-Purpose OS (see [3] in [Example Code Organization](#)) run directly on your host computer without any embedded hardware. These examples are simple console applications. This group includes examples described in the [PSiCC2](#) book.

#### Note

Examples in this directory can also be used on the *embedded* versions of the desktop operating systems, such as embedded-Linux and Windows-embedded.

### 4.2.4 Examples for 3rd-party Middleware

Examples for 3rd-party Middleware are located in sub-directories named after the middleware (see [4] in [Example Code Organization](#)), such as `lwIP` for the lwIP TCP/IP stack. Under that directory, the sub-directories, such as `arm-cm`, contain examples for the specified CPU architecture, such as ARM Cortex-M here.

### 4.2.5 Examples for QUTest

Examples for QUTest (see [5] in [Example Code Organization](#)) illustrate unit testing of embedded event-driven code.

#### Note

Examples in this directory can run on various General-Purpose OSes (Linux, macOS, Windows) (Windows, Linux, MacOS), as well as embedded targets.

### 4.2.6 Examples for QWin-GUI

Examples for QUTest (see [6] in [Example Code Organization](#)) illustrate prototyping of embedded software on Windows ("dual targeting"). The examples contain are designed for Visual Studio and contain the solution files.

## Chapter 5

# Software Requirements Specification



This *Software Requirements Specification* is part of the [SafeQP Certification Kit↑](#), but applies to the whole [QP Framework family↑](#). This document is the *best source of information* about the underlying **concepts** and **functionality** of the QP Frameworks and the QP Applications based on the frameworks.

### Revision History

QP version	Document revision	Date (YYYY-MM-DD)	By	Description
7.3.4	A	2024-05-05	MMS	Initial release for IEC-61508 SIL-3 and IEC-62304 Class-C.
7.4.0	B	2024-07-30	MMS	Updated for QP 7.4.0.
8.0.0	C	2024-11-17	MMS	Updated for QP 8.0.0.
8.0.3	D	2025-03-27	MMS	Updated for QP 8.0.3.

## 5.1 About this Document

### 5.1.1 DOC\_SRS\_QP

Software Requirements Specification (SRS)

## Description

This *Software Requirements Specification*, with Unique Identifier: [DOC\\_SRS\\_QP](#), defines the software requirements for the [QP Frameworks](#) as well as the most important *concepts* and *context of use* of the QP Frameworks and [QP Applications derived from the frameworks](#).

## Scope

The scope of this SRS document is to define what the QP Frameworks [shall do](#) under the **normal operating conditions** (the success view point). However, equally important is specifying what the QP Frameworks as well as QP Applications [shall NOT do](#) to avoid *failures* and what the software [shall do](#) under the identified **failure conditions** (the failure view point). The *failure space* is the focus of the separate document *Software Safety Requirements* ([DOC\\_SSR\\_QP](#), available in the SafeQP editions).

## Audience

This Software Requirements Specification is primarily intended for:

- **Application Developers** who develop *QP Applications* based on the QP Frameworks.

This requirements specification can also be of interest to:

- Quality-Assurance Engineers,
- Test Engineers,
- Software Architects,
- System Engineers,
- Hardware Engineers, as well as
- Managers who oversee the software development.

## Document Organization

After a high-level [overview](#), this requirements specification contains sections devoted to specific feature areas:

- Overview
- Active Objects
- Events
- State Machines
- Event Delivery Mechanisms
- Event Memory Management
- Time Management
- Software Tracing
- Non-Preemptive Kernel
- Preemptive Non-Blocking Kernel
- Preemptive Dual-Mode Kernel
- Non-Functional Requirements

Each section starts with an introduction of the **relevant concepts** and the description of the feature, followed by the associated requirements.

## Remarks

The presented features are in order of relevance for the *Application Developers* working on QP Applications, who are the primary audience of this requirements specification:

## 5.2 Document Conventions

Requirement *definitions* use consistent terminology to indicate whether something is mandatory, desirable, or allowed.

### 5.2.1 General Requirement UIDs

For traceability, this Software Requirements Specification uses the Unique Identifiers (UIDs) with the following structure:

```
+----- [1] Work artifact class (e.g., 'SRS' for Software Requirement Specification)
| +----- [2] Project identifier ('QP' for QP Framework or 'QA' for QP Application)
| | +----- [3] Work artifact ID
| | | +---- [4] Work artifact number
| | | | +--- [5] Optional variant letter ('A', 'B', 'C'...)
| | | | |+-- [6] Optional version number (1, 2, 3...)
| | | |
SRS_QP_xx_yy [-A2]
```

Examples: [SRS\\_QP\\_EVT\\_30](#), [SRS\\_QP\\_SM\\_32](#)

### 5.2.2 Use of "shall"

*Shall* is used to denote **mandatory** behavior.

### 5.2.3 Use of "shall not"

*Shall* is used to denote **prohibited** behavior.

### 5.2.4 Use of "should"

*Should* is used to denote a **desirable** behavior that should typically occur but might not happen all the time or might be optional in exceptional cases. The special cases are typically clarified in sub-requirements.

### 5.2.5 Use of "may"

*May* is used to denote **allowed** behavior that is optional but possible.

## 5.3 References

[IEEE 29148]	Requirement Specification Standard, <a href="#">ISO/IEC/IEEE 29148:2018</a>
[IEC 61508-3:2010]	IEC 61508-3:2010, Functional safety of electrical/electronic/programmable electronic safety-related systems- Part 3: Software requirements
[IEC 62304:2015]	IEC 62304:2006, Medical device software - Software life-cycle process, IEC 62304:2006 + IEC 62304:2006/Amd1:2015
[ISO 26262-6:2018]	ISO 26262-6:2018(en) Road vehicles - Functional safety - Part 6: Product development at the software level. International Standardization Organization.
[DOC_SRS_QP]	Software Safety Requirements
[DOC_SAS_QP]	<a href="#">Software Architecture Specification</a>
[DOC_SDS_QP]	<a href="#">Software Design Specification</a>
[PSiCC:02]	Miro Samek, <i>Practical Statecharts in C/C++</i> , CMP Books 2002. <a href="https://www.state-machine.com/psicc">https://www.state-machine.com/psicc</a>
[PSiCC2:08]	Miro Samek, <i>Practical UML Statecharts in C/C++</i> , 2nd Edition, Newnes 2008. <a href="https://www.state-machine.com/psicc2">https://www.state-machine.com/psicc2</a>

[Cummings:10]	David M. Cummings, "Managing Concurrency in Complex Embedded Systems", 2010 Workshop on Cyber-Physical Systems. <a href="https://www.state-machine.com/doc/Cummings2006.pdf">https://www.state-machine.com/doc/Cummings2006.pdf</a>
[ROOM:94]	Bran Selic, Garth Gullekson, Paul T. Ward: <i>Real-Time Object-Oriented Modeling</i> , New York, John Wiley & Sons Inc, 1994, ISBN 978-0-471-59917-3
[Samek:07]	Miro Samek, "Use an MCU's low-power modes in foreground/background systems", <i>Embedded Systems Design</i> , 2007, <a href="https://www.state-machine.com/doc/Samek0710.pdf">https://www.state-machine.com/doc/Samek0710.pdf</a>
[CODE2:04]	Steve McConnell, <i>Code Complete</i> , 2nd Ed, Microsoft Press 2004.
[UML-2.5]	OMG, "OMG Unified Modeling Language (OMG UML) Version 2.5.1", <i>formal/2017-12-05</i> , 2017 <a href="https://www.omg.org/spec/UML">https://www.omg.org/spec/UML</a> .
[Sutter:10]	Herb Sutter, "Prefer Using Active Objects Instead of Naked Threads", <i>Dr.Dobbs Journal</i> , June 2010. <a href="https://www.state-machine.com/doc/Sutter2010a.pdf">https://www.state-machine.com/doc/Sutter2010a.pdf</a> )
[SRP:90]	Theodore P. Baker, "A Stack-Based Resource Allocation Policy for Realtime Processes", <i>IEEE Real-Time Systems Symposium</i> , 1990
[OSEK:03]	OSEK/VDX, "Operating System Specification 2.2.1", <a href="http://osek-vdx.org/mirror/os221.pdf">osek-vdx.org, 2003</a> <a href="http://osek-vdx.org/mirror/os221.pdf">https://www.osek-vdx.org/mirror/os221.pdf</a>
[PTS:07]	Rony Ghattas and Alexander G. Dean, "Preemption Threshold Scheduling: Stack Optimality, Enhancements and Analysis", <i>Conference Paper</i> , April 2007
[RMS/RMA:91]	Lui Sha Mark H. Klein John B. Goodenough, "Rate Monotonic Analysis for Real-Time Systems", <i>Technical Report CMU/SEI-91-TR-6 ESD-91-TR-6</i> , 2017 <a href="https://insights.sei.cmu.edu/documents/1021/1991_005_001_15923.pdf">https://insights.sei.cmu.edu/documents/1021/1991_005_001_15923.pdf</a> .
[DMS:91]	N. C. Audsley A. Burns M. F. Richardson A. J. Wellings, "Hard Real-Time Scheduling: The Deadline-Monotonic Approach", 1991 <a href="https://www.cs.cmu.edu/~ssaewong/research/audsley_DMS.pdf">https://www.cs.cmu.edu/~ssaewong/research/audsley_DMS.pdf</a> .
[Yiu:14]	Joseph Yiu, "The Definitive Guide to ARM Cortex M3 and Cortex-M4 Processors Third Edition", <i>ARM Ltd., Cambridge, UK</i> , June 2014.
[OOP-C:08]	Quantum Leaps, <i>Object-Oriented Programming in C</i> , <a href="https://www.state-machine.com/oop">https://www.state-machine.com/oop</a>
[DbC:16]	Quantum Leaps, <i>Key Concept: Design by Contract</i> , <a href="https://www.state-machine.com/dbc">https://www.state-machine.com/dbc</a>

## 5.4 Overview

**QP real-time event frameworks** (RTEFs) form a family of lightweight implementations of the [Active Object model of computation](#) specifically designed for real-time embedded (RTE) systems (such as ARM Cortex-M based MCUs). QP Frameworks provide both a *software infrastructure* for building QP Applications consisting of Active Objects (Actors) and a *runtime environment* for executing the Active Objects in real-time. Additionally, QP Framework supports [Hierarchical State Machines](#) with which to specify the behavior of Active Objects [\[UML-2.5\]](#), [\[Sutter:10\]](#), [\[ROOM:94\]](#).

### Remarks

The features and requirements described in this *Software Requirement Specification* can be ultimately implemented in various programming languages, so this document applies to a whole family of QP RTEFs, currently consisting of **QP/C**, **QP/C++**, **SafeQP/C**, and **SafeQP/C++** frameworks implemented in C and C++, respectively. Other possible implementations (e.g., QP/Rust) of these requirements and features might be added in the future.

### 5.4.1 Goals

The main goals of the QP Framework are:

- to provide a modern, event-driven model of concurrency based on the **best practices of concurrent programming** [\[Sutter:10\]](#), [\[Cummings:10\]](#), [\[ROOM:94\]](#), collectively known as the [Active Object \(Actor\) model of computation](#);
- to provide an **inherently safer** concurrent programming model than the traditional "shared-state concurrency, mutual-exclusion, and blocking" approach based on a conventional Real-Time Operating System (RTOS);
- to provide a **higher-level of abstraction** closer to the problem domain than the "naked" RTOS threads;
- to provide the **right abstractions** for applying modern techniques like [Hierarchical State Machines](#), visual modeling, and automatic code generation;
- to **bridge the semantic gap** between the higher level UML concepts (such as [Events](#), [Hierarchical State Machines](#), and [Active Objects](#)) and the traditional programming languages like C or C++.

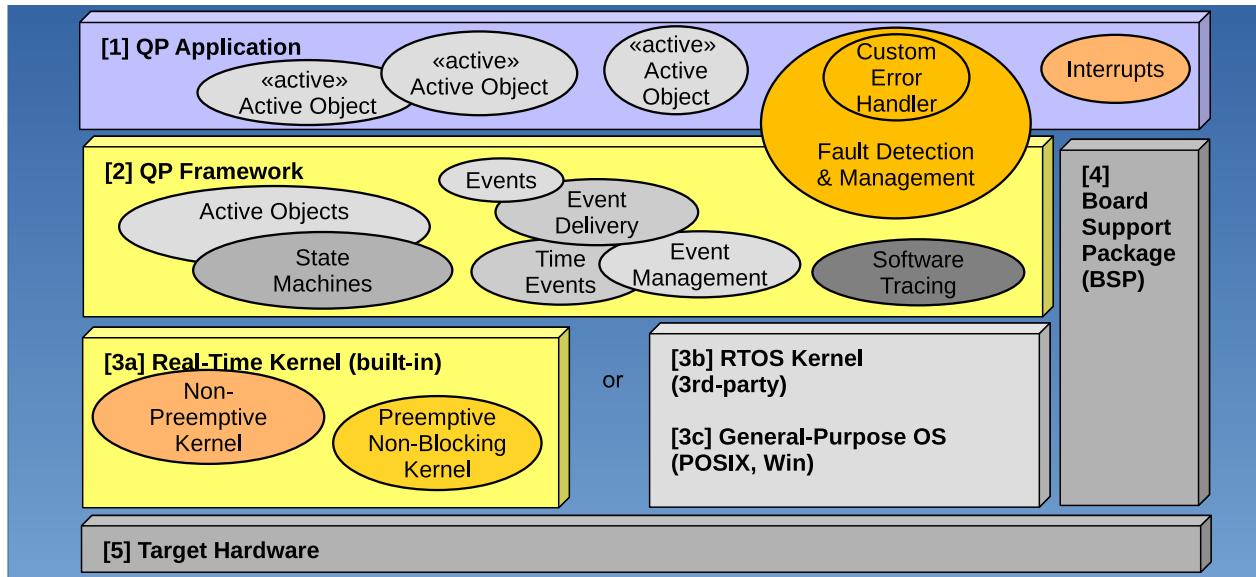
### 5.4.2 Functional Safety

In the context of **functional safety** standards and certification, QP Frameworks offer numerous advantages over the traditional "shared state concurrency" based on a "naked" Real-Time Operating System (RTOS). QP Frameworks implement an [inherently safer](#) model of concurrency and many best practices recommended by functional safety standards (e.g., [\[IEC 61508-7:2010\]](#)) such as:

- Structured methods ([\[IEC 61508-7:2010\] C.2.1](#))
- Semi-formal methods, such as Hierarchical State Machines ([\[IEC 61508-7:2010\] Table.B.7](#))
- Computer-aided design tools ([\[IEC 61508-7:2010\] B.3.5](#))
- Defensive programming, including Failure Detection and Failure Assertion Programming ([\[IEC 61508-7:2010\] C.2.5](#))
- Modular approach ([\[IEC 61508-7:2010\] Table.B.9](#))
- Design and coding standards, including safe subsets of C or C++ ([\[IEC 61508-7:2010\] C.2.6](#))
- Structured programming ([\[IEC 61508-7:2010\] C.2.7](#))
- Traceability between requirements and software design ([\[IEC 61508-7:2010\] C.2.11](#))

### 5.4.3 Context of Use

The following block diagram shows the context of use and the high-level functional decomposition of a system built with the QP Framework.



*Figure SRS-BLK: Block diagram showing the context and functional decomposition of a system based on QP Framework.*

[ 1 ] **QP Application:**

- defines event-driven **Active Objects** that collectively deliver the intended functionality;
- defines Custom Error Handler that is responsible for achieving Safe State;
- defines interrupts (Interrupt Service Routines) that produce events for the Active Objects

[ 2 ] **QP Framework:**

- executes **Active Objects** by dispatching **Events** to their internal **State Machines**;
- **delivers Events**;
- **manages memory for the mutable events**;
- provides **timing services**;
- provides Fault Detection and Management for QP Framework and QP Application;
- provides **software tracing system** for debugging, optimizing, and testing.

[ 3a ] **Real-Time Kernel** provides scheduling and execution context for the Active Objects. This could be one of the kernels built-into QP Frameworks (**QV**, **QK**, or **QXK**); or

[ 3b ] Alternatively, QP Framework can run on top of a 3-rd party **RTOS kernel**; or

[ 3c ] Alternatively, QP Framework can run on top of a **General-Purpose OS** (e.g., Linux/POSIX or Windows);

[ 4 ] **Board Support Package (BSP)** consists of the low-level support code;

[ 5 ] **Target Hardware** (MCU + external components).

### 5.4.3.1 Inversion of Control

Like most event-driven systems, QP Framework is based on [inversion of control](#), which means that the control of code execution resides in the QP Framework rather than the QP Application based on QP. Specifically, to handle events, QP Framework calls the QP Application, and not the other way around. Of course, QP Applications can also call services provided in QP Framework, but the *main* flow of control always begins with the QP Framework.

### 5.4.3.2 Framework, NOT a Library

That *inversion of control* gives the event-driven infrastructure (QP) all the defining characteristics of a [framework](#) rather than a library.

The main difference between a framework and a library is that when you use a library, such as a conventional RTOS, *you* write the main body of each thread, and *you* call the library code that you want to reuse (e.g., a semaphore). When you use a framework, such as QP, you reuse the main body and write the code *it* calls (*inversion of control*).

### 5.4.3.3 Deriving Application from a Framework

The mechanism for **deriving an application** from a software framework, such as QP, is by *inheriting* the base classes provided by the framework and *specializing* them for the application at hand. (See also the **extensibility** characteristics of a framework enumerated above.)

## 5.4.4 Main Use Cases

The use case diagram depicted in [Figure SRS-USE](#) shows the main users and typical use cases they engage in.

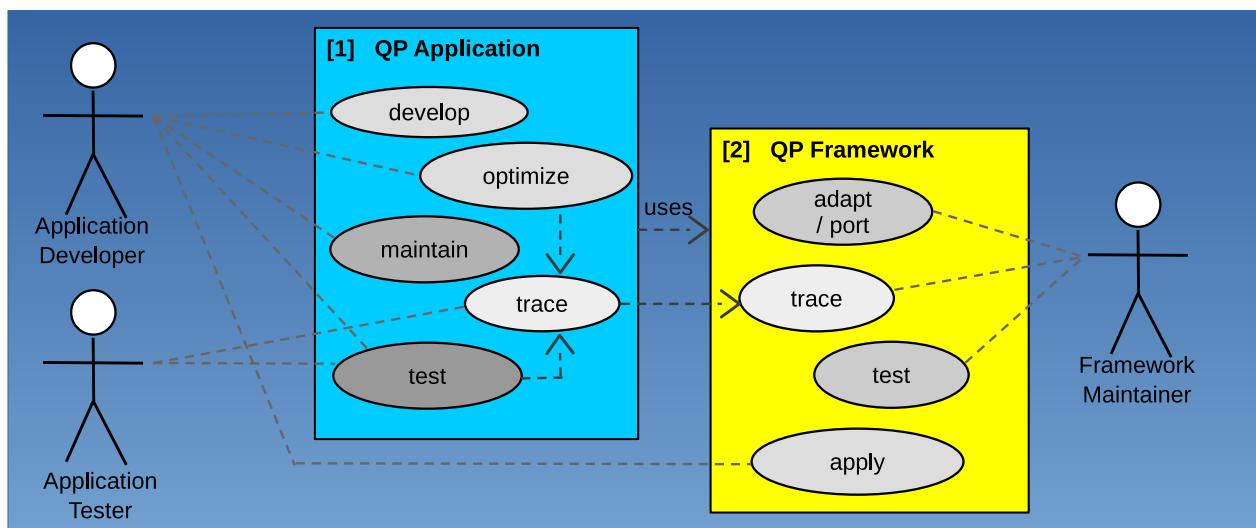


Figure SRS-USE: Users and main use cases of QP Application and QP Framework.

## 5.4.5 Portability & Configurability

A reusable architecture, like QP, needs to be adaptable to a wide range of application areas and target hardware selections. For that, QP needs to be highly configurable. Whenever this requirements specification mentions "configurability", it explicitly specifies which of the following types of configurability is required:

### 5.4.5.1 Compile-Time Configurability

Compile-time configurability means that the specific configuration option (out of a given range of possibilities) is chosen at compile-time and cannot be changed later (e.g., at run-time). Compile-time configurability is typically used for:

- Turning features on or off
- Determining the size (and dynamic range) of various data types
- Determining the maximum numbers of various objects in the framework or Application
- Determining the presence (or absence) of class members

#### Remarks

In C or C++, compile-time configurability is typically implemented with the preprocessor macros, but can also be achieved by composition of the selected modules.

#### 5.4.5.2 Run-Time Configurability

Run-time configurability means that the specific configuration option is chosen at run-time. Unlike compile-time configurability, run-time configurability is finer granularity, usually on an object-by-object basis. It is typically used for:

- Choosing a specific implementation of a class operation based on the object type (polymorphism)
- Selecting features by calling or not calling the specific QP API
- Selecting features by passing parameters to the specific QP API

## 5.5 Active Objects

### 5.5.1 Concepts & Definitions

As described in the [Overview](#), the main goal of the QP Framework is to provide a lightweight and efficient implementation of the **Active Object Model of Computation** with the specific focus on real-time embedded (RTE) systems, such as single-chip microcontroller units (MCUs).

#### 5.5.1.1 Active Object Model of Computation

The Active Object Model of Computation represents a paradigm shift compared to the traditional "shared state concurrency" based on explicit mutual exclusion and managing threads by blocking. The following subsections describe Active Object properties and explain why this model of computation is **safer** and makes it easier to write **correct concurrent software**.

#### 5.5.1.2 Active Objects

**Active Objects** (a.k.a., Actors) are autonomous software objects, each possessing an [event queue](#) and [execution context](#). They encapsulate state and behavior and communicate [asynchronously](#) by exchanging events. [Figure SRS-AOS](#) below shows a QP Application consisting of multiple, event-driven Active Objects that collectively deliver the desired functionality:

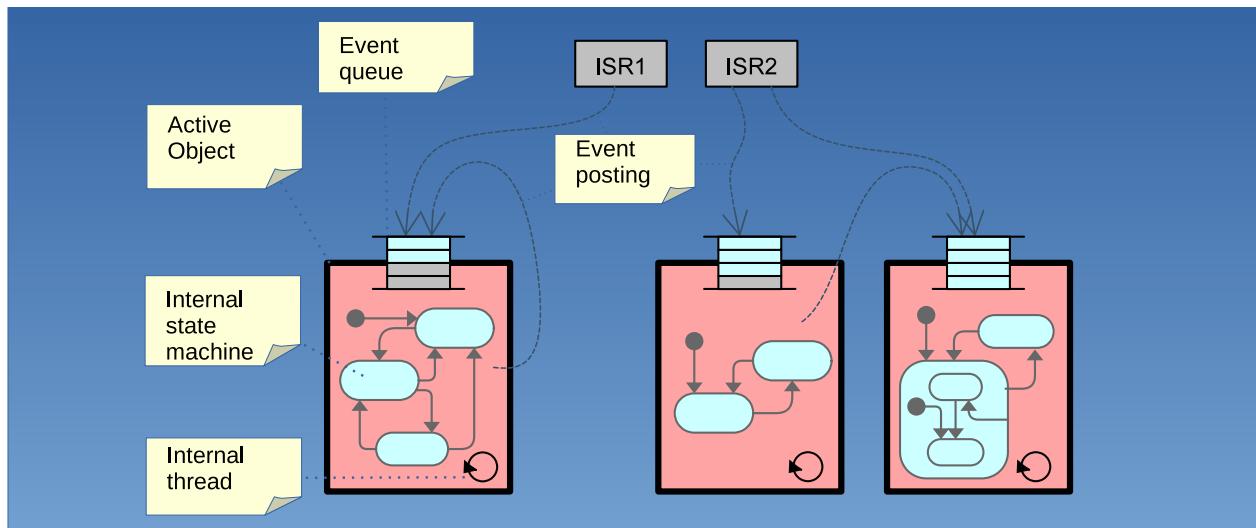


Figure SRS-AOS: Active Objects in QP

#### 5.5.1.3 Encapsulation for Concurrency

The traditional object-oriented encapsulation, as provided by C++, C# or Java, does not really encapsulate anything in terms of **concurrency**. As shown in [Figure SRS-SEQ-TH](#), any operation on a *passive* object still runs in the caller's thread. If that passive object is shared among multiple threads, the object's attributes are subject to the same race conditions as global data, not encapsulated at all. To become concurrency-safe, operations need to be explicitly protected by an appropriate mutual exclusion mechanism, such as a mutex (for threads) or a critical section (for Interrupt Service Routines). However, this reduces responsiveness (blocking or increased latency), causes contention, and often leads to missed real-time deadlines [Sutter:10]. Also this style of managing concurrency is **inherently unsafe** because developers might simply forget to apply mutual exclusion or use an incorrect mechanism, which can lead to latent, highly intermittent, hard-to-find and hard-to-fix concurrency bugs.

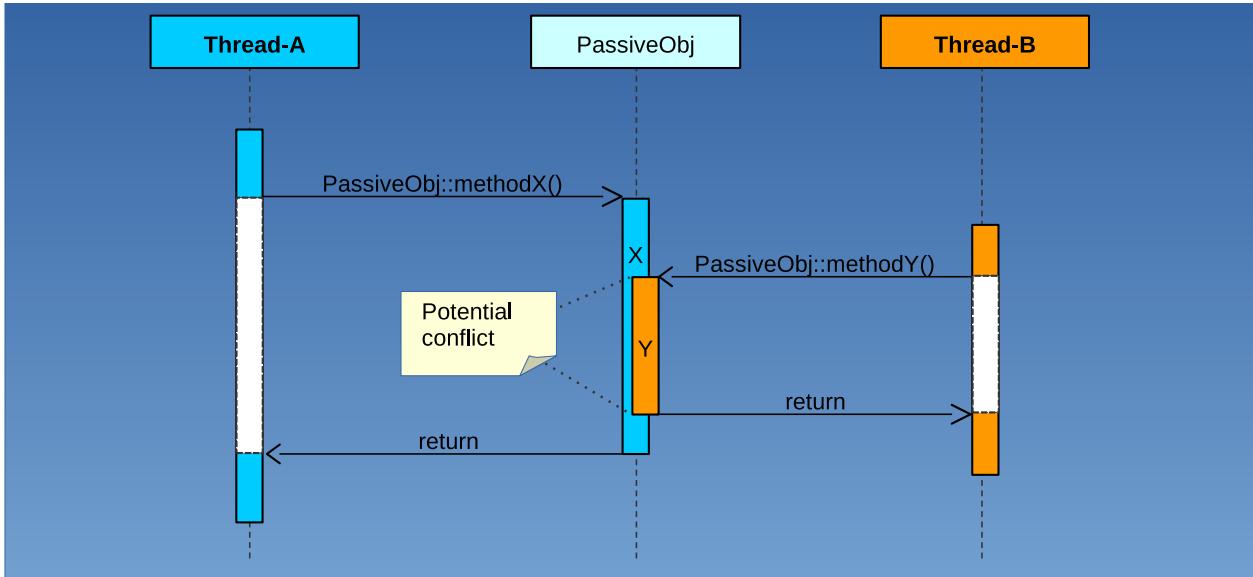


Figure SRS-SEQ-TH: Threads interacting (synchronously) with a passive object

In contrast, as shown in [Figure SRS-AOS](#), all interactions with an *Active Object* occur by posting events, which are all handled in the execution context of the Active Object. As long as there is **no sharing** of data or resources among Active Objects (or any other concurrent entities), there are **no concurrency hazards**. Also, because each event is processed to completion (see [Run-to-Completion processing](#)), event processing is naturally serialized. This means that an Active Object is truly encapsulated without any mutual exclusion mechanisms. In this sense Active Objects are the most stringent form of object-oriented programming because they enable strict **encapsulation for concurrency**, which is much **safer** than the "naked" threads.

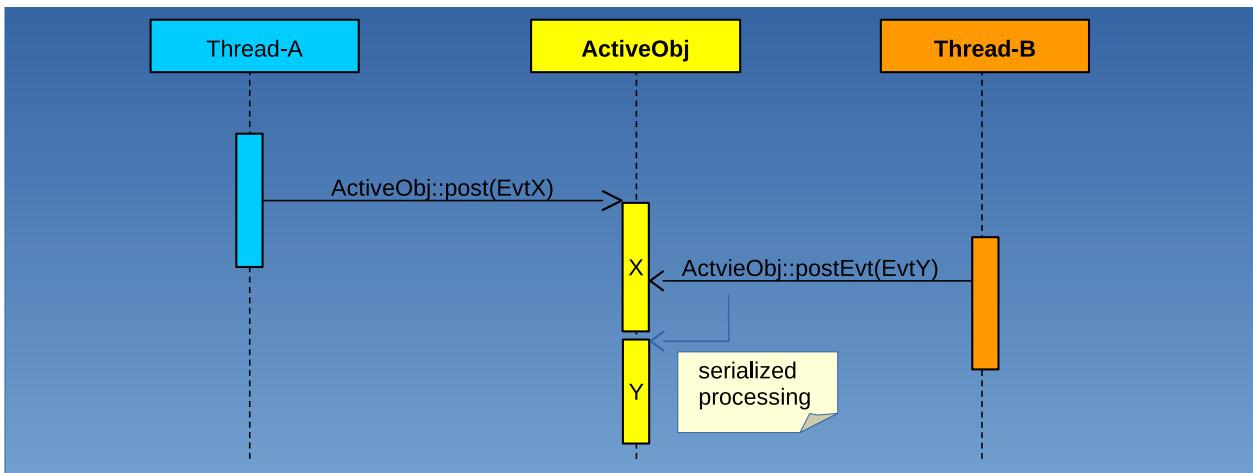


Figure SRS-SEQ-AO: Threads interacting (asynchronously) with an Active Object

#### 5.5.1.4 Shared-Nothing Principle

Encapsulation for concurrency is not a programming language feature, so it is no more difficult to achieve in C as in C++, but it requires a *programming discipline* on behalf of the application developers to **avoid sharing** resources ("shared-nothing" principle). However, the event-based communication helps immensely, because instead of sharing a resource, a dedicated Active Object can become the manager/broker of the resource and the rest of the system can access the resource only via *events* posted to this broker Active Object.

#### Attention

QP Framework by itself cannot and is not required to guarantee or enforce the *shared-noting* principle. Achieving the strict no-sharing of resources among Active Objects is the responsibility of the QP Applications. But at the very least, the QP Framework architecture and design must provide adequate and thread-safe *event-based* information exchange mechanisms, which can replace the traditional sharing of data or resources.

#### 5.5.1.5 Execution Context

In the UML, Active Object is defined as: "*the object having its own thread of control*" [UML 2.5]. A traditional thread might indeed be used for Active Objects when the QP Framework runs on top of a traditional multitasking kernel (e.g., traditional RTOS or general-purpose OS).

However, the Active Object model of computation can also work with real-time kernels that don't necessarily support the notion of traditional *blocking* threads. Active Objects have [no need for blocking](#) while handling events, which opens up possibilities of using lightweight, *non-blocking* kernels that might be non-preemptive or fully preemptive. (See also lightweight kernels provided in QP, such as [Non-Preemptive Kernel](#), [Preemptive Non-Blocking Kernel](#), and [Preemptive Dual-Mode Kernel](#).)

#### 5.5.1.6 Priority

The execution context of an Active Object is closely related to its **priority** relative to other Active Objects or "naked" threads in the system. In fact, an Active Object can be viewed primarily as a *priority level* for executing the functionality it encapsulates. In the QP Framework, each Active Object is required to have a **unique** priority. (See also [SRS\\_QP\\_AO\\_01](#).)

#### 5.5.1.7 Event Queue

Each Active Object has its own event queue and receives all events exclusively through that queue. This means that the event queue has only a single consumer (the Active Object that owns the queue). On the other hand, the event queue must accommodate multiple producers that don't need to be only Active Objects, but also interrupts (ISRs), or other software components (see [Figure SRS-AOS](#)). The Active Object infrastructure, such as the QP Framework in this case, is responsible for delivering and queuing the events in a *deterministic* and *thread-safe* manner.

#### 5.5.1.8 Asynchronous Communication

All events are delivered to Active Objects **asynchronously**, meaning that an event producer merely posts an event to the event queue of the recipient Active Object but doesn't wait in line for the actual processing of the event.

The QP Framework makes no distinction between external events generated from interrupts and internal events originating from Active Objects. As shown in [Figure SRS-AOS](#), an Active Object can post events to any other Active Object, including to self. All events are treated uniformly, regardless of their origin.

#### 5.5.1.9 Run-to-Completion (RTC)

Each Active Object processes events in **run-to-completion (RTC) fashion**, which must be guaranteed by the underlying QP Framework. RTC means that Active Objects process the events one at a time and the next event can only be processed after the previous event has been processed completely. RTC event processing is the essential requirement for proper execution of [state machines](#).

#### Note

It is very important to clearly distinguish the notion of RTC from the concept of preemption [OMG 07]. In particular, **RTC** does *not* mean that the Active Object thread has to monopolize the CPU until the RTC step is complete. In fact, RTC steps *can* be preempted by interrupts or other threads executing on the same CPU. Such thread preemption is determined by the scheduling policy of the underlying multitasking kernel, not by the Active Object model. When the preempted Active Object is assigned the CPU time again, it resumes its event processing from the point of preemption and, eventually, completes its RTC step. As long as the preempting and the preempted threads don't share any resources (see [Encapsulation](#)), there are no concurrency hazards.

### 5.5.1.10 Current Event

**Current event** is the event being processed in the run-to-completion (RTC) step. The event-driven infrastructure (QP Framework in this case) must guarantee that the *current event* remains available and unchanging for the whole duration of the RTC step.

### 5.5.1.11 No Blocking

Most traditional operating systems manage the threads and all inter-thread communication based on *blocking*, such as waiting on a semaphore or a time-delay. However, blocking (as in the middle of the RTC step) is *incompatible* with the **RTC event processing requirement**. This is because every blocking call is really another way to deliver an *event* (event is delivered by unblocking and return from a blocking call). Such "backdoor" event delivery happening in the middle of the RTC step violates the RTC semantics, because after unblocking the Active Object needs to process two events at a time (the original one and the new one delivered by unblocking).

Another detrimental consequence of blocking (or polling for events) inside RTC steps is that Active Objects become *unresponsive* to events delivered to their event queues. This, in turn, can cause Active Objects to miss their hard-real time deadlines and also can cause overflow of the internal event queue.

Finally, blocking (or polling for events) means that the expected sequences of events are *hard-coded*, which is inherently inflexible and not extensible, especially if the system must handle multiple event sequences (which turns out to be the case in most real-life systems).

### 5.5.1.12 Support For State Machines

Event-driven components, like Active Objects, must often retain the execution context from one event to the next. This must be done *without blocking*, so the context can't be preserved on the call-stack as it is done in the traditional, blocking RTOS threads. Instead, the context between events must be preserved in some other way, which often leads to multitude of variables and flags checked and modified in a convoluted if-then-else logic (a.k.a. "spaghetti code"). A well known alternative to such "improvised context management" is the concept of a **state machine**, which manages the execution context using "states". QP Framework augments and complements the Active Object model of computation by providing support for **state machines** to represent the internal behavior of Active Objects.

#### Remarks

The relationship between Active Objects and state machines is mutually synergistic. On one hand, the Active Objects provide the execution context and event queuing that the state machines need to process the events in a *run-to-completion* fashion. On the other hand, state machines provide the structure and clear design for the event-driven behavior running inside the Active Objects. State machines are also the most constructive part of the design amenable to modeling and automatic code generation.

### 5.5.1.13 Inversion of Control

Event-driven systems require a distinctly different way of thinking than traditional sequential threads. When a sequential thread needs some incoming event, it explicitly *blocks* and waits in-line until the event arrives. Thus the sequential thread remains "in control" at all times, but while waiting for one kind of event, it cannot respond (at least for the time being) to any other events.

In contrast, most event-driven applications are structured according to the Hollywood principle, which essentially means "Don't call us, we'll call you." So, an event-driven Active Object is not in control while waiting for an event; in fact, it's not even active. Only once the event arrives, the event-driven program is called to process the event and then it quickly relinquishes the control again. This arrangement allows an event-driven program to wait for many events in parallel, so the system remains *responsive* to all events it needs to handle. This scheme implies that in an event-driven system the control resides within the event-driven infrastructure (QP Framework), rather than in the application. In other words, the control is *inverted* compared to a traditional sequential thread.

### 5.5.1.14 Framework vs. Toolkit

Inversion of control is the key property that makes a software framework different from a software toolkit. A toolkit, such as a traditional RTOS, is essentially a set of predefined functions that you can call. When you use a toolkit, you write

the main body of the application, such as the body of all RTOS threads, and call the various blocking functions from the RTOS. When you use a framework (such as QP), you reuse the main body (codified inside the framework) and provide the application code that it calls, so the control resides in the framework rather than in your code. Indeed, this **inversion of control** gives the event-driven infrastructure all the defining characteristics of a framework:

*"One important characteristic of a framework is that the methods defined by the user to tailor the framework will often be called from within the framework itself, rather than from the user's application code. The framework often plays the role of the main program in coordinating and sequencing application activity. This <u>inversion of control</u> gives frameworks the power to serve as extensible skeletons. The methods supplied by the user tailor the generic algorithms defined in the framework for a particular application."*

—Ralph Johnson and Brian Foote

### 5.5.1.15 Low Power Architecture

Most modern embedded microcontrollers (MCUs) provide an assortment of low-power sleep modes designed to conserve power by gating the clock to the CPU and various peripherals. However, the sleep modes are entered under the software control and therefore require an appropriate software infrastructure.

An event-driven framework, like QP, based on inversion of control is particularly suitable for taking advantage of these power-savings features because the framework can easily detect situations in which the system has no more events to process, called the *idle condition*. In that case the framework can place the MCU into a low-power sleep mode **safely** and without creating race conditions with active interrupts.

## 5.5.2 Requirements

### 5.5.2.1 SRS\_QP\_AO\_00

QP Framework shall provide the Active Object abstraction to QP Application

#### Description

The Active Object abstraction provided by QP Framework shall be customizable by QP Application and executed by QP Framework according to the Active Object model of computation.

#### Use Case

QP Framework can meet this requirement by proving the Active Object abstraction as a *class*, which the QP Applications can customize by subclassing. Such an Active Object base class must be customizable at compile-time for a wide variety of real-time kernels, including traditional blocking RTOS and event-driven, non-blocking kernels.

#### Forward Traceability (truncated to 2 level(s))

- [SRS\\_QP\\_AO\\_01](#): *QP Framework shall be able to manage a compile-time configurable number of Active Objects not exceeding 64 instances*
- [SAS\\_QP\\_CLS](#): *QP Framework base classes*

---

### 5.5.2.2 SRS\_QP\_AO\_01

QP Framework shall be able to manage a compile-time configurable number of Active Objects not exceeding 64 instances

#### Description

The maximum number of Active Object instances managed by QP Framework at any given time shall be compile-time configurable with the maximum of 64 instances. The actual number of Active Object instances registered with QP Framework can be lower than the configured compile-time limit, but it cannot exceed the limit.

### Backward Traceability

- [SRS\\_QP\\_AO\\_00](#): *QP Framework shall provide the Active Object abstraction to QP Application*

### Use Case

For best memory and CPU performance, the maximum number of Active Object instances can be configured below the 64 maximum, and often it might be advantageous to use powers of 2, for example, configure the limit as 16 or 32 AO instances.

### Forward Traceability (truncated to 2 level(s))

---

#### 5.5.2.3 SRS\_QP\_AO\_10

Active Object abstraction shall provide the unique priority for each Active Object instance

##### Description

The Active Object **priority** is a positive integer number between 1 and the maximum configured number of Active Objects (see [SRS\\_QP\\_AO\\_20](#)). The priority shall conform to the numbering scheme defined as follows: priority 1 corresponds to the lowest priority and higher numbers correspond to higher priorities (direct priority numbering scheme). The priority 0 cannot be assigned to any Active Object, and is reserved for the idle thread (or the idle condition) of the underlying real-time kernel.

##### Use Case

The QP priority numbering scheme is fixed and does not change, even if the underlying real-time kernel uses a different priority scheme (e.g., reversed priority numbering). However, the QP priority must remain consistent with the priority of the underlying kernel in that higher QP priority numbers must correspond to a higher (or at least not lower) *urgency* threads.

### Forward Traceability (truncated to 2 level(s))

---

#### 5.5.2.4 SRS\_QP\_AO\_11

Active Object abstraction may provide second "auxiliary priority" for each Active Object instance

##### Description

The second "auxiliary priority" may be used for different purposes, depending on the underlying real-time kernel.

##### Use Cases

In some real-time kernels, the "auxiliary priority" might be used to represent preemption threshold. In other kernels, it might represent the native thread priority according to the priority numbering scheme of the kernel.

### Forward Traceability (truncated to 2 level(s))

---

### 5.5.2.5 SRS\_QP\_AO\_20

Active Object abstraction shall provide an event queue for each Active Object instance

#### Description

The event queue type shall be compile-time configurable to allow various blocking/no-blocking mechanisms corresponding to the chosen real-time kernel. The capacity of the event queue (maximum number of events it can hold) shall be run-time configurable before the Active Object instance starts executing.

#### Use Case

QP Framework can meet this requirement by proving event queue as an attribute inside the Active Object class. The type of the event queue must be compile-time configurable to match the underlying real-time kernel that executes the Active Objects in QP.

#### Forward Traceability (truncated to 2 level(s))

- [SRS\\_QP\\_AO\\_21](#): Active Object event queue shall provide FIFO policy for posting events from outside the Active Object
  - [SRS\\_QP\\_EDM\\_00](#): QP Framework shall provide direct event posting to Active Object instances based on the FIFO policy
  - [SRS\\_QP\\_EDM\\_01](#): QP Framework shall provide direct event self-posting to Active Object instances based on the LIFO policy
- [SRS\\_QP\\_AO\\_22](#): Active Object event queue shall additionally provide LIFO policy for self-posting events from within the Active Object
- [SRS\\_QP\\_AO\\_23](#): The maximum capacity of the Active Object event queue shall be run-time configurable
- [SRS\\_QP\\_EDM\\_00](#): QP Framework shall provide direct event posting to Active Object instances based on the FIFO policy
  - [SAS\\_QP\\_API](#): QP Framework API
- [SRS\\_QP\\_EDM\\_01](#): QP Framework shall provide direct event self-posting to Active Object instances based on the LIFO policy
  - [SAS\\_QP\\_API](#): QP Framework API

### 5.5.2.6 SRS\_QP\_AO\_21

Active Object event queue shall provide FIFO policy for posting events from outside the Active Object

#### Description

FIFO stands for First-In-First-Out and means that the events are extracted from the queue in the *same* order in which they have been inserted into the queue.

#### Backward Traceability

- [SRS\\_QP\\_AO\\_20](#): Active Object abstraction shall provide an event queue for each Active Object instance

#### Forward Traceability (truncated to 2 level(s))

- [SRS\\_QP\\_EDM\\_00](#): QP Framework shall provide direct event posting to Active Object instances based on the FIFO policy
  - [SAS\\_QP\\_API](#): QP Framework API
- [SRS\\_QP\\_EDM\\_01](#): QP Framework shall provide direct event self-posting to Active Object instances based on the LIFO policy
  - [SAS\\_QP\\_API](#): QP Framework API

### 5.5.2.7 SRS\_QP\_AO\_22

Active Object event queue shall additionally provide LIFO policy for self-posting events from within the Active Object Description

LIFO stands for Last-In-First-Out and means that the events are extracted from the queue in the *reversed* order in which they have been inserted into the queue. The support for the LIFO policy for self-posting should be in *addition* to supporting the standard FIFO policy. In other words, QP Application can choose to self-post any given event either with the FIFO or LIFO policy.

Backward Traceability

- [SRS\\_QP\\_AO\\_20](#): Active Object abstraction shall provide an event queue for each Active Object instance

Forward Traceability (truncated to 2 level(s))

---

### 5.5.2.8 SRS\_QP\_AO\_23

The maximum capacity of the Active Object event queue shall be run-time configurable Description

The maximum capacity of an event queue is the maximum number of events that can be inside the queue at any given time. This maximum capacity shall be determined at run-time.

Use Case

The event queue can be supplied with the buffer memory to hold the events at run-time. The size of that memory buffer will determine the maximum capacity of the event queue.

Backward Traceability

- [SRS\\_QP\\_AO\\_20](#): Active Object abstraction shall provide an event queue for each Active Object instance

Forward Traceability (truncated to 2 level(s))

---

### 5.5.2.9 SRS\_QP\_AO\_30

Active Object abstraction may provide an optional execution context for each Active Object instance Description

The execution context (e.g., thread) attribute shall be compile-time configurable to allow various thread types corresponding to the chosen real-time kernel.

Use Case

In case QP uses a traditional RTOS, the execution context might be a thread-control-block (TCB) or just a thread handle/pointer. In case of other real-time kernels, the execution context might be a thread identifier or might not be needed at all.

Forward Traceability (truncated to 2 level(s))

---

### 5.5.2.10 SRS\_QP\_AO\_31

QP Framework shall allow Active Object instances to be started at runtime

#### Description

Starting an Active Object instance means registering it with the QP Framework, so that the framework can start managing the Active Object. Only after an Active Object instance has been started, it can receive and process events. Starting at runtime means that Active Object instances can start at any time during the normal system operation (as opposed to starting only during system initialization).

#### Use Case

Starting an Active Object instance might involve initializing its event queue and creating and/or starting the execution context. Only after that, the underlying real-time kernel can include the Active Object in the scheduling process.

#### Forward Traceability (truncated to 2 level(s))

---

### 5.5.2.11 SRS\_QP\_AO\_32

QP Framework may allow Active Object instances to be stopped at runtime

#### Description

Stopping an Active Object instance means stopping its execution context and unregistering it from QP Framework. Stopping an Active Object does not mean that it is deleted, destroyed, or its memory is recycled. Stopping only means that the Active Object is no longer managed by QP Framework and stops participating in scheduling and event processing.

#### Background

The biggest challenge in stopping an Active Object is to perform it **cleanly**, without disrupting the rest of the application. For example, the stopped Active Object should not receive any events from the rest of the application. Also, the Active Object should not have any events to process in its queue. For these reasons, stopping an Active Object is optional, and is **not recommended**, especially in safety-related applications. Instead of stopping an Active Object, a better design is to post a special event to that Active Object, which could trigger a transition to a special "stopped" state in the Active Object's state machine.

#### Forward Traceability (truncated to 2 level(s))

---

### 5.5.2.12 SRS\_QP\_AO\_40

Active Object abstraction may provide optional "operating-system object" for each Active Object instance

#### Description

The optional "operating system object" shall be compile-time configurable to allow various operating-system object types corresponding to the selected real-time kernel.

**Use Case**

For example, the Active Object's event-queue for the POSIX operating system might require additional condition-variable attribute ("operating system object") to implement blocking on an empty queue.

**Forward Traceability (truncated to 2 level(s))**

---

**5.5.2.13 SRS\_QP\_AO\_50**

Active Object abstraction shall encapsulate its internals

**Description**

The Active Object abstraction should hide and protect its internals, both for reading and writing by any outside entities. Additionally, QP Framework shall allow the QP Application to hide and protect any additional attributes added to the derived Active Objects.

**Forward Traceability (truncated to 2 level(s))**

- [SRS\\_QP\\_AO\\_51](#): *Active Object abstraction shall allow Applications to easily access the internal attributes from inside the Active Object*
- 

**5.5.2.14 SRS\_QP\_AO\_51**

Active Object abstraction shall allow Applications to easily access the internal attributes from inside the Active Object

**Description**

Notwithstanding Requirement [SRS\\_QP\\_AO\\_10](#), QP Framework shall allow for easy and computationally inexpensive access to the internal attributes of an Active Object from *within* the AO, such as from its internal state machine.

**Use Case**

A good example of implementing such a policy is the concept of class encapsulation in OOP, where the internal attributes are accessible to the class operations (e.g., via the `this` pointer) and are harder to access from the outside.

**Backward Traceability**

- [SRS\\_QP\\_AO\\_50](#): *Active Object abstraction shall encapsulate its internals*
- [SRS\\_QP\\_SM\\_24](#): *All State Machine Implementation Strategies provided by QP shall allow Applications to easily access the instance variables associated with a given state machine object*

**Forward Traceability (truncated to 2 level(s))**

**5.5.2.15 SRS\_QP\_AO\_60**

Active Object abstraction shall support run-to-completion event processing

**Description**

QP Framework must guarantee that every event is processed to completion, regardless of the real-time kernel used, and that during the RTC step the [current event](#) remains available and unchanged.

---

**5.5.2.16 SRS\_QP\_AO\_70**

Active Object abstraction shall provide support for state machines.

**Description**

Each Active Object instance shall have an internal state machine, with the features and semantics specified in Section [State Machines](#). QP Framework shall guarantee execution of such internal state machines in Run-to-Completion fashion.

**Backward Traceability**

- [SRS\\_QP\\_SM\\_00](#): *QP Framework shall provide support for hierarchical state machines both for Active Objects and for passive event-driven objects in the Application*

**Forward Traceability (truncated to 2 level(s))**

## 5.6 Events

### 5.6.1 Concepts & Definitions

An **Event** is the specification of some occurrence that may potentially have significance to the system under consideration. Upon such an occurrence, a corresponding Event (*class*) is instantiated, and the generated **Event Instance** (*object*) outlives the instantaneous occurrence that generated it.

Event Instances (a.k.a. *Messages*) are objects specifically designed for communication and synchronization within an event-driven system, such as a QP Application. Once generated, an Event Instance (a.k.a. Message) is propagated to the system, where it can trigger various *Actions*. These *Actions* might involve generating secondary Event Instances.

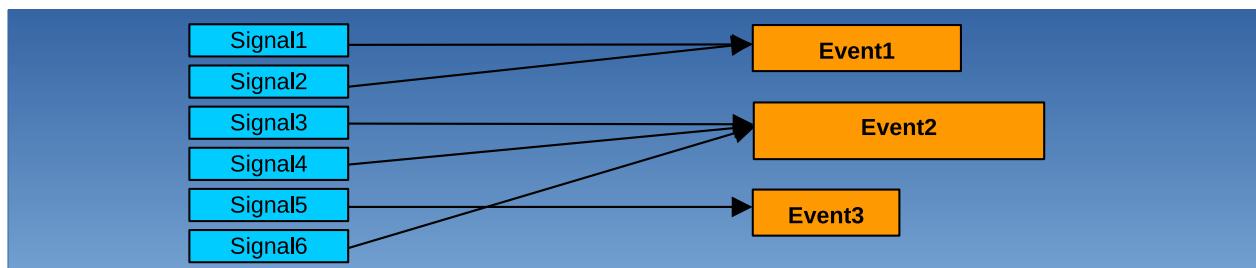
#### Remarks

This section describes events from the perspective of using them in [State Machines](#) and [Active Objects](#).

Additional concepts and requirements related to event management in QP Framework are discussed in sections [Event Delivery Mechanisms](#) and [Event Memory Management](#).

#### 5.6.1.1 Event Signal

An Event always has information about the type of occurrence that generated the Event Instance ("what happened"), which will be called **Signal** in this document. Event *Signals* are typically enumerated constants that indicate which specific occurrence out of a set of all possible (enumerated) occurrences has generated the Event Instance.



*Figure SRS-EVT-SIG: Relationship between Event-Signals and Events. A Signal must unambiguously identify an Event type.*

#### 5.6.1.2 Event Parameters

An Event can have (optionally) associated **Parameters**, allowing the Event Instance to convey the quantitative information regarding that occurrence. For example, a Keystroke event generated by pressing a key on a computer keyboard might have associated parameters that carry the character scan code and the status of the Shift, Ctrl, and Alt keys.

## 5.6.2 Requirements

### 5.6.2.1 SRS\_QP\_EVT\_00

QP Framework shall provide Event abstraction to QP Application

#### Description

The QP Framework shall define a common Event abstraction, so that it can be instantiated, exchanged, and correctly interpreted by both the QP Application and QP Framework.

Forward Traceability (truncated to 2 level(s))

- [SAS\\_QP\\_CLS](#): *QP Framework base classes*
  - [SDS\\_QP\\_QEvt](#): *QEvt event class*.
    - [QEvt](#): *Event class*
- 

### 5.6.2.2 SRS\_QP\_EVT\_20

Each event instance shall contain the event Signal

#### Description

The event Signal carries the information about the occurrence that generated the given event instance. The Signal needs to be easily accessible to the QP Application to determine how to handle a given event.

Forward Traceability (truncated to 2 level(s))

- [SDS\\_QP\\_QEvt](#): *QEvt event class*.
    - [QEvt](#): *Event class*
  - [QEvt::sig](#): *Signal of the event (see Event Signal)*
- 

### 5.6.2.3 SRS\_QP\_EVT\_21

The dynamic range of the event signal shall be compile-time configurable

#### Description

QP Framework should be compile-time configurable to allow choosing the optimal dynamic range for the event signals while minimizing the size of the event instances. The following dynamic range of event signals should be configurable: 255 signals (1 byte of memory per Signal), 64K signals (2 bytes of memory per Signal), and 4G signals (4 bytes of memory per Signal).

Forward Traceability (truncated to 2 level(s))

---

### 5.6.2.4 SRS\_QP\_EVT\_22

QP Framework may reserve some signals for its internal use

#### Description

QP Framework may need some reserved signals to implement features inside hierarchical state machines. The reserved signals shall have the lowest numerical values.

Forward Traceability (truncated to 2 level(s))

---

### 5.6.2.5 SRS\_QP\_EVT\_23

QP Framework shall provide the lowest numerical value of signals allowed in the Application

#### Description

To avoid conflicts around any *reserved* event signals, QP Framework shall provide the lowest numerical limit for Application-level signals. In other words, the reserved signals will have numerical values 0 .. (SIGNAL\_OFFSET - 1), and the Application-level signals will have values SIGNAL\_OFFSET .. SIGNAL\_DYNAMIC\_RANGE.

Forward Traceability (truncated to 2 level(s))

---

### 5.6.2.6 SRS\_QP\_EVT\_30

QP Framework shall allow Application to create event instances with Parameters defined by the Application

#### Description

The Event abstraction shall provide a mechanism for QP Application to define event parameters useful for the QP Application, and then to create event instances with these parameters.

Forward Traceability (truncated to 2 level(s))

- [SDS\\_QP\\_QEvt](#): *QEvt event class.*
    - [QEvt](#): *Event class*
  - [QEvt::refCtr\\_](#): *Event reference counter.*
  - [QEvt::poolNum\\_](#): *Event pool number of this event*
  - [Q\\_EVT\\_CAST](#): *Perform downcast of an event onto a subclass of [QEvt class](#)*
- 

### 5.6.2.7 SRS\_QP\_EVT\_31

Event abstraction may contain other data items for internal event management inside QP Framework

#### Description

Event management in QP Framework might require additional information to be stored inside each event instance. Please see [Event Memory Management](#) for requirements related to event management inside QP Framework.

Forward Traceability (truncated to 2 level(s))

- [SDS\\_QP\\_QEvt](#): *QEvt event class.*
    - [QEvt](#): *Event class*
  - [QEvt::refCtr\\_](#): *Event reference counter.*
  - [QEvt::poolNum\\_](#): *Event pool number of this event*
-

### 5.6.2.8 SRS\_QP\_EVT\_40

QP Framework may be compile-time configurable to allow customized constructor and other custom operations on event instances

#### Description

When such customization is configured, QP Framework shall invoke the custom construction upon instantiation of an event.

#### Forward Traceability (truncated to 2 level(s))

- [QEvt::QEvt\\_init\(\)](#): *Event without parameters initialization*
- 

### 5.6.2.9 SRS\_QP\_EVT\_41

QP Framework may be compile-time configurable to allow customized destructor of event instances

#### Description

When such customization is configured, QP Framework shall invoke the custom destructor right before recycling the event instance.

#### Forward Traceability (truncated to 2 level(s))

---

## 5.7 State Machines

### 5.7.1 Concepts & Definitions

Event-driven systems work by [responding to Events](#). In general, the system's response to a given Event depends both on the nature of that Event (captured in its [Signal](#)) and on the *history of events* the system has received. In practice not all aspects of the full "history of past events" are relevant. The simplified history consisting only of aspects that are consequential for the system's response to *future events* is called the **Relevant History**.

#### 5.7.1.1 State

**State** is an *equivalence class* of past histories of a system, all of which are equivalent in the sense that the future behavior of the system given any of these past histories will be identical. Thus, the concept of "State" is the most efficient representation of the Relevant History of the system. It is the minimum information that captures only the relevant aspects for the future behavior and abstracts away all irrelevant aspects.

#### 5.7.1.2 Transition

**Transition** is a change from one State to another during the lifetime of a system. In event-driven systems, a change from one state to another can be caused only by an event. The events that triggers a Transition is called **Triggering Event** or just **Trigger** of the Transition.

#### 5.7.1.3 State Machine

**State Machine** is the set of all States (*equivalence classes* of relevant histories), plus all the Transitions (rules for changing States). An important benefit of the State Machine formalism is the expressive graphical representation of State Machines in form of **state diagrams**.

#### Note

This definition pertains to event-driven State Machines, which is the only kind supported in QP Framework. The definition does not cover "input-driven" state machines or other types of state machines.

#### 5.7.1.4 Hierarchical State Machine

**Hierarchical State Machine** (a.k.a. UML statechart) is an advanced formalism which extends the traditional state machines in several ways. The most important innovation of UML state machines over classical state machines is the introduction of **hierarchically nested states**. The value of state nesting lies in avoiding repetitions, which are inevitable in the traditional "flat" state machine formalism. The semantics of state nesting allow substates to define only the *differences* in behavior from the superstates, thus promoting sharing and reuse of behavior.

#### 5.7.1.5 State Machine Implementation Strategy

State Machines, and Hierarchical State Machines, in particular, can be implemented in many different ways. A specific way of implementing a state machine will be called here a **State Machine Implementation Strategy**, and it can be characterized by the following properties:

- efficiency in time (CPU cycles)
- efficiency in data space (RAM footprint)
- efficiency in code space (ROM footprint)
- monolithic vs. partitioned with various levels of granularity
- maintainability (with manual coding)
- maintainability (via automatic code generation)

- traceability from design (e.g., state diagram) to code
- traceability from code back to design
- other, quality attributes (non-functional requirements)

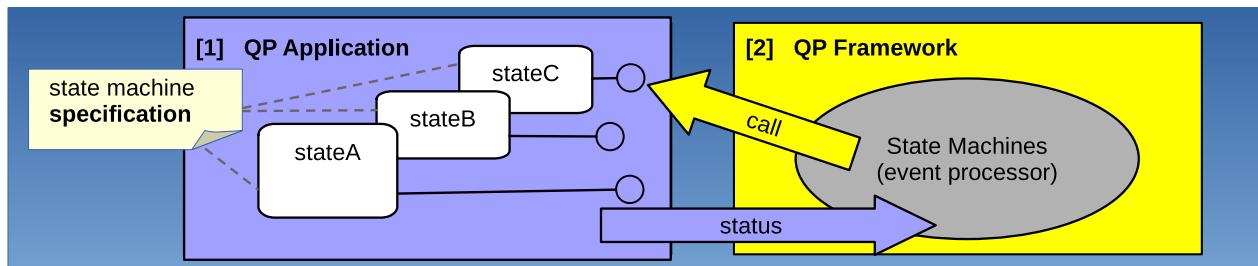
No single state machine implementation strategy can be optimal for all circumstances, and therefore QP Framework shall support multiple and interchangeable strategies (see [SRS\\_QP\\_SM\\_20](#)).

#### 5.7.1.6 Initializing a State Machine

Every state machine must be initialized before it can process any events. The initialization is accomplished by executing the top-most initial transition, which must be present in every well-formed state machine.

#### 5.7.1.7 Dispatching Events to a State Machine

The event processing inside a state machine is called **dispatching** an event to the state machine, and it requires interaction between the QP Framework and the QP Application, as illustrated in [Figure RS-SM-DIS](#). The process of *event dispatching* consists of multiple interactions between the **State Machine Processor** inside QP Framework and **State Machine Specification** inside QP Application. The framework chooses with elements of the "State Machine Specification" to call (e.g., states). The "Specification" then performs the actions and returns the *status* of processing (e.g., a transition has been taken) back to the "State Machine Processor". Based on this status, the "State Machine Processor" decides which element of the "Specification" to call next or that the processing is complete.



*Figure SRS-SM-DIS:Event Dispatching to a State Machine in QP Framework*

##### 5.7.1.7.1 State Machine Specification

The "State Machine Specification" is provided inside the QP Application and is prepared according to the rules defined by the chosen **State Machine Implementation Strategy** in QP Framework. Typically an implementation strategy represents a state machine as several elements, such as states, transitions, etc.

The "State Machine Specification" can mean state machine code (when the state machine is coded manually) or a state machine model (when the state machine is specified in a modeling tool, like ["QM"](#)). Either way, it is highly recommended to *think* of the state machine implementation as the **specification** of state machine elements, not merely code. This notion of "specifying" a state machine rather than coding it can be reinforced by selecting an expressive and fully *traceable* state machine implementation strategy, see [SRS\\_QP\\_SM\\_40](#). The advantage of a traceable implementation is that each artifact at all levels of abstraction (design to code) unambiguously represents an element of a state machine.

##### 5.7.1.7.2 State Machine Processor

A state machine is executed in QP Framework by the "State Machine Processor" that decides which elements of the "State Machine Specification" to call. Once called, the chosen part of the "State Machine Specification" executes some actions and *returns* back to the "State Machine Processor" (QP Framework) with the status information as to what has happened. For example, the returned status might inform the "State Machine Processor" that a state transition needs to be taken, or that the event needs to be propagated to the superstate in the hierarchical state machine.

### 5.7.1.7.3 Run To Completion (RTC) Processing

The "State Machine Processor" is a *passive* software component that needs to be explicitly called from some context of execution (e.g., thread) to dispatch each event to the given state machine object. The most important restriction is that the dispatch operation must necessarily run to completion (**Run-to-Completion** processing) before another event can be dispatched to the same state machine object.

RTC event processing means, among others, that a state machine should **NOT** block or busy-poll for events (e.g., a semaphore-wait or busy-delay) because every such blocking or busy-polling represents waiting for an *event*, which will be delivered immediately after the call unblocks. The problem is that such a "backdoor" event is delivered before the original RTC step completes, thus violating the RTC semantics. Blocking inside a state machine also extends the RTC processing and makes the state machine unresponsive to new events.

### 5.7.1.7.4 Current State

Between the discrete **RTC steps**, the state machine remains in a stable state configuration, which is called the **current state**. The *current state* changes in real-time as the state machine executes RTC steps and transitions from one state to another. Because the state changes occur in run-time, every state machine must store its state in a *variable*, which is called **state variable**.

### 5.7.1.7.5 Current Event

The event that has been dispatch to the state machine is called the **current event**. This current event must not change and must be accessible to the state machine over the entire **RTC step**.

## 5.7.2 Requirements

### 5.7.2.1 SRS\_QP\_SM\_00

QP Framework shall provide support for hierarchical state machines both for Active Objects and for passive event-driven objects in the Application

#### Description

Support for hierarchical state machines (HSMs) means that QP Framework shall provide a set of rules for **State Machine Specifications** (rules for coding state machines in the QP Application) as well as the matching implementation of the **State Machine Processor** (inside the QP Framework) to handle events according to HSM semantics defined in requirements in this section.

#### Background

In QP Framework, state machines can be associated only with *objects*, which provide the execution context (e.g., the data and other resources accessed by the state machine, see also [SRS\\_QP\\_SM\\_22](#)). These objects can be both active and passive. Active Objects are specified in the dedicated [section of this requirement specification document](#). Passive objects with a state machine can be useful as event-driven components ("Orthogonal Components") inside Active Objects or inside device drivers, Interrupt Service Routines (ISRs), or other parts of the system.

#### Forward Traceability (truncated to 2 level(s))

- [SRS\\_QP\\_AO\\_70](#): Active Object abstraction shall provide support for state machines.
- [SAS\\_QP\\_CLS](#): QP Framework base classes
- [SDS\\_QP\\_QAsm](#): QAsm Abstract state machine class.
  - [QAsm](#): Abstract State Machine class (state machine interface)
- [QHsm](#): Hierarchical State Machine class (QHsm-style state machine implementation strategy)
  - [QHsm::QHsm\\_ctor\(\)](#): Constructor of the [QHsm](#) base class

- `QHsm::QHsm_init_()`: Implementation of the top-most initial transition in `QHsm`
  - `QHsm::QHsm_dispatch_()`: Implementation of dispatching events to a `QHsm`
  - `QHsm::QHsm_isIn_()`: Check if a given state is part of the current active state configuration
  - `QHsm::QHsm_state()`: Obtain the current active state from a HSM (read only)
  - `QHsm::QHsm_childState()`: Obtain the current active child state of a given parent in `QHsm`
  - `QActive`: Active object class (based on the `QHsm` implementation strategy)
- 

### 5.7.2.2 SRS\_QP\_SM\_01

Hierarchical state machines shall maintain their current state between RTC steps.

#### Description

The main job of a state machine is to "remember" its `current state` between the RTC steps.

#### Forward Traceability (truncated to 2 level(s))

---

### 5.7.2.3 SRS\_QP\_SM\_10

QP Framework shall support multiple and interchangeable State Machine Implementation Strategies

#### Description

QP Application can choose the State Machine Implementation Strategy (out of a set of supported strategies) through the `type` of a state machine object. Based on that type, QP Framework shall then resolve the matching "State Machine Processor" (matching `dispatch` method) at run-time (e.g., by virtual call). Moreover, QP Framework shall allow Applications to add their State Machine Implementation Strategies, and QP Framework shall still resolve the matching (application-defined) `dispatch` method based on the type of the state machine object.

#### Background

Application-defined State Machine Implementation Strategies might be useful for special purposes, such as components with stringent performance requirements (but perhaps fewer state machine features) or test doubles (in TDD).

#### Forward Traceability (truncated to 2 level(s))

- `SDS_QP_QAsm`: *QAsm Abstract state machine class.*
    - `QAsm`: *Abstract State Machine class (state machine interface)*
    - `QAsmVtable`: *Virtual table for the `QAsm` class*
  - `QHsm`: *Hierarchical State Machine class (QHsm-style state machine implementation strategy)*
    - `QHsm::QHsm_ctor()`: *Constructor of the `QHsm` base class*
    - `QHsm::QHsm_init_()`: *Implementation of the top-most initial transition in `QHsm`*
    - `QHsm::QHsm_dispatch_()`: *Implementation of dispatching events to a `QHsm`*
    - `QHsm::QHsm_isIn_()`: *Check if a given state is part of the current active state configuration*
    - `QHsm::QHsm_state()`: *Obtain the current active state from a HSM (read only)*
    - `QHsm::QHsm_childState()`: *Obtain the current active child state of a given parent in `QHsm`*
    - `QActive`: *Active object class (based on the `QHsm` implementation strategy)*
  - `QASM_DISPATCH`: *Virtual call to dispatch an event to a HSM*
-

#### 5.7.2.4 SRS\_QP\_SM\_20

QP Framework shall provide a State Machine Implementation Strategy optimized for "manual coding"

##### Description

"Optimized for manual coding" means that changing a single element in the state machine design (e.g., nesting of the state hierarchy) should require changing only a single matching element in the implementation.

##### Background

The State Machine Implementation Strategy "optimized for manual coding" imposes restrictions on the implementation strategy but does *not* mean that the code must be written manually. In the presence of a modeling tool, such code can also be generated automatically.

##### Forward Traceability (truncated to 2 level(s))

- [SDS\\_QP\\_QHsm](#): *QHsm State machine class*.
    - [QHsm](#): *Hierarchical State Machine class (QHsm-style state machine implementation strategy)*
- 

#### 5.7.2.5 SRS\_QP\_SM\_21

QP Framework should provide a State Machine Implementation Strategy optimized for "automatic code generation"

##### Description

"Optimized for automatic code generation" means the implementation may contain some redundant information to improve the *efficiency* of the state machine execution. Also, such a strategy can support *more advanced* state machine features (see [SRS\\_QP\\_SM\\_21](#)) than a strategy constrained by the limitations of "manual coding" (see [SRS\\_QP\\_SM\\_20](#)).

##### Background

Automatically generated code not intended for manual maintenance allows relaxing the restrictions imposed by "manual coding". In that case, a State Machine Implementation Strategy "optimized for automatic code generation" offers the application developers a choice of higher-performance and/or more features than the strategy "optimized for manual coding." For example, an implementation may contain "transition tables" with information about the chains of state exit and entry actions to execute for a given transition (instead of determining the state exit and entry at run-time). This optimization might require adjusting multiple "transition tables" when changing the hierarchical nesting of a single state, which is considered unsuitable for manual coding (see [SRS\\_QP\\_SM\\_20](#)). However, optimization of that kind is trivial for an automatic code generator.

##### Forward Traceability (truncated to 2 level(s))

- [SDS\\_QP\\_QMsm](#): *QMsm State machine class*.
  - [QMsm](#): *Hierarchical State Machine class (QMsm-style state machine implementation strategy)*
- [QMState](#): *State object for the QMsm class (QM State Machine)*
- [QMsm](#): *Hierarchical State Machine class (QMsm-style state machine implementation strategy)*
  - [QMsm::QMsm\\_init\\_\(\)](#): *Implementation of the top-most initial transition in QMsm*
  - [QMsm::QMsm\\_dispatch\\_\(\)](#): *Implementation of dispatching events to a QMsm*
  - [QMsm::QMsm\\_isIn\\_\(\)](#): *Tests if a given state is part of the current active state configuration*
  - [QMsm::QMsm\\_getStateHandler\\_\(\)](#): *Implementation of getting the state handler in a QMsm subclass*
  - [QMsm::QMsm\\_stateObj\(\)](#): *Obtain the current state from a MSM (read only)*

- `QMsm::QMsm_childStateObj()`: Obtain the current active child state of a given parent in `QMsm`
  - `QMsm::QMsm_ctor()`: Constructor of `QMsm`
  - `QMsm::QMsm_execTatbl_()`: Execute transition-action table
  - `QMsm::QMsm_exitToTranSource_()`: Exit the current state up to the explicit transition source
  - `QMsm::QMsm_enterHistory_()`: Enter history of a composite state
  - `QMActive`: Active object class (based on `QMsm` implementation strategy)
  - `QMActive`: Active object class (based on `QMsm` implementation strategy)
    - `QMActive::QMActive_ctor()`: Constructor of `QMActive` class
- 

### 5.7.2.6 SRS\_QP\_SM\_22

All State Machine Implementation Strategies provided by QP Framework shall be bidirectionally traceable

#### Description

Bi-directional traceability of a State Machine Implementation Strategy means that the rules of the "State Machine Specification" are such that:

1. Each state machine element in the *design* (in the diagram) is represented by exactly one element in the implementation; and
2. Each state machine element in the *implementation* corresponds to exactly one element in the design (diagram).

#### Background

Traceability between design and implementation is a required property for many functional safety standards. Additionally, traceability of state machine implementation is an extremely valuable property is a cornerstone for effective debugging and tracing of state machine execution. For example, traceable implementation allows a developer to set a breakpoint on a specific state transition, state-entry action, a specific guard condition etc. Without a one-to-one traceability between state machine design and code, such elements (e.g., transitions) might be repeated, which would hinder debugging.

#### Forward Traceability (truncated to 2 level(s))

- `SDS_QP_QHsm`: *QHsm* State machine class.
    - `QHsm`: Hierarchical State Machine class (*QHsm*-style state machine implementation strategy)
  - `SDS_QP_QMsm`: *QMsm* State machine class.
    - `QMsm`: Hierarchical State Machine class (*QMsm*-style state machine implementation strategy)
- 

### 5.7.2.7 SRS\_QP\_SM\_23

QP Framework shall ensure that the current event does not change and is accessible to the state machine implementation over the entire RTC step.

#### Description

The unchangeability of the current event means that both its Signal and Parameters remain unchanged throughout the RTC step within the state machine. Also the access to the current event should be computationally inexpensive (e.g., via a pointer or a reference to the current event).

## Background

The most important aspect of this requirement is preventing any changes to the current event throughout all RTC steps that the event might be involved in (in case the same event is dispatched to multiple state machines). Also, QP Framework shall make provisions for protecting the current event (e.g., by making it `const` in the implementation).

## Forward Traceability (truncated to 2 level(s))

- [SDS\\_QP\\_QHsm](#): *QHsm State machine class.*
    - *QHsm: Hierarchical State Machine class (QHsm-style state machine implementation strategy)*
  - [SDS\\_QP\\_QMsm](#): *QMsm State machine class.*
    - *QMsm: Hierarchical State Machine class (QMsm-style state machine implementation strategy)*
- 

### 5.7.2.8 SRS\_QP\_SM\_24

All State Machine Implementation Strategies provided by QP shall allow Applications to easily access the instance variables associated with a given state machine object

#### Description

QP framework shall allow for easy and computationally inexpensive access to the internal attributes of the object associated with the state machine from *within* that object. A good example of implementing such a policy is the concept of class encapsulation in OOP, where the internal attributes are accessible to the class operations (e.g., via the `this` pointer) and are harder to access from the outside.

## Background

At the same time, QP Framework shall provide [encapsulation](#) of the state machine objects. While QP Framework alone cannot rigorously enforce such encapsulation, the framework should allow the QP Application to hide such access from the *outside* of the state machine object.

## Forward Traceability (truncated to 2 level(s))

- [SRS\\_QP\\_AO\\_51](#): *Active Object abstraction shall allow Applications to easily access the internal attributes from inside the Active Object*
  - [SDS\\_QP\\_QHsm](#): *QHsm State machine class.*
    - *QHsm: Hierarchical State Machine class (QHsm-style state machine implementation strategy)*
  - [SDS\\_QP\\_QMsm](#): *QMsm State machine class.*
    - *QMsm: Hierarchical State Machine class (QMsm-style state machine implementation strategy)*
- 

### 5.7.2.9 SRS\_QP\_SM\_25

All State Machine Implementation Strategies provided by QP Framework might supply a method for checking if a state machine is in a given state

#### Description

The "is-in" state operation returns 'true' if the [current state](#) of the state machine is equal or is a substate of the given state. Otherwise, the "is-in" operation returns 'false'. Please note that in a hierarchical state machine, to "be in a state" means also to be in a superstate of the given state.

## Background

This operation is intended to be used only for state machines that run in the same thread of execution. For example, a given Active Object could use the "is-in" check on one of the "Orthogonal Components" owned by that Active Object.

## Forward Traceability (truncated to 2 level(s))

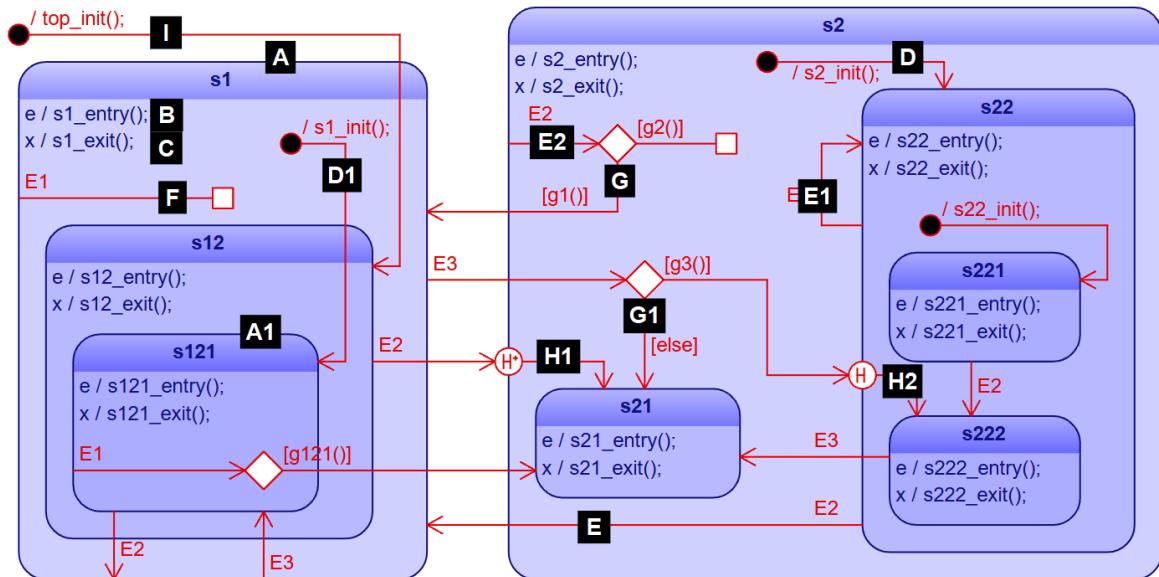
- [SDS\\_QP\\_QAsm](#): *QAsm Abstract state machine class.*
    - [QAsm](#): *Abstract State Machine class (state machine interface)*
  - [QHsm::QHsm\\_isIn\(\)](#): *Check if a given state is part of the current active state configuration*
  - [QMsm::QMsm\\_isIn\(\)](#): *Tests if a given state is part of the current active state configuration*
  - [QASM\\_IS\\_IN](#): *Virtual call to check whether a SM is in a given state*

### **5.7.2.10 SRS\_QP\_SM\_30**

All State Machine Implementation Strategies provided by QP Framework shall support hierarchical state machines with features specified in the sub-requirements SRS\_QP\_SM\_3x

## Description

The state diagram shown in Figure SRS-SM-HSM below demonstrates a Hierarchical State Machine with all features that need to be supported by all State Machine Implementation Strategies provided in QP Framework:



*Figure SRS-SM-HSM: Hierarchical State Machine diagram with labeled features corresponding to the sub-requirements*

## Background

The hierarchical state machine shown in Figure SRS-SM-HSM, demonstrates only a **subset of features** found in UML Statecharts [UML-05]. Most notably, the UML Statecharts features **not** supported in the QP Framework include "orthogonal regions" and several kinds of "pseudostates".

#### Forward Traceability (truncated to 2 level(s))

- **SDS\_QP\_QHsm:** *QHsm State machine class.*
    - *QHsm: Hierarchical State Machine class (QHsm-style state machine implementation strategy)*
  - **SDS\_QP\_QMsm:** *QMsm State machine class.*
    - *QMsm: Hierarchical State Machine class (QMsm-style state machine implementation strategy)*
- 

#### 5.7.2.11 SRS\_QP\_SM\_31

All State Machine Implementation Strategies provided by QP Framework shall support states capable of holding hierarchically nested substates

##### Description

An example state is shown in [Figure SRS-SM-HSM \[A\]](#). This is a *composite* state because it holds other states (called substates). A state that holds no other states is shown in [Figure SRS-SM-HSM \[A1\]](#). Such a state is called a *leaf* state. The State Machines Implementation Strategies in QP need to represent both types of states. Moreover, it should be possible to simply add substates to a given state thus making it a composite state as well as remove substates, thus making it a leaf state. Also, it should be possible to simply change the nesting of a given state from one superstate to another (including moving it to the implicit "top" superstate).

#### Forward Traceability (truncated to 2 level(s))

- **SDS\_QP\_QHsm:** *QHsm State machine class.*
    - *QHsm: Hierarchical State Machine class (QHsm-style state machine implementation strategy)*
  - **SDS\_QP\_QMsm:** *QMsm State machine class.*
    - *QMsm: Hierarchical State Machine class (QMsm-style state machine implementation strategy)*
  - **Q\_SUPER:** *Designates the superstate of a given state. Applicable only to QHsm subclasses*
- 

#### 5.7.2.12 SRS\_QP\_SM\_32

All State Machine Implementation Strategies provided by QP Framework shall support entry actions to states

##### Description

Example entry actions to a state are shown in [Figure SRS-SM-HSM \[B\]](#). Entry actions to a state are optional, meaning that a given state might specify entry actions or not. If any entry actions are defined in a given state, the State Machine Processor in QP must execute these actions whenever that state is entered. Also, entry actions to superstates must be always executed *before* entry actions to substates.

##### Background

Entry actions to a state provide an important mechanism to initialize that state context and QP must guarantee such initialization on any transition path leading to a given state.

#### Forward Traceability (truncated to 2 level(s))

- **SDS\_QP\_QHsm:** *QHsm State machine class.*
    - *QHsm: Hierarchical State Machine class (QHsm-style state machine implementation strategy)*
  - **SDS\_QP\_QMsm:** *QMsm State machine class.*
    - *QMsm: Hierarchical State Machine class (QMsm-style state machine implementation strategy)*
  - **Q\_HANDLED:** *Indicate that an action has been "handled". Applies to entry/exit actions and to internal transitions*
-

### 5.7.2.13 SRS\_QP\_SM\_33

All State Machine Implementation Strategies provided by QP Framework shall support exit actions from states

#### Description

Example exit actions from a state are shown in [Figure SRS-SM-HSM \[C\]](#). Exit actions from a state are optional, meaning that a given state might specify exit actions or not. If any exit actions are defined in a given state, the State Machine Processor in QP must execute these actions whenever that state is exited. Also, exit actions to superstates must be always executed *after* exit actions from substates.

#### Background

Exit actions from a state provide an important mechanism to cleanup that state context, and QP must guarantee such cleanup on any transition path leading out of a given state.

#### Forward Traceability (truncated to 2 level(s))

- [SDS\\_QP\\_QHsm](#): *QHsm State machine class*.
  - [QHsm](#): *Hierarchical State Machine class (QHsm-style state machine implementation strategy)*
- [SDS\\_QP\\_QMsm](#): *QMsm State machine class*.
  - [QMsm](#): *Hierarchical State Machine class (QMsm-style state machine implementation strategy)*
- [Q\\_HANDLED](#): *Indicate that an action has been "handled". Applies to entry/exit actions and to internal transitions*

### 5.7.2.14 SRS\_QP\_SM\_34

All State Machine Implementation Strategies provided by QP Framework shall support nested initial transitions in composite states

#### Description

An example nested initial transition is shown in [Figure SRS-SM-HSM \[D\]](#). A composite can have at most one initial transition nested directly in that state. The nested initial transition can have actions and can target any direct substate or indirect substate of the parent state (at a deeper level of state nesting). If a given state has an initial transition and other transition (regular or initial) targets that state, QP must execute the initial transition.

#### Initial Transition Execution Sequence

The execution sequence of nested initial transition is as follows:

1. actions associated with the initial transition;
2. entry actions to the target state configuration, starting with the states at the highest levels of nesting;
3. if the main target state of the initial transition contains its nested initial transition, it should be executed according to the same rules, applied recursively.

#### Examples

The execution sequence for the initial transition nested directly in state "s2" in [Figure SRS-SM-HSM](#) is as follows:  
`s2_init(); s22_entry(); s22_init(); s211_entry(); @uid_litem{}` On the other hand, the execution sequence for the initial transition nested directly in state "s1" in [Figure SRS-SM-HSM](#) is as follows:  
`s1_init(); s12_entry(); s121_entry();`

**Note**

It is also legal in QP to have a composite state with substates, but *without* a nested initial transition. If such a composite state is the main target of a state transition, the state becomes the *current state*, without any of its substates becoming active.

**Forward Traceability (truncated to 2 level(s))**

- **SDS\_QP\_QHsm:** *QHsm State machine class.*
    - **QHsm:** *Hierarchical State Machine class (QHsm-style state machine implementation strategy)*
  - **SDS\_QP\_QMsm:** *QMsm State machine class.*
    - **QMsm:** *Hierarchical State Machine class (QMsm-style state machine implementation strategy)*
- 

**5.7.2.15 SRS\_QP\_SM\_35**

All State Machine Implementation Strategies provided by QP Framework shall support transitions between states at any level of nesting

**Description**

An example of a transition is shown in [Figure SRS-SM-HSM \[E\]](#). A transition in QP must have an **explicit trigger**, which is the **Signal** of the event that triggered the transition.

**Main-Source State**

The state where the transition originates is called the *main-source* and in QP this main-source state "owns" the transition. Please note that the main-source state might be different from the current state when the transition is "inherited" from a higher-level state.

**Main-Target State**

The state where the transition terminates is called the *main-target*. Please note that the main-target state might be different from the new current state after the transition when the main-target state is composite and contains a nested initial transition.

**Self-Transition**

In a special case of the main source being the same as the main target (see [Figure SRS-SM-HSM \[E1\]](#)), the transition is called *self-transition*.

**Transition Execution Sequence**

The execution sequence of a state transition is as follows:

1. Actions associated with the transition, which might include the whole [guard evaluation sequence](#);
2. Exit actions from the source state configuration, starting with the states at the lowest levels of nesting, up to the LCA (main-source, main-target) state, whereas  $LCA(s_1, s_2)$  denotes the state that is the Least Common Ancestor of states  $s_1$  and  $s_2$ , based on the state containment hierarchy.
3. Entry actions to the target state configuration, starting with the states at the highest levels of nesting. If the main-target state contains a nested initial transition, it should be executed according to the rules described in [SRS\\_QP\\_SM\\_34](#).

In most state transitions, the main-source state is exited, and the main target is entered. The only exceptional cases are explained below:

### Local State Transition Semantics

*Special Case 1:* If the main-source state of the transition contains the main-target state (e.g., transition E3 in state "s1" in [Figure SRS-SM-HSM](#)), the main-source state is *not* exited.

*Special Case 2:* If the main-target state contains the main\_source (e.g., transition E2 in state "s121" in [Figure SRS-SM-HSM](#)), the main-target is *not* entered.

### Remarks

In the UML Specification [\[UML-2.5\]](#), Special Cases 1 and 2 correspond to the **local state transition** semantics.

### Note

The transition execution sequence in QP [\[PSICC2:08\]](#) is *different* than in the UML Specification [\[UML-2.5\]](#), because the **guard evaluation sequence** is executed in QP *before* the exit from the source state configuration and entry to the target state configuration. It is necessary first to determine the main-target state of the transition based on the evaluation of guards. The guards' evaluation might also determine that the event is to be propagated to the higher-level states, or that only an internal transition should be executed, in which cases no states should be exited or entered at all. In the UML Specification [\[UML-2.5\]](#), transition actions are executed *after* exiting the source state configuration but *before* entering the target state configuration, which immensely complicates the semantics and implementation of guards.

### Examples

Assuming that "s222" is the current state, the execution sequence for the transition s22:E2 (see [Figure SRS-SM-HSM\[E\]](#)) is as follows:

```
s22_E2(); s22_exit(); s22_exit(); s2_exit(); s1_entry(); s1_init();
s12_entry(); s121_entry(); @uid_litem{} Assuming that "s121" is the current state, the execution sequence for the transition s1:E3 (Special Case 1) is as follows:
```

```
s1_E3(); s121_exit(); s12_exit(); s12_entry(); s121_entry();
@uid_litem{} Assuming that "s121" is the current state, the execution sequence for the transition s121:E1 (Special Case 2) is as follows:
```

```
s121_E2(); s121_exit(); s12_exit(); s1_init(); s12_entry(); s121_entry();
@uid_litem{} Assuming that "s222" is the current state, the execution sequence for the self-transition s22:E1 in (see Figure SRS-SM-HSM\[E1\]) is as follows:
```

```
s22_E1; s222_exit(); s22_exit(); s22_entry(); s22_init(); s221_entry();
```

### Note

In a self-transition, the main-source **is** exited, and the main-target (same as main-source) *is* entered. Thus self-transition becomes an idiom for resetting a given state context by cleanly exiting and re-entering a given state.

### Forward Traceability (truncated to 2 level(s))

- [SDS\\_QP\\_QHsm](#): *QHsm State machine class.*
  - [QHsm](#): *Hierarchical State Machine class (QHsm-style state machine implementation strategy)*
- [SDS\\_QP\\_QMsm](#): *QMsm State machine class.*
  - [QMsm](#): *Hierarchical State Machine class (QMsm-style state machine implementation strategy)*

### 5.7.2.16 SRS\_QP\_SM\_36

All State Machine Implementation Strategies provided by QP Framework shall support internal transitions in states

#### Description

An example of an internal transition is shown in [Figure SRS-SM-HSM \[F\]](#). This type of transition causes only the execution of the associated actions. Still, it never leads to a change of the current state, and consequently, it never causes execution of any state exit or state entry actions. An alternative name for internal transition is a *state reaction*.

#### Background

Internal transitions (state reactions) are very common in practice. Internal transitions are also different from self-transitions because an internal transition never causes execution of any state exit or state entry actions.

#### Forward Traceability (truncated to 2 level(s))

- [SDS\\_QP\\_QHsm](#): *QHsm State machine class*.
  - *QHsm*: *Hierarchical State Machine class (QHsm-style state machine implementation strategy)*
- [SDS\\_QP\\_QMsm](#): *QMsm State machine class*.
  - *QMsm*: *Hierarchical State Machine class (QMsm-style state machine implementation strategy)*
- [Q\\_HANDLED](#): *Indicate that an action has been "handled". Applies to entry/exit actions and to internal transitions*

### 5.7.2.17 SRS\_QP\_SM\_37

All State Machine Implementation Strategies provided by QP Framework shall support guard conditions to be attached to regular and internal transitions

#### Description

An example of a transition with an attached guard condition is shown in [Figure SRS-SM-HSM \[G\]](#). A *guard condition* (or simply *\*guard\_\**) is a Boolean expression that disables a given transition path when it evaluates to FALSE. In QP, guard conditions are always "attached" to a transition via a choice pseudostate (UML Specification [[UML-2.5](#)]). A given choice pseudostate may have multiple attached guards, each starting a separate transition path and associated with its own (optional) action.

#### Disabled Transitions

If all guards attached to a transition (via a choice pseudostate) evaluate to FALSE, the whole transition is *disabled*. Such a transition shall be treated as though it was not present, so the triggering event is propagated to the higher-level states in the state hierarchy.

#### Guard Evaluation

Conceptually, you can think of a choice pseudostate and the attached guard conditions as an *if-then-else* sequence. Each guard is evaluated *dynamically* when the control reaches the guard in that sequence. For example, the following pseudocode shows the sequence for the transition s2:E2 ([Figure SRS-SM-HSM \[E2\]](#)):

```
s2_E2(); // action associated with the original transition
if (g1()) { // evaluate guard g1()
    s2_E2_g1(); // action associated with the path following [g1()]
    transition_to(s1); // regular state transition
}
else if (g2()) { // evaluate guard g2()
    s2_E2_g2(); // action associated with the path following [g2()]
    internal_transition(); // internal state transition
}
else { // disabled transition
    propagate_to_superstate(top); // event not handled at this level
}
```

**Note**

The **guard evaluation sequence** determines the **main-target** of the transition. For example, in the guard evaluation sequence shown above, if the guard `g1()` evaluates to TRUE, the main target is set to state "s1". Otherwise, if `g2()` evaluates to TRUE, the main target will not be set, and the **State Machine Specification** will return status to the **QP State Machine Processor** to indicate only an **internal transition**. Otherwise, if both `g1()` and `g2()` evaluate to FALSE, the whole transition is considered *disabled*, and the **State Machine Specification** will return status to the **QP State Machine Processor** to propagate the event to the higher-level state.

**The Complementary [else] Guard**

QP shall also support the special, complementary guard `[else]` (see [Figure SRS-SM-HSM \[G1\]](#)), which will explicitly complement all other guards attached to the same choice pseudostate. For example, the following pseudocode shows the sequence for the transition `s1:E3` ([Figure SRS-SM-HSM](#)):

```
s1_E3();           // action associated with the original transition
if (g3()) {        // evaluate guard g3()
    s1_E3_g3();    // action associated with the path following [g3()]
    transition_to_deep_history_of(s22); // transition to history (deep)
}
else {            // explicit complementary [else] guard
    s1_E3_else(); // action associated with the path following [else]
    transition_to(s21); // regular state transition
}
```

**Forward Traceability (truncated to 2 level(s))**

- [SDS\\_QP\\_QHsm](#): *QHsm State machine class*.
    - [QHsm](#): *Hierarchical State Machine class (QHsm-style state machine implementation strategy)*
  - [SDS\\_QP\\_QMsm](#): *QMsm State machine class*.
    - [QMsm](#): *Hierarchical State Machine class (QMsm-style state machine implementation strategy)*
- 

**5.7.2.18 SRS\_QP\_SM\_38**

All State Machine Implementation Strategies provided by QP Framework shall support top-most initial transition that shall be explicitly triggered independently from instantiation of the state machine object

**Description**

An example top-most initial transition is shown in [Figure SRS-SM-HSM \[I\]](#). The top-most initial transition has the same semantics as nested initial transitions (see [SRS\\_QP\\_SM\\_34](#)) except the top-most initial transition nests in the implicit "top" superstate and it is mandatory rather than optional.

The execution of the top-most initial transition is intentionally separated from the instantiation of the state machine object, to allow applications to fully control the initialization performed in the actions to the top-most initial transition.

**Background**

The instantiation of state machine objects might occur in an undefined order, even before the entry point into the application (before the `main()` function in C++). This is typically before the target hardware or the underlying real-time kernel has been properly initialized.

**Forward Traceability (truncated to 2 level(s))**

- [SDS\\_QP\\_QHsm](#): *QHsm State machine class*.
    - [QHsm](#): *Hierarchical State Machine class (QHsm-style state machine implementation strategy)*
  - [SDS\\_QP\\_QMsm](#): *QMsm State machine class*.
    - [QMsm](#): *Hierarchical State Machine class (QMsm-style state machine implementation strategy)*
  - [QASM\\_INIT](#): *QVirtual call to the top-most initial transition in a HSM*
-

### 5.7.2.19 SRS\_QP\_SM\_39

All State Machine Implementation Strategies provided by QP Framework should support transitions to history. Both shallow and deep histories shall be supported

#### Description

An example of a transition to *deep history* is shown in [Figure SRS-SM-HSM \[H1\]](#). An example of a transition path to *shallow history* is shown in [Figure SRS-SM-HSM \[H2\]](#). Transitions to history (deep or shallow) apply only to composite states and represent the most recently active substate. In the case of deep history, the actual current substate is remembered upon the exit from the given composite state. In the case of shallow history, only the direct substate containing the current substate is remembered. Transition to state history means transitioning to that remembered substate. Upon initialization, when a given composite state has never been active before, the transition to history is initialized with the *default history*, which is the substate pointed to by the transition coming out of the history circle (e.g., [Figure SRS-SM-HSM \[H1\]](#)).

#### Background

To support transitions to history, QP Framework needs to supply a mechanism to access the current state (deep history) and the direct substate of the current state. This information needs to be stored upon the exit of a given composite state. Also, the QP Framework needs to transition dynamically to the stored history substate.

#### Forward Traceability (truncated to 2 level(s))

- [SDS\\_QP\\_QHsm](#): *QHsm State machine class*.
    - [QHsm](#): *Hierarchical State Machine class (QHsm-style state machine implementation strategy)*
  - [SDS\\_QP\\_QMsm](#): *QMsm State machine class*.
    - [QMsm](#): *Hierarchical State Machine class (QMsm-style state machine implementation strategy)*
  - [Q\\_TRAN\\_HIST](#): *Take transition to the specified history of a given state. Applicable only to HSMs*
- 

### 5.7.2.20 SRS\_QP\_SM\_40

State Machine Implementation Strategies provided by QP Framework might supply the top-state

#### Description

The *top-state* is the ultimate root of state hierarchy and typically it is not rendered in the state diagrams. However, the concept can be useful in State Machine Specification as the superstate of states not nested in any other state. In case a given State Machine Implementation Strategy uses the concept of the top-state, QP Framework may provide a top-state element with the default behavior of silently ignoring all events.

#### Forward Traceability (truncated to 2 level(s))

- [SDS\\_QP\\_QHsm](#): *QHsm State machine class*.
    - [QHsm](#): *Hierarchical State Machine class (QHsm-style state machine implementation strategy)*
  - [SDS\\_QP\\_QMsm](#): *QMsm State machine class*.
    - [QMsm](#): *Hierarchical State Machine class (QMsm-style state machine implementation strategy)*
-

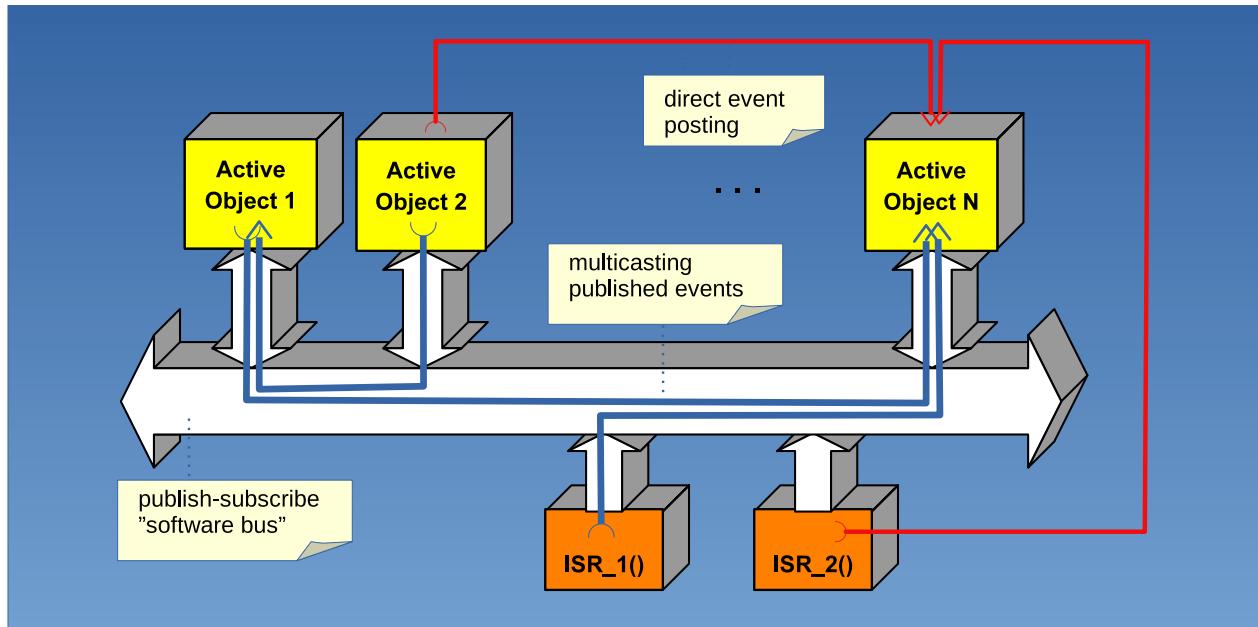
## 5.8 Event Delivery Mechanisms

### 5.8.1 Concepts & Definitions

One of the main responsibilities of the QP Framework is to reliably and safely deliver [QP events](#) from various producers to Active Objects. The event delivery is [asynchronous](#), meaning that the producers only post events to Active Objects, but don't wait in-line for the processing of the events. Any part of the system can produce events, not necessarily only the Active Objects. For example, ISRs, device drivers, or legacy code running outside the framework can produce events. On the other hand, only Active Objects can consume QP events, because only Active Objects are guaranteed to have [event queues](#).

Event delivery mechanisms can be broadly classified into the following two categories (see [Figure SRS-EDM](#)):

1. [Direct event posting](#), when the producer of an event directly posts the event to the event queue of the consumer Active Object.
2. [Publish-subscribe](#), where a producer "publishes" an event to the framework, and the framework then delivers the event to all Active Objects that had "subscribed" to the event.



*Figure SRS-EDM: Event delivery mechanisms in the QP Framework*

#### 5.8.1.1 Direct Event Posting

Direct event posting is the simplest mechanism that allows producers to post events directly to the event queue of the recipient Active Object. [Figure SRS-EDM](#) illustrates this form of communication as red arrows directly connecting event producers and the consumer Active Objects.

Direct event posting is a "push-style" communication mechanism, in which recipients receive unsolicited events whether they "want" them or not. Direct event posting is well suited for situations where a group of Active Objects, or an Active Object and an ISR, form a subsystem providing a particular service, such as a communication stack, GPS capability, digital camera subsystem in a mobile phone, or the like. This style of event passing requires that the event producers "knows" the recipients and their interests in various events. The "knowledge" that a sender needs is, at a minimum, the handle (e.g., a pointer) to the recipient Active Object.

Direct event posting mechanism might increase the coupling among the components, especially when the recipients of the events are hard-coded inside the event producers. However, one way of reducing the coupling is to allow the recipients to "register" with the producers at runtime (e.g., event producer can store a pointer to the direct consumer of the events). That way the producer does not need to hard-code the recipient(s).

### Remarks

QP Framework also provides "raw" thread-safe event queues without Active Objects behind them. Such "raw" thread-safe queues can be also used for direct event posting, but they cannot block (even when QP runs on top of a blocking RTOS kernel) and are mainly intended to directly deliver events to ISRs, that is, provide a communication mechanism from the Active Object level to the ISR level.

#### 5.8.1.2 Publish-Subscribe

Publish-subscribe event delivery is shown in [Figure SRS-EDM](#) as a "software bus" into which Active Objects "plug in" through the specified interface. Active Objects interested in certain events **subscribe** to one or more **event signals** by the QP Framework. When an event producer chooses to publish an event, QP Framework delivers the event to *all* subscribers, which entails event *multicasting* the event (sometimes incorrectly called broadcasting). Publication requests can originate asynchronously from many sources, not necessarily just from Active Objects. For example, events can be published from interrupts (ISRs), device drivers, or the "naked" threads (in case QP runs on top of a conventional RTOS/OS).

Publish-subscribe is a "pull-style" communication mechanism in which recipients receive only solicited (subscribed) events. The properties of the publish-subscribe model are:

- Producers and consumers of events don't need to know each other (loose coupling).
- The events exchanged via this mechanism must be publicly known and must have the same semantics to all parties.
- Publish-subscribe implies **event multicasting** in case a given **event signal** is subscribed by multiple Active Objects.
- A "mediator" (QP Framework) is required to accept published events and to deliver them to all interested subscribers.
- Many-to-many interactions (object-to-object) are replaced with one-to-many (object-to-mediator) interactions.

#### 5.8.1.3 Event Delivery Guarantee

In any event-driven system event delivery is critical for proper function of the entire system. If the underlying event-driven infrastructure does not or cannot guarantee event delivery, the application has to compensate by checking, verifying, and potentially repeating delivery of most events. However, in the single address space of an embedded MCU there is really no reason why event delivery (via shared memory) should fail. Under such circumstances, event delivery mechanism is similar to the basic function call mechanism. In both cases these basic mechanisms can fail if the system does not provide enough resources (event queues and event pools in case of event delivery and stack space in case of function call mechanism). Virtually all systems consider the basic function call mechanism to be guaranteed, which assumes the adequate stack resources (this involves multiple stacks if a traditional RTOS is used). Similarly, event-driven systems running in a single address space can consider the basic event delivery mechanism to be guaranteed, which assumes the adequate event-queue and event-pool sizes. Such **event delivery guarantee** is the single most powerful feature that drastically simplifies QP Applications.

### Remarks

This discussion is limited to non-distributed systems executing in a single address space. Distributed systems connected with unreliable communication media pose quite different challenges. In this case, neither synchronous communications such as remote procedure call (RPC) nor asynchronous communications via event passing can make strong guarantees.

#### 5.8.1.4 Best-Effort Event Delivery

Not all events in the system are critical. Some events are not essential and the application can afford to occasionally lose them. If events of this kind are produced frequently and would overwhelm the system, the event delivery guarantee would not be the desired approach. Instead, events of this type should be delivered by an alternative "best-effort event delivery" policy.

## 5.8.2 Requirements

### 5.8.2.1 SRS\_QP\_EDM\_00

QP Framework shall provide direct event posting to Active Object instances based on the FIFO policy

#### Description

The direct event posting mechanism shall post events to the [event queue](#) of the recipient Active Object utilizing the FIFO (First-In-First-Out) policy. Direct event posting with the default FIFO policy shall be available to all possible event producers, such as Active Objects, but also interrupts (ISRs), "device drivers", and "naked" threads of an RTOS (if an RTOS is used to run QP).

#### Backward Traceability

- [SRS\\_QP\\_AO\\_20](#): *Active Object abstraction shall provide an [event queue](#) for each Active Object instance*
- [SRS\\_QP\\_AO\\_21](#): *Active Object event queue shall provide [FIFO policy](#) for posting events from outside the Active Object*

#### Forward Traceability (truncated to 2 level(s))

- [SAS\\_QP\\_API](#): *QP Framework API*

---

### 5.8.2.2 SRS\_QP\_EDM\_01

QP Framework shall provide direct event self-posting to Active Object instances based on the LIFO policy

#### Description

Self-posting means that a given Active Object instance posts event to its own event queue. Such self-posting mechanism shall use the same [event queue](#) as the default FIFO policy, but in addition to the default FIFO policy it should provide also the LIFO (Last-In-First-Out) policy. This self-posting with the LIFO policy shall use a distinctly different API than the default direct event posting with the FIFO policy.

#### Use case

Self-posting of events with the LIFO policy can be useful for recalling events that have been deferred by the Active Object instance.

#### Backward Traceability

- [SRS\\_QP\\_AO\\_20](#): *Active Object abstraction shall provide an [event queue](#) for each Active Object instance*
- [SRS\\_QP\\_AO\\_21](#): *Active Object event queue shall provide [FIFO policy](#) for posting events from outside the Active Object*

#### Forward Traceability (truncated to 2 level(s))

- [SAS\\_QP\\_API](#): *QP Framework API*

### 5.8.2.3 SRS\_QP\_EDG\_10

QP Framework shall provide event delivery guarantee for the direct event posting mechanism.

#### Description

The event delivery guarantee shall be implemented with the SSFs based on assertion programming specified in SRS\_QP\_FDM\_00. QP Framework shall detect all conditions that could prevent a posted event from reaching the recipient Active Object (in particular event queue overflow and event-pool depletion for mutable events). In case any such condition occurs, QP Framework shall relinquish control and invoke the custom error handler defined in QP Application (see SRS\_QP\_FDM\_20). That way, QP Application does not need to check whether direct event posting was successful (because continuing execution means that it was).

Forward Traceability (truncated to 2 level(s))

---

### 5.8.2.4 SRS\_QP\_EDM\_50

QP Framework shall provide publish-subscribe event delivery mechanism

#### Description

Publish-subscribe shall use direct event posting (default, reliable variant based on FIFO policy) as the underlying low-level mechanism to multicast the subscribed events.

Forward Traceability (truncated to 2 level(s))

- **SRS\_QP\_EDM\_51:** *The publish-subscribe event delivery mechanism shall be configurable and optional*
  - **SRS\_QP\_EDM\_52:** *QP Framework shall allow Active Object instances to subscribe to a given event signal at run-time.*
    - **SAS\_QP\_API:** *QP Framework API*
  - **SRS\_QP\_EDM\_53:** *QP Framework shall allow Active Object instances to unsubscribe from a subscribed event signal at run-time.*
    - **SAS\_QP\_API:** *QP Framework API*
  - **SRS\_QP\_EDM\_55:** *Event multicasting during publishing shall complete before the processing of the published events.*
    - **FMEDA\_QP\_43: Failure Mode:** *Active Object gets preempted in the middle of multicasting a published event.*
  - **SAS\_QP\_API:** *QP Framework API*
- 

### 5.8.2.5 SRS\_QP\_EDM\_51

The publish-subscribe event delivery mechanism shall be configurable and optional

#### Description

The QP Application shall initialize and configure the publish-subscribe delivery mechanism by supplying the maximum number of event signals that can be subscribed and a memory buffer to store the subscription information. As a special case, QP Application might choose not to initialize publish-subscribe, in which case the feature shall be inactive and shall not cause any waste of resources (RAM).

**Backward Traceability**

- [SRS\\_QP\\_EDM\\_50](#): *QP Framework shall provide publish-subscribe event delivery mechanism*

**Use case**

QP Application initializes the publish-subscribe event delivery for a given maximum number of event signals and provides memory buffer capable of holding that many signals for the maximum allowed number of Active Objects in QP Application (see [SRS\\_QP\\_AO\\_01](#)).

**Forward Traceability (truncated to 2 level(s))****5.8.2.6 SRS\_QP\_EDM\_52**

QP Framework shall allow Active Object instances to subscribe to a given event signal at run-time.

**Description**

QP Framework shall provide API for Active Objects to subscribe one [event signal](#) at a time. The API shall be callable multiple times by a given Active Object to subscribe to multiple event signals. Event subscription requires initialization of the publish-subscribe feature and attempts to subscribe without prior initialization shall be treated as a programming error. Similarly, subscribing to an already subscribed signal (by the same Active Object) shall be treated as a programming error.

**Backward Traceability**

- [SRS\\_QP\\_EDM\\_50](#): *QP Framework shall provide publish-subscribe event delivery mechanism*

**Forward Traceability (truncated to 2 level(s))**

- [SAS\\_QP\\_API](#): *QP Framework API*

**5.8.2.7 SRS\_QP\_EDM\_53**

QP Framework shall allow Active Object instances to unsubscribe from a subscribed event signal at run-time.

**Description**

QP Framework shall provide API for Active Objects to unsubscribe one [event signal](#) at a time. The API shall be callable multiple times by a given Active Object to unsubscribe from multiple event signals. Event un-subscription requires initialization of the publish-subscribe feature and attempts to unsubscribe without prior initialization shall be treated as a programming error. Similarly, unsubscribing from a signal that has not been subscribed shall be treated as a programming error.

**Backward Traceability**

- [SRS\\_QP\\_EDM\\_50](#): *QP Framework shall provide publish-subscribe event delivery mechanism*

**Forward Traceability (truncated to 2 level(s))**

- [SAS\\_QP\\_API](#): *QP Framework API*

### 5.8.2.8 SRS\_QP\_EDM\_54

QP Framework shall allow Active Object instances to unsubscribe from all subscribed event signals at run-time.

#### Description

QP Framework shall provide API for Active Objects to unsubscribe from all [event signals](#) at once. Event un-subscription requires initialization of the publish-subscribe feature and attempts to unsubscribe without prior initialization shall be treated as a programming error.

#### Forward Traceability (truncated to 2 level(s))

- [SAS\\_QP\\_API](#): *QP Framework API*
- 

### 5.8.2.9 SRS\_QP\_EDM\_55

Event multicasting during publishing shall complete before the processing of the published events.

#### Description

The purpose of this requirement is to prevent event publishing from creating unexpected event sequences when QP Framework runs on top of a *preemptive* kernel. As specified in [SRS\\_QP\\_EDM\\_50](#), event multicasting (required when a given event signal is subscribed by multiple Active Objects) uses the default direct event posting mechanism. However, if the priority of the event producer is lower than the priority of the recipient Active Object, direct event posting would normally lead to preemption (before multicasting completes). Such a preemption would then cause the high-priority Active Object to immediately process the published event, which might produce other events. These other events would then appear in the event queues of Active Objects before the originally published event. This would generate a **confusing event sequence**.

#### Backward Traceability

- [SRS\\_QP\\_EDM\\_50](#): *QP Framework shall provide publish-subscribe event delivery mechanism*

#### Use Case

QP Framework can fulfill this requirement by preventing preemption during the event multicasting (only needed when QP Framework runs on top of a preemptive kernel). The preemption should be prevented only for the priorities of Active Objects involved in this particular multicasting. An ideal mechanism for that is selective scheduler locking based on priority ceiling protocol. The priority ceiling shall be set to the highest priority subscriber to a given event.

#### Forward Traceability (truncated to 2 level(s))

- [FMEDA\\_QP\\_43](#): *Failure Mode: Active Object gets preempted in the middle of multicasting a published event.*
- 

### 5.8.2.10 SRS\_QA\_EDM\_60

QP Application shall adequately size all event queues and event pools.

#### Description

According to the QP Framework memory allocation policy defined in the Software Architecture Specification, the size of event queues and event pools is determined by QP Application, not by QP Framework. Therefore, to achieve [event delivery guarantee](#) required in [SRS\\_QP\\_EDG\\_10](#), QP Application must supply adequate sizes for the event queues and event pools.

Forward Traceability (truncated to 2 level(s))

---

### 5.8.2.11 SRS\_QP\_EDM\_61

QP Framework may provide alternative best-effort event posting mechanism without event delivery guarantee

#### Description

The alternative **best-effort event posting** mechanism without event delivery guarantee shall detect that a given event cannot be posted, but QP Framework should pass this information to QP Application instead of entering a fail-safe state. The alterative direct event posting mechanism without event delivery guarantee must use a different API than the default mechanism with delivery guarantee.

#### Use Case

Event posting without event delivery guarantee is useful for events that the application can afford to occasionally lose and can apply only best-effort to handle.

Forward Traceability (truncated to 2 level(s))

---

### 5.8.2.12 SRS\_QP\_EDM\_62

The alternative unreliable event posting mechanism shall not interfere with the default reliable event posting

#### Description

An example of interference between the two event posting mechanisms is exhausting queue capacity by the unreliable event posting, so that the reliable event posting fails. Consequently, the unreliable event posting mechanism should not exhaust the event queue resource completely, but rather leave a specified number of unused queue entries (a safety margin) for the reliable event delivery.

#### Use Case

When using the unreliable direct event posting mechanism, QP Application shall specify a non-zero safety margin of the event queue to which it posts events that can be lost. The unreliable posting should fail (and convey this failure to QP Application) when the safety margin is reached. That way some space in the queue remains available to the reliable (default) event posting mechanism.

#### Backward Traceability

- SRS\_QP\_EDG\_20

Forward Traceability (truncated to 2 level(s))

---

### 5.8.2.13 SRS\_QP\_EDM\_64

Mutable event allocation shall be reliable. This requirement also means that QP Application is responsible for adequately sizing all event pools.

#### Description

"Reliable" event allocation means that QP Framework shall detect if the mutable event cannot be allocated (e.g., due to depletion of an event pool of a given block size). In case allocation fails, QP Framework shall enter a fail-safe state. This reliable event allocation policy shall be the default.

#### Use Case

The default (reliable) event allocation policy frees QP Application from checking whether event allocation was successful (because continuing execution means that it was).

#### Forward Traceability (truncated to 2 level(s))

---

### 5.8.2.14 SRS\_QP\_EDM\_65

QP Framework may provide alternative unreliable method of allocating mutable events.

#### Description

The alternative, best-effort only method of allocating mutable events must use a different API than the default reliable event allocation. QP Framework still shall detect that a mutable event cannot be allocated, but the framework should pass this information to QP Application instead of entering a fail-safe state. The unreliable event allocation policy shall be a special case and should be used with caution.

#### Use Case

Unreliable event allocation is useful for events that the application can afford to occasionally lose and can apply only "best effort" to allocate and deliver.

#### Forward Traceability (truncated to 2 level(s))

---

### 5.8.2.15 SRS\_QP\_EDM\_66

The alternative best-effort event allocation method shall not interfere with the default reliable event allocation.

#### Description

The alterative unreliable event allocation must co-exist with the default reliable event allocation. An example of interference between the two event allocation methods is exhausting queue capacity by the unreliable event posting, so that the reliable event posting fails and causes the system to enter fail-safe state.

#### Backward Traceability

- SRS\_QP\_EDG\_21

#### Forward Traceability (truncated to 2 level(s))

---

**5.8.2.16 SRS\_QP\_EDM\_80**

All event delivery mechanisms shall be free of concurrency hazards.

**Description**

"Free of concurrency hazards" means free of such hazards as race conditions and data races. It also means that QP Framework must ensure that the [current event](#) does not change (e.g., is not corrupted or prematurely recycled) throughout all RTC steps it is involved in.

Forward Traceability (truncated to 2 level(s))

---

**5.8.2.17 SRS\_QP\_EDM\_81**

All event delivery mechanism shall be deterministic.

**Description**

"Deterministic" means that the process of event posting or publishing has a known and constant upper bound of its execution time. In case of event publishing, QP Framework cannot meet this requirement alone because the time of event multicasting depends on the number of subscribers to a given event, so it is not constant. Therefore, QP Application must be designed in such a way that the multicasting time is acceptable.

Forward Traceability (truncated to 2 level(s))

---

## 5.9 Event Memory Management

### 5.9.1 Concepts & Definitions

In QP Framework, as in any event-driven system, events are frequently passed from producers to consumers. The exchanged event instances can be either [immutable](#) or [mutable](#).

#### 5.9.1.1 Immutable Events

**Immutable events** are event instances that never change at runtime. Such immutable events can be pre-allocated statically (typically and preferably in ROM) and can be [shared safely](#) among any number of concurrent entities (Active Objects, ISRs, "naked" threads, etc.) rather than being created and recycled dynamically every time. QP Framework does not need to manage memory for immutable events, but needs to clearly distinguish them from [mutable events](#), precisely to *avoid* any memory management for them.

#### Remarks

Immutability is a desirable property of event instances because exchanging such immutable events is faster and inherently safer compared to [mutable events](#) described below. QP Applications should take advantage of immutable events whenever possible.

#### 5.9.1.2 Mutable Events

Many event instances, especially [events with parameters](#) cannot be easily made immutable because their main function is specifically to deliver information produced at *runtime*. Consequently, such **mutable events** must be dynamically created, filled with information (mutated), passed around, and eventually recycled. The management of these processes at runtime is one of the most valuable services QP Framework can provide to QP Application. The main challenge of managing mutable events is to guarantee that once a mutable event gets posted or published (which might involve event multicasting), it does *not change* and does not get recycled until all Active Objects have finished their [Run-to-Completion](#) processing of that event. In fact, changing or premature recycling the [current event](#) constitutes a violation of the RTC semantics.

#### 5.9.1.3 Automatic Event Recycling

Every dynamically created mutable event must be eventually recycled, or the memory would leak. Some event-driven systems leave the event recycling to the application, but in the safety-related context, this is considered unreliable and unsafe. Therefore, the event memory management in QP Framework must also include [automatic event recycling](#).

#### 5.9.1.4 Zero-Copy Event Management

From the safety point of view, the ideal would be to *copy* entire mutable events into and out of the event queues, as it is often done with the message queues of a traditional RTOS. Unfortunately, it is prohibitively expensive in RAM and nondeterministic in CPU cycles for larger event instances (events with large parameters). However, an event-driven framework, like QP, can be far more sophisticated than a traditional RTOS because, due to [inversion of control](#), the framework manages an event's *whole life cycle*. The framework extracts an event from the Active Object's event queue and dispatches it for processing. After the [Run-to-Completion processing](#), the framework *regains control* of the event and can [automatically recycle the event](#).

An event-driven framework can also easily control the dynamic allocation of mutable events (e.g., the QP Framework provides API for this purpose). All this permits the framework to implement controlled, concurrency-safe sharing of mutable events, which, from the application standpoint, is almost indistinguishable from copying entire events. Such event management is called [zero-copy event management](#).

The whole event memory management must also carefully avoid *concurrency hazards* around the shared mutable events. Failure in any of those aspects results in defects (bugs) that are the hardest to detect, isolate, and fix.

### 5.9.1.5 Event Pools

To manage the memory for mutable events, QP Framework needs deterministic, efficient, and concurrency-safe method of dynamically allocating and recycling the event memory. The general-purpose, variable-block-size heap does not fit this bill and is inappropriate for safety-related applications, anyway. However, simpler, higher-performance, and *safer* options exist to the general-purpose heap. A well-known alternative, commonly supported by RTOSs, is a fixed-block-size heap, also known as a memory partition or *memory pool*. Memory pools are a much better choice for a real-time event framework like QP to manage mutable event memory than the general-purpose heap. Unlike the conventional (variable-block-size) heap, a memory pool is deterministic, has guaranteed capacity, and is not subject to fragmentation because all blocks are exactly the same size.

The most obvious drawback of a memory pool is that it does not support variable-sized blocks. Consequently, the blocks have to be oversized to handle the biggest possible allocation. Such a policy is often too wasteful if the actual sizes of allocated objects (mutable events, in this case) vary a lot. A good compromise is often to use not one but *multiple* memory pools with memory blocks of different sizes. QP Framework chooses that option to implement **event pools**, which are multiple memory pools specialized to hold mutable events.

## 5.9.2 Requirements

### 5.9.2.1 SRS\_QP\_EMM\_00

QP Framework shall support immutable events.

#### Description

Support for immutable events means that QP Framework shall provide a way to create and initialize them (including allocation and initialization in ROM). Also, QP Framework must recognize and distinguish immutable events from mutable events and should never attempt to manage immutable events as mutable.

#### Use Case

QP Framework can fulfill this requirement by embedding a unique immutable-event signature in the internal data of the event (see [SRS\\_QP\\_EVT\\_31](#)). QP Framework must then provide a constant initializer for immutable events.

#### Forward Traceability (truncated to 2 level(s))

- [SAS\\_QP\\_EMM](#): *Event memory management policy*
  - [SAS\\_QP\\_FUSA\\_02](#): *Error detecting codes mechanisms to protect critical variables*.
  - [SDS\\_QA\\_START](#): *QA Application startup sequence*
  - [SDS\\_QP\\_POST](#): *QP event posting sequence*
  - [SDS\\_QP\\_PUB](#): *QP event publishing sequence*
  - [SDS\\_QP\\_MELC](#): *Mutable event ownership rules*

---

### 5.9.2.2 SRS\_QP\_EMM\_10

QP Framework shall support mutable events.

#### Description

Support for mutable events means that QP Framework shall provide a way to create and initialize them. Also, QP Framework must recognize and distinguish mutable events from immutable events and should never attempt to manage mutable events as immutable.

Forward Traceability (truncated to 2 level(s))

- [SAS\\_QP\\_EMM](#): *Event memory management policy*
    - SAS\_QP\_FUSA\_02: *Error detecting codes mechanisms to protect critical variables.*
    - [SDS\\_QA\\_START](#): *QA Application startup sequence*
    - [SDS\\_QP\\_POST](#): *QP event posting sequence*
    - [SDS\\_QP\\_PUB](#): *QP event publishing sequence*
    - [SDS\\_QP\\_MELC](#): *Mutable event ownership rules*
- 

### 5.9.2.3 SRS\_QP\_EMM\_11

QP Framework shall support up to 15 event pools for mutable events.

#### Description

Support for mutable events means that QP Framework shall support multiple, deterministic [event pools](#) with different block sizes. The maximum number of event pools managed by QP Framework at any given time shall be compile-time configurable with the maximum of 15 event pools.

#### Use Case

QP Application configures QP for the maximum number of event pools to 5 but initializes only 3 event pools, each for a different event size.

Forward Traceability (truncated to 2 level(s))

---

### 5.9.2.4 SRS\_QP\_EMM\_20

QP Framework shall provide a method of allocating mutable events at runtime.

#### Description

The allocation shall choose the appropriate event pool from which to allocate the event based on the event size. The method of allocating mutable events shall be available to all event producers, such as Active Objects, but also ISRs, "device drivers", or "naked" thread of an RTOS (if a traditional RTOS is used).

#### Use Case

QP Framework might meet this requirement by sorting the event pools internally based on their increasing block size. Then any given event is allocated from the smallest block-size event pool that fits the requested event size.

---

### 5.9.2.5 SRS\_QP\_EMM\_30

QP Framework shall support automatic recycling of mutable events according to the "zero-copy" memory management policy.

#### Description

QP Framework shall keep track of every use of a mutable event, so that it can support the [zero-copy event management](#) policy. In particular, QP Framework shall automatically recycle an unused event.

**Use Case**

One way to meet this requirement is for QP Framework to use the standard reference-counting algorithm for mutable events.

**Forward Traceability (truncated to 2 level(s))**

---

**5.9.2.6 SRS\_QP\_EMM\_40**

QP Framework shall provide a method of explicitly recycling mutable events.

**Description**

The method of recycling mutable events back to the appropriate event pool shall be available to all event producers, such as Active Objects, but also ISRs, "device drivers", or "naked" thread of an RTOS (if a traditional RTOS is used).

---

## 5.10 Time Management

### 5.10.1 Concepts & Definitions

Time management available in a traditional RTOS includes delaying a calling thread by timed blocking (`delay()`) or blocking with a timeout on various kernel objects (e.g., semaphores or event flags). These blocking mechanisms are *not* useful in Active Object-based systems where **blocking is not allowed**. Instead, to be compatible with the **Active Object model of computation**, time management must be based on the *event-driven paradigm* in which every relevant for the system occurrence manifests itself as an event instance.

#### 5.10.1.1 Time Events

An event-driven framework like QP needs an event-driven time management mechanism based on **Time Events** (sometimes called timers). A Time Event is a UML term and denotes a point in time (at the specified time, the event occurs [UML 2.5]). A Time Event is said to be *armed* when it is actively timing out towards the specified time in the future. When the specified time arrives, the Time Event *expires* and QP Framework directly posts the Time Event to the event queue of the *recipient* Active Object. An expired Time Event armed originally for one-shot gets automatically **disarmed**. A Time Event armed for periodic expiration gets automatically **re-armed** to expire again in the specified time interval in the future. A disarmed Time Event is dormant and must be explicitly armed to time out in the future.

#### 5.10.1.2 System Clock Tick

Most real-time systems, including traditional RTOSes and Active Object Frameworks like QP, require a periodic time source called the **System Clock Tick** to keep track of time delays, timeouts, and armed Time Events in case of the event-driven QP Framework. The System Clock Tick must call a special service in the QP Framework to periodically update all armed Time Events.

The System Clock Tick is typically a periodic interrupt (asynchronous with respect to the baseline code execution) that occurs at a predetermined rate, typically between 10Hz and 1000Hz. This rate establishes the basic time units as "clock ticks" and the resolution of Time Events as integer number of "clock ticks". The faster the tick rate, the finer the granularity of "clock ticks" and higher resolution of Time Events, but also the more overhead the time management implies.

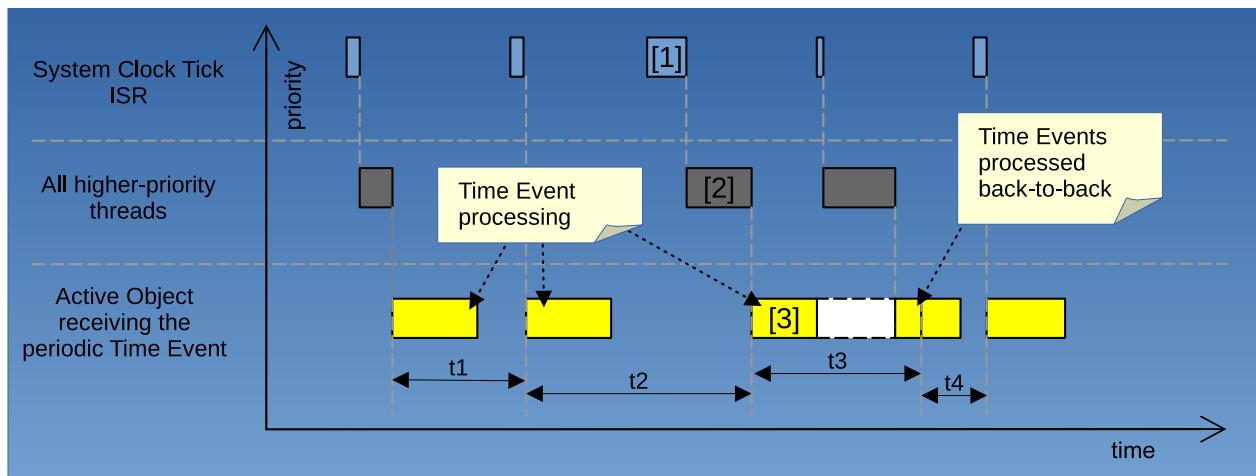


Figure SRS-TE-JIT: Jitter of a periodic Time Event expiring every tick

Even though the resolution of Time Events is one "clock tick", it does not mean that the accuracy is also one "clock tick". Figure SRS-TE-JIT shows in a somewhat exaggerated manner that the Time Event delivery and eventual handling is always subject to **jitter**. The main sources of this jitter are the following variable delays:

- [1] Delay caused by the varying length of processing inside the System Clock Tick ISR

[2] Delay caused by the varying length of processing inside the higher-priority Active Objects or other "naked" threads in the system (such processing might also be triggered by the system Clock Tick ISR)

[3] Delay caused by the varying length of processing inside the recipient Active Object.

Generally the jitter in Time Event processing gets worse as the priority of the recipient Active Object gets lower. In heavily loaded systems, the jitter might even exceed one "clock tick" period. In particular, a Time Event armed for just one tick might expire almost immediately because the System Clock Tick is *asynchronous* with respect to Active Object execution. To guarantee at least one "clock tick" timeout, you need to arm a Time Event for two clock ticks. Note too that Time Events are generally not lost due to event queuing. This is in contrast to clock ticks of a traditional RTOS, which can be lost during periods of heavy loading.

#### 5.10.1.3 Power Efficiency

Time management has a big impact on the **power efficiency** of the system. Specifically the need to periodically service Time Events (or various timeouts in a traditional RTOS kernel) has the unfortunate side effect of constantly interrupting the CPU, which means that the CPU can spend less time in the low-power sleep mode.

#### 5.10.1.4 "Tickless Mode"

While a fixed System Clock Tick is very useful, it often interrupts the CPU regardless if real work needs to be done or not. To mitigate that problem, some real-time kernels use the low-power optimization called the "Tickless Mode" (a.k.a. "tick suppression" or "dynamic tick"). In this mode, instead of indiscriminately making the System Clock Tick expire with a fixed period, the kernel adjusts the clock ticks dynamically, as needed. Specifically, after each clock tick the kernel re-calculates the time for the next clock tick and then sets the clock tick interrupt for the earliest timeout it has to wait for. For example, if the shortest timeout the kernel has to attend to is 300 milliseconds into the future, then the clock interrupt will be set for 300 milliseconds.

This approach maximizes the amount of time the CPU can remain asleep, but requires the kernel to perform the additional work to calculate the dynamic tick intervals and to program them into the hardware. This additional bookkeeping adds complexity to the kernel, might be an additional source of jitter and, most importantly, extends the time CPU spends in the high-power active mode and thus eliminates some of the power gains the "Tickless Mode" was supposed to bring.

Also, the "Tickless Mode" requires a more capable hardware timer that must be able of being reprogrammed for every interrupt in a wide dynamic range and also must accurately keep track of the elapsed time to correct for the irregular (dynamic) tick intervals. Still, even with various precautions and corrections, "Tickless Mode" often causes a **drift** in the timing of the clock tick.

#### 5.10.1.5 Multiple Tick Rates

For the reasons just described, QP Framework does not support "Tickless Mode". Instead, QP Framework supports **multiple clock tick rates** as an alternative way of achieving similar goals (to avoid interrupting the CPU at higher rate than necessary.)

Support for multiple tick rates means that each Time Event instance in QP is associated with a particular clock tick rate. For example, TimeEvt0 instance might be associated with rate #0 of only 10Hz, while TimeEvt1 instance might be associated with rate #1 of 1000Hz. The higher tick rate #1 is needed only occasionally (only in certain modes of the system), which means that TimeEvt1 is armed only during these periods and otherwise is disarmed. This provides an opportunity to shut down the high tick rate #1 of 1000Hz when it is not needed, and QP Framework provides a method for finding out when there are no armed Time Events for any given rate. The high tick rate #1 can be re-started again only when some Time Events (like TimeEvt1) get armed again. In an extreme case, a system can shut down all clock tick rates until some external event (typically interrupt) wakes the system up.

"Shutting down" a clock rate might be implemented in various ways. For example, a hardware timer that generates an interrupt at that rate can be disabled. But it is also possible to generate multiple tick rates from one hardware timer. In that case, shutting down a clock rate might mean reprogramming that timer to generate interrupts at a different period. All of this is under control of QP Application.

From the point of view of QP Framework, the support for multiple static clock tick rates is significantly simpler than the "Tickless Mode", and essentially does not increase the complexity of the framework because the same code for the single tick rate can handle other tick rates the same way. Also, multiple static tick rates require much simpler hardware

timers, which can be clocked specifically to the desired tick rate and don't need particularly wide dynamic range. For example, 16-bit timers or even 8-bit timers with or without prescalers are completely adequate. Yet the multiple clock rates can deliver similar low-power performance for the system, while keeping QP Framework much simpler and easier to certify than "Tickless" Kernels.

### 5.10.2 Requirements

#### 5.10.2.1 SRS\_QP\_TM\_00

QP Framework shall support Time Events.

##### Description

Support for Time Events means that QP Framework shall provide a Time Event abstraction that behaves like other [events in %QP](#), but is additionally equipped with the notion of time passage. The time passage consists of discrete steps (clock ticks).

##### Use Case

The basic usage model of the Time Events is as follows. An Active Object allocates and initializes one or more Time Event instances. When the Active Object needs to arrange for a timeout, it arms one of its time events to expire either just once (one-shot) or periodically. Each Time Event times out independently from the others, so a QP Application can make multiple parallel timeout requests (from the same or different Active Objects). When QP Framework service associated with the armed Time Events detects that the appropriate moment has arrived, it inserts the expiring Time Event instance directly into the recipient's event queue using the default FIFO policy with event delivery guarantee ([SRS\\_QP\\_EDM\\_00](#)). The recipient Active Object then processes the Time Event instance just like any other event.

##### Forward Traceability (truncated to 2 level(s))

- [SAS\\_QP\\_API](#): *QP Framework API*
  - [SAS\\_QP\\_EMM](#): *Event memory management policy*
    - [SAS\\_QP\\_FUSA\\_02](#): *Error detecting codes mechanisms to protect critical variables*.
    - [SDS\\_QA\\_START](#): *QA Application startup sequence*
    - [SDS\\_QP\\_POST](#): *QP event posting sequence*
    - [SDS\\_QP\\_PUB](#): *QP event publishing sequence*
    - [SDS\\_QP\\_MELC](#): *Mutable event ownership rules*
  - [SVP\\_QP\\_inst](#): *QP/C Framework installation files*.
- 

#### 5.10.2.2 SRS\_QP\_TM\_10

QP Framework shall support up to 15 clock tick rates.

##### Description

Support for multiple clock tick rates means that QP Framework shall provide that many different ways of grouping Time Events and associating them with a given clock tick rate. The maximum number of clock tick rates supported by QP Framework shall be compile-time configurable with the maximum of 15.

##### Forward Traceability (truncated to 2 level(s))

---



---

### 5.10.2.3 SRS\_QP\_TM\_11

For every clock rate QP Framework shall provide a clock-tick processing operation that QP Application must call periodically to service the armed Time Events.

#### Description

The clock-tick processing operation shall update all armed Time Events associated with a given tick rate. The clock-tick processing operation must be callable from the interrupt level and also from the thread level (e.g., from an Active Object). Also, clock-tick processing operations for different tick rates must be allowed to preempt each other (e.g., higher clock tick rates might be serviced from interrupts while others from threads).

#### Use Case

QP Framework can provide the clock-tick processing operation parameterized by the tick rate. For each tick rate, QP Application must then invoke the corresponding clock-tick processing operation at the right period either from a time-tick ISR or from the thread level.

#### Forward Traceability (truncated to 2 level(s))

- [SAS\\_QP\\_API](#): *QP Framework API*
- 

### 5.10.2.4 SRS\_QP\_TM\_20

QP Framework shall provide Time Event initialization.

#### Description

The association between a Time Event and the event signal, recipient Active Object, and the tick rate shall be made only during initialization (e.g., via a constructor) and shall not be changed later during the lifetime of the Time Event.

---

### 5.10.2.5 SRS\_QP\_TM\_21

QP Framework shall allow a Time Event to be armed both for one-shot and periodic expiry.

#### Description

A Time Event shall provide a service to arm it to expire in a specified number of clock ticks of the associated clock tick rate (one-shot expiration). Additionally, the service shall allow specifying an interval for automatic re-arming the Time Event (periodic expiration). The special value zero (interval == 0) shall denote no periodic re-arming.

Arming an already armed Time Event shall be considered as a programming error and the QP Framework shall enter a safe state.

When the Timeout Expires, it gets directly posted (using the default FIFO policy with event delivery guarantee) into the event queue of the recipient Active Object. After posting, a one-shot time event gets automatically disarmed while a periodic time event (interval != 0) is automatically re-armed.

#### Forward Traceability (truncated to 2 level(s))

- [SAS\\_QP\\_API](#): *QP Framework API*
-

### 5.10.2.6 SRS\_QP\_TM\_22

QP Framework shall allow a Time Event to be explicitly disarmed.

#### Description

A Time Event shall provide a service to disarm it. This service shall be available for Time Events armed for one-shot or periodic expiration. Disarming an already disarmed Time Event is fine and not considered an error.

#### Use Case

Due to the asynchronous nature of System Clock Tick, an Active Object sometimes cannot know whether a one-shot Time Event was automatically disarmed. Therefore the disarm operation shall return the status of the Time Event. The return of 'true' shall mean that the Time Event was still armed. The return of 'false' means that the Time Event was not truly disarmed because it has already expired. The 'false' return is only possible for one-shot Time Events that have been automatically disarmed upon expiration. In this case the 'false' return means that the Time Event has already been posted and should be expected in the recipient Active Object's event queue.

#### Forward Traceability (truncated to 2 level(s))

- [SAS\\_QP\\_API](#): QP Framework API
- 

### 5.10.2.7 SRS\_QP\_TM\_23

QP Framework shall allow a Time Event to be rearmed.

#### Description

A Time Event shall provide a service to rearm it to expire in a specified number of clock ticks (of the associated tick rate). This service shall be available for Time Events armed for one-shot or periodic expiration. Rearming a Time Event that is disarmed is fine and not considered an error.

#### Use Case

The rearm service can be used to adjust the current period of a periodic time event or to prevent a one-shot time event from expiring (e.g., a watchdog Time Event). Rearming a periodic timer leaves the interval unchanged and is a convenient method to adjust the phasing of a periodic time event.

Due to the asynchronous nature of System Clock Tick, an Active Object sometimes cannot know whether a one-shot Time Event was automatically rearmed. Therefore the rearm operation shall return the status of the Time Event. Return 'true' shall mean that the Time Event was running as it was re-armed. The 'false' return means that the Time Event was not truly rearmed because it has already expired. The 'false' return is only possible for one-shot Time Events that have been automatically disarmed upon expiration. In this case the 'false' return means that the Time Event has already been posted and should be expected in the Active Object's event queue.

#### Forward Traceability (truncated to 2 level(s))

- [SAS\\_QP\\_API](#): QP Framework API
- 

### 5.10.2.8 SRS\_QP\_TM\_30

QP Framework shall provide operation to check whether any Time Events are armed for a given tick rate.

**Description**

The operation shall return 'true' if no Time Events are armed at the given tick rate and 'false' otherwise. Also, due to the asynchronous nature of system clock tick, the operation shall be designed to run inside a critical section thus preventing preemption.

**Forward Traceability (truncated to 2 level(s))**

- [SAS\\_QP\\_API](#): *QP Framework API*
- 

**5.10.2.9 SRS\_QP\_TM\_40**

QP Framework shall support low-power sleep modes.

**Description**

QP Framework shall detect the *idle condition* of the system and provide a mechanism for QP Application to enter the desired sleep mode **safely**. "Entering a sleep mode safely" means that there should be no race conditions and situations, where the system would enter a sleep mode with some events present in the Active Object event queues and thus requiring processing.

**Forward Traceability (truncated to 2 level(s))**

## 5.11 Software Tracing

### 5.11.1 Concepts & Definitions

In any real-life project, getting the code written, compiled, and successfully linked is only the first step. The system still needs to be tested, validated, and tuned for best performance and resource consumption. A single-step debugger is frequently not helpful because it stops the system and exactly hinders seeing live interactions within the application. Clogging up high-performance code with `printf()` statements is usually too intrusive and simply unworkable in most embedded systems. So the questions are: How can you monitor the behavior of a running real-time system without degrading the system itself? How can you discover and document elusive, intermittent bugs that are caused by subtle interactions among concurrent components? How do you design and execute repeatable unit and integration tests of your system? How do you ensure that a system runs reliably for long periods of time and gets top processor performance? Techniques based on [Software Tracing](#) can answer many of these questions.

#### 5.11.1.1 What is Software Tracing?

**Software Tracing** is a method for obtaining diagnostic information in a *live* environment without the need to stop or even significantly slow down the application to get the system feedback. Software Tracing always involves some form of a target system **instrumentation** to log the relevant discrete occurrences inside the target for subsequent retrieval from the target to the host computer and analysis. An example of Software Tracing is sprinkling the code with `printf()` statements, which are a crude tracing instrumentation in this case. Of course, a professional Software Tracing system can be far less intrusive and more powerful than the primitive `printf()`.

#### Attention

In the context of safety certification, the Software Tracing system described in this part of the Software Requirement Specification plays a **critical role**. It provides the backbone of testing, verification, and validation strategies for both QP Framework and QP Applications.

The [Figure SRS-QS-SET](#) below shows a typical setup for Software Tracing, which always consists of two components:

1. **Target-resident component** for generating and sending the trace data. This component might also *receive* commands from the host to execute on the target; and
2. **Host-resident component** to receive, parse, validate, visualize, and analyze the data.

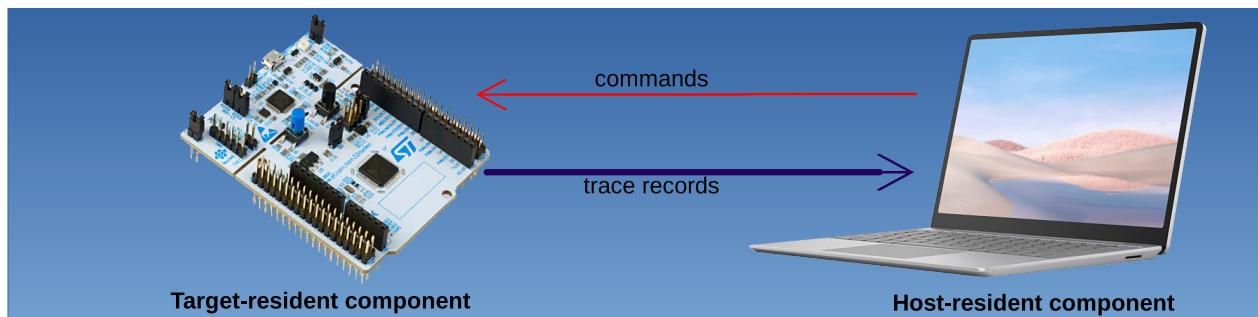


Figure SRS-QS-SET: Typical setup for Software Tracing.

#### Remarks

The discrete occurrences logged by a Software Tracing system are sometimes referred to as "events". However, in the context of an event-driven framework like QP, these occurrences will be called **trace records**, to avoid confusing them with the [application-level events](#).

### 5.11.1.2 Software Tracing & Active Objects

Software Tracing is particularly effective and powerful in combination with the event-driven **Active Object model of computation**. Due to the **inversion of control**, a running application built of Active Objects is a highly structured affair where all important system interactions funnel through the underlying event-driven framework. This arrangement offers a unique opportunity for applying Software Tracing in a framework like QP.

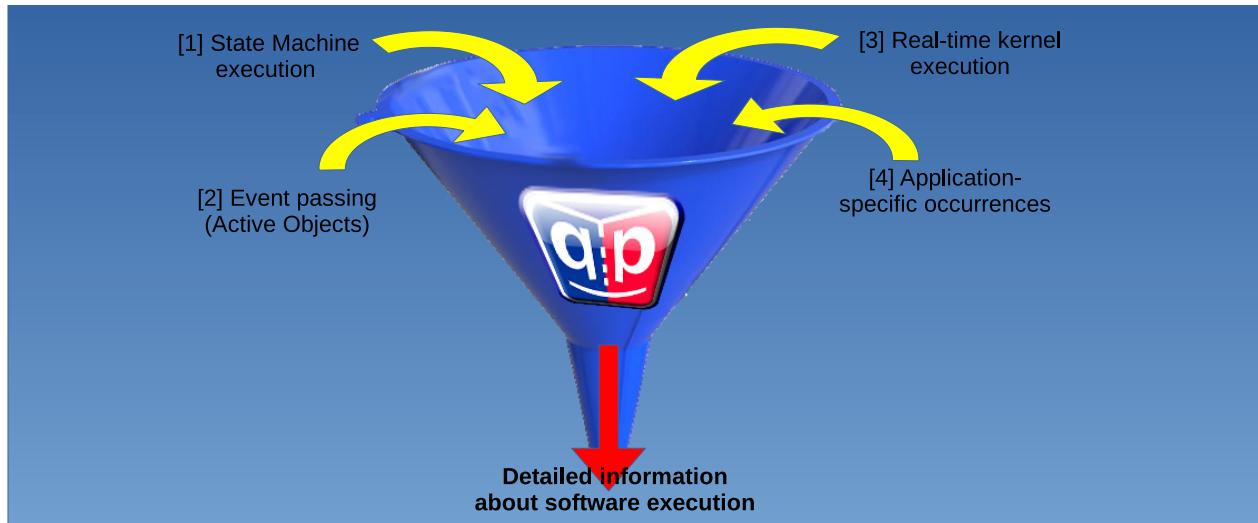


Figure SRS-QS-FUN: Software tracing in QP Framework

Tracing instrumentation inside just the QP Framework provides an unprecedented wealth of information about the running system, far more detailed and comprehensive than any traditional RTOS can provide (because RTOS is *not* based on control inversion.) The Software Trace data from the framework alone can produce:

- [1] detailed state machine activity in all Active Objects and other state machines in the system.
- [2] detailed information about event posting/publishing, queuing, recycling. This also includes complete, time-stamped sequence diagrams.
- [3] detailed information about real-time kernel activity, like: kernel objects, context switches, scheduler, etc.
- [4] custom application-specific trace records

This ability can form the foundation of the whole **testing** strategy for QP Application. In addition, individual Active Objects are natural entities for unit testing, which you can perform simply by injecting events into the Active Objects and collecting the generated execution trace. Software Tracing at the framework level makes all this comprehensive information available to the application developers, even with *no instrumentation* added to the application-level code.

### 5.11.1.3 QP/Spy Software Tracing System

QP/Spy is a Software Tracing and testing system specifically designed for and embedded in the QP Framework. As shown in [Figure SRS-QS-STR](#), QP/Spy consists of the following components:

1. Target-resident component called **QS**. This component is part of QP Framework and is the main subject of this Software Requirements Specification. The QS target-resident component consists of the QS-TX RAM buffer, the QS Filters, as well as the [instrumentation embedded in the %QP Framework](#). Additionally, the QS target-resident component contains the [receive-channel \(QS-RX\)](#) with its own RAM buffer, which can receive data from the QSPY host component.
2. Host-resident component called **QSPY**. This component is not part of QP Framework and is described in the [QTools collection↑](#).

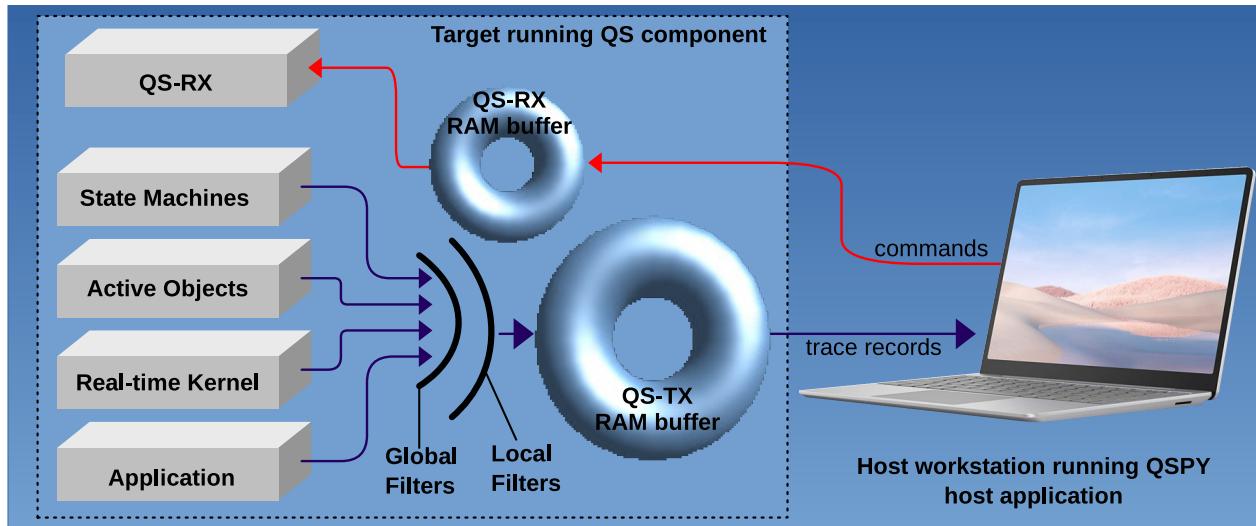


Figure SRS-QS-STR: Structure of the QP/Spy Software Tracing system.

#### Note

The QS tracing instrumentation is **inactive** by default. It becomes active only when explicitly enabled (by defining a special macro). This means that the QS instrumentation can be safely left in the code for future development, testing, and maintenance.

#### 5.11.1.4 Data Protocol

The QS target-resident component inserts trace records into the QS-TX RAM buffer using a **binary** data protocol. The QP/Spy protocol must be lightweight, but must support clearly delimited **frames**, as well as provisions to check data **continuity** and **integrity** of the frames. These features are necessary to allow flexible removal of data from the RAM buffer in any chunks typically not aligned with the frame boundaries. Finally, the data transmitted from the target with the QS data protocol must also allow the host to **instantaneously re-synchronize** after any buffering or transmission error to minimize loss of useful data.

#### 5.11.1.5 Run-time Filtering

To minimize the intrusiveness of tracing, the QS target-resident component must perform efficient, selective logging of trace records using as little processing and memory resources of the target as possible. Selective logging means that the tracing system provides user-definable, fine-granularity **filters** so that the QS target-resident component only collects trace records of interest, and you can filter out as many or as few instrumented trace records as you need.

#### 5.11.1.6 Predefined Trace Records

QP Framework contains the tracing instrumentation for **pre-defined trace records**, such as state machine activity (dispatching events, entering/exiting a state, executing transitions, etc.), Active Object activity (allocating events, posting/publishing events, time events, etc.), and more. These QS records have predefined (hard-coded) structure both in the QS target-resident component and in the QSPY host-based application.

#### 5.11.1.7 Application-Specific Trace Records

In addition to the predefined QS records, QP Application can add its own, flexible, **application-specific trace records**, whose structure is not known in advance to the QSPY host-resident component. Application-specific trace records carry the format information in them.

### 5.11.1.8 QS Dictionaries

Every Software Tracing system, just like every single-step debugger, needs the *symbolic information*, such as the names of various objects, names of functions, and symbolic names of event signals. This is because by the time source code is compiled and loaded into the target, the symbolic information is stripped. Therefore, the symbolic information must be somehow provided to the QSPY host-resident component, so that it can associate the symbolic names with binary addresses and other binary information received from the target and then display the symbolic names in the human-readable trace. Various Software Tracing systems approach this problem differently.

The QS target-resident component provides special **dictionary trace records** designed expressly for providing the symbolic information about the target code in the trace itself. These "dictionary records" provide mapping between the unique object or function addresses in the target memory and the corresponding symbolic names from the source code. The dictionary trace records are typically generated during the system initialization and this is the only time they are sent to the QSPY host component. Generating the "dictionaries" is the responsibility of the QP Application.

### 5.11.1.9 QS-RX Receive Channel

The QS target-resident component can also implement a *receive-channel* (QS-RX), which allows receiving, parsing and executing commands from the QSPY host application. Such a QS-RX channel can be the backbone for interacting with the target system and implementing such features as **unit testing** and **monitoring** of the target system.

### 5.11.1.10 Reentrancy

Finally, the QS target-resident tracing component must allow consolidating data from all parts of the system, including concurrently executing Active Objects, "naked" threads (if used), and interrupts. This means that the QS API must be **reentrant** (i.e., both thread-safe and interrupt-safe).

## 5.11.2 Requirements

### 5.11.2.1 SRS\_QP\_QS\_00

QP Framework shall support Software Tracing.

#### Description

Support for Software Tracing means that QP Framework shall implement the QS target-resident component. This also means that QP Framework shall provide the Software Tracing API for initializing the QS target-resident component, setting up the filters, encoding QS trace records, accessing the QS trace buffers, etc.

#### Forward Traceability (truncated to 2 level(s))

---

### 5.11.2.2 SRS\_QP\_QS\_01

QS target-resident component shall be inactive by default and activated only when explicitly enabled.

#### Description

The QS tracing API shall be inactive by default, meaning it should not produce any executable code. The QS instrumentation shall become activate only when explicitly enabled. This requirement does NOT mean that the QS tracing instrumentation gets removed or changed in the QP Framework or QP Application source code. Instead, this requirement means that the (inactive) QS instrumentation can be safely left in the source code (both QP Framework and QP Application) to help in future development, testing, and maintenance.

### Use Case

The QS API can be implemented as preprocessor macros (in C or C++), which are defined to nothing by default resulting in no code generation by the compiler. Only when a special macro is defined (e.g., `_QSPY`), the QS API can be defined such that it actually generates code. That way also avoids contaminating the source code with conditional compilation for every QS API, which could run the risk of inadvertently leaving some QS APIs active (should the developer forget to surround the QS API with conditional compilation).

### Forward Traceability (truncated to 2 level(s))

---

#### 5.11.2.3 SRS\_QP\_QS\_10

QS target-resident component shall use the binary data protocol.

##### Description

The QS target-resident component shall produce tracing data into the RAM buffer encoded by means of a binary protocol. The main feature required from the applied protocol is maintaining clearly delimited **frames**, each containing one trace record. The protocol "frames" shall contain the following elements:

- **sequence-number** to enable checking data continuity (range 0..255, "wrapping around")
- **record-id** to identify the type of the trace record (range 0..127)
- optional **time-stamp** data element (configurable length integer of 1, 2, or 4 bytes)
- optional **data-payload** (length 0..n bytes)
- **checksum** (range 0..255) to enable checking integrity of the frame

##### Use Case

This requirement can be met, for example, by a High Level Data Link Control (HDLC) protocol, which is characterized by establishing a very easily identifiable frames in the serial data stream. Any receiver of such a protocol can instantaneously synchronize to the frame boundary by simply finding the Flag byte (typically 0x7E).

### Forward Traceability (truncated to 2 level(s))

---

#### 5.11.2.4 SRS\_QP\_QS\_11

QS target-resident component shall allow flexible buffering schemes and decoupling trace generation from transmission to the host.

##### Description

The QS data protocol is the main enabler for a flexible *buffering* policy. Specifically, the protocol inserts only complete trace records as clearly delimited "frames" into the RAM buffer, which has two important consequences:

1. For each trace record the QS transmission protocol maintains both the continuity and the integrity checks (see [SRS\\_QP\\_QS\\_10](#)), which means that any data corruption caused by overrunning the old data with the new data can be always reliably detected on the host side. Therefore, the new trace data can be simply inserted into the trace RAM buffer, regardless if it perhaps overwrites the old data that hasn't been removed and sent out yet. The detection of any data corruption can be thus removed from the target system and deferred to the QSPY host component.

2. The insertion of delimited "frames" in the trace buffers enables **decoupling** data insertion into the trace buffers from data removal out of the trace buffers. QP Application can remove the trace data in arbitrary chunks, without any consideration for frame boundaries. QP Application can employ any physical data link available in the target for transferring the trace data the host.

Forward Traceability (truncated to 2 level(s))

---

#### 5.11.2.5 SRS\_QP\_QS\_20

QS target-resident component shall support Global-Filter.

##### Description

The QS Global-Filter is based on the **record-id** associated with each QS trace record (see [SRS\\_QP\\_QS\\_10](#)). The Global-Filter shall allow QP Application to disable or enabling each individual record-id or an arbitrary subset of record-ids. For example, QP Application might enable or disable only state-machine-entry records, or all state-machine group of records. This filter works globally for all trace records in the entire system.

Forward Traceability (truncated to 2 level(s))

---

#### 5.11.2.6 SRS\_QP\_QS\_21

QS target-resident component shall support Local-Filter.

##### Description

The QS Local-Filter is based on the individual **object-id** associated with various objects in the target memory. The object-ids are small integer numbers in the range 0..127. The object-ids in the range 0..64 are reserved for the Active Objects, where object-id corresponds to the [unique priority](#) of the Active Object. QP Application can associate other remaining object-ids (65..127) with other objects. Then, QP Application can set up the QS Local-Filter to enable only a specific object-id or any subset of object-ids.

##### Use Case

The main use case for the Local-Filter is an application where certain objects (e.g., Active Objects) are very "noisy", and would overwhelm the trace. The Local-Filter allows QP Application to silence the "noisy" objects and let the others through. Please note that the Global-Filter cannot achieve such a selection and must be complemented by the Local-Filter.

Forward Traceability (truncated to 2 level(s))

---

### 5.11.2.7 SRS\_QP\_QS\_30

QS target-resident component shall support predefined trace records.

#### Description

Predefined trace records are trace records with a fixed format known upfront by both the QS target-resident component and the QSPY host-resident component. Every predefined trace record is uniquely identified by its **record-id** (see [SRS\\_QP\\_QS\\_10](#)), and the record-id range 0..100 is reserved for the predefined records. This one-to-one mapping between record-ids and predefined records allows the QSPY host-resident component to easily recognize and correctly parse all the "pre-defined" records. Most predefined trace records have the timestamp data element (see [SRS\\_QP\\_QS\\_10](#)), but some (e.g., dictionary trace records) do not.

#### Use Case

Predefined trace records are used for tracing instrumentation embedded in the QP Framework, such as state machine activity (dispatching events, entering/exiting a state, executing transitions, etc.), Active Object activity (allocating events, posting/publishing events, time events, etc.), and more.

#### Forward Traceability (truncated to 2 level(s))

---

### 5.11.2.8 SRS\_QP\_QS\_31

QS target-resident component shall support application-specific trace records.

#### Description

Application-specific trace records have flexible format not known in advance to the QSPY host-resident component. Instead, application-specific trace records carry the format information in them (which makes them somewhat less efficient than predefined trace records). The record-ids of application-specific trace records are in the range 101-127, and are used only for the Local-Filter (see [SRS\\_QP\\_QS\\_20](#)). However, the record-ids in this case do not determine any specific format of the application-specific record. In other words, many application-specific trace records can have the same record-id. All application-specific trace records have the timestamp data element (see [SRS\\_QP\\_QS\\_10](#)).

#### Use Case

Application-specific trace records are used for tracing instrumentation embedded in the QP Applications. Their flexible format allows the QP Application to add arbitrary information to the software trace.

#### Forward Traceability (truncated to 2 level(s))

---

### 5.11.2.9 SRS\_QP\_QS\_40

QS target-resident component shall provide symbolic information in the trace by means of dictionary trace records.

**Description**

The QS target-resident component provides special predefined dictionary trace records designed expressly for providing the symbolic information about the target code in the trace itself. These dictionary records are similar to the symbolic information embedded in the object files for the traditional single-step debugger. QS supports five types of dictionary trace records:

1. object dictionary
2. function dictionary
3. signal dictionary
4. enumeration dictionary
5. user dictionary (for the application-specific trace records)

**Forward Traceability (truncated to 2 level(s))****5.11.2.10 SRS\_QP\_QS\_50**

QS target-resident component shall provide the receive channel to allow interaction between the host and the target.

**Description**

A QS-RX channel is required for interacting with the target system and implementing such features as testing, validation, verification, and monitoring of the target system. The QS-RX channel provides the following services:

- Remotely reset the Target
- Request target information (version, all sizes of objects, build time-stamp)
- Execute a user-defined command inside the target with arguments supplied from the QSPY host component
- Inject an arbitrary event to the target (dispatch, post or publish)
- Changing the Global-Filter" inside the target (see @ref SRS\_QP\_QS\_20) - Changing the Local-Filter" inside the target (see [SRS\\_QP\\_QS\\_21](#))
- Changing the Current-Object inside the target
- Peek data inside the target memory
- Poke data into the target memory
- Fill a specified target memory area with the supplied bit pattern
- Execute clock tick inside the target
- Execute test setup inside the target
- Execute test teardown inside the Target
- Store a "Test Probe" inside the target

**Forward Traceability (truncated to 2 level(s))**

## 5.12 Non-Preemptive Kernel

### 5.12.1 Concepts & Definitions

The [Active Object model of computation](#) can work with a wide range of real-time kernels. Specifically for the kernel discussed in this section, an Active Object requires an [execution context](#) only during the [RTC \(Run-to-Completion\) processing](#) and an Active Object that is merely *waiting* for events does not need an execution context at all. This opens up possibility of executing multiple Active Objects in a single thread (e.g., the `main()` function in C or C++).

#### 5.12.1.1 QV Non-Preemptive Kernel

QV is a simple non-preemptive, cooperative, fixed-priority, event-driven kernel integrated with the QP Framework. As shown in [Figure SRS-QV-LOOP](#), QV executes all Active Objects in the system in a single loop (similar to the traditional "superloop", a.k.a. "main+ISRs" architecture), so Active Objects cannot preempt each other (**non-preemptive kernel**). The QV priority-based scheduler engages only at the top of the loop and selects the highest-priority Active Object ready to run (with some event(s) in its queue). When such Active Object is found, QV executes the [RTC step](#) in this Active Object and then loops back. That way, Active Objects **cooperate** to share a single "superloop" and implicitly yield to each other after every [RTC step](#). When the QV scheduler finds no Active Objects ready to run, it invokes the [idle processing](#).

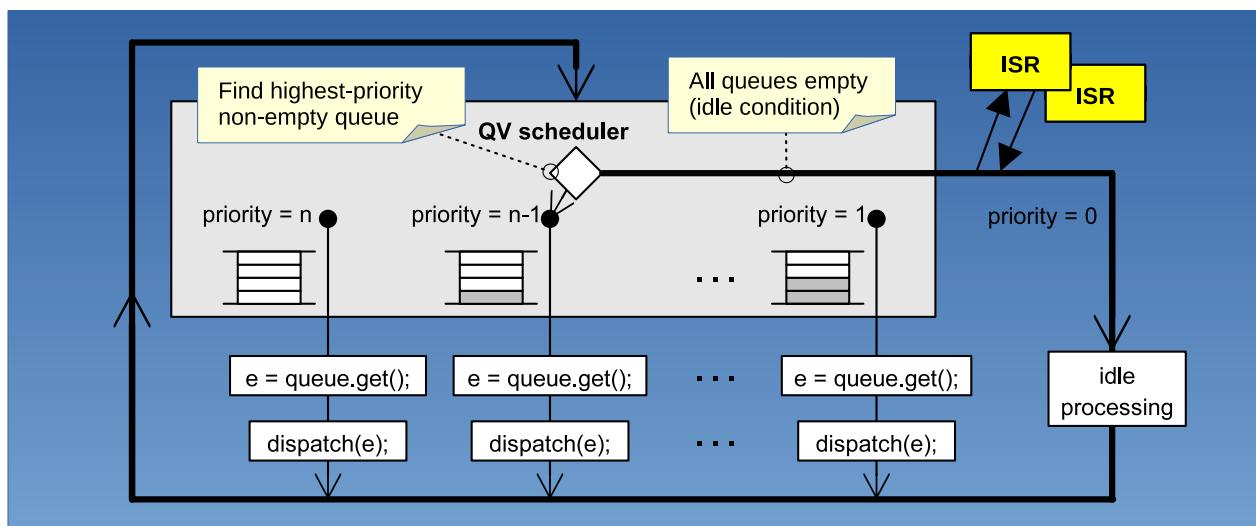


Figure SRS-QV-LOOP: QV non-preemptive kernel loop.

#### Note

The QV kernel enables partitioning the application into separate Active Objects, while preserving the simplicity and portability of the "superloop" architecture. Due to the simplicity and restrictions on preemption (which is only allowed for ISRs), the QV kernel is a *recommended* choice for **safety-critical** applications.

#### 5.12.1.2 Sharing Resources in QV

As described in the [Shared-Nothing Principle](#) for Active Objects, QP Applications should generally strive to avoid any sharing of resources among Active Objects. However, the simplistic and *non-preemptive* nature of the QV kernel allows applications to relax the "shared-nothing" principle and safely share some resources among Active Objects.

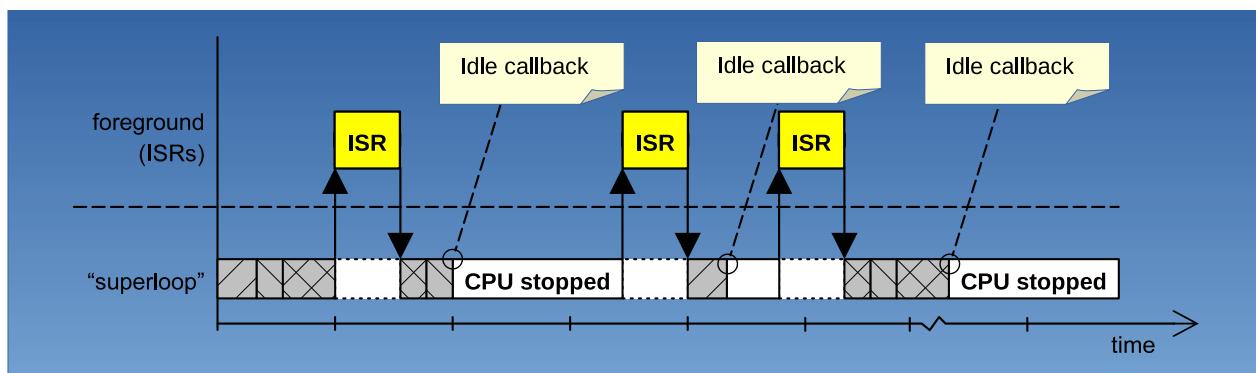
#### Note

As soon as any resource sharing among Active Objects is introduced, the application becomes locked to a *non-preemptive* kernel and stops being portable to a *preemptive* kernel. Conversely, an application that heeds the [Shared-Nothing Principle](#) remains widely portable to any real-time kernel, including *preemptive* kernels.

### 5.12.1.3 Idle Processing in QV

The situation when QV scheduler finds no events for processing in a given pass through the "superloop" is called the **idle condition**. In that case, the QV kernel executes **idle processing** to let the application, among others, put the CPU and peripherals in a *low-power sleep mode* (see [Figure SRS-QV-IDL](#)). Existence of such a single point to apply low-power modes is a hallmark of a **power-friendly architecture**.

However, as in the traditional "superloop" (a.k.a., "foreground/background" architecture) QV kernel must detect the *idle condition* inside a critical section (with interrupts disabled) and must be careful to enter the low-power mode **safely** without re-enabling interrupts too soon (see [\[Samek:07\]](#)). Otherwise any interrupt allowed after determining the *idle condition* but before calling the *idle processing* could post events to Active Objects, thus invalidating the *idle condition*. However, due to the simplistic, non-preemptive nature of the QV kernel, the idle processing would still be called and would enter the sleep mode while some events might be available and waiting (indefinitely) for processing.



*Figure SRS-QV-IDL: QV non-preemptive kernel idle processing*

#### Note

The need to enter the *low-power sleep mode* with interrupts still disabled (or atomically with re-enabling interrupts) is a unique requirement of a *non-preemptive* kernel, such as QV. This requirement does not apply to *preemptive* kernels.

### 5.12.1.4 Task-Level Response in QV

The maximum time an event for the highest-priority Active Object can be delayed is called the **task-level response**. The task-level response in the QV kernel is equal to the longest RTC step of all Active Objects in the system. Note that this task-level response is still a lot better than the traditional "superloop" (a.k.a. main+ISRs) architecture, where the task-level response is typically the *sum* of the worst-case execution times of all tasks in the "superloop".

Due to the [non-blocking](#) nature of event processing inside Active Objects, the RTC steps tend to be short (typically microseconds), which can deliver adequate real-time performance to surprisingly wide range of applications. Also, the task-level response can be often improved by breaking up the longest RTC steps into shorter pieces (multi-stage processing). For example, an Active Object can perform only a fraction of the overall event processing and post event to self to trigger continuation next time ("Reminder" state pattern).

#### Remarks

Sometimes the task-level response of the simple QV kernel might be too slow and it is impractical to break up all the long RTC steps into shorter pieces. In such cases a *preemptive* kernel (such as [QK](#), [QXK](#), or a traditional RTOS) can provide a more robust solution. The big advantage of a preemptive kernel is that it effectively decouples high-priority Active Objects from low-priority Active Objects in the *time domain*. The timeliness of execution of high-priority Active Object is almost independent on the low-priority Active Objects. However, preemptive kernels open the whole new class of problems, collectively known as *concurrency hazards*.

## 5.12.2 Requirements

### 5.12.2.1 SRS\_QP\_QV\_00

QP Framework shall provide non-preemptive QV kernel as one of the built-in kernels.

#### Description

QP Framework shall provide QV kernel implementation and ports to the supported CPU/compiler combinations as one of the optional software components. QP Application can then choose the QV kernel to execute Active Objects. Such a selection is exclusive, meaning that when QP Application selects the QV kernel, other kernels are excluded and cannot be used.

#### Description

QP Framework can implement the QV kernel component by re-using already existing mechanisms, such as event queues for Active Objects, event delivery mechanisms, event memory management, etc. That way, the QV kernel implementation can be quite small and consist only of the missing pieces, such as the "superloop" with the QV scheduler.

#### Forward Traceability (truncated to 2 level(s))

---

### 5.12.2.2 SRS\_QP\_QV\_10

QV kernel shall provide an idle-callback defined in QP Application.

#### Description

When no events are available in a given pass through the "superloop" (see [Figure SRS-QV-SCH](#)), the QV kernel executes **idle processing**. This idle processing shall invoke an **idle-callback** (a function) defined in QP Application. The *idle-callback* can perform any processing, including putting the CPU and peripherals in a low-power sleep mode.

#### Note

The *idle-processing* of the QV kernel can be viewed as the lowest-priority task that can't be preempted by other Active Objects. Consequently the idle-processing time also counts as another RTC step, which must be considered for the [task-level response](#) of the QV kernel.

#### Use Case

The *idle-callback* can perform [Software Tracing](#) data transfer to the host, or other processing.

#### Forward Traceability (truncated to 2 level(s))

---

### 5.12.2.3 SRS\_QP\_QV\_11

The idle-callback shall be invoked with interrupts disabled to allow a safe transition to a low-power sleep mode.

**Description**

As described in Section [Idle Processing in QV](#), a **safe** transition to low-power sleep mode requires the QV *idle-callback* to be invoked with **interrupts disabled** (inside the same critical section as the detection of the *idle condition*).

**Forward Traceability (truncated to 2 level(s))****5.12.2.4 SRS\_QP\_QV\_12**

The idle-callback shall always return with interrupts enabled.

**Description**

Regardless whether the *idle-callback* switches to a low-power mode, QP Application must define the *idle-callback* such that it re-enables interrupts in every path through the code. The QV kernel assumes that this will be the case, and will not work correctly if the *idle-callback* fails to enable interrupts.

**Forward Traceability (truncated to 2 level(s))****5.12.2.5 SRS\_QP\_QV\_20**

QV kernel may provide API to selectively disable scheduling Active Objects below the specified scheduler-disable ceiling priority.

**Description**

The selective scheduler disabling API shall prevent scheduling any Active Object whose **unique priority** is below the specified *scheduler-disable ceiling priority*.

**Use Case**

The main use case for selective scheduler disabling is in **time-triggered** designs and Active Objects with multi-stage processing. An Active Object that breaks up its long RTC steps into shorter pieces (multi-stage processing) would self-post a special "Reminder" event to trigger subsequent stages of processing. However, to prevent such processing from overrunning the next clock period, the Active Object can explicitly disable the QV scheduler up to its own priority level.

**Forward Traceability (truncated to 2 level(s))****5.12.2.6 SRS\_QP\_QV\_21**

If QV kernel provides API to disable the scheduler it shall provide the API to enable the scheduler.

**Description**

The scheduler enabling API shall complement the scheduler disabling API (see [SRS\\_QP\\_QV\\_20](#)). The scheduler enabling API shall restore the *scheduler-disable ceiling* established in the most recent call to the scheduler disabling API.

**Use Case**

The main use case for scheduler enabling (matching the use case in [SRS\\_QP\\_QV\\_20](#)) is enabling the QV scheduler in the system clock tick.

**Forward Traceability (truncated to 2 level(s))**

---

## 5.13 Preemptive Non-Blocking Kernel

### 5.13.1 Concepts & Definitions

The Active Object model of computation can work with a wide range of real-time kernels. Specifically for the kernel discussed in this section, the [non-blocking](#) and [run-to-completion](#) characteristics of the Active Object model open up a possibility of using a *non-blocking*, run-to-completion kernel. Such a kernel can be **preemptive** and fully compatible with the requirements of Rate Monotonic Scheduling/Analysis (RMS/RMA) method [RMS/RMA:91]. Still, a non-blocking kernel is much simpler and significantly more efficient than any traditional blocking RTOS kernel.

#### 5.13.1.1 QK Preemptive Non-Blocking Kernel

QK is a preemptive, non-blocking, fixed-priority, run-to-completion, single-stack, event-driven kernel integrated with the QP Framework. QK executes all Active Objects in discrete, one-shot, non-blocking [RTC steps](#) that can **preempt** each other. This preemption is similar to how prioritized interrupts can preempt each other and nest on a **single stack** (e.g., see description of interrupt handling in ARM Cortex-M [Yiu:14]). Interrupts, just like [RTC steps](#) in Active Objects, are also one-shot, run-to-completion steps that are not allowed to block.

#### Remarks

The one-shot, run-to-completion, non-blocking, preemptible schedulable units of work are known in the literature as "jobs" (Stack Resource Policy [SRP:90]) or "basic tasks" (OSEK/AUTOSAR terminology [OSEK:03]). Similar concepts are also called "fibers" (e.g., Q-Kernel), "deferred interrupts" (e.g., SMX kernel), or "software interrupts" (e.g., TI-RTOS).

#### 5.13.1.2 Preemptions in QK

As a fully *preemptive*, fixed-priority kernel, QK must ensure that at all times the CPU executes the highest-priority Active Object (AO) as soon as it *becomes ready* to run. Fortunately, only two scenarios can lead to preemption of a lower-priority AO1 by a higher-priority AO2.

#### Note

The QK kernel is one of the simplest and most efficient kernels fully compatible with RMS/RMA/DMS. Due to the simplicity, the QK kernel is a *recommended* choice for **safety-critical** applications.

#### Synchronous Preemption

When a lower-priority AO1 posts an event to a higher-priority AO2, the QK kernel must immediately suspend the execution of the lower-priority AO1 and start the higher-priority AO2. This type of preemption is called **synchronous preemption** because it happens synchronously with posting an event to the AO2's event queue.

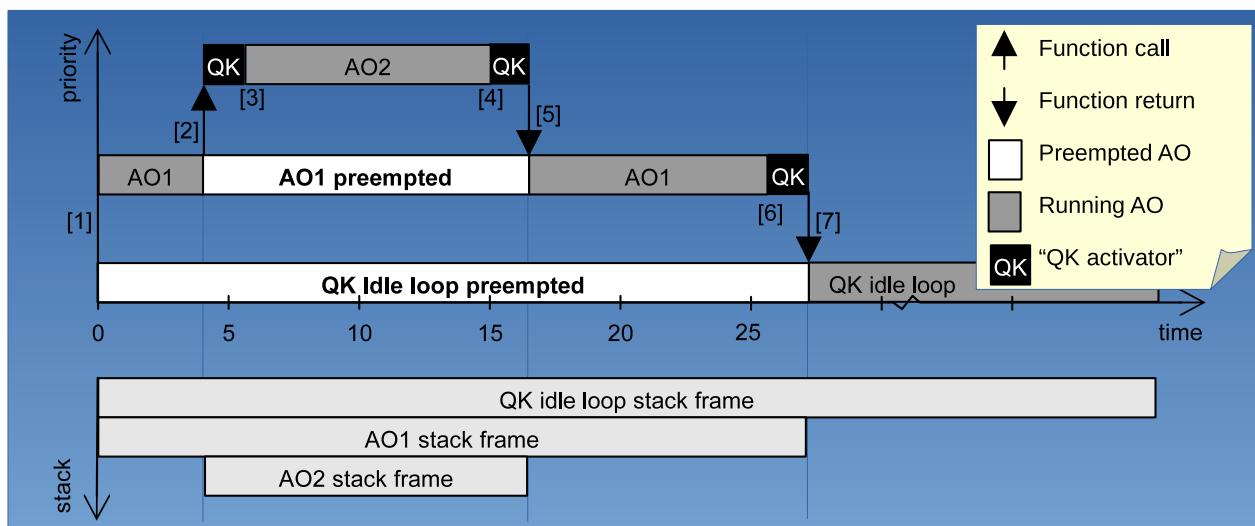


Figure SRS-QK-SYN: Synchronous Preemption in QK. The stack is shown in the bottom panel grows "down" (as on ARM Cortex-M).

Figure SRS-QK-SYN illustrates a **synchronous** **preemption** scenario:

- [1] QK idle loop (QK idle task) is preempted by an AO1, which executes its RTC step
- [2] AO1 posts an event to a higher-priority AO2, which calls the "QK activator"
- [3] "QK activator" activates (calls) the AO2 RTC step
- [4] AO2 RTC step completes and returns to the "QK activator"
- [5] "QK activator" returns back to AO1, which remained *synchronously* preempted
- [6] AO1 completes its RTC step and returns to "QK activator" (previous instance)
- [7] "QK activator" finds no more AOs to run and returns to the QK idle loop.

#### Remarks

A traditional RTOS kernel does not distinguish between the synchronous and asynchronous preemptions (see next) and makes all preemptions look like the more stack-intensive asynchronous preemptions. In contrast, a run-to-completion kernel like QK can implement synchronous preemption as a simple function call, which is more efficient than a full context-switch.

#### Asynchronous Preemption

When an *interrupt* posts an event to a higher-priority AO2 than the interrupted AO1, upon completion of the ISR the QK kernel must start execution of the higher-priority AO2 instead of resuming the lower-priority AO1. This type of preemption is called **asynchronous** **preemption** because it can happen asynchronously, any time interrupts are not explicitly disabled.

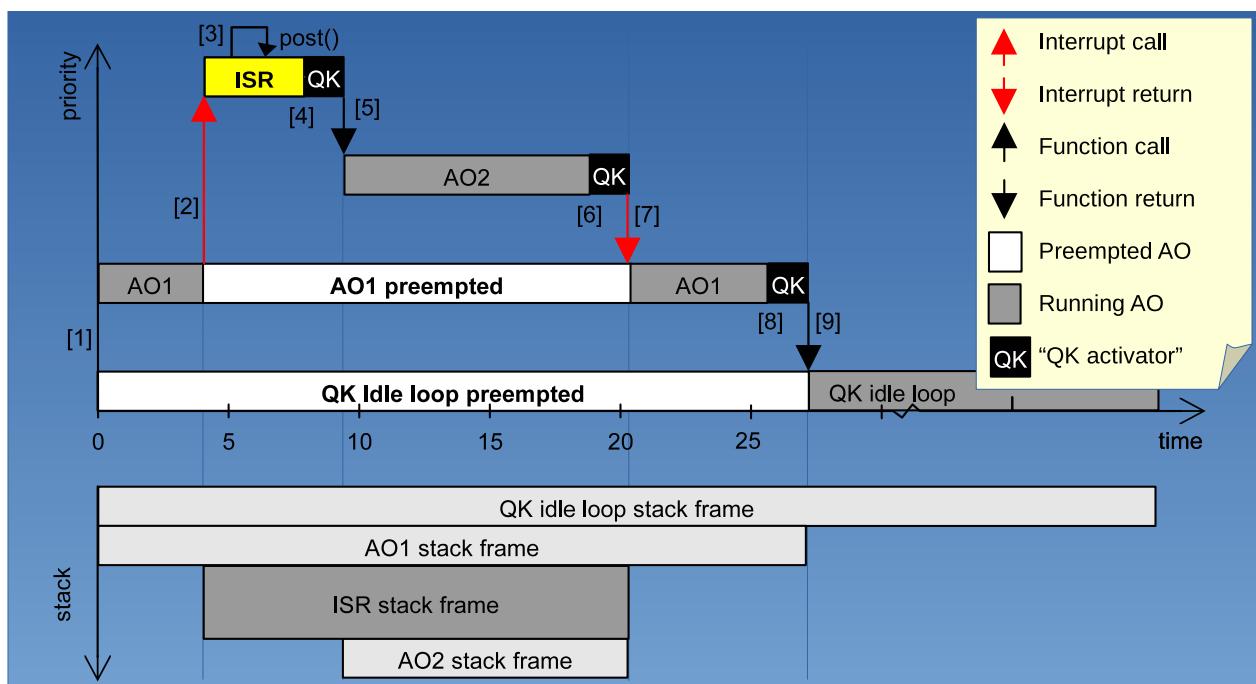


Figure SRS-QK-ASN: Asynchronous Preemption in QK. The stack is shown in the bottom panel grows "down" (as on ARM Cortex-M).

Figure SRS-QK-ASN illustrates an **asynchronous** **preemption** scenario:

- [1] QK idle loop (QK idle task) is preempted by an AO1, which executes its RTC step
- [2] An interrupt fires and asynchronously preempts the AO1 RTC step
- [3] The ISR for the interrupt starts running and posts an event to high-priority AO2
- [4] The ISR sends the EOI (End-of-Interrupt) command to the interrupt-controller, but does NOT return to the preempted context (so the interrupt stack frame remains on the stack). Instead, the ISR calls "QK activator".

### Remarks

The details of this step depend on the CPU and interrupt controller and might be more complex. For example ARM Cortex-M CPU requires additional step of activating the PendSV exception.)

- [ 5 ] "QK activator" activates (calls) the AO2 RTC step
- [ 6 ] AO2 RTC step completes and returns to the "QK activator"
- [ 7 ] "QK activator" performs **interrupt**-return back to AO1, which remained *asynchronously* preempted
- [ 8 ] AO1 completes its RTC step and returns to "QK activator" (previous instance)
- [ 9 ] "QK activator" finds no more AOs to run and returns to the QK idle loop.

#### 5.13.1.3 Single-Stack Kernel

By requiring that all AOs run-to-completion and enforcing fixed-priority scheduling, a non-blocking kernel like QK can manage all context information using only a **single stack** and the CPU's *natural stack protocol*. Whenever an AO posts an event to a higher-priority AO, QK kernel uses a regular C function call to build the higher-priority AO context on top of the existing stack context ([synchronous preemption](#)). Whenever an interrupt preempts an AO and the interrupt posts an event to a higher-priority AO, the QK kernel uses the already established interrupt stack frame on top of which to build the higher-priority AO context, again using a regular C function call ([asynchronous preemption](#)).

This simple form of context management is adequate because every AO, just like every ISR, runs to completion. Because the preempting AO must also run to completion, the lower-priority context will never be needed until the preempting AO (and any higher-priority AO that might preempt it) has completed and returned — at which time the preempted AO will, naturally, be at the top of the stack, ready to be resumed.

#### 5.13.1.4 Idle Processing in QK

The situation when QK kernel finishes processing all [RTC steps](#) in all Active Objects is called the **idle condition**. In that case, the QK kernel executes *idle processing*. Idle processing in QK can be viewed as the lowest-priority task (of priority 0), but unlike all other one-shot tasks, it has a structure of an *endless loop* (see "QK idle loop" in [Figure SRS-QK-SYN](#) and [Figure SRS-QK-ASN](#)).

This QK idle loop invokes a QK-idle-callback defined in the application, to let the application, among others, put the CPU and peripherals in a *low-power sleep mode*. Existence of such a single point to apply low-power modes is a hallmark of a **power-friendly architecture**.

### Remarks

The QK-idle-callback is invoked with interrupts enabled because transition to a sleep mode in a *preemptive* QK kernel is safe from the hazards described for the [non-preemptive QV kernel](#).

#### 5.13.1.5 Selective Scheduler Locking

As all *preemptive* kernels, QK requires the QP Application to be very careful with any resource sharing among Active Objects. Ideally, the Active Objects should communicate exclusively via events and otherwise should [not share](#) any resources. However, at the cost of increased coupling among Active Objects, QP Application might choose to share selected resources. However, such QP Application takes the burden on itself to apply a **mutual exclusion mechanism** while accessing any shared resources.

QK kernel provides **selective scheduler locking** as a powerful mutual exclusion mechanism. Specifically, before accessing a shared resource, an Active Objects can **lock** the QK scheduler up to the specified *priority ceiling*. This prevents Active Object preemption up to the specified *priority ceiling*, while not affecting Active Objects (or interrupts) of priority higher than the *priority ceiling*. After accessing the shared resource the Active Object must **unlock** the QK scheduler. Selective scheduler locking in QK is related to IPCP/SRP (Immediate Priority Ceiling Protocol/Stack Resource Policy) and is immune to unbound priority inversion [[SRP:90](#)].

Selective scheduling locking is a *non-blocking* mechanism. If an Active Object that needs to protect a shared resource is running, it means that all Active Objects of higher priority have no events to process. Consequently, simply preventing activation of higher-priority AOs that might access the resource is sufficient to guarantee the mutually exclusive access to the resource. Of course, you don't need to worry about any lower-priority AOs that might be preempted because they never resume until the current AO runs to completion.

Selective scheduler lock requests made by the same Active Object *can nest*. The nested lock requests can only increase the priority ceiling. Also, the nested lock requests must unlock the scheduler by *restoring* the previous priority ceiling.

#### 5.13.1.6 Preemption-Threshold Scheduling

While preemption is a desirable property that enables techniques like RMS/RMA by decoupling higher-priority Active Objects from lower-priority ones in the *time domain*, too much preemption has also negative effects. These include the overhead of preemption and restrictions on sharing resources. For example, sometimes a group of Active Objects forms a cohesive subsystem, where preemption of one group member by another might be unnecessary and undesirable.

To ease some of the inherent problems of preemption, QK provides an advanced feature called **preemption threshold scheduling** (PTS), [PTS:07]. PTS allows an Active Object to specify a *preemption threshold* to selectively **restrict preemption** by other Active Objects. Active Objects that have priorities higher than the *preemption threshold* are still allowed to preempt, while those less than the threshold are not allowed to preempt.

For example, all Active Objects in a group might specify the same *preemption threshold* (alongside their unique priorities). Such *preemption threshold* will prevent preemption within the group, while still allowing preemption by other Active Objects with higher priorities than the *preemption threshold*.

#### 5.13.1.7 Task-Level Response

The maximum time an event for the highest-priority Active Object can be delayed is called the *task-level response*. The preemptive QK kernel always executes the highest-priority Active Object *immediately* as it becomes ready to run (see [Preemptions in QK](#)). Therefore, the task-level response in the QK kernel is determined only by the context switch time, which is independent on the lower-priority Active Objects. That is what is meant that a preemptive kernel decouples tasks in the time domain.

##### Remarks

[Selective scheduler locking](#) and [preemption-threshold scheduling](#) re-introduce some time-domain coupling among Active Objects and can also negatively impact the task level response of the QK kernel.

As a preemptive kernel with fixed-priority scheduling, QK fulfills all requirements of the Rate Monotonic Scheduling/Analysis (RMS/RMA) [RMS/RMA:91] and the related Deadline-Monotonic Scheduling (DMS) [RMS/RMA:91][DMS:91]. Additionally, the no-blocking nature of QK makes the applications easier to analyze and thus actually even more suitable for RMS/RMA than a traditional blocking RTOS.

### 5.13.2 Requirements

#### 5.13.2.1 SRS\_QP\_QK\_00

QP Framework shall provide preemptive non-blocking QK kernel as one of the built-in kernels.

##### Description

QP Framework shall provide QK kernel implementation and ports to the supported CPU/compiler combinations as one of the optional software components. QP Application can then choose the QK kernel to execute Active Objects. Such a selection is exclusive, meaning that when QP Application selects the QK kernel, other kernels are excluded and cannot be used.

##### Description

QP Framework can implement the QK kernel component by re-using already existing mechanisms, such as event queues for Active Objects, event delivery mechanisms, event memory management, etc. That way, the QK kernel implementation can be quite small and consist only of the missing pieces, such as the QK scheduler and "activator".

Forward Traceability (truncated to 2 level(s))

---

### 5.13.2.2 SRS\_QP\_QK\_10

QK kernel shall provide an idle-callback defined in QP Application.

#### Description

When QK kernel finishes processing all [RTC steps](#) in all Active Objects it shall execute the [QK idle loop](#). This *QK idle loop* shall invoke an **idle-callback** (a function) defined in QP Application. The *idle-callback* can perform any processing, including putting the CPU and peripherals in a low-power sleep mode.

#### Use Case

The *idle-callback* shall be invoked with interrupts enabled and can perform [Software Tracing](#) data transfer to the host, or other processing.

Forward Traceability (truncated to 2 level(s))

---

### 5.13.2.3 SRS\_QP\_QK\_20

QK kernel shall provide API to selectively lock scheduling Active Objects below the specified scheduler-lock ceiling priority.

#### Description

The [selective scheduler locking](#) API shall prevent scheduling any Active Objects whose [unique priority](#) is below the specified *scheduler-lock ceiling priority*.

#### Use Case

The main use case for selective scheduler locking is during *multicasting events* to prevent unexpected and confusing event sequences (see [SRS\\_QP\\_EDM\\_55](#)).

Forward Traceability (truncated to 2 level(s))

---

### 5.13.2.4 SRS\_QP\_QK\_21

QK kernel shall provide API to unlock scheduling Active Objects.

#### Description

The scheduler unlocking API shall complement the scheduler locking API (see [SRS\\_QP\\_QK\\_20](#)). The scheduler unlocking API shall restore the *scheduler-lock ceiling* established in the most recent call to the scheduler locking API.

**Use Case**

The main use case for selective scheduler locking (matching the use case in [SRS\\_QP\\_QK\\_20](#)) is enabling the QV scheduler in the system clock tick.

**Forward Traceability (truncated to 2 level(s))**

---

**5.13.2.5 SRS\_QP\_QK\_30**

QK kernel shall support preemption-threshold scheduling (PTS).

**Description**

Support for [preemption-threshold scheduling \(PTS\)](#) means that each Active Object can be assigned a *preemption-threshold* (alongside its [unique %QP priority](#)). The Active Object with a given *preemption-threshold* cannot be preempted by Active Objects with the unique QP priorities below the *preemption threshold*.

**Forward Traceability (truncated to 2 level(s))**

---

**5.13.2.6 SRS\_QP\_QK\_31**

The preemption-threshold assigned to an Active Object must be consistent with its unique QP priority.

**Description**

The *preemption-threshold* assigned to an Active Object can be only higher than the unique QP priority. Additionally, the *preemption-threshold* cannot exceed *preemption thresholds* assigned to any higher-priority Active Objects.

**Forward Traceability (truncated to 2 level(s))**

## 5.14 Preemptive Dual-Mode Kernel

### 5.14.1 Concepts & Definitions

The [Active Object model of computation](#) can work with a wide range of real-time kernels. Among others, event-driven Active Objects can be executed by a traditional, blocking RTOS kernel, where each Active Object runs in its own RTOS thread structured as an *event-loop*. However, using a traditional, blocking RTOS to execute non-blocking Active Objects is wasteful. As demonstrated in the [QK Preemptive Non-Blocking Kernel](#), non-blocking Active Objects can be executed more efficiently as one-shot, run-to-completion steps.

#### 5.14.1.1 QXK Dual-Mode Kernel

QXK is a preemptive, fixed-priority, dual-mode (blocking / non-blocking) kernel that executes Active Objects like the [QK kernel \(basic tasks\)](#), but can also execute traditional **blocking** threads ([extended tasks](#)). In this respect, QXK behaves like a traditional RTOS.

##### Remarks

QXK adopts the "basic/extended tasks" terms from the [OSEK/AUTOSAR Operating System](#) specification [[OSEK:03](#)]. Other real-time kernels might use different terminology for similar concepts.

QXK has been designed specifically for combining event-driven Active Objects with traditional code that requires **blocking**, such as commercial middleware (TCP/IP stacks, UDP stacks, embedded file systems, etc.) or legacy software. To this end, QXK is not only more efficient than running QP on top of a [traditional 3rd-party RTOS](#) (because non-blocking **basic tasks** take less stack space and CPU cycles for context switch than the much heavier **extended tasks**). But the biggest advantage of QXK is that it *protects* the application-level code from inadvertent mixing of blocking calls inside the event-driven Active Objects. Specifically, QXK "knows" the type of the task context (extended/basic) and asserts internally if a blocking call (e.g., semaphore-wait or a time-delay) is attempted in a basic task (Active Object).

##### Note

The QXK dual-mode kernel is significantly more complex than the [QV non-preemptive kernel](#) and the [QK preemptive non-blocking kernel](#), and therefore it is **not recommended** for safety-critical applications.

#### 5.14.1.2 Basic Tasks

**Basic** tasks are one-shot, non-blocking, run-to-completion activations. The basic-task can nest on the same stack. Also, context switching from a basic-task to another basic-task requires only activation of the basic-task, which is simpler and faster than full context-switch required for [extended](#)-tasks (that QXK also supports, see below).

#### 5.14.1.3 Extended tasks

**Extended** tasks are endless loops allowed to block. Extended tasks require private per-task stacks, as in conventional RTOS kernels. Any switching from basic-to-extended task or extended-to-extended task requires full context switch.

##### Remarks

QXK is a unique dual-mode kernel on the market that supports interleaving the priorities of basic tasks and extended tasks. Other dual-mode kernels typically limit the priorities of basic tasks to be always higher (more urgent) than any of the extended tasks.

#### 5.14.1.4 QXK Blocking Mechanisms

QXK provides most blocking mechanisms found in traditional blocking RTOS kernels:

- **Counting Semaphores** with optional timeout that can block multiple extended-tasks;
- **Binary Semaphores** (as a subset of counting semaphores);

- **Blocking Event Queue** with optional timeout bound to each extended-task;
- Priority-Ceiling, recursive **Mutexes** with optional timeout;

#### 5.14.1.5 Selective Scheduler Locking

As all *preemptive* kernels, QXK requires the QP Application to be very careful with any resource sharing among Active Objects. Ideally, the Active Objects should communicate exclusively via events and otherwise should **not share** any resources. However, at the cost of increased coupling among Active Objects, QP Application might choose to share selected resources. However, such QP Application takes the burden on itself to apply a **mutual exclusion mechanism** while accessing any shared resources.

QXK kernel provides **selective scheduler locking** as a powerful mutual exclusion mechanism. Specifically, before accessing a shared resource, an Active Objects can **lock** the QXK scheduler up to the specified *priority ceiling*. This prevents Active Object preemption up to the specified *priority ceiling*, while not affecting Active Objects (or interrupts) of priority higher than the *priority ceiling*. After accessing the shared resource the Active Object must **unlock** the QXK scheduler.

Selective scheduling locking is a *non-blocking* mechanism. If an Active Object that needs to protect a shared resource is running, it means that all Active Objects of higher priority have no events to process. Consequently, simply preventing activation of higher-priority AOs that might access the resource is sufficient to guarantee the mutually exclusive access to the resource. Of course, you don't need to worry about any lower-priority AOs that might be preempted because they never resume until the current AO runs to completion.

Selective scheduler lock requests made by the same Active Object *can nest*. The nested lock requests can only increase the priority ceiling. Also, the nested lock requests must unlock the scheduler by *restoring* the previous priority ceiling.

#### 5.14.1.6 Task-Level Response

As a fully *preemptive*, fixed-priority kernel, QXK always executes the highest-priority task that is ready to run (is not blocked). The scheduling algorithm used in QXK meets all the requirement of the Rate Monotonic Scheduling (a.k.a. Rate Monotonic Analysis RMA) and can be used in hard real-time systems.

### 5.14.2 Requirements

#### Note

As [stated above](#), due to its complexity the QXK dual-mode kernel is **not recommended** for safety-critical applications. Therefore, this section contains only high-level requirements for the QXK kernel without the details needed to verify and validate QXK for functional safety.

#### 5.14.2.1 SRS\_QP\_QXK\_00

QP Framework shall provide preemptive non-blocking QXK kernel as one of the built-in kernels.

#### Description

QP Framework shall provide QXK kernel implementation and ports to the supported CPU/compiler combinations as one of the optional software components. QP Application can then choose the QXK kernel to execute Active Objects and extended tasks. Such a selection is exclusive, meaning that when QP Application selects the QXK kernel, other kernels are excluded and cannot be used.

#### Description

QP Framework can implement the QXK kernel component by re-using already existing mechanisms, such as event queues for Active Objects, event delivery mechanisms, event memory management, etc.

Forward Traceability (truncated to 2 level(s))

---

#### 5.14.2.2 SRS\_QP\_QXK\_10

QXK kernel shall support basic tasks for execution of Active Objects and shall handle them like the QK kernel.

##### Description

QXK kernel shall provide API for designating an Active Object, which will require a unique priority and event queue, but not a private stack. Subsequently, QXK kernel shall execute thus designated Active Object as a basic-task by dispatching events to its state machine.

Forward Traceability (truncated to 2 level(s))

---

#### 5.14.2.3 SRS\_QP\_QXK\_11

QXK kernel shall support extended tasks for execution of blocking threads and shall execute them like a conventional RTOS kernel.

##### Description

QXK kernel shall provide API for designating an extended task (distinct from designating an Active Object), which will require a unique priority, optional event queue, and a private stack. Subsequently, QXK kernel shall execute thus designated object as an extended task. If an extended task is configured with an event queue, such an extended task can *block* on that queue and receive events through that queue.

Forward Traceability (truncated to 2 level(s))

---

#### 5.14.2.4 SRS\_QP\_QXK\_12

QXK kernel shall support mixing basic and extended tasks.

##### Description

QXK shall allow combining basic tasks (Active Objects) and extended tasks (threads) in a single application. QXK shall allow interleaving the unique priorities of basic tasks and extended tasks and prioritize them according to the same uniform prioritization scheme described in [SRS\\_QP\\_AO\\_10](#).

Forward Traceability (truncated to 2 level(s))

---

#### 5.14.2.5 SRS\_QP\_QXK\_13

QXK kernel shall support interactions between basic and extended tasks.

##### Description

QXK shall support the following integrated mechanisms for communication between extended and basic tasks:

- Extended tasks can post or publish events to Active Objects;
- Extended tasks can post or publish events to other extended tasks (that provide an event queue);
- Extended tasks that provide an event queue can subscribe to events and thus can participate in the publish/subscribe event delivery;
- Basic tasks (Active Objects) can signal semaphores and post events to extended tasks (those that provide an event queue).

Forward Traceability (truncated to 2 level(s))

---

#### 5.14.2.6 SRS\_QP\_QXK\_20

QXK kernel shall support counting semaphores with optional timeout.

##### Description

QXK shall support counting semaphores as one of the standard blocking mechanisms. As a special case, counting semaphore with maximum 1 shall behave as a binary semaphore. The blocking semaphore-wait operation (with optional timeout) shall be allowed only inside the extended threads. However, signaling a semaphore is allowed both from Active Objects and ISRs.

Forward Traceability (truncated to 2 level(s))

---

#### 5.14.2.7 SRS\_QP\_QXK\_21

QXK kernel shall support blocking event queues with optional timeout.

##### Description

For extended tasks configured with an event queue, QXK shall support blocking on that event queue (with optional timeout) inside the endless loop of the extended task.

Forward Traceability (truncated to 2 level(s))

---

### 5.14.2.8 SRS\_QP\_QXK\_22

QXK kernel shall support blocking recursive mutexes with optional timeout.

#### Description

QXK shall support mutexes as one of the standard mutual exclusion mechanisms. The blocking mutex-lock operation (with optional timeout) shall be allowed only inside the extended threads. The mutex shall support the priority-ceiling protocol to prevent unbounded priority inversion. The mutex shall also allow recursive locking (by the same extended task).

#### Forward Traceability (truncated to 2 level(s))

---

## 5.15 Non-Functional Requirements

This section lists and explains **non-functional requirements** (a.k.a., quality attributes) of the QP Framework.

### 5.15.1 Standards Compliance

#### 5.15.1.1 SRS\_QP\_NF\_01

QP Framework shall be implemented in a standardized programming language.

##### Description

A standardized programming language means a language specified by an international standard, such as C99, C11, C++11, C++14, etc.

##### Rationale

A standardized programming language provides a well-defined, documented, and stable set of programming constructs and features.

##### Forward Traceability (truncated to 1 level(s))

---

#### 5.15.1.2 SRS\_QP\_NF\_02

QP Framework shall be implemented in a programming language recommended for safety-critical software.

##### Description

Recommended for safety-critical software means a programming language that is highly recommended for SIL3 and SIL4 according to IEC-61508. For example, the full programming languages like C99 or C++11 are generally *not* recommended for higher SILs. However, **safe subsets** of these languages, such as MISRA-C:2023 or MISRA-C++:2023 are highly recommended (see [IEC-61508-3] Table A.3, [IEC-61508-7] C.4.6 Table C.1).

##### Rationale

QP Framework shall be certifiable to the international functional safety standards, which require an appropriate implementation programming language.

##### Forward Traceability (truncated to 1 level(s))

---

#### 5.15.1.3 SRS\_QP\_NF\_03

QP Framework shall be modular.

##### Description

QP Framework shall apply the following rules:

- information hiding/encapsulation
- fully defined interface with explicit specification of function signatures
- fully documented interfaces

- modules shall guarantee correct use by preconditions and postconditions (see failure assertion programming)
- all subprograms (functions) shall have a single entry and a single exit
- all subprograms (functions) shall have number of parameters limited to 4
- all subprograms (functions) shall have complexity limited to 20 (cyclomatic complexity)
- global variables shall be limited to less than 10 for the whole QP Framework

**Rationale**

Modularity is highly recommended by the international functional safety standards.

**Forward Traceability (truncated to 1 level(s))**

- [SAS\\_QP\\_FRM](#): *QP Framework layer*
  - [SAS\\_QP\\_OSAL](#): *OS abstraction layer (OSAL)*
  - [SAS\\_QP\\_OS](#): *OS layer*
- 

**5.15.1.4 SRS\_QP\_NF\_04**

QP Framework shall be layered.

**Description****Rationale**

Layered design is highly recommended by the international functional safety standards.

**Forward Traceability (truncated to 1 level(s))**

- [SAS\\_QP\\_FRM](#): *QP Framework layer*
  - [SAS\\_QP\\_OSAL](#): *OS abstraction layer (OSAL)*
  - [SAS\\_QP\\_OS](#): *OS layer*
  - [SDS\\_QP\\_QEP](#): *QEP Event Processor*
  - [SDS\\_QP\\_QF](#): *QF Active Object Framework*
- 

**5.15.2 Determinism****5.15.2.1 SRS\_QP\_NF\_20**

All services provided by QP must be deterministic.

**Description**

Deterministic services means that the service has a well-defined upper bound on the CPU and memory utilization.

**Rationale**

QP Framework shall be certifiable to the international functional standards, which require determinism.

**Forward Traceability (truncated to 1 level(s))**

### 5.15.2.2 SRS\_QP\_NF\_21

All services provided by QP shall use limited and known amount of call stack

#### Description

The call stack (or multiple call stacks, if used) is a limited resource taking up a significant portion of the precious RAM in deeply embedded applications targeted by QP Framework.

#### Rationale

The stack is a *critical* resource in that its exhaustion (a.k.a. stack overflow) represents a system *failure*, rather than merely some degradation of performance. Therefore, it is imperative for QP Framework (and QP Application by extension) to use the call stack efficiently and deterministically.

Forward Traceability (truncated to 1 level(s))

---

## 5.15.3 Portability

### 5.15.3.1 SRS\_QP\_NF\_40

QP Framework shall be portable to 16-/32-/64-bit CPUs.

#### Description

QP Framework shall be implemented such that

#### Rationale

QP Framework shall internally be implemented without implicit assumptions about the CPU register size for correctness of the internal algorithms. Among others, the bit-widths of integer variables must be specified in the implementation, so that they provide adequate dynamic range. Also, integer overflow and underflow (e.g., "wrap-around"), if relevant for a given algorithm, need to be handled without assumptions on the CPU register size.

Forward Traceability (truncated to 1 level(s))

- [SAS\\_OSAL\\_API](#): OSAL API

---

### 5.15.3.2 SRS\_QP\_NF\_41

QP Framework shall be portable to a wide range of real-time kernels.

#### Description

As an implementation of the [Active Object model of computation](#), QP Framework shall preserve the inherent ability of Active Objects work with a variety of real-time kernels. This includes traditional blocking RTOS kernels, special non-blocking kernels (e.g., [QV](#), [QK](#), [QXK](#)), and general-purpose operating systems (POSIX, Windows).

#### Rationale

Portability of QP Framework (and QP Applications based on the framework) to a wide range of kernels allows applications to use the most appropriate kernel for the job at hand. Additionally, portability to GP-OS allows running and developing and testing QP Applications on the host computers.

Forward Traceability (truncated to 1 level(s))

- [SAS\\_OSAL\\_API](#): OSAL API

---

### 5.15.4 Ease of Development

#### 5.15.4.1 SRS\_QP\_NF\_50

All State Machine Implementation Strategies provided by QP Framework shall be easy to debug.

##### Description

"Easy to debug" state machine implementation means that it should always be possible to set a single debugger breakpoint in the code to stop upon the execution of a given state machine element, such as a state transition or entry to a given state, exit from a given state, a given guard condition, etc. An example of a hard to debug implementation would be one in which a given transition or entry to a state would be represented more than once in the code. A single breakpoint would not necessarily be sufficient to guarantee that the breakpoint is reached when that element is executed. Hard to debug state machine implementations are often the result of "flattening" the hierarchical state machine to the non-hierarchical representation, which causes repetitions of transitions.

##### Rationale

The ease of debugging is essential for effective development and maintenance of the state machine code.

##### Forward Traceability (truncated to 1 level(s))

---

### 5.15.5 Constraints

#### 5.15.5.1 SRS\_QP\_NF\_10

QP Framework implementation shall NOT depend on any standard libraries.

##### Description

##### Forward Traceability (truncated to 1 level(s))

---

#### 5.15.5.2 SRS\_QP\_NF\_11

QP Framework implementation shall NOT use floating point.

##### Description

##### Forward Traceability (truncated to 1 level(s))

---

### 5.15.5.3 SRS\_QP\_NF\_12

QP Framework shall NOT use multiple inheritance.

#### Description

Multiple inheritance is complex and can lead to problems (e.g., problematic diamond-shaped inheritance). Therefore, safe subsets of OO programming languages often disallow use of multiple inheritance.

Forward Traceability (truncated to 1 level(s))

---

### 5.15.5.4 SRS\_QP\_NF\_13

All services provided by QP must NOT use the standard heap.

#### Description

QP Framework must not force Applications to use potentially unsafe and indeterministic memory allocation policies, like the standard heap. In other words, the QP Framework should internally use only static memory with worst-case memory consumption known at compile-time.

---

### 5.15.5.5 SRS\_QP\_NF\_14

QP Framework shall NOT internally pre-allocate memory for application-specific objects such as event-queue buffers; event-pool buffers; tracing-buffers etc.

#### Description

Forward Traceability (truncated to 1 level(s))

---

## Chapter 6

# Software Architecture Specification



This *Software Architecture Specification* is part of the [SafeQP Certification Kit](#), but applies to the whole [QP Framework family](#). This document is the best source of information about the *master plan* for the [overall organization](#) of QP Framework as well as *QP Applications derived from the framework*. The detailed QP Framework *design* is described in a separate document: [QP Software Design Specification \[DOC\\_SDS\\_QP\]](#).

### Revision History

QP version	Document revision	Date (YYYY-MM-DD)	By	Description
7.3.4	A	2024-05-05	MMS	Initial release for IEC-61508 SIL-3 and IEC-62304 Class-C.
7.4.0	B	2024-07-30	MMS	Updated for QP 7.4.0.
8.0.0	C	2024-11-17	MMS	Updated for QP 8.0.0.

## 6.1 About this Document

### 6.1.1 DOC\_SAS\_QP

Software Architecture Specification (SAS)

#### Description

This *Software Architecture Specification* (SAS), with Unique Identifier: [DOC\\_SAS\\_QP](#), describes the **software architecture** of QP Framework that satisfies the [QP Software Requirements Specification \(DOC\\_SRS\\_QP\)](#) and the QP Software Safety Requirements Specification ([DOC\\_SSR\\_QP](#)).

### Note

The architecture specified in this document can be ultimately implemented in various programming languages, so this document applies to a whole family of QP Real-Time Event Frameworks (RTEFs), currently consisting of **QP/C**, **QP/C++**, **SafeQP/C**, and **SafeQP/C++** frameworks implemented in C and C++, respectively. Other possible implementations (e.g., QP/Rust) of these requirements and features might be added in the future.

### Architectural Viewpoints

The QP Framework architecture is presented according to the international standard [\[ISO-42010:2022\] Architecture Description](#) by means of the following **architectural viewpoints** (each consisting of various *architectural views*):

- [Technology Viewpoint](#)
- [Context Viewpoint](#)
- [Resource Viewpoint](#)

### Stakeholders

This Software Architecture Specification is primarily intended for the following **stakeholders**:

- **Application Developers** who develop QP Applications based on the QP Framework,
- Software Architects,
- Quality-Assurance Engineers,
- System Engineers,
- Test Engineers, as well as
- Managers who oversee the software development.

### Concerns

This architecture specification addresses the following general **concerns** (understood here as topics of interest [\[ISO-42010:2022\]](#)):

- the applied programming paradigms and software technologies;
- the context, layering, and assignment of functionality to QP Framework and QP Application;
- the interface between QP Framework and the QP Application based on the framework as well as between QP Framework and the Operating System underlying the framework;
- main policies with respect to resource management and ownership.

## 6.2 Document Conventions

### 6.2.1 Architecture Specification UIDs

For traceability, this Software Architecture Specification uses the Unique Identifiers (UIDs) with the following structure:

```
+----- [1] Work artifact class (e.g., 'SAS' for Software Architecture Specification)
| +----- [2] Project identifier ('QP' for QP Framework or 'QA' for QP Application)
| | +----- [3] Architecture view (e.g., 'OSAL' for OS Abstraction Layer)
| |
SAS_QP_view
```

Examples: [SAS\\_QP\\_OSAL](#), [SAS\\_QP\\_OO](#)

### 6.2.2 UML Semantics

Most diagrams presented in this Software Architecture Specification conform to the established and precisely defined semantics of the Unified Modeling Language [UML2.5:17]. In case a diagram uses any non-normative" elements, the semantics of those are explained in the diagram description.

## 6.3 References

[ISO-42010:2022]	ISO/IEC/IEEE, "International Standard ISO/IEC/IEEE 4210, Software, systems and enterprise engineering - Architecture description", 2022
[DOC_SRS_QP]	<a href="#">Software Requirements Specification</a>
[DOC_SSR_QP]	<a href="#">Software Safety Requirements</a>
[DOC_SDS_QP]	<a href="#">Software Design Specification</a>
[QM-Tool:2024]	Quantum Leaps, <a href="#">QM Model-Based Design Tool↑</a>
[OO-in-C:2023]	<a href="#">Object-Oriented Programming in C↑</a> , Quantum Leaps, GitHub, 2023
[UML2.5:17]	"OMG Unified Modeling Language (OMG UML) Version 2.5.1", document formal/2017-12-05, OMG 2017

## 6.4 Technology Viewpoint

The purpose of the **Technology Viewpoint** is to classify the elements of the problem and the solution around well-known software paradigms and technologies.

### 6.4.1 Views

The following traceable architectural views explain the applied programming paradigms and technologies:

#### 6.4.1.1 SAS\_QP\_OO

Object-oriented view

##### Description

This architecture specification of the QP Framework assumes an **object-oriented** view, utilizing the concepts such as:

- class
- inheritance
- polymorphism
- object-oriented design patterns
- UML notation and semantics

##### Concerns

- programming paradigm
- software organization
- software interfaces
- information hiding

##### Anti-Concerns

The object-oriented view does not impose the choice of object-oriented programming language. In traditionally procedural languages, such as C, object-oriented concepts can be applied as design patterns. A set of such patterns for the C programming language is described in the reference [Object-Oriented Programming in C \[OO-in-C:23\]](#).

---

#### 6.4.1.2 SAS\_QP\_EDA

Event-driven architecture view

##### Description

The QP Framework is a reusable **event-driven architecture** (EDA), where all interactions are triggered by events, which are delivered asynchronously and are processed *without blocking*.

##### Concerns

- programming paradigm
- software organization
- real-time processing

#### Anti-Concerns

The EDA architecture underlying QP Framework is in contrast to the sequential architectures, such as a traditional Real-Time Operating System (RTOS), where the application explicitly awaits events in-line in the hard-coded *blocking* calls (e.g., time-delay, semaphore, etc.)

---

#### 6.4.1.3 SAS\_QP\_AF

##### Application framework view

###### Description

QP Framework is an **application framework** defined as a *reusable* architecture for a specific problem domain (embedded, real-time systems in case of QP Framework).

###### Concerns

Application framework has the following key characteristics:

- **inversion of control**: In a framework, unlike in a toolkit or in a standard end-application, the overall program's flow of control is not dictated by the application, but *by the framework* (the *framework* calls the application, not the other way around).
- **extensibility**: application developers can *extend* the framework by deriving and specializing base classes provided inside the framework.
- **non-modifiable framework code**: the framework code is not supposed to be modified, while accepting application-implemented extensions. In other words, developers can *extend* the framework, but *cannot modify* the framework.

###### Anti-concerns

Software organized as a framework significantly differs from software organized as a "toolkit", such as a traditional Real-Time Operating System (RTOS).

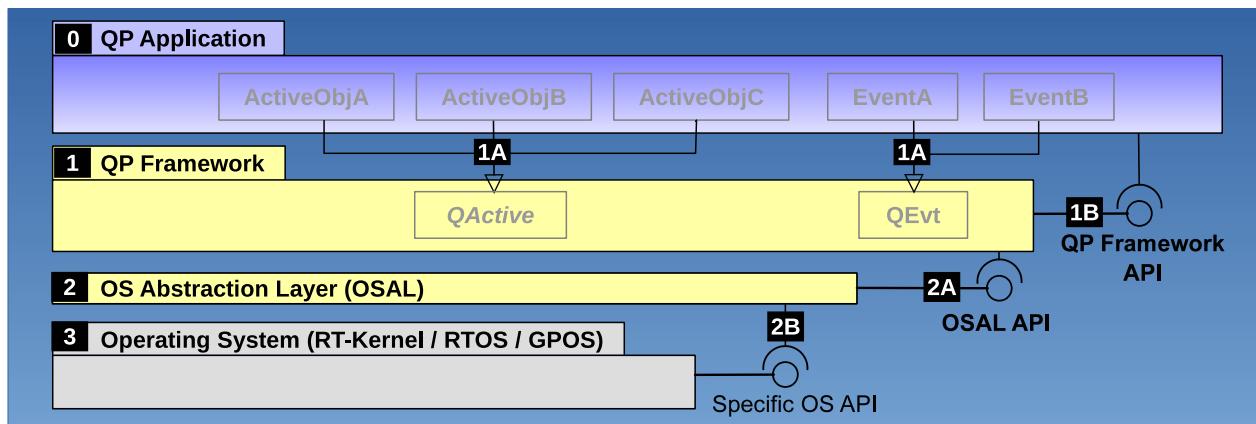
---

## 6.5 Context Viewpoint

The **Context Viewpoint** provides a "black box" perspective on the architecture subject (QP Framework in this case) in the surrounding context (QP Application and underlying Operating System). The views within this viewpoint identify the stratification of services (**layers**) and the **interfaces** between the layers. The Context Viewpoint frames the following **concerns**:

- context of the architecture subject
- functional decomposition of the system into layers
- interfaces between the layers

The **model kind** used to illustrate the Context Viewpoint is the UML package diagram [Figure SAS-CTXT \(static view\)](#). It shows the QP Framework in the context of other software elements. The architecture is **layered** with clearly defined **interfaces**. As required in a truly *layered* architecture, the dependencies between layers point in one direction only (from higher-level to lower-level layers).



*Figure SAS-CTXT: Context of use with layers and QP Framework interfaces.*

### Remarks

In compliance with the UML conventions, the diagrams shown in this section intentionally **omit** many elements of QP Framework or QP Applications. The presented views contain only selected elements necessary to explain the high-level structure, but without cluttering the diagrams with irrelevant details.

### Note

QP Framework classes, their responsibilities, and relationships are described in the [Structural View in Software Design Specification](#).

### 6.5.1 Layer View

The Layers View explains the QP Framework context in terms of distinct layers and their relationships. The objectives of the Layer View are:

- simplification of the interactions inside the system because a software layer can only interact with the layer immediately below.
- promote modularity by encapsulating functionalities within layers. This makes it easier to manage, test, and update individual parts of the system without affecting others.
- enhance scalability by allowing layers to be scaled independently.

- encourage reusability of components within and across projects. For example well-defined [QP Application layer](#) can be reused in different contexts (e.g., with different operating system or different target hardware), saving development time and effort.
- improve maintainability by isolating changes to specific layers. This reduces the risk of unintended side effects when modifying the system.
- facilitate interoperability by allowing layers to interact only through the well defined interfaces (see [Interface View](#)).
- provide flexibility to allow layers to be modified or replaced without impacting the entire system.
- enhance security by isolating sensitive operations within specific layers. This limits the exposure of critical functionalities and data.

### 6.5.1.1 SAS\_QP\_APP

QP Application layer

Description

**QP Application** layer ([Figure SAS-CTX \[0\]](#)) consists of specific [Active Object](#) classes and [Event](#) classes. The QP Application classes *inherit and specialize* the base classes from QP Framework. QP Application also uses services provided by the [QP Framework API](#).

Note

The specific classes inside the QP Application layer, such as `ActiveObjA` or `EventA`, are derived from the QP Framework base classes, which are described in the [Structural View in Software Design Specification](#).

---

### 6.5.1.2 SAS\_QP\_FRM

QP Framework layer

Description

**QP Framework** layer ([Figure SAS-CTX \[1\]](#)) provides the base classes to be specialized by the QP Application. QP Framework also provides the runtime environment to execute QP Application. Finally QP Framework provides the [QP Framework API](#), which are services that the specific Active Objects can call directly without subclassing and specializing the base classes.

Note

The QP Framework base classes, such as `QActive` and `QEvt` are described in the [Structural View in Software Design Specification](#).

Backward Traceability

- [SRS\\_QP\\_NF\\_03](#): *QP Framework shall be modular.*
- [SRS\\_QP\\_NF\\_04](#): *QP Framework shall be layered.*

## Forward Traceability (truncated to 1 level(s))

- [SDS\\_QP\\_QEP](#): *QEP Event Processor*
- [SDS\\_QP\\_QF](#): *QF Active Object Framework*
- [SDS\\_QP\\_QActive](#): *QActive Active Object class.*
- [SDS\\_QP\\_QMctive](#): *QMctive Active Object class.*
- [SDS\\_QP\\_QTimeEvt](#): *QTimeEvt time event class.*
- [SDS\\_QA\\_QHsm\\_decl](#): *Declaring QHsm subclass.*
- [SDS\\_QA\\_QHsm\\_top\\_init](#): *Implementing the top-most initial pseudo-state.*
- [SDS\\_QA\\_QHsm\\_state](#): *Implementing a state and its nesting.*
- [SDS\\_QA\\_QHsm\\_entry](#): *Implementing state entry action.*
- [SDS\\_QA\\_QHsm\\_exit](#): *Implementing state exit action.*
- [SDS\\_QA\\_QHsm\\_nest\\_init](#): *Implementing nested initial transition.*
- [SDS\\_QA\\_QHsm\\_tran](#): *Implementing state transition.*
- [SDS\\_QA\\_QHsm\\_intern](#): *Implementing internal transition.*
- [SDS\\_QA\\_QHsm\\_choice](#): *Implementing choice point and guard conditions.*
- [SDS\\_QA\\_QHsm\\_hist](#): *Implementing state history.*
- [SDS\\_QA\\_QHsm\\_hist\\_tran](#): *Implementing transition to history.*
- [SDS\\_QA\\_QMsm\\_decl](#): *Declaring QMsm subclass.*
- [SDS\\_QA\\_QMsm\\_top\\_init](#): *Implementing the top-most initial pseudo-state.*
- [SDS\\_QA\\_QMsm\\_state](#): *Implementing a state and its nesting.*
- [SDS\\_QA\\_QMsm\\_entry](#): *Implementing state entry action.*
- [SDS\\_QA\\_QMsm\\_exit](#): *Implementing state exit action.*
- [SDS\\_QA\\_QMsm\\_nest\\_init](#): *Implementing nested initial transition.*
- [SDS\\_QA\\_QMsm\\_tran](#): *Implementing state transition.*
- [SDS\\_QA\\_QMsm\\_intern](#): *Implementing internal transition.*
- [SDS\\_QA\\_QMsm\\_choice](#): *Implementing choice point and guard conditions.*
- [SDS\\_QA\\_QMsm\\_hist](#): *Implementing state history.*
- [SDS\\_QA\\_QMsm\\_hist\\_tran](#): *Implementing transition to history.*

**6.5.1.3 SAS\_QP\_OSAL**

OS abstraction layer (OSAL)

Description

**Operating System Abstraction** layer ([Figure SAS-CTX \[2\]](#)) insulates the QP Framework from the specifics of the underlying [Operating System \(OS\)](#). The OS Abstraction Layer provides an abstracted OSAL API that QP Framework *uses* internally. The OS Abstraction Layer also *implements* the OSAL API for the underlying [OS layer](#), which is called a specific [QP port](#).

## Backward Traceability

- [SRS\\_QP\\_NF\\_03](#): *QP Framework shall be modular.*
- [SRS\\_QP\\_NF\\_04](#): *QP Framework shall be layered.*

## Forward Traceability (truncated to 1 level(s))

#### 6.5.1.4 SAS\_QP\_OS

OS layer

Description

**Operating System** layer ([Figure SAS-CTX \[3\]](#)) provides the **execution context** for Active Objects as well as other services used by the **OS abstraction layer (OSAL)**. The OS Layer might consist of one of the real-time kernels provided in the QP Framework (e.g., **QV**, **QK**, or **QXK**) or might be a 3rd-party RTOS/OS.

Backward Traceability

- [SRS\\_QP\\_NF\\_03](#): *QP Framework shall be modular.*
- [SRS\\_QP\\_NF\\_04](#): *QP Framework shall be layered.*
- SRS\_QP\_AO\_80

Forward Traceability (truncated to 1 level(s))

## 6.5.2 Interface View

The Interface View defines how different layers of a system interact with each other. The objectives are:

- define clear layer boundaries, ensuring each layer's responsibilities and interactions are well-defined.
- ensure compatibility to allow seamless communication and data exchange between layers.
- promote modularity to allow components to be developed, tested, and maintained independently, which enhances flexibility and scalability.
- facilitate integration to provide a blueprint for integrating different components, making it easier to assemble the system and ensure that all parts work together harmoniously.
- enhance reusability by designing interfaces that can be reused across different projects or components, reducing development time and effort.
- support interoperability to ensure that the system can interact with other systems or external components, which is especially important in real-time embedded systems where integration with hardware and other software is common.
- maintain consistency in how components interact, which helps in maintaining the integrity and reliability of the system.

#### 6.5.2.1 SAS\_QP\_CLS

QP Framework base classes

Description

**Base classes in QP Framework** ([Figure SAS-CTX \[1A\]](#)) provide an interface for QP Application to inherit and specialize.

### Backward Traceability

- [SRS\\_QP\\_EVT\\_00](#): QP Framework shall provide Event abstraction to QP Application
- [SRS\\_QP\\_SM\\_00](#): QP Framework shall provide support for hierarchical state machines both for Active Objects and for passive event-driven objects in the Application
- [SRS\\_QP\\_AO\\_00](#): QP Framework shall provide the Active Object abstraction to QP Application

### Forward Traceability (truncated to 1 level(s))

---

## 6.5.2.2 SAS\_QP\_API

### QP Framework API

#### Description

**QP Framework API** ([Figure SAS-CTX \[1B\]](#)) provides an interface to various QP Framework services without the need to subclass and specialize the QP Framework base classes. Examples of such services include: direct event posting, subscribing and publishing events, or arming and disarming time events.

### Backward Traceability

- [SRS\\_QP\\_EDM\\_00](#): QP Framework shall provide direct event posting to Active Object instances based on the FIFO policy
- [SRS\\_QP\\_EDM\\_01](#): QP Framework shall provide direct event self-posting to Active Object instances based on the LIFO policy
- [SRS\\_QP\\_EDM\\_50](#): QP Framework shall provide publish-subscribe event delivery mechanism
- [SRS\\_QP\\_EDM\\_52](#): QP Framework shall allow Active Object instances to subscribe to a given event signal at run-time.
- [SRS\\_QP\\_EDM\\_53](#): QP Framework shall allow Active Object instances to unsubscribe from a subscribed event signal at run-time.
- [SRS\\_QP\\_EDM\\_54](#): QP Framework shall allow Active Object instances to unsubscribe from all subscribed event signals at run-time.
- [SRS\\_QP\\_EMM\\_40](#): QP Framework shall provide a method of explicitly recycling mutable events.
- [SRS\\_QP\\_TM\\_00](#): QP Framework shall support Time Events.
- [SRS\\_QP\\_TM\\_11](#): For every clock rate QP Framework shall provide a clock-tick processing operation that QP Application must call periodically to service the armed Time Events.
- [SRS\\_QP\\_TM\\_20](#): QP Framework shall provide Time Event initialization.
- [SRS\\_QP\\_TM\\_21](#): QP Framework shall allow a Time Event to be armed both for one-shot and periodic expiry.
- [SRS\\_QP\\_TM\\_22](#): QP Framework shall allow a Time Event to be explicitly disarmed.
- [SRS\\_QP\\_TM\\_23](#): QP Framework shall allow a Time Event to be rearmed.
- [SRS\\_QP\\_TM\\_30](#): QP Framework shall provide operation to check whether any Time Events are armed for a given tick rate.

### Forward Traceability (truncated to 1 level(s))

---

### 6.5.2.3 SAS\_OSAL\_API

OSAL API

Description

**OSAL API** ([Figure SAS-CTX \[2A\]](#)) is an abstract interface between QP Framework and the underlying OS to insulate the framework from the OS.

Backward Traceability

- [SRS\\_QP\\_NF\\_40](#): *QP Framework shall be portable to 16-/32-/64-bit CPUs.*
- [SRS\\_QP\\_NF\\_41](#): *QP Framework shall be portable to a wide range of real-time kernels.*

Forward Traceability (truncated to 1 level(s))

---

### 6.5.2.4 SAS\_OS\_API

OS API

Description

**OS API** ([Figure SAS-CTX \[2B\]](#)) is the specific Operating System API. This interface is not part of QP Framework, but is used in every specific [OS Abstraction layer \(OSAL\)](#).

Forward Traceability (truncated to 1 level(s))

---

## 6.6 Resource Viewpoint

The **Resource Viewpoint** covers resource management, resource ownership, and resource utilization. This architectural viewpoint frames the following concerns:

- resource management policy
- resource ownership
- resource utilization

### 6.6.1 Views

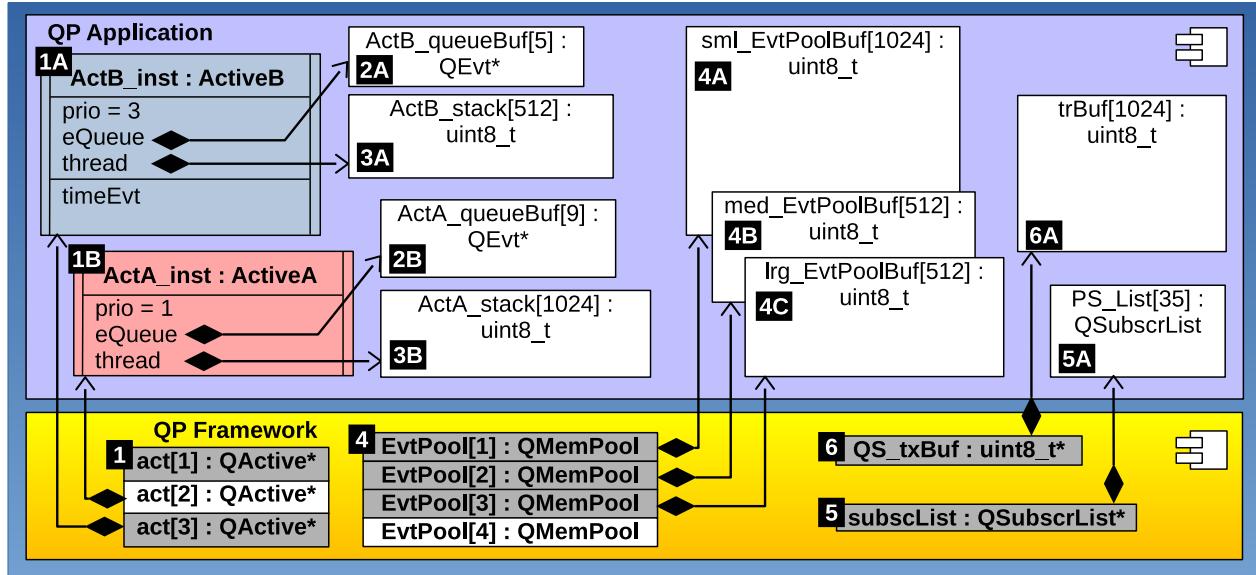
The following traceable architectural views explain the memory management policies, event management, and other resources:

#### 6.6.1.1 SAS\_QP\_MEM

General memory allocation policy

## Model Kind

The composite structure diagram in [Figure SAS-MEM](#) shows a snapshot of memory after the startup of QP Application. This *dynamic* view illustrates the **memory allocation policy** in QP Framework.



*Figure SAS-MEM: Memory allocation in QP Framework and QP Application.*

QP Framework allocates (statically) only the following kinds of objects, whose sizes are known at compile-time:

- An array of pointers to Active Objects managed by the framework (array `act []` in [Figure SAS-MEM \[1\]](#))
- An array of event-pools of different block-sizes (array `EvtPool []` in [Figure SAS-MEM \[4\]](#))
- A pointer to "subscriber list" for publish-subscribe (pointer `subscrList []` in [Figure SAS-MEM \[5\]](#))
- A pointer to a software tracing buffer (pointer `QS_txBuf []` in [Figure SAS-MEM \[6\]](#))
- other, similar objects known at compile-time

QP Application is responsible for allocation of the following objects:

- **Active Object instances** (see `ActB_inst` and `ActA_inst` in [Figure SAS-MEM \[1A,1B\]](#)). Note that the sizes of Active Objects depend on the attributes added during subclassing and customizing the `QActive` or `QMActive` base classes.
- **event-queue buffers** for Active Objects ([Figure SAS-MEM \[2A,2B\]](#)). Note that the event-queue control structure (`eQueue` attribute) is known at compile-time and is part of the `QActive` base class. However, the size of the queue *buffer* depends on the needs of the specific Active Object and therefore is allocated by the application.
- **thread stacks**, if needed ([Figure SAS-MEM \[3A,3B\]](#)). Note that the thread control structure (`thread` attribute) is known at compile-time and is part of the `QActive` base class. However, the size of the thread *stack* depends on the needs of the specific Active Object and therefore is allocated by the application. (NOTE: the thread stack allocation policy depends on the real-time kernel chosen for QP Framework. Some kernels don't need per-thread stacks at all. Some kernels need the stacks, but pre-allocate them internally. In those cases QP Application might not need to allocate the per-stack stacks);
- **event-pool buffers**, if used ([Figure SAS-MEM \[4A,4B,4C\]](#)). Event pools are part of the QP Framework `event memory management` feature. QP Application can allocate the up to compile-time-configured number of **event pools** of different block-sizes. The event-pool control structure is known at compile-time (see array `EvtPool []` in [Figure SAS-MEM \[4\]](#)), but the size of the event-pool *buffers* depends on the application needs and therefore is allocated by the application.

- **publish-subscribe lists** (Figure SAS-MEM [5A]). If the **publish-subscribe** feature is used, QP Application needs to provide the subscribe-list buffer, whose size depends on the number of events that can be published.
- **tracing-buffers** (Figure SAS-MEM [6A]). If the **software tracing** feature is used, QP Application needs to provide the tracing buffer, whose size depends on the volume of software tracing data produced.

#### Note

QP Framework's memory allocation policy leaves it up to QP Application how it allocates its objects. In particular, this **freedom** allows QP Application to use any allocation scheme, including allocating the object statically (preferred) or even from the standard heap (not recommended). Also, QP Application might choose the **memory type** for allocating various objects. For example, QP Application might choose to allocate event-pools in special memory regions (e.g., shared memory).

#### Backward Traceability

- **SRS\_QP\_NF\_13:** All services provided by QP must NOT use the standard heap.

#### Forward Traceability (truncated to 1 level(s))

- **SAS\_QP\_FUSA\_02:** Error detecting codes mechanisms to protect critical variables.
- **SDS\_QA\_START:** QA Application startup sequence
- **SDS\_QP\_MELC:** Mutable event ownership rules

### 6.6.1.2 SAS\_QP\_EMM

#### Event memory management policy

##### Model Kind

The composite structure diagram in Figure SAS-EMM illustrates how QP Framework manages the event memory. The diagram shows a snapshot in time including an Active Object's event-queue buffer as well as the events in memory.

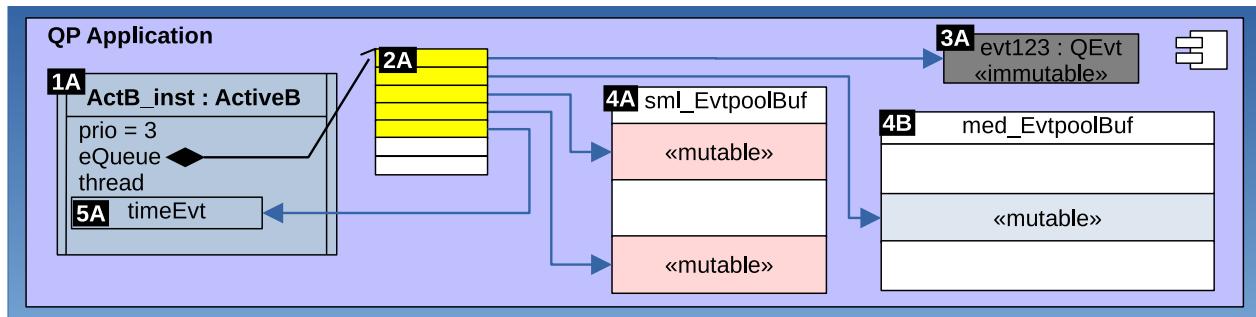


Figure SAS-EMM: Event memory management in QP Framework.

### Description

In order to uniformly treat all types of events, the Active Object [event-queue](#) stores only **pointers** to [event instances](#). These pointers can point to immutable and mutable events located in event-pools as well as Time Events located typically inside Active Objects. QP Framework manages the memory for event-queues, queue-buffers, and event-instances as illustrated in a scenario shown in [Figure SAS-EMM](#) and described below:

[Figure SAS-EMM \[1A\]](#): Active Object instance allocated by QP Application has an internal event-queue (the control structure);

[Figure SAS-EMM \[2A\]](#): The external queue-buffer (allocated by QP Application) holds *pointers* to events. The used queue entries are highlighted.

[Figure SAS-EMM \[3A\]](#): One of the event-queue entries points to an [immutable event](#) instance (allocated in ROM).

[Figure SAS-EMM \[4A\]](#): The event-pool buffer holds dynamically allocated [mutable events](#). The used pool entries are highlighted. Two of the event-queue entries point to events in the "small" pool (event-pool with a "small" block-size).

[Figure SAS-EMM \[4B\]](#): Another event-pool buffer holds dynamically allocated [mutable events](#). The used pool entries are highlighted. One of the event-queue entries points to an event in the "medium" pool (event-pool with a "medium" block-size).

[Figure SAS-EMM \[5A\]](#): One of the event-queue entries points to an [Time Event](#) instance (allocated inside the Active Object instance).

### Backward Traceability

- [SRS\\_QP\\_EMM\\_00](#): *QP Framework shall support immutable events.*
- [SRS\\_QP\\_EMM\\_10](#): *QP Framework shall support mutable events.*
- [SRS\\_QP\\_TM\\_00](#): *QP Framework shall support Time Events.*

### Forward Traceability (truncated to 1 level(s))

- [SAS\\_QP\\_FUSA\\_02](#): *Error detecting codes mechanisms to protect critical variables.*
- [SDS\\_QA\\_START](#): *QA Application startup sequence*
- [SDS\\_QP\\_POST](#): *QP event posting sequence*
- [SDS\\_QP\\_PUB](#): *QP event publishing sequence*
- [SDS\\_QP\\_MELC](#): *Mutable event ownership rules*

## Chapter 7

# Software Design Specification



This *Software Design Specification* is part of the [SafeQP Certification Kit](#), but applies to the whole [QP Framework family](#). This document is the best source of information about the *design*, and internal *implementation* of QP Framework as well as *QP Applications derived from the framework*.

### Revision History

QP version	Document revision	Date (YYYY-MM-DD)	By	Description
7.3.4	A	2024-05-05	MMS	Initial release for IEC-61508 SIL-3 and IEC-62304 Class-C.
7.4.0	B	2024-07-30	MMS	Updated for QP 7.4.0.
8.0.0	C	2024-10-18	MMS	Updated for QP 8.0.0.

## 7.1 About This Document

### 7.1.1 DOC\_SDS\_QP

Software Design Specification (SDS)

#### Description

This *Software Design Specification*, with Unique Identifier: [DOC\\_SDS\\_QP](#), describes the **software design** for the QP Framework that realizes the architecture specified in the [QP Software Architecture Specification \(DOC\\_SAS\\_QP\)](#), requirements specified in the [QP Software Requirements Specification \(DOC\\_SRS\\_QP\)](#), and QP Software Safety Requirements Specification ([DOC\\_SSR\\_QP](#)). @uid{Design Viewpoints} The Software Design Specification is organized according to the international standard [\[IEEE-1016:2009\] Software Design Descriptions](#) by means of the following **design viewpoints**, each consisting

of various **design views**. The described viewpoints are followed by the traceable [Software-Design-Specifications](#), which describe and specify the relevant views.

- Structure Viewpoint
- Interaction Viewpoint
- State Dynamics Viewpoint
- Time Viewpoint
- Algorithm Viewpoint
- Interface Viewpoint

### Stakeholders

This Software Design Specification is primarily intended for the following **stakeholders**:

- *Application Developers* who develop QP Applications based on the QP Framework,
- System Engineers,
- Quality-Assurance Engineers,
- Test Engineers, as well as
- Managers who oversee the software development.

### Concerns

This design specification addresses the following **concerns** (understood here as areas of interest with respect to a software design [\[IEEE-1016:2009\]](#)):

- logical structure of QP Framework and QP Applications
- interaction by means of events
- state dynamics by means of hierarchical state machines
- time management by means of Time Events
- algorithms used to implement various functions
- interface between QP Framework and the Operating System underlying the framework;
- safe programming techniques

## 7.2 Document Conventions

### 7.2.1 Software-Design-Specification UIDs

For traceability, this Software Design Specification uses the Unique Identifiers (UIDs) with the following structure:

```
+----- [1] Work artifact class (e.g., 'SDS' for Software Design Specification)
| +----- [2] Project identifier ('QP' for QP Framework or 'QA' for QP Application)
| | +----- [3] Design view (e.g., 'OSAL' for OS Abstraction Layer)
| |
SDS_QP_view
```

Examples: [SDS\\_QP\\_QHsm](#), [SDS\\_QA\\_START](#)

## 7.3 References

[IEEE-1016:2009]	IEEE Computer Society, "IEEE Standard for Information Technology - Systems Design - Software Design Descriptions", 2009
[ISO-42010:2011]	ISO/IEC/IEEE, "International Standard ISO/IEC/IEEE 4210, Systems and software engineering - Architecture description", 2011
[IEC 61508-1:2010]	IEC 61508-1:2010 Functional safety of electrical/electronic/programmable electronic safety-related systems- Part 1: General requirements
[IEC 61508-2:2010]	IEC 61508-2:2010 Functional safety of electrical/electronic/programmable electronic safety-related systems- Part 2: Requirements for E/E/PE safety-related systems
[IEC 61508-3:2010]	IEC 61508-3:2010 Functional safety of electrical/electronic/programmable electronic safety-related systems- Part 3: Software requirements
[IEC 61508-7:2010]	IEC 61508-7:2010 Functional safety of electrical/electronic/programmable electronic safety-related systems- Part 7: Overview of techniques and measures
[ISO 26262-1:2018]	ISO 26262-1:2018(en) Road vehicles — Functional safety — Part 1: Vocabulary. International Standardization Organization.
[ISO 26262-2:2018]	ISO 26262-2:2018(en) Road vehicles - Functional safety - Part 2: Management of functional safety. International Standardization Organization.
[ISO 26262-3:2018]	ISO 26262-3:2018(en) Road vehicles - Functional safety - Part 3: Concept phase. International Standardization Organization.
[ISO 26262-4:2018]	ISO 26262-3:2018(en) Road vehicles - Functional safety - Part 4: Definitions and abbreviations. International Standardization Organization.
[ISO 26262-6:2018]	ISO 26262-6:2018(en) Road vehicles - Functional safety - Part 6: Product development at the software level. International Standardization Organization.
[ISO 26262-8:2018]	ISO 26262-8:2018(en) Road vehicles - Functional safety - Part 8: Supporting processes. International Standardization Organization.
[DOC_SRS_QP]	<a href="#">Software Requirements Specification</a>
[DOC_SAS_QP]	<a href="#">Software Architecture Specification</a>
[PSiCC:02]	Miro Samek, <i>Practical Statecharts in C/C++</i> , CMP Books 2002. <a href="https://www.state-machine.com/psiccc">https://www.state-machine.com/psiccc</a>
[PSiCC2:08]	Miro Samek, <i>Practical UML Statecharts in C/C++, 2nd Edition</i> , Newnes 2008. <a href="https://www.state-machine.com/psiccc2">https://www.state-machine.com/psiccc2</a>
[OO-in-C:23]	<a href="#">Object-Oriented Programming in C↑</a> , Quantum Leaps, GitHub, 2023
[GoF:94]	Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, <i>Design Patterns: Elements of Reusable Object-Oriented Software</i> , Addison-Wesley 1994.
[UML2.5:17]	"OMG Unified Modeling Language (OMG UML) Version 2.5.1", document formal/2017-12-05, OMG 2017
[UML-Dist:04]	Martin Fowler, "UML Distilled, 3rd Edition", Addison-Wesley, 2004

## 7.4 Structure Viewpoint

**Structure Viewpoint** elaborates the base classes and interfaces identified in the architecture [Context Viewpoint](#) with their design details and structural static relationships. The Structure Design Viewpoint also provides examples of using the QP Framework classes in QP Applications and is used to address the following concerns:

- packages and classes
- interfaces
- functionality and responsibilities
- data structures
- control flow
- concurrency and parallelism
- resource management

The **model kind** used to illustrate the Structure Viewpoint is the UML class diagram shown in [Figure SDS-CLS](#) (static view). It depicts base classes comprising QP Framework and their specializations in the QP Application shown at the bottom of [Figure SDS-CLS](#).

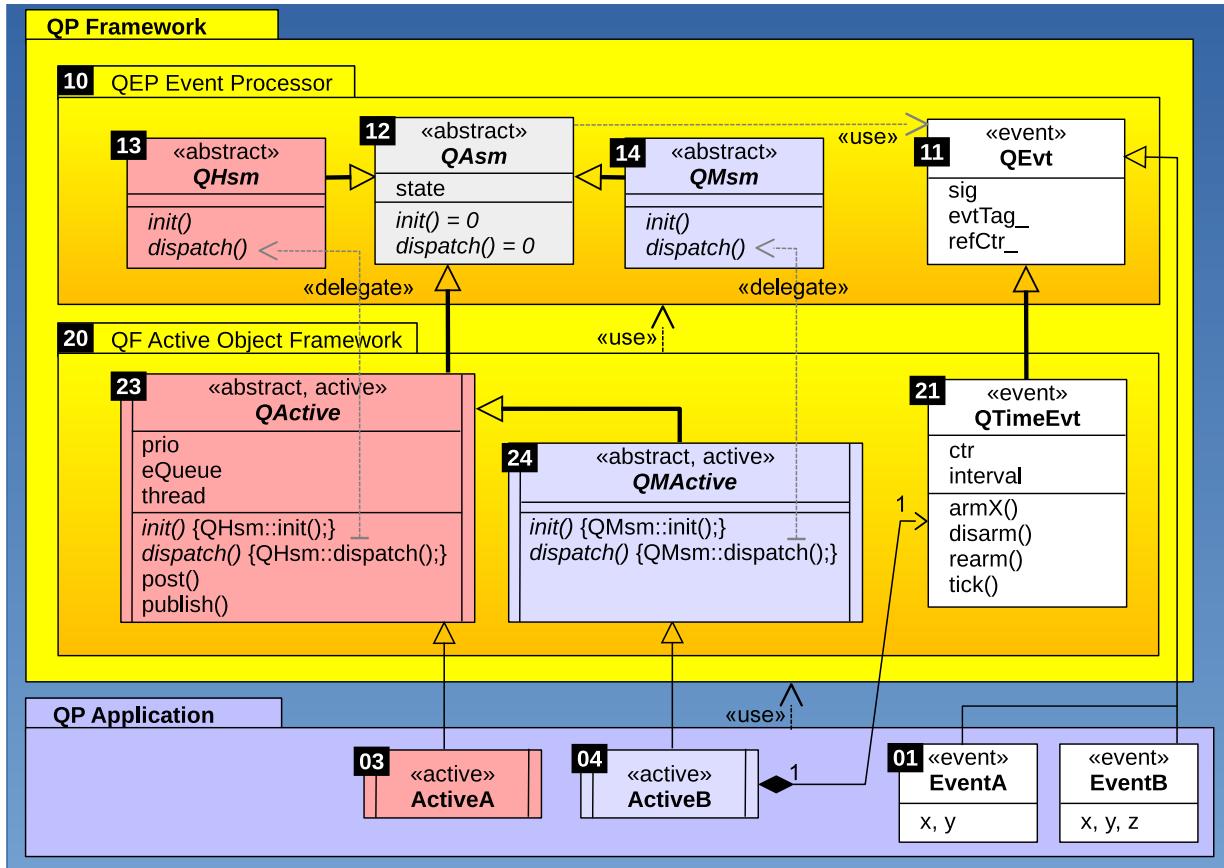


Figure SDS-CLS: Core classes in QP Framework and their relation to QP Application.

### 7.4.1 Sub-Layer View

The Sub-Layer View view in the Structure Viewpoint focuses on the internal layering of the QP Framework, which has the following objectives:

- promote modularity.
- separation of concerns into *passive* and *active* elements.
- enable QP Application to use of only the [passive layer](#) without pulling in any of the Active Object layer.

#### 7.4.1.1 SDS\_QP\_QEP

QEP Event Processor

##### Description

*QEP Event Processor* sub-layer of QP Framework (see [Figure SDS-CLS \[10\]](#)) implements [Events](#) and *passive State Machines*.

##### Note

QEP Event Processor is intentionally separated out to allow QP Applications to directly use just the **passive** state machines (without [Active Objects](#)). QP Applications that use just the QEP Event Processor can still benefit from the [QM modeling tool↑](#), which can generate code that requires only passive state machines.

##### Backward Traceability

- [SAS\\_QP\\_FRM](#): *QP Framework layer*
- [SRS\\_QP\\_NF\\_04](#): *QP Framework shall be layered.*

##### Forward Traceability (truncated to 1 level(s))

- [QAsm](#): *Abstract State Machine class (state machine interface)*
- [QHsm](#): *Hierarchical State Machine class (QHsm-style state machine implementation strategy)*
- [QMsm](#): *Hierarchical State Machine class (QMsm-style state machine implementation strategy)*

---

#### 7.4.1.2 SDS\_QP\_QF

QF Active Object Framework

##### Description

*QF Active Object Framework* sub-layer of QP Framework (see [Figure SDS-CLS \[20\]](#)) implements [Active Objects](#) and [Time Events](#).

##### Note

QF Active Object Framework requires the [QEP Event Processor](#) to provide events and implement the state machine behavior of Active Objects.

##### Backward Traceability

- [SAS\\_QP\\_FRM](#): *QP Framework layer*
- [SRS\\_QP\\_NF\\_04](#): *QP Framework shall be layered.*

#### Forward Traceability (truncated to 1 level(s))

- **QPSet:** Set of Active Objects of up to *QF\_MAX\_ACTIVE* elements
- **QSubscrList:** Subscriber List (for publish-subscribe)
- **QActive:** Active object class (based on the *QHsm* implementation strategy)
- **QActive::QActive\_getQueueMin():** This function returns the minimum of free entries of the given event queue
- **QMActive:** Active object class (based on *QMsm* implementation strategy)
- **QTicker:** "Ticker" Active Object class
- **QF::QF\_init():** QF initialization
- **QF::QF\_stop():** Invoked by the application layer to stop the QF framework and return control to the OS/Kernel (used in some QF ports)
- **QF::QF\_run():** Transfers control to QF to run the application
- **QF::QF\_onStartup():** Startup QF callback
- **QF::QF\_onCleanup():** Cleanup QF callback
- **QF::QF\_poolInit():** Event pool initialization for dynamic allocation of events
- **QF::QF\_getPoolMin():** Obtain the minimum of free entries of the given event pool
- **QF::QF\_newX\_():** Internal QF implementation of creating new mutable (dynamic) event
- **QF::QF\_gc():** Recycle a mutable (mutable) event
- **QF::QF\_newRef\_():** Internal QF implementation of creating new event reference
- **QF::QF\_deleteRef\_():** Internal QF implementation of deleting event reference
- **QF::QF\_onContextSw():** QF context switch callback used in built-in kernels (QV/QK/QXK)
- **Q\_NEW:** Allocate a mutable (dynamic) event
- **Q\_NEW\_X:** Non-asserting allocate a mutable (dynamic) event
- **Q\_NEW\_REF:** Create a new reference of the current event *e*
- **Q\_DELETE\_REF:** Delete the event reference

#### 7.4.2 Class View

The Class View in the Structure Viewpoint focuses on the following key design concerns:

- describing the *logical structure* and organization of the system, including the identification of classes and their relationships.
- defining the QP Framework *base classes* that are the basis for [derivation of QP Applications](#) from the framework.
- describing the *responsibilities* of each class to achieve high cohesion within a class and loose coupling among classes.
- defining the *inheritance hierarchies* and *polymorphic behavior* to promote reuse and flexibility.
- ensuring data *encapsulation* within classes to maintain safety, integrity, and security.

### 7.4.2.1 SDS\_QP\_QEvt

[QEvt](#) event class.

#### Description

The [QEvt](#) class (see [Figure SDS-CLS \[11\]](#)) represents both immutable and mutable events in QP Framework. It can be instantiated directly (concrete class), in which case it represents events *without parameters*. QP Applications can also inherit and extend [QEvt](#) to add custom [event parameters](#).

#### Attention

Event parameters must be included in the event instance *directly* (as opposed to being referenced by pointers or references). Therefore, the [QEvt](#) subclasses are restricted to have both "standard layout" and be "trivially copyable". In QP this means that [QEvt](#) subclasses should:

- contain **NO pointers** (including NO virtual pointer); The only exception is passing an *opaque* pointer to Active Object intended as the recipient of future events.

#### Event Attributes

The [QEvt](#) class has the following data attributes:

- [event signal](#) of type [QSignal](#) with compile-time configurable dynamic range
- event-pool number:
  - for mutable (dynamic) events event-pool number is in the range 1 .. [QF\\_MAX\\_EPOOL](#). For immutable (static) events, the event-pool number is 0.
- event reference counter with dynamic range 0 .. (2 \* [QF\\_MAX\\_ACTIVE](#)). For [mutable events](#), the reference counter stores the number of outstanding references to that event instance. For [immutable events](#) and time events (see [SDS\\_QP\\_QTimeEvt](#)), the reference counter attribute is not used for counting but instead indicates the origin of the event.

#### Backward Traceability

- [SRS\\_QP\\_EVT\\_00](#): *QP Framework shall provide Event abstraction to QP Application*
- [SRS\\_QP\\_EVT\\_20](#): *Each event instance shall contain the event Signal*
- [SRS\\_QP\\_EVT\\_30](#): *QP Framework shall allow Application to create event instances with Parameters defined by the Application*
- [SRS\\_QP\\_EVT\\_31](#): *Event abstraction may contain other data items for internal event management inside QP Framework*

#### Forward Traceability (truncated to 2 level(s))

- [QEvt](#): *Event class*
  - [QEvt::sig](#): *Signal of the event (see [Event Signal](#))*
  - [QEvt::refCtr](#)\_: *Event reference counter*.
  - [QEvt::poolNum](#)\_: *Event pool number of this event*
  - [QEvt::sig\\_dis](#): *Duplicate Inverse Storage for [QEvt::sig](#)*
  - [QEvt::poolNum\\_dis](#): *Duplicate Inverse Storage for [QEvt::poolNum](#)\_*
  - [QEvt::refCtr\\_dis](#): *Duplicate Inverse Storage for [QEvt::refCtr](#)\_*
  - [QEvt::QEvt\\_init\(\)](#): *Event without parameters initialization*

### 7.4.2.2 SDS\_QP\_QAsm

**QAsm** Abstract state machine class.

#### Description

The **QAsm** class (see [Figure SDS-CLS \[12\]](#)) is the abstract base class (ABC) for all **state machines** in QP Framework. The **QAsm** class specifies the abstract state machine interface.

#### State Machine Interface

The state machine interface defined in the **QAsm** class consists of:

- `init()` operation executes the top-most initial transition. It must be executed only once.
- `dispatch()` operation dispatches a given even to the state machine for RTC processing. It must be executed for every event.
- `isIn()` (*is in-state*) operation returns 'true' if the state machine is-in the given state. Note that being "in-a-state" means also being in all the superstates of that state.

#### State Machine Attributes

- `current state` (state variable): stores the current state of the state machine.
- temporary attribute: stores transition target during a state transition. Also used to implement the Safety Function described below.

#### Safety Function

In the stable state configuration (between the RTC steps) the subclasses of **QAsm** use the temporary attribute for the Duplicate Inverse Storage of the `QAsm::state` attribute.

#### Backward Traceability

- **SRS\_QP\_SM\_00**: *QP Framework shall provide support for hierarchical state machines both for Active Objects and for passive event-driven objects in the Application*
- **SRS\_QP\_SM\_10**: *QP Framework shall support multiple and interchangeable State Machine Implementation Strategies*
- **SRS\_QP\_SM\_25**: *All State Machine Implementation Strategies provided by QP Framework might supply a method for checking if a state machine is in a given state*

#### Forward Traceability (truncated to 2 level(s))

- **QAsm**: *Abstract State Machine class (state machine interface)*
  - `QAsm::vptr`: *Virtual pointer inherited by all QAsm subclasses (see also SAS\_QP\_OO)*
  - `QAsm::temp`: *Temporary storage for target/act-table etc.*
  - `QAsm::QAsm_ctor()`: *Constructor of the QAsm base class*

---

### 7.4.2.3 SDS\_QP\_QHsm

**QHsm** State machine class.

#### Description

The `QHsm` class (see [Figure SDS-CLS \[13\]](#)) derived from `QAsm` implements the **state machine interface** `init()` and `dispatch()` according to the [State Machine Implementation Strategy optimized for manual coding](#). `QHsm` provides support for hierarchical nesting of states, entry/exit actions, initial transitions, and transitions to history in any composite state. This class is designed for ease of manual coding, but it is also supported by the [QM modeling and code-generation tool↑](#).

#### Safety Function

The `QHsm` class implements the Fixed Loop Bound technique for all not explicitly terminated loops. This prevents malformed [state machines specifications](#) (from the QP Application layer) to "hang" or crash the event processor.

#### Backward Traceability

- [SRS\\_QP\\_SM\\_20](#): *QP Framework shall provide a State Machine Implementation Strategy optimized for "manual coding"*
- [SRS\\_QP\\_SM\\_22](#): *All State Machine Implementation Strategies provided by QP Framework shall be bidirectionally traceable*
- [SRS\\_QP\\_SM\\_23](#): *QP Framework shall ensure that the current event does not change and is accessible to the state machine implementation over the entire RTC step.*
- [SRS\\_QP\\_SM\\_24](#): *All State Machine Implementation Strategies provided by QP shall allow Applications to easily access the instance variables associated with a given state machine object*
- [SRS\\_QP\\_SM\\_30](#): *All State Machine Implementation Strategies provided by QP Framework shall support hierarchical state machines with features specified in the sub-requirements [SRS\\_QP\\_SM\\_3x](#)*
- [SRS\\_QP\\_SM\\_31](#): *All State Machine Implementation Strategies provided by QP Framework shall support states capable of holding hierarchically nested substates*
- [SRS\\_QP\\_SM\\_32](#): *All State Machine Implementation Strategies provided by QP Framework shall support entry actions to states*
- [SRS\\_QP\\_SM\\_33](#): *All State Machine Implementation Strategies provided by QP Framework shall support exit actions from states*
- [SRS\\_QP\\_SM\\_34](#): *All State Machine Implementation Strategies provided by QP Framework shall support nested initial transitions in composite states*
- [SRS\\_QP\\_SM\\_35](#): *All State Machine Implementation Strategies provided by QP Framework shall support transitions between states at any level of nesting*
- [SRS\\_QP\\_SM\\_36](#): *All State Machine Implementation Strategies provided by QP Framework shall support internal transitions in states*
- [SRS\\_QP\\_SM\\_37](#): *All State Machine Implementation Strategies provided by QP Framework shall support guard conditions to be attached to regular and internal transitions*
- [SRS\\_QP\\_SM\\_38](#): *All State Machine Implementation Strategies provided by QP Framework shall support top-most initial transition that shall be explicitly triggered independently from instantiation of the state machine object*
- [SRS\\_QP\\_SM\\_39](#): *All State Machine Implementation Strategies provided by QP Framework should support transitions to history. Both shallow and deep histories shall be supported*
- [SRS\\_QP\\_SM\\_40](#): *State Machine Implementation Strategies provided by QP Framework might supply the top-state*

### Forward Traceability (truncated to 2 level(s))

- **QHsm:** *Hierarchical State Machine class (QHsm-style state machine implementation strategy)*
    - **QHsm::QHsm\_ctor()**: *Constructor of the QHsm base class*
    - **QHsm::QHsm\_init\_()**: *Implementation of the top-most initial transition in QHsm*
    - **QHsm::QHsm\_dispatch\_()**: *Implementation of dispatching events to a QHsm*
    - **QHsm::QHsm\_isIn\_()**: *Check if a given state is part of the current active state configuration*
    - **QHsm::QHsm\_state()**: *Obtain the current active state from a HSM (read only)*
    - **QHsm::QHsm\_childState()**: *Obtain the current active child state of a given parent in QHsm*
    - **QActive**: *Active object class (based on the QHsm implementation strategy)*
- 

#### 7.4.2.4 SDS\_QP\_QMsm

**QMsm** State machine class.

##### Description

The **QMsm** class ([Figure SDS-CLS \[14\]](#)) derived from **QAsm** implements the **state machine interface** `init()` and `dispatch()` according to the

[State Machine Implementation Strategy optimized for automatic code generation](#). **QMsm** provides support for all hierarchical state machine features listed in [SRS\\_QP\\_SM\\_20](#). This class is designed for automatic code generation, and **requires** the [QM modeling and code-generation tool↑](#).

##### Safety Function

The **QMsm** class implements the Fixed Loop Bound technique for all not explicitly terminated loops. This prevents malformed [state machines specifications](#) (from the QP Application layer) to "hang" or crash the event processor.

##### Backward Traceability

- [SRS\\_QP\\_SM\\_21](#): *QP Framework should provide a State Machine Implementation Strategy optimized for "automatic code generation"*
- [SRS\\_QP\\_SM\\_22](#): *All State Machine Implementation Strategies provided by QP Framework shall be bidirectionally traceable*
- [SRS\\_QP\\_SM\\_23](#): *QP Framework shall ensure that the current event does not change and is accessible to the state machine implementation over the entire RTC step.*
- [SRS\\_QP\\_SM\\_24](#): *All State Machine Implementation Strategies provided by QP shall allow Applications to easily access the instance variables associated with a given state machine object*
- [SRS\\_QP\\_SM\\_30](#): *All State Machine Implementation Strategies provided by QP Framework shall support hierarchical state machines with features specified in the sub-requirements [SRS\\_QP\\_SM\\_3x](#)*
- [SRS\\_QP\\_SM\\_31](#): *All State Machine Implementation Strategies provided by QP Framework shall support states capable of holding hierarchically nested substates*
- [SRS\\_QP\\_SM\\_32](#): *All State Machine Implementation Strategies provided by QP Framework shall support entry actions to states*
- [SRS\\_QP\\_SM\\_33](#): *All State Machine Implementation Strategies provided by QP Framework shall support exit actions from states*
- [SRS\\_QP\\_SM\\_34](#): *All State Machine Implementation Strategies provided by QP Framework shall support nested initial transitions in composite states*
- [SRS\\_QP\\_SM\\_35](#): *All State Machine Implementation Strategies provided by QP Framework shall support transitions between states at any level of nesting*

- [SRS\\_QP\\_SM\\_36](#): All State Machine Implementation Strategies provided by QP Framework shall support internal transitions in states
- [SRS\\_QP\\_SM\\_37](#): All State Machine Implementation Strategies provided by QP Framework shall support guard conditions to be attached to regular and internal transitions
- [SRS\\_QP\\_SM\\_38](#): All State Machine Implementation Strategies provided by QP Framework shall support top-most initial transition that shall be explicitly triggered independently from instantiation of the state machine object
- [SRS\\_QP\\_SM\\_39](#): All State Machine Implementation Strategies provided by QP Framework should support transitions to history. Both shallow and deep histories shall be supported
- [SRS\\_QP\\_SM\\_40](#): State Machine Implementation Strategies provided by QP Framework might supply the top-state

Forward Traceability (truncated to 2 level(s))

- [QMsm](#): Hierarchical State Machine class (QMsm-style state machine implementation strategy)
  - [QMsm::QMsm\\_init\\_\(\)](#): Implementation of the top-most initial transition in [QMsm](#)
  - [QMsm::QMsm\\_dispatch\\_\(\)](#): Implementation of dispatching events to a [QMsm](#)
  - [QMsm::QMsm\\_isIn\\_\(\)](#): Tests if a given state is part of the current active state configuration
  - [QMsm::QMsm\\_getStateHandler\\_\(\)](#): Implementation of getting the state handler in a [QMsm](#) subclass
  - [QMsm::QMsm\\_stateObj\(\)](#): Obtain the current state from a MSM (read only)
  - [QMsm::QMsm\\_childStateObj\(\)](#): Obtain the current active child state of a given parent in [QMsm](#)
  - [QMsm::QMsm\\_ctor\(\)](#): Constructor of [QMsm](#)
  - [QMsm::QMsm\\_execTatbl\\_\(\)](#): Execute transition-action table
  - [QMsm::QMsm\\_exitToTranSource\\_\(\)](#): Exit the current state up to the explicit transition source
  - [QMsm::QMsm\\_enterHistory\\_\(\)](#): Enter history of a composite state
  - [QMActive](#): Active object class (based on [QMsm](#) implementation strategy)

#### 7.4.2.5 SDS\_QP\_QActive

[QActive](#) Active Object class.

##### Description

The [QActive](#) class (Figure SDS-CLS [23]) is a base class for derivation of concrete Active Objects in the QP Application. [QActive](#) inherits [QAsm](#), which means that an Active Object in QP can be treated as a state machine with the standard interface. [QActive](#) implements this interface by **delegating** to the [QHsm](#) state machine implementation.

##### Active Object Priority

[QActive](#) owns the unique priority attribute, which must be in range 1..[QF\\_MAX\\_ACTIVE](#), inclusive.

##### Event Queue

[QActive](#) owns the event-queue attribute, whose type depends on the underlying real-time kernel. Therefore, the event-queue type is compile-time configurable. When QP is used with one of the built-in kernels ([QV](#), [QK](#), [QXK](#)), the [QActive](#) event queue is configured to be [QEQueue](#). However, when QP Framework runs on a 3rd-party RTOS, typically the RTOS' message queue is adapted as an event queue.

## Remarks

The event queue for Active Objects requires multiple-write, but only single-read capability. Also, blocking the queue is only required when event-queue is empty but not when it is full. Finally, the Active Object queue in QP Framework passes only *pointers* to events. Standard RTOS message queues are significantly more complex than required by Active Objects because they typically allow multiple-write as well as multiple-read access and often support variable-length data (not only pointer-sized data). Usually message queues also allow blocking when the queue is empty and when the queue is full, and both types of blocking can be timed out. Naturally, all this extra functionality, which you don't really need in QP Framework, comes at an extra cost in CPU and memory usage.

## Thread of Execution

`QActive` owns the execution thread attribute, whose type depends on the underlying real-time kernel. Therefore, the thread type is compile-time configurable. When QP is used with one of the built-in kernels (`QV`, `QK`, `QXK`), the `QActive` thread attribute isn't really needed and is used to hold MPU settings. However, when QP Framework runs on a 3rd-party RTOS, the `QActive` thread attribute holds the RTOS thread.

## OS Object

The "operating system object" attribute is

## Backward Traceability

- `SAS_QP_FRM`: *QP Framework layer*

## Forward Traceability (truncated to 2 level(s))

- `QActive`: *Active object class (based on the `QHsm` implementation strategy)*
  - `QActive::eQueue`: *Port-dependent event-queue type (often `QEQueue`)*
  - `QActive::thread`: *Port-dependent representation of the thread of the active object*
  - `QActive::prio`: *QF-priority [1..`QF_MAX_ACTIVE`] of this AO*
  - `QActive::pthre`: *Preemption-threshold [1..`QF_MAX_ACTIVE`] of this AO*
  - `QActive::QActive_ctor()`: *`QActive` constructor (abstract base class)*
  - `QActive::QActive_start()`: *Starts execution of an active object and registers the object with the framework*
  - `QActive::QActive_stop()`: *Stops execution of an active object and removes it from the framework's supervision*
  - `QActive::QActive_post()`: *Posts an event  $e$  directly to the event queue of the active object using the First-In-First-Out (FIFO) policy*
  - `QActive::QActive_postLIFO()`: *Posts an event  $e$  directly to the event queue of the active object using the Last-In-First-Out (LIFO) policy*
  - `QActive::QActive_get()`: *Get an event from the event queue of an active object*
  - `QActive::QActive_subscribe()`: *Subscribes for delivery of signal  $sig$  to the active object*
  - `QActive::QActive_unsubscribe()`: *Unsubscribes from the delivery of signal  $sig$  to the active object*
  - `QActive::QActive_unsubscribeAll()`: *Unsubscribes from the delivery of all signals to the active object*
  - `QActive::QActive_psInit()`: *Publish event to all subscribers of a given signal  $e \rightarrow sig$*
  - `QActive::QActive_publish()`: *Publish event to all subscribers of a given signal  $e \rightarrow sig$*
  - `QActive::QActive_defer()`: *Defer an event to a given separate event queue*
  - `QActive::QActive_evtLoop()`: *Event loop thread routine for executing an active object  $act$  (defined some in QP ports)*
  - `QActive::QActive_register()`: *Register this active object to be managed by the framework*
  - `QActive::QActive_unregister()`: *Un-register the active object from the framework*

- `QActive::QActive_subscrList_`: *Static (one per-class) pointer to all subscriber AOs for a given event signal*
  - `QACTIVE_POST`: *Invoke the direct event posting facility `QActive_post()`*
  - `QACTIVE_POST_X`: *Invoke the direct event posting facility `QActive_post()` without delivery guarantee*
  - `QACTIVE_POST_LIFO`: *Post an event to an active object using the Last-In-First-Out (LIFO) policy*
  - `QACTIVE_PUBLISH`: *Publish an event to all subscriber Active Objects*
- 

#### 7.4.2.6 SDS\_QP\_QMactive

QMactive Active Object class.

##### Description

The `QMActive` class (Figure SDS-CLS [24]) derives from `QActive` base class, so it inherits all its properties. The only difference is that, `QMActive` implements the state machine interface by **delegating** to the `QMsm` state machine implementation.

##### Backward Traceability

- `SAS_QP_FRM`: *QP Framework layer*

Forward Traceability (truncated to 2 level(s))

---

#### 7.4.2.7 SDS\_QP\_QTimeEvt

`QTimeEvt` time event class.

##### Description

The `QActive` class (Figure SDS-CLS [23]) is a base class for derivation of concrete Active Objects in the QP Application. `QActive` encapsulates the unique priority, event-queue, execution context, and state machine for the Active Object. `QActive` inherits `QAsm`, which means that an Active Object in QP can be treated as a state machine with the standard interface. `QActive` implements this interface by **delegating** to the `QHsm` state machine implementation.

##### Backward Traceability

- `SAS_QP_FRM`: *QP Framework layer*

Forward Traceability (truncated to 2 level(s))

- `QTimeEvt`: *Time Event class*
  - `QTimeEvt::next`: *Link to the next time event in the list*
  - `QTimeEvt::ctr`: *Down-counter of the time event*
  - `QTimeEvt::interval`: *Interval for periodic time event (zero for one-shot time event)*
  - `QTimeEvt::QTimeEvt_ctorX()`: *The "extended" constructor to initialize a Time Event*
  - `QTimeEvt::QTimeEvt_armX()`: *Arm a time event (extended version for one shot or periodic time event)*
  - `QTimeEvt::QTimeEvt_disarm()`: *Disarm a time event*
  - `QTimeEvt::QTimeEvt_rearm()`: *Rearm a time event*

- `(QTimeEvt::QTimeEvt_wasDisarmed())` "(QTimeEvt::QTimeEvt\_wasDisarmed()): Check the "was disarmed" status of a time event
  - `QTimeEvt::QTimeEvt_currCtr()`: Get the current value of the down-counter of a time event
  - `QTimeEvt::QTimeEvt_tick_()`: Processes all armed time events at every clock tick
  - `QTimeEvt::QTimeEvt_tick1_()`: Processes one clock tick for QUTest
  - `QTimeEvt::QTimeEvt_noActive()`: Check if any time events are active at a given clock tick rate
  - `QTimeEvt::QTimeEvt_timeEvtHead_`: Static array of heads of linked lists of time events (one for every clock tick rate)
  - `QTIMEEVNT_TICK_X`: Invoke the system clock tick processing `QTimeEvt_tick_()`
-

## 7.5 Interaction Viewpoint

The **Interaction Design Viewpoint** provides an analysis of interactions within QP Framework elements. This design viewpoint frames the following concerns:

- behavior of the system under various scenarios of use
- interactions in terms of events exchanged among Active Objects
- required sequences of operations
- control flow
- concurrency and parallelism
- resource management

### 7.5.1 Behavior View

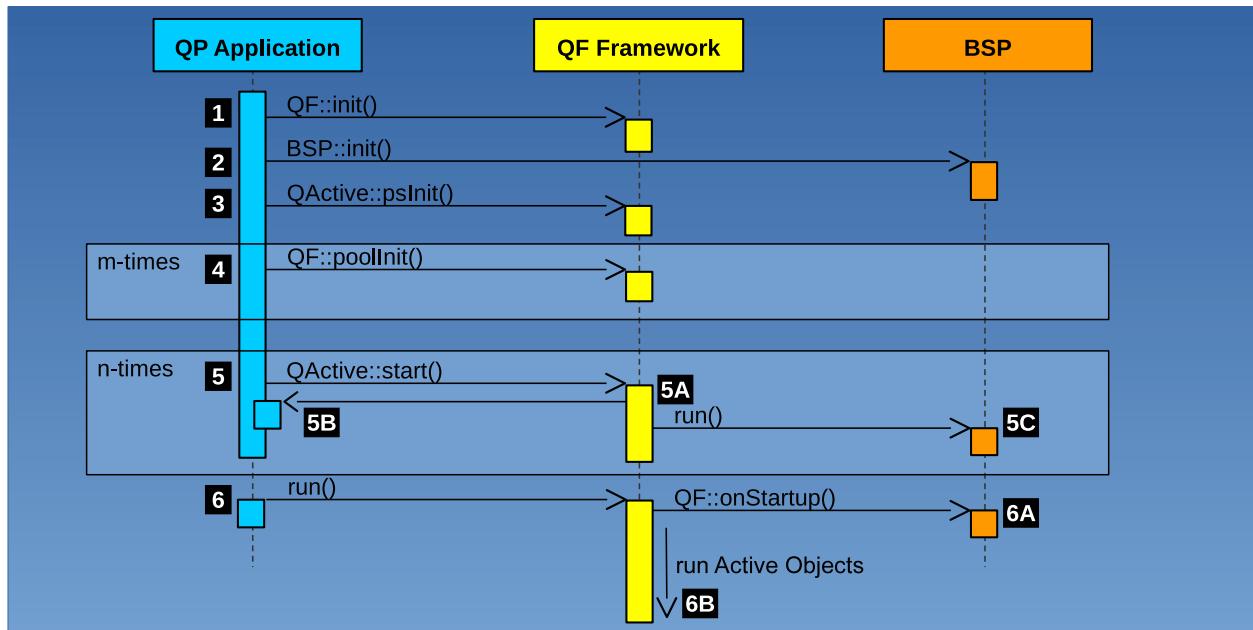
#### 7.5.1.1 SDS\_QA\_START

QA Application startup sequence

##### Model Kinds

The QA Application startup sequence is illustrated with two kinds of models:

- the UML sequence diagram shown in [Figure SDS-START](#) (*dynamic view*)
- the pseudocode shown in [Listing SDS-START](#) (*static view*)



*Figure SDS-START: QP Application startup sequence.*

```
[0] int main(int argc, char *argv[]) {
[1]     QF_init(); // initialize the QF Active Object framework
[2]     BSP_init(); // initialize the Board Support Package
[3]     // initialize publish-subscribe...
[4]     static QSubscrList subscrSto[MAX_PUB_SIG];
```

```

QActive_psInit(subscrSto, Q_DIM(subscrSto));

[4]   // initialize event pools...
static QF_MPOOL_EL(SmallEvt) smlPoolSto[10];
QF_poolInit(smlPoolSto, sizeof(smlPoolSto), sizeof(smlPoolSto[0]));

static QF_MPOOL_EL(MediumEvt) medPoolSto[5];
QF_poolInit(medPoolSto, sizeof(medPoolSto), sizeof(medPoolSto[0]));
.

[5]   // instantiate and start Active Objects...
ActiveA_ctor();
static QEvt const *aoA_QueueSto[5];
QActive_start(AO_activeA,
    2U,                                // QP priority of the AO
    aoA_QueueSto,                      // event queue storage
    Q_DIM(aoA_QueueSto),               // queue length [events]
    (void *)0, 0U,                     // no stack storage
    (void *)0);                        // no initialization param

ActiveB_ctor();
static QEvt const *aoB_QueueSto[3];
QActive_start(AO_activeB,
    5U,                                // QP priority of the AO
    aoB_QueueSto,                      // event queue storage
    Q_DIM(aoB_QueueSto),               // queue length [events]
    (void *)0, 0U,                     // no stack storage
    (void *)0);                        // no initialization param
.

[6]   // transfer control to QF framework to run the Active Objects
return QF_run();
}

```

*Listing SDS-START: QP Application startup sequence.*

### Description

The following descriptions pertain to both [Figure SDS-START](#) and [Listing SDS-START](#) because the steps are labeled consistently in these two model kinds.

- [0] The QP Application startup sequence occurs in the `main()` function.
- [1] The Active Object framework layer gets initialized, which also performs initialization of the real-time kernel.
- [2] The Board Support Package (BSP) initialization sets up the hardware, software tracing (if used), etc.
- [3] If this QP Application uses [publish-subscribe](#), it is initialized with a call to [QActive\\_psInit\(\)](#).

### Note

The QP Application must provide the memory for the subscriber-lists (`subscrSto`), which is allocated statically in this case.

- [4] The QP Application initializes all event pools that it is using by repeated calls to [QF\\_poolInit\(\)](#)

### Note

The QP Application must provide the memory for each of the initialized event pools (e.g., `medPoolSto`), which is allocated statically in this case.

- [5] The QP Application instantiates (by calling the constructor) and starts all Active Objects by calling [QActive\\_start\(\)](#). Each Active Objects is assigned [unique priorities](#), and provided with event-queue buffers and the stacks (if required). Also, this step involves executing the top-most initial transition in the Active Object ([Figure SDS-START \[5B\]](#)). Additionally, this step triggers the top-most initial transition in the Active Object's state machine, which might involve some interaction with the BSP to initialize the hardware controlled by this Active Object ([Figure SDS-START \[5C\]](#)).

### Note

The QP Application must provide the event-queue buffer (e.g., `aoA_QueueSto`) and the stack (if required by the underlying kernel) for each started Active Object.

- [ 6 ] The QP Application transfers control to QF Active Object Framework by calling `QF_run()`. `QF_run()` begins by calling the `QF_onStartup()` callback to configure and enable **interrupts** (Figure SDS-START [6A]), as the system is only now ready to receive them. After this, QF Framework starts execution of Active Objects (Figure SDS-START [6B]), which typically involves transferring control to the underlying real-time kernel.

### Note

In deeply embedded applications `QF_run()` does not return. In case QP runs on top of a General-Purpose OS (e.g., POSIX), `QF_run()` might return to the OS. The behavior of `QF_run()` depends on the QP Port (see [QP OSAL](#)).

### Backward Traceability

- `SAS_QP_MEM`: General memory allocation policy
- `SAS_QP_EMM`: Event memory management policy

### Forward Traceability (truncated to 2 level(s))

- `QActive::QActive_psInit()`: Publish event to all subscribers of a given signal  $e \rightarrow sig$
- `QF::QF_init()`: QF initialization
- `QF::QF_run()`: Transfers control to QF to run the application
- `QF::QF_onStartup()`: Startup QF callback
- `QF::QF_poolInit()`: Event pool initialization for dynamic allocation of events

## 7.5.2 Event Exchange View

### 7.5.2.1 SDS\_QP\_POST

#### QP event posting sequence

##### Model Kinds

Event posting to Active Objects is illustrated with UML sequence diagrams shown in Figure SDS-POST1 and Figure SDS-POST2 (*dynamic views*). The views are explained in the labeled descriptions following the sequence diagrams. The chosen scenarios pertain to mutable events dynamically allocated from event pools. Also the shown order of processing corresponds to a *preemptive* scheduler.

Scenario: Event posting from ISR to Active Object

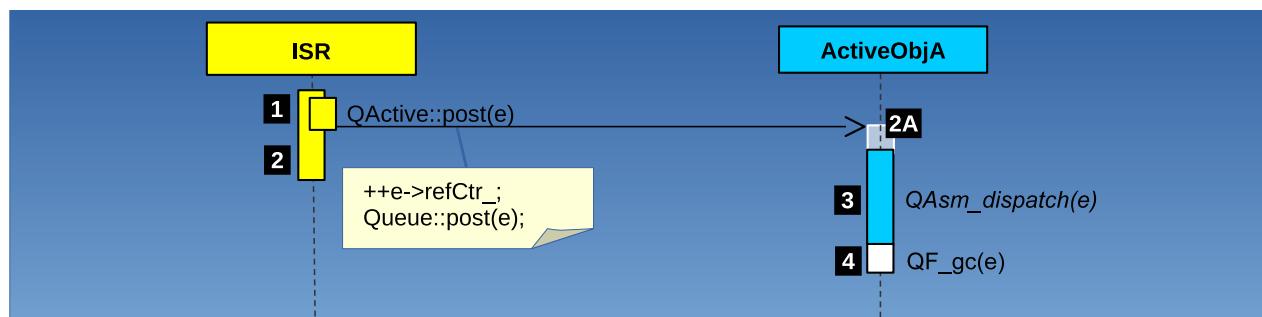


Figure SDS-POST1: Posting event from ISR to an Active Object.

- [1] Event posting begins with incrementing the reference counter of the event (for mutable events dynamically allocated from event pools), which happens within a critical section.
- [2] The event is posted to the event queue of the recipient Active Object. The default behavior of the queue is to assert internally that the queue does not overflow and can accept the event (this is part of the event delivery guarantee).
- [2A] Assuming that the recipient Active Object didn't have events before, adding an event to its queue makes the Active Object ready to run. However, the Active Object is not assigned the CPU just yet because the ISR has a higher priority and continues to completion. (NOTE: the same behavior would occur if the event was posted from an Active Object of a higher priority than the recipient.)
- [3] Only after the ISR completes, the received event is dispatched to the internal state machine of the recipient Active Object. The virtual `dispatch()` function runs-to-completion (RTC) in the thread context of the recipient Active Object.
- [4] After the RTC step, the event is garbage-collected, which decrements the reference counter (for a mutable dynamic event). The event is recycled back to the original event pool only when the reference count drops to zero.

Scenario: Event posting from low-priority to high-priority Active Object



Figure SDS-POST2: Posting event from a lower-priority Active Object to higher-priority Active Object (preemptive scheduler).

- [1] As before in Figure SDS-POST1, event posting begins with incrementing the reference counter of the event (for mutable events dynamically allocated from event pools), which happens within a critical section.
- [2] As before, the event is posted to the event queue of the recipient Active Object.
- [2A] However, assuming that the system executes under a *preemptive*, priority-based scheduler, the recipient Active Object immediately preempts the sender Active Object. This happens because the recipient has a higher priority than the sender and a preemptive scheduler must always give control to the highest-priority Active Object ready to run.
- [3] The recipient Active Object dispatches the event to its internal state machine.
- [4] The recipient Active Object calls the garbage-collector to decrement the reference count and recycles the event back to the original event pool.
- [5] The CPU is assigned back to the preempted sender Active Object. The sender completes its RTC step.

#### Note

The sender Active Object should NOT access the event after posing because it might be recycled by that time, as illustrated in this scenario. Please see also dynamic event life-cycle in Figure SRS-EVT-LIFE.

#### Backward Traceability

- [SAS\\_QP\\_EMM](#): Event memory management policy

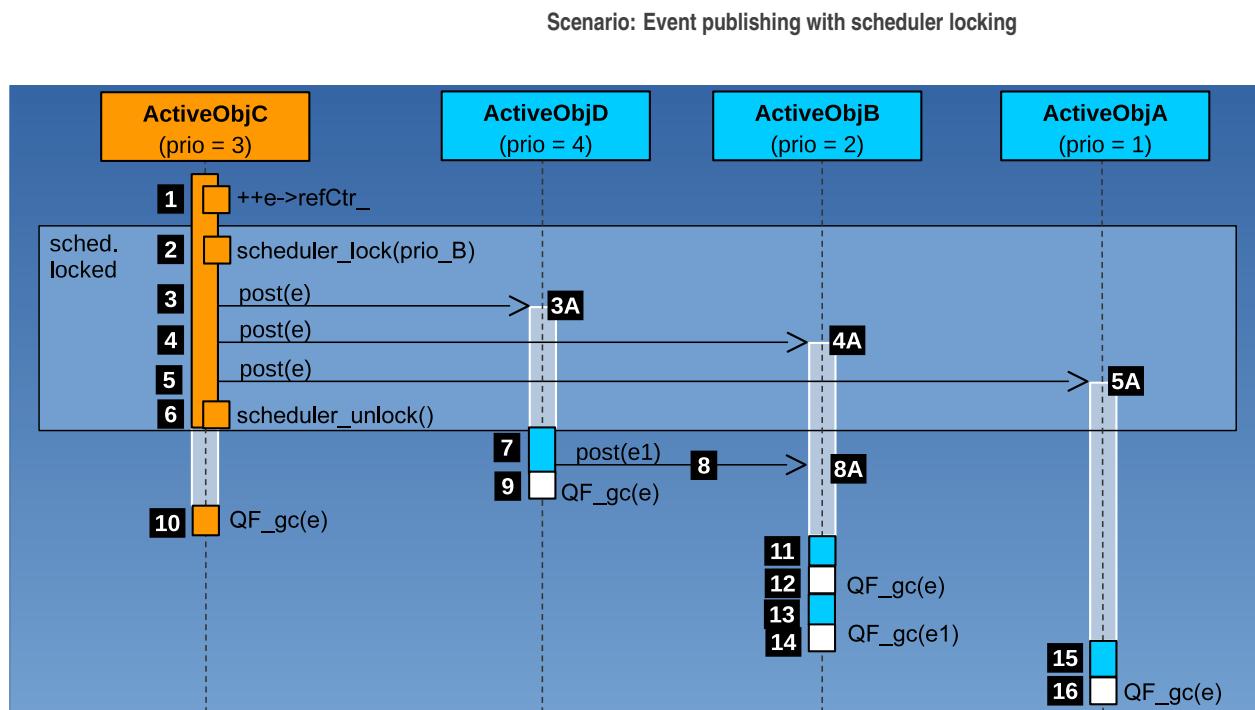
#### Forward Traceability (truncated to 2 level(s))

### 7.5.2.2 SDS\_QP\_PUB

QP event publishing sequence

Model Kinds

Event publishing to Active Objects is illustrated with UML sequence diagrams shown in [Figure SDS-PUB1](#) and [Figure SDS-PUB2 \(dynamic view\)](#). The view is explained in the labeled descriptions following the sequence diagram. The chosen scenarios pertain to mutable events dynamically allocated from event pools. Also the shown order of processing corresponds to a *preemptive* scheduler.



*Figure SDS-PUB1: Publishing event form a medium-priority Active Object (preemptive scheduler).*

- [1] Event publishing begins with incrementing the reference counter of the event (for mutable events dynamically allocated from event pools), which happens within a critical section.
- [2] Next, the publish operation determines the highest-priority subscriber and selectively locks the scheduler up to that priority. (NOTE: selective scheduler locking is supported by modern real-time kernels. Older kernels support only indiscriminate, global scheduler locking).
- [3] Event is posted to the highest-priority subscriber Active Object, which increments the reference counter of the event (per [SDS\\_QP\\_POST](#)).
- [3A] Even if the priority of that subscriber is higher than the sender, preemption does not happen because of the scheduler lock.
- [4] Event is posted to the medium-priority subscriber Active Objects, which increments the reference counter of the event (per [SDS\\_QP\\_POST](#)).
- [5] Event is posted to the low-priority subscriber Active Objects, which increments the reference counter of the event (per [SDS\\_QP\\_POST](#)).
- [4A–5A] The events are only posted, but the recipients Active Objects don't run yet.
- [6] The publish operation unlocks the scheduler to the previous level before the publishing.
- [7] The highest-priority recipient Active Object immediately preempts the lower-priority publisher and dispatches the published event to its state machine.
- [8] During the processing, the highest-priority Active Object posts another event  $e_1$  to ActiveObjB.
- [8A] The posted event  $e_1$  is only enqueued, but is not processed.

[ 9 ] The highest-priority Active Object continues and garbage-collects the published event. This only decrements its reference counter, but does not recycle the published event.

[10] The event publisher garbage-collects the original event. This decrements the reference counter incremented in step [1].

#### Note

The garbage-collect step [10] prevents event leak in case there are **no subscribers** to that event.

[11] ActiveObjB dispatches the published event to its state machine.

[12] The event publisher garbage-collects published event. This decrements the reference counter, but does not recycle the event yet.

[13] ActiveObjB dispatches the posted event e1 to its state machine.

#### Remarks

ActiveObjB processes events in the expected order (based on cause and effect): published event e followed by posted event e1 (caused by the published event e).

[14] ActiveObjB garbage-collects the posted event e1, which decrements its reference counter and recycles the event.

[15] The lowest-priority ActiveObjA dispatches the published event to its state machine.

[16] The lowest-priority ActiveObjA garbage-collects the original event, which decrements its reference counter. This time, the counter drops to zero, so the event is finally recycled to its original event pool.

Scenario: Event publishing WITHOUT scheduler locking

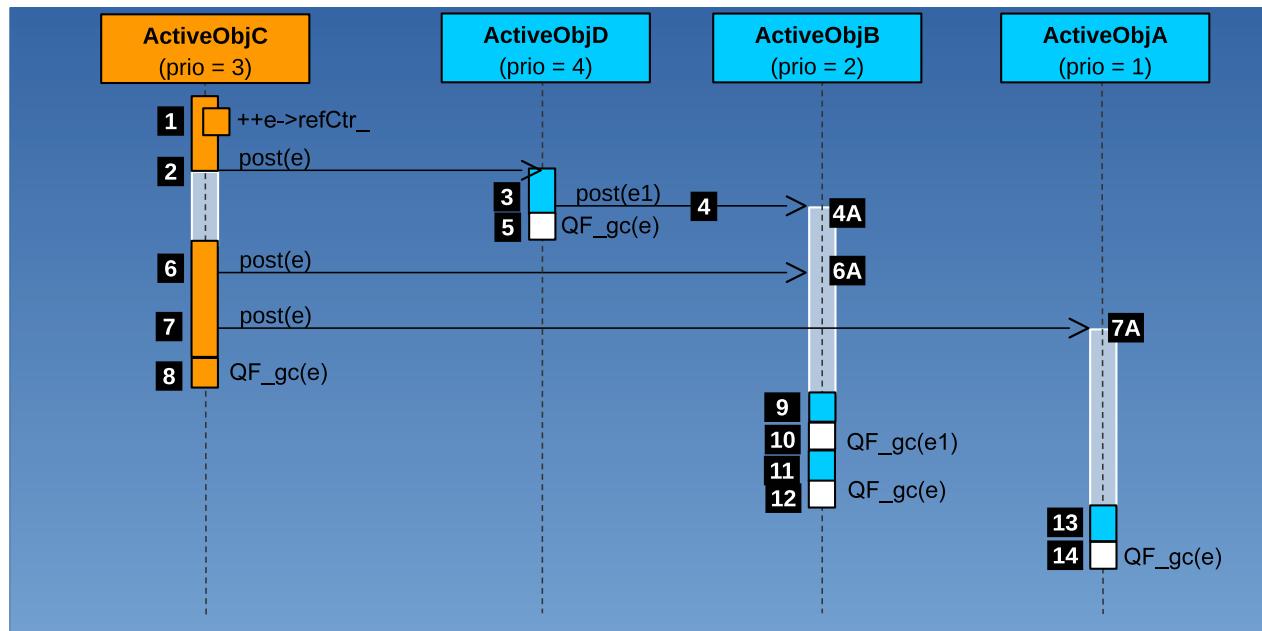


Figure SDS-PUB2: Publishing event WITHOUT scheduler locking (preemptive scheduler).

[ 1 ] As before, event publishing begins with incrementing the reference counter of the event (for mutable events dynamically allocated from event pools), which happens within a critical section.

[ 2 ] The publish operation determines the highest-priority subscriber and posts the event to that Active Object. This increments the reference counter of the event (per [SDS\\_QP\\_POST](#)).

[ 3 ] However, this time the scheduler is NOT locked, so the highest-priority Active Object immediately preempts the lower-priority publisher.

[4] During the processing, the highest-priority Active Object posts another event  $e_1$  to ActiveObjB.

[4A] The posted event  $e_1$  is only enqueued, but is not processed.

[5] The highest-priority Active Object continues and garbage-collects the published event. This only decrements its reference counter, but does not recycle the published event because its reference counter has been incremented in step [1].

[6] The preemptive scheduler resumes the preempted publisher, which posts the original event to ActiveObjB.

#### Note

At this point, the medium-priority ActiveObjB has two events, which are enqueued in the following order:  $e_1$  followed by  $e$ . This is an **unexpected** order because event  $e_1$  is caused by event  $e$ , yet it precedes event  $e$ . This unexpected re-ordering of events is the result of NOT locking the scheduler.

[6A] The posted event is only enqueued, but is not processed.

[7] The preemptive scheduler posts the original event to ActiveObjC.

[7A] The posted event is only enqueued, but is not processed.

[8] The event publisher garbage-collects the original event. This decrements the reference counter incremented in step [1].

#### Note

The garbage-collect step [8] prevents event leak in case there are **no subscribers** to that event.

[9] The medium-priority ActiveObjB dispatches the event  $e_1$  to its internal state machine.

[10] The medium-priority ActiveObjB garbage-collects event  $e_1$ , which causes recycling of that event.

[11] The medium-priority ActiveObjB dispatches the event  $e$  to its internal state machine.

[12] The medium-priority ActiveObjB garbage-collects published event  $e$ , which decrements its reference counter but does not recycle the event yet.

[13] The lowest-priority ActiveObjA dispatches the published event  $e$  to its internal state machine.

[14] The lowest-priority ActiveObjA garbage-collects the published event  $e$ , which is no longer referenced, so it is recycled.

#### Remarks

Even without scheduler locking, the event publishing performs event multicasting without event leaks. However, without scheduler locking event publishing might end up with an unexpected re-ordering of events, as illustrated for the medium-priority ActiveObjB in this scenario.

#### Backward Traceability

- [SAS\\_QP\\_EMM: Event memory management policy](#)

#### Forward Traceability (truncated to 2 level(s))

### 7.5.3 Mutable Event Life Cycle

The [zero-copy event management](#) is designed to be intuitive and transparent to the application-level code. However, for the *zero-copy event management* abstraction to behave exactly like true event copying, QP/C Application needs to obey specific **event ownership rules** similar to the rules of working with objects allocated with the C++ operator `new` and summarized in the life cycle diagram of a mutable event (see [Figure SDS-EVT-LIFE](#)). In exchange, QP/C Framework can safely and deterministically deliver your mutable events with hard real-time performance, which does not complicate the RMS/RMA method and is superior to the copying entire events approach.

### 7.5.3.1 SDS\_QP\_MELC

Mutable event ownership rules

Model Kind

The Mutable event life cycle is illustrated with UML state diagram.

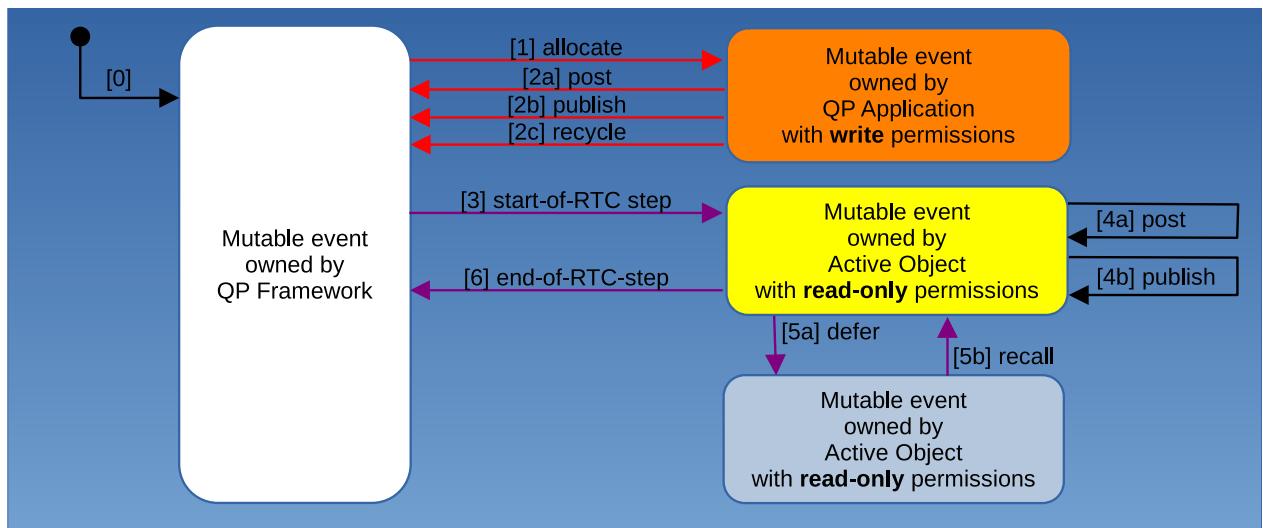


Figure SDS-EVT-LIFE: Mutable Event Life Cycle and Ownership Rules

Figure SDS-EVT-LIFE illustrates the mutable event life cycle and possible transfers of ownership rights to the event:

[0] All mutable events are initially owned by QP/C Framework.

[1] An event producer might gain ownership of a new event only by allocating it. At this point, the producer gains the ownership rights with the permissions to **write** to the event. Indeed, the purpose of this stage in the mutable event's life cycle is to initialize the event and fill it with data. The event producer might keep the event as long as it needs. For example, the producer (e.g., ISR) might fill the event with data over many invocations. Eventually, however, the producer must transfer the ownership back to the framework.

[2a, 2b, 2c] Typically the producer posts [2a] or publishes [2b] the event. As a special case, the producer might decide that the event is not good, in which case the producer must *explicitly* recycle [2c] the event. After any of these three operations, the producer immediately loses ownership of the event and can no longer access it. In particular, it is illegal to post, publish, or recycle the event again.

[3] The recipient Active Object gains ownership of the current event upon the start of the RTC step. This time, the Active Object gains the **read-only** permissions to the *current event*.

[4a, 4b] During the RTC step, the recipient Active Object is allowed to re-post [4a] or re-publish [4b] the *current event* any number of times without losing ownership of the event.

[5a] As a special case, the recipient Active Object may defer the current event. Event deferral extends the read-only ownership rights beyond the current RTC step.

[5b] Eventually, however, the deferred event must be recalled, which self-posts the event into the Active Object's event queue (using the LIFO policy). Recalling ends the ownership of the original deferred event.

[6] The end of the RTC step terminates the ownership of the *current event*. The Active Object cannot use the event in any way past the RTC step. In particular, if any data from that event is needed in the future, QP/C Application must save that data (typically in some attributes inside the Active Object).

Backward Traceability

- [SAS\\_QP\\_MEM](#): General memory allocation policy
- [SAS\\_QP\\_EMM](#): Event memory management policy

Forward Traceability (truncated to 2 level(s))

---

## 7.6 State Dynamics Viewpoint

The **State Dynamics Viewpoint** explains how QP Application shall implement hierarchical state machines. This design viewpoint is most important and relevant to **QP Application developers**, who's primary activity is designing and ultimately implementing the hierarchical state machines of Active Objects.

This viewpoint consists of two parts:

- state machine implementation strategy based on the **QHsm class** (designed for "manual coding", see [SRS\\_QP\\_SM\\_20](#))
- state machine implementation strategy based on the **QMsm class** (designed for "automatic code generation", see [SRS\\_QP\\_SM\\_21](#))

The design concerns in this viewpoint correspond to the "recipes" for implementing various state machine elements. The state diagram and the table below list the elements and provides links to the design views in the "QHsm-based implementation" and "QMsm-based implementation".

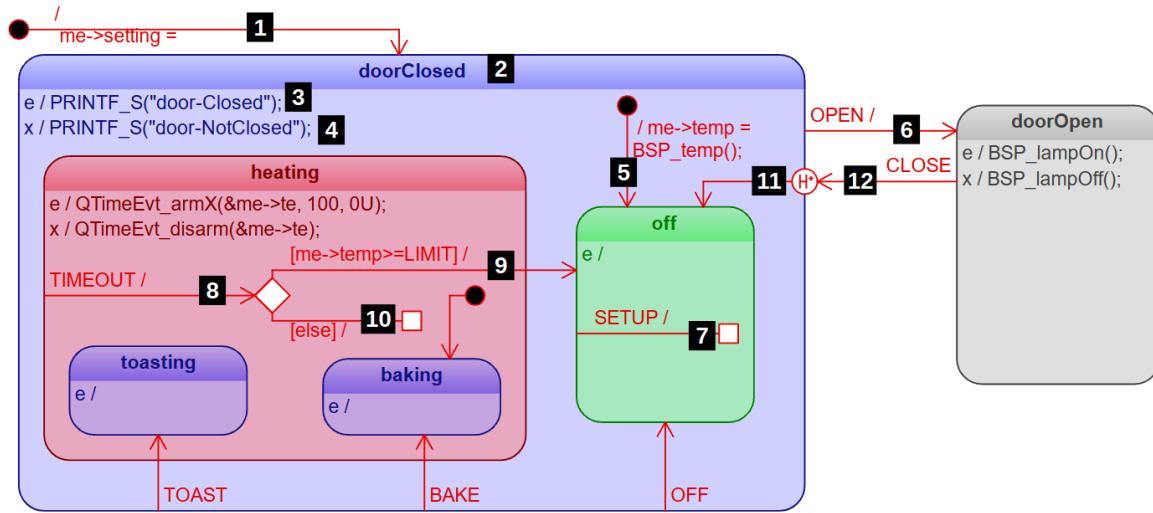


Figure SDS-SM: Example Hierarchical State Machine (a toaster oven)

The state machine elements labeled in [Figure SDS-SM](#) are as follows:

Label	Design concern	QHsm-based implementation	QMsm-based implementation
	declaring application-level state machine sub-class	<a href="#">SDS_QA_QHsm_decl</a>	<a href="#">SDS_QA_QMsm_decl</a>
[ 1 ]	implementing top-most initial pseudo-state	<a href="#">SDS_QA_QHsm_top_init</a>	<a href="#">SDS_QA_QMsm_top_init</a>
[ 2 ]	implementing states (including states nested in other states)	<a href="#">SDS_QA_QHsm_state</a>	<a href="#">SDS_QA_QMsm_state</a>
[ 3 ]	implementing state entry actions	<a href="#">SDS_QA_QHsm_entry</a>	<a href="#">SDS_QA_QMsm_entry</a>
[ 4 ]	implementing state exit actions	<a href="#">SDS_QA_QHsm_exit</a>	<a href="#">SDS_QA_QMsm_exit</a>
[ 5 ]	implementing nested initial transitions	<a href="#">SDS_QA_QHsm_nest_init</a>	<a href="#">SDS_QA_QMsm_nest_init</a>
[ 6 ]	implementing state transitions	<a href="#">SDS_QA_QHsm_tran</a>	<a href="#">SDS_QA_QMsm_tran</a>
[ 7 ]	implementing internal transitions	<a href="#">SDS_QA_QHsm_intern</a>	<a href="#">SDS_QA_QMsm_intern</a>
[ 8 ]	implementing choice points and guard conditions	<a href="#">SDS_QA_QHsm_choice</a>	<a href="#">SDS_QA_QMsm_choice</a>
[ 9 ]	Guard condition with internal transition	<a href="#">SDS_QA_QHsm_choice</a>	<a href="#">SDS_QA_QMsm_choice</a>
[ 10 ]	Guard condition with regular transition	<a href="#">SDS_QA_QHsm_choice</a>	<a href="#">SDS_QA_QMsm_choice</a>

Label	Design concern	QHsm-based implementation	QMsm-based implementation
[11]	implementing state history	<a href="#">SDS_QA_QHsm_hist</a>	<a href="#">SDS_QA_QMsm_hist</a>
[12]	implementing transition to state history	<a href="#">SDS_QA_QHsm_hist_tran</a>	<a href="#">SDS_QA_QMsm_hist_tran</a>

**Note**

The design concerns included in this State Dynamics viewpoint apply to **QP Applications** rather than QP Framework and therefore they are use the UIDs `SDS_QA_...` (with the component part set to `QA` rather than `QP`).

The **model kind** used to illustrate the State Dynamics Viewpoint is the UML *state diagram* shown in [Figure SDS-SM](#) and [Figure SDS-SM](#) (static views). These state diagrams depict hierarchical state machines (for a toaster oven) with labeled elements, which are subsequently elaborated in various traceable Design Views.

## 7.6.1 QHsm Design View

This section presents the [hierarchical state machine implementation strategy](#) "optimized for manual coding", which means that changing a single element in the state machine design (e.g., nesting of the state hierarchy) should require changing only a single matching element in the implementation. This strategy imposes restrictions on the implementation, but does not mean that the code must be written manually. In the presence of a modeling tool, such code can also be generated automatically and most of the code presented in this section has indeed been generated by the [QM modeling tool↑](#) (from the state diagram presented in [Figure SDS-SM](#)).

### 7.6.1.1 SDS\_QA\_QHsm\_decl

Declaring `QHsm` subclass.

#### Description

The first step in coding a hierarchical state machine based on the `QHsm` implementation is to declare a subclass of the `QHsm` or the `QActive` base class.

#### Example

The following snippet of code shows the complete declaration of the `ToasterOven` subclass of `QHsm` shown in [Figure SDS-SM](#):

```

[1]  typedef struct {
[2]      // protected:
[2]      QHsm super;
[3]      // private:
[3]      uint32_t settings;
[3]      uint16_t temp;
[3]      QTimeEvt te;
[4]      // private state histories
[4]      QStateHandler hist_doorClosed;
[1]  } ToasterOven;
[5]  // public:
[10] void ToasterOven_ctor(ToasterOven * const me) {
[11]     QHsm_ctor(&me->super, _QSTATE_CAST(&ToasterOven_initial));
[12]     QTimeEvt_ctorX(&me->te, &me->super, TIMEOUT_SIG, 0U),
[1] }
[13] // protected:
[20] QState ToasterOven_initial(ToasterOven * const me, void const * const par);
[30] QState ToasterOven_doorClosed(ToasterOven * const me, QEvt const * const e);
[30] QState ToasterOven_heating(ToasterOven * const me, QEvt const * const e);
[30] QState ToasterOven_toasting(ToasterOven * const me, QEvt const * const e);
[30] QState ToasterOven_baking(ToasterOven * const me, QEvt const * const e);

```

```

QState ToasterOven_off(ToasterOven * const me, QEvt const * const e);
QState ToasterOven_doorOpen(ToasterOven * const me, QEvt const * const e);

// private:
[40] void ToasterOven_actionA(ToasterOven * const me);
void ToasterOven_actionB(ToasterOven * const me);

[1] Declaration of class ToasterOven

```

[2] Class ToasterOven inherits [QHsm](#), so it can have a state machine that must be implemented according to the rules specified in the [QHsm Design View](#).

#### Note

The state machine implementation is identical, whether it inherits [QHsm](#), [QActive](#), or subclasses of these base classes.

[3] The class can have data members (typically private), which will be accessible inside the state machine as the "extended-state variables".

[4] If the state machine uses state history ([Figure SDS-SM \[11\]](#)), a data member must be provided to remember each state's history.

[10] The class needs to provide a *constructor*, typically without any parameters.

#### Note

The role of the constructor is just to initialize the state machine instance to bind it to the top-most initial pseudo-state (see step [5]). Most other data members are initialized in the top-most initial transition, which is taken later, when the state machine is initialized (see [SDS\\_QA\\_QHsm\\_top\\_init](#)).

[11] The class *constructor*, must call the superclass' constructor ([QHsm\\_ctor\(\)](#) constructor in this case). The superclass' constructor initializes this state machine to the top-most initial pseudostate declared in step [5].

[12] The class *constructor*, must call the constructors of its data members, if such constructors exist.

[20] Each state machine must have exactly one initial pseudo-state, which is a member function of the class as shown in the listing. By convention, the initial pseudo-state should be always called `initial`.

[30] All states are represented as member functions of the state machine class (hence they are called *state-handler functions*). Every state-handler function takes the const pointer to the current event as a parameter.

#### Note

All states are declared the same way regardless of their nesting level.

[40] The class might also provide any number of action functions that will be called in the state machine.

#### Backward Traceability

- [SAS\\_QP\\_FRM](#): *QP Framework layer*

#### Forward Traceability (truncated to 2 level(s))

### 7.6.1.2 SDS\_QA\_QHsm\_top\_init

Implementing the top-most initial pseudo-state.

#### Description

The top-most initial pseudostate is a member function of the state machine class (see [Figure SDS-SM \[1\]](#)). This pseudostate defines the top-most initial transition, which will be taken when the state machine is initialized (see [SDS\\_QP\\_QAsm](#) interface `init()` function).

### Example

The following example code shows a top-most initial pseudo-state function with various elements explained in the labeled annotations following the listing.

```
[1] QState ToasterOven_initial(ToasterOven * const me, void const * const par) {
[2]     me->setting = DEFAULT_SETTINGS;
me->temp = BSP(Temp());
// state history attributes
[3]     me->hist_doorClosed = Q_STATE_CAST(&ToasterOven_off);

[4]     QS_FUN_DICTIONARY(&ToasterOven_doorClosed);
QS_FUN_DICTIONARY(&ToasterOven_heating);
QS_FUN_DICTIONARY(&ToasterOven_toasting);
QS_FUN_DICTIONARY(&ToasterOven_baking);
QS_FUN_DICTIONARY(&ToasterOven_off);
QS_FUN_DICTIONARY(&ToasterOven_doorOpen);

[5]     return Q_TRAN(&ToasterOven_doorClosed);
}
```

[1] The top-most initial pseudostate function signature (see also [SDS\\_QA\\_QHsm\\_decl](#) [5])

[2] The top-most initial pseudostate initializes internal variables (see also [SDS\\_QA\\_QHsm\\_decl](#) [3])

[3] The top-most initial pseudostate initializes state histories for all states with history (see also [SDS\\_QA\\_QHsm\\_hist](#))

[4] If [QS](#) software tracing is used, the top-most initial pseudostate produces function dictionaries for all its state-handler functions

[5] The top-most initial pseudostate must return with the [Q\\_TRAN\(\)](#) macro. The argument of this macro is the target state indicated in the diagram (see [Figure SDS-SM](#) [1])

### Backward Traceability

- [SAS\\_QP\\_FRM](#): *QP Framework layer*

### Forward Traceability (truncated to 2 level(s))

---

#### 7.6.1.3 SDS\_QA\_QHsm\_state

Implementing a state and its nesting.

##### Description

States are member functions of the state machine class (see [Figure SDS-SM](#) [2]). State-handler functions are called during the process of [dispatching events](#). Their purpose is to perform *your* specific actions and then to *tell* the [%QP event processor](#) what your state-handler has just done. For example, if your state-handler performs a state transition, it executes some actions and then it calls the special [Q\\_TRAN\(\)](#) macro, where it specifies the target state of this state transition. The state-handler then **returns** the status from the [Q\\_TRAN\(\)](#) macro, and through this return value it informs the [dispatch operation](#) what needs to be done with the state machine. Based on this information, the event-processor might decide to call this or other state-handler functions to process the same current event.

### Example

The following example code shows two state-handler functions with almost identical structure. The first one implements a top-level state. The second one implements a state that is explicitly nested in another state.

**Note**

The general structure of state-handler functions is the same regardless of their nesting level.

```
// top-level state
[1] QState ToasterOven_doorClosed(ToasterOven * const me, QEvt const * const e) {
[2]     QState status_;
[3]     switch (e->sig) {
[4]         .
[5]         default: {
[6a]             status_ = Q_SUPER(&QHsm_top);
[6b]             break;
[7]         }
[7]     return status_;
}

// state explicitly nested in another state
[1] QState ToasterOven_heating(ToasterOven * const me, QEvt const * const e) {
[2]     QState status_;
[3]     switch (e->sig) {
[4]         .
[5]         default: {
[6b]             status_ = Q_SUPER(&ToasterOven_doorClosed);
[6c]             break;
[7]         }
[7]     return status_;
}
```

[1] The state function signature (see also [SDS\\_QA\\_QHsm\\_decl](#) [6]). The function takes the `me` pointer (OOP) and a `const` pointer to the current event `e`.

[2] The local `status_` variable to return at the end of the state-handler function

[3] The state-handler function is structured as a `switch` statement that discriminates based on the signal of the current event `e->sig` (see [SDS\\_QP\\_QEvt](#))

[4] the case of the `switch` implement various state elements explained in the remaining Design Elements in this viewpoint.

[5] The `switch` must always have the default case, which should be executed **without side effects**.

[6a] For a state that does not nest in any other state, you need to set the `status_` variable to the [Q\\_TRAN\(\)](#) macro, whereas the argument [QHsm\\_top\(\)](#) is the top-most state handler function defined in the [QHsm](#) base class.

[6b] For a state that explicitly nests in another state, you need to set the `status_` variable to the [Q\\_TRAN\(\)](#) macro, whereas the argument is the superstate of that state.

**Backward Traceability**

- [SAS\\_QP\\_FRM](#): *QP Framework layer*

**Forward Traceability (truncated to 2 level(s))****7.6.1.4 SDS\_QA\_QHsm\_entry**

Implementing state entry action.

**Description**

State entry action (see [Figure SDS-SM \[3\]](#)) is coded as special case for the reserved signal [Q\\_ENTRY\\_SIG](#). The QP event processor executes the entry action during a state transition processing by calling the state-handler function passing in the reserved event with the [Q\\_ENTRY\\_SIG](#). Please note that the entry action has no access to the original event that triggered the state transition.

### Example

The following example code shows a state-handler function with an entry action.

```
QState ToasterOven_heating(ToasterOven * const me, QEvt const * const e) {
    QState status_;
    switch (e->sig) {
[1]        case Q_ENTRY_SIG: {
[2]            QTimeEvt_armX(&me->te, 100, 0U);
[3]            status_ = Q_HANDLED();
            break;
        }
        . .
    }
    return status_;
}
```

- [1] State with an entry action needs a case-statement on the reserved signal [Q\\_ENTRY\\_SIG](#)

### Note

The case `Q_ENTRY_SIG` is needed only if a state actually has something to do upon the state entry. If there is nothing to do, the whole case can be just omitted.

- [2] Inside the case, you code all the actions to be called upon the entry to the state.

- [3] The action list must end with setting the status to the `Q_HANDLED()` macro, which informs the QP event processor that the entry action has been handled.

### Backward Traceability

- [SAS\\_QP\\_FRM](#): *QP Framework layer*

### Forward Traceability (truncated to 2 level(s))

---

#### 7.6.1.5 SDS\_QA\_QHsm\_exit

Implementing state exit action.

### Description

State exit action (see [Figure SDS-SM \[4\]](#)) is coded as special case on the reserved signal `Q_EXIT_SIG`. The QP event processor executes the exit action during a state transition processing by calling the state-handler function passing in the reserved event with the `Q_EXIT_SIG`. Please note that the entry action has no access to the original event that triggered the state transition.

### Example

The following example code shows a state-handler function with an exit action.

```
QState ToasterOven_heating(ToasterOven * const me, QEvt const * const e) {
    QState status_;
    switch (e->sig) {
        . .
[1]        case Q_EXIT_SIG: {
[2]            (void) QTimeEvt_disarm(&me->te);
[3]            status_ = Q_HANDLED();
            break;
        }
        . .
    }
    return status_;
}
```

- [1] State with an exit action needs a case statement on the reserved signal [Q\\_EXIT\\_SIG](#)

**Note**

The case `Q_EXIT_SIG` is needed only if a state actually has something to do upon the state exit. If there is nothing to do, the whole case can be just omitted.

[2] Inside the case, you code all the actions to be called upon the exit from the state.

[3] The action list must end with setting the status to the `Q_HANDLED()` macro, which informs the QP event processor that the exit action has been handled.

**Backward Traceability**

- `SAS_QP_FRM`: *QP Framework layer*

**Forward Traceability (truncated to 2 level(s))****7.6.1.6 SDS\_QA\_QHsm\_nest\_init**

Implementing nested initial transition.

**Description**

Nested initial transition (see [Figure SDS-SM \[5\]](#)) in a state is coded as special case for the reserved signal `Q_INIT_SIG`. If a state has no nested initial transition, the whole case can be just omitted.

**Example**

The following example code shows a state-handler function with nested initial transition.

```
QState ToasterOven_heating(ToasterOven * const me, QEvt const * const e) {
    QState status_;
    switch (e->sig) {
        . .
        case Q_INIT_SIG: {
            [1]     me->temp = BSP_temp(); // get temperature
            [2]     status_ = Q_TRAN(&ToasterOven_baking);
            [3]     break;
        }
        . .
    }
    return status_;
}
```

[1] State with a nested initial transition needs a case-statement on the reserved signal `Q_INIT_SIG`

[2] Inside the case, you code all the actions to be called upon the initial transition.

[3] The action list must end with setting the status to the `Q_TRAN()` macro with the argument being the substate-handler, which informs the QP event processor about the next state and that the nested initial transition has been executed.

**Attention**

The target of the nested initial transition specified with the macro `Q_TRAN()` must be a substate of the given state.

**Backward Traceability**

- `SAS_QP_FRM`: *QP Framework layer*

**Forward Traceability (truncated to 2 level(s))**

### 7.6.1.7 SDS\_QA\_QHsm\_tran

Implementing state transition.

#### Description

A state transition (see [Figure SDS-SM \[6\]](#)) is coded as special case for the user-defined signal (transition trigger).

#### Example

The following example code shows a state-handler function with a state transition.

```
QState ToasterOven_doorClosed(ToasterOven * const me, QEvt const * const e) {
    QState status_;
    switch (e->sig) {
        .
        .
        .
        [1]     case OPEN_SIG: {
        [2]         ToasterOven_actionA(me);
        [3]         status_ = Q_TRAN(&ToasterOven_doorOpen);
        [4]         break;
        [5]     }
        .
        .
    }
    return status_;
}
```

[1] State with a transition needs a case-statement on the user-defined signal (transition trigger).

[2] Inside the case, you code all the actions to be called upon the transition.

[3] The action list must end with setting the status to the `Q_TRAN()` macro. The argument of this macro is the target state indicated in the diagram (see [Figure SDS-SM \[6\]](#))

#### Backward Traceability

- [SAS\\_QP\\_FRM](#): *QP Framework layer*

#### Forward Traceability (truncated to 2 level(s))

---

### 7.6.1.8 SDS\_QA\_QHsm\_intern

Implementing internal transition.

#### Description

An internal transition (see [Figure SDS-SM \[7\]](#)) is coded as special case for the user-defined signal (transition trigger).

#### Example

The following example code shows a state-handler function with an internal transition.

```
QState ToasterOven_off(ToasterOven * const me, QEvt const * const e) {
    QState status_;
    switch (e->sig) {
        .
        .
        .
        [1]     case SETUP_SIG: {
        [2]         me->setup = Q_EVT_CAST(SetupEvt)->setup;
        [3]         status_ = Q_HANDLED();
        [4]         break;
        [5]     }
        .
        .
    }
    return status_;
}
```

[1] State with a transition needs a case-statement on the user-defined signal (transition trigger).

- [2] Inside the case, you code all the actions to be called upon the transition.  
 [3] The action list must end with setting the status to the `Q_HANDLED()` macro, which informs the QP event processor that the *internal* transition has been handled.

#### Backward Traceability

- `SAS_QP_FRM`: *QP Framework layer*

#### Forward Traceability (truncated to 2 level(s))

---

### 7.6.1.9 SDS\_QA\_QHsm\_choice

Implementing choice point and guard conditions.

#### Description

A choice point with guard conditions (see [Figure SDS-SM \[8\]](#)) is coded as special case for the user-defined signal (transition trigger).

#### Example

The following example code shows a state-handler function with an internal transition.

```
QState ToasterOven_heating(ToasterOven * const me, QEvt const * const e) {
    QState status_;
    switch (e->sig) {
        . .
        [1]     case TIMEOUT_SIG: {
        [2]         me->temp = BSP_temp();
        [3]         if (me->temp >= LIMIT) {
        [4]             // any actions
        [5]             status_ = Q_TRAN(&ToasterOven_off);
        }
        [6]         else {
        [7]             // any actions
        [8]             status_ = Q_HANDLED();
        }
        break;
    }
    . .
}
return status_;
```

- [1] State with a choice needs a case-statement for the user-defined signal (transition trigger).  
 [2] Inside the case, you code all the actions to be called before any guard conditions are evaluated.  
 [3] The guard condition is represented as a condition of the `if` statement.  
 [4] any actions to be executed if the guard evaluates to 'true'.  
 [5] regular state transition is coded as usual with the `Q_TRAN()` macro with the target of the transition specified as the argument.  
 [6] the complementary guard condition to all evaluated before is specified with the `else` statement.

#### Note

A choice point might have many outgoing guards, each specified in `else if (guard)`

- [7] any actions to be executed for this guard condition.  
 [8] The action list must end with setting the status to the `Q_HANDLED()` macro, which informs the QP event processor that the *internal* transition has been handled..

### Backward Traceability

- [SAS\\_QP\\_FRM](#): *QP Framework layer*

### Forward Traceability (truncated to 2 level(s))

---

#### 7.6.1.10 SDS\_QA\_QHsm\_hist

Implementing state history.

##### Description

State history (see [Figure SDS-SM \[11\]](#)) requires remembering the most recent active substate when the given state is exited, which is accomplished in the exit action from the state coded as already described in [SDS\\_QA\\_QHsm\\_exit](#).

##### Example

The following example code shows an exit action that additionally saves the state history.

```
QState ToasterOven_doorClosed(ToasterOven * const me, QEvt const * const e) {
    QState status_;
    switch (e->sig) {
        . .
        [1]     case Q_EXIT_SIG: {
            [2]         // any exit actions
            [2]         // save deep history
            [3]         me->hist_doorClosed = QHsm_state(Q_HSM_UPCAST(me));
            [4]         status_ = Q_HANDLED();
            [4]         break;
        }
        . .
    }
    return status_;
}
```

[1] State with an exit action needs a case-statement on the reserved signal [Q\\_EXIT\\_SIG](#)

[2] Inside the case, you code all the actions to be called upon the exit from the state.

[3] One of the actions must be to set the history variable for the given state to the currently active state. The [QHsm](#) base class provides two member functions to obtain the current state:

- [QHsm\\_state\(\)](#) to obtain the currently active state for **deep history**
- [QHsm\\_childState\(\)](#) to obtain the immediate active child state of the given state for **shallow history**

[4] The action list must end with setting the status to the [Q\\_HANDLED\(\)](#) macro, which informs the QP event processor that the exit action has been executed.

### Backward Traceability

- [SAS\\_QP\\_FRM](#): *QP Framework layer*

### Forward Traceability (truncated to 2 level(s))

---

### 7.6.1.11 SDS\_QA\_QHsm\_hist\_tran

Implementing transition to history.

#### Description

Transition to state history (see [Figure SDS-SM \[12\]](#)) is coded similar as a regular state transition, but takes the transition with the

#### Example

The following example code shows a transition to state history.

```
QState ToasterOven_doorOpen(ToasterOven * const me, QEvt const * const e) {
    QState status_;
    switch (e->sig) {
        .
        .
        .
        [1]     case CLOSE_SIG: {
        [2]         // any actions
        [3]         status_ = Q_TRAN_HIST(me->hist_doorClosed);
        [4]         break;
        [5]     }
        .
        .
    }
    return status_;
}
```

[1] State with a transition to history needs a case-statement for the user-defined signal (transition trigger).

[2] Inside the case, you code all the actions to be called upon the transition.

[3] The action list must end with setting the status to the [Q\\_TRAN\\_HIST\(\)](#) macro with the state history variable of the transition specified as the argument.

#### Backward Traceability

- [SAS\\_QP\\_FRM](#): QP Framework layer

#### Forward Traceability (truncated to 2 level(s))

## 7.6.2 QMsm Design View

This section presents the [hierarchical state machine implementation strategy](#) "optimized for automatic code generation" (see [SRS\\_QP\\_SM\\_21](#)), which means the implementation may contain some redundant information to improve the *speed* of the state machine execution at the expense of larger code and data (typically in ROM).

This strategy requires a code generator, such as the [QM modeling tool](#). Indeed, the presented code in the Design Views below has been automatically generated by QM (from the state diagram presented in [Figure SDS-SM](#)).

#### Note

The purpose of this section is to describe the design of the generated code for the purpose of its *readability* and the ability to *debug* such code, but not to implement it manually.

### 7.6.2.1 SDS\_QA\_QMsm\_decl

Declaring [QMsm](#) subclass.

#### Description

The first step in coding a hierarchical state machine based on the [QMsm](#) implementation is to declare a subclass of the [QMsm](#) or the [QMActive](#) base class.

### Example

The following snippet of code shows the complete declaration of the `ToasterOven` subclass of `QMsm` shown in [Figure SDS-SM](#):

```
[1]  typedef struct {
[2]      // protected:
[2]      QMsm super;
[3]
[3]      // private:
[3]      uint32_t settings;
[3]      uint16_t temp;
[3]      QTimeEvt te;
[4]
[4]      // private state histories
[4]      QMState const *hist_doorClosed;
[1]  } ToasterOven;
[1]
[1]  // public:
[10] void ToasterOven_ctor(ToasterOven * const me) {
[10]     QHsm_ctor(&me->super, _STATE_CAST(&ToasterOven_initial));
[10]     QTimeEvt_ctorX(&me->te, &me->super, TIMEOUT_SIG, 0U),
[10] }
[1]
[1]  // protected:
[20] QState ToasterOven_initial(ToasterOven * const me, void const * const par);
[1]
[30] QState ToasterOven_doorClosed (ToasterOven * const me, QEvt const * const e);
[31] QState ToasterOven_doorClosed_e(ToasterOven * const me);
[32] QState ToasterOven_doorClosed_x(ToasterOven * const me);
[33] QState ToasterOven_doorClosed_i(ToasterOven * const me);
[34] extern QMState const ToasterOven_doorClosed_s;
[1]
[35] QState ToasterOven_heating (ToasterOven * const me, QEvt const * const e);
[36] QState ToasterOven_heating_e(ToasterOven * const me);
[37] QState ToasterOven_heating_x(ToasterOven * const me);
[38] QState ToasterOven_heating_i(ToasterOven * const me);
[39] extern QMState const ToasterOven_heating_s;
[1]
[40] QState ToasterOven_toasting (ToasterOven * const me, QEvt const * const e);
[41] QState ToasterOven_toasting_e(ToasterOven * const me);
[42] extern QMState const ToasterOven_toasting_s;
[1]
[43] QState ToasterOven_baking (ToasterOven * const me, QEvt const * const e);
[44] QState ToasterOven_baking_e(ToasterOven * const me);
[45] extern QMState const ToasterOven_baking_s;
[1]
[46] QState ToasterOven_off (ToasterOven * const me, QEvt const * const e);
[47] QState ToasterOven_off_e(ToasterOven * const me);
[48] extern QMState const ToasterOven_off_s;
[1]
[49] QState ToasterOven_doorOpen (ToasterOven * const me, QEvt const * const e);
[50] QState ToasterOven_doorOpen_e(ToasterOven * const me);
[51] QState ToasterOven_doorOpen_x(ToasterOven * const me);
[52] extern QMState const ToasterOven_doorOpen_s;
[1]
[53] // private:
[40] void ToasterOven_actionA(ToasterOven * const me);
[41] void ToasterOven_actionB(ToasterOven * const me);
[1]
```

#### [1] Declaration of class `ToasterOven`

[2] Class `ToasterOven` inherits `QMsm`, so it can have a state machine that must be implemented according to the rules specified in the [QMsm Design View](#).

#### Note

The state machine implementation is identical, whether it inherits `QMsm`, `QMActive`, or subclasses of these base classes.

[3] The class can have data members (typically private), which will be accessible inside the state machine as the "extended-state variables".

[4] If the state machine uses state history ([Figure SDS-SM \[11\]](#)), a data member must be provided to remember each state's history.

[10] The class needs to provide a *constructor*, typically without any parameters.

**Note**

The role of the constructor is just to initialize the state machine instance to bind it to the top-most initial pseudo-state (see step [5]). Most other data members are initialized in the top-most initial transition, which is taken later, when the state machine is initialized (see [SDS\\_QA\\_QHsm\\_top\\_init](#)).

[11] The class *constructor*, must call the superclass' constructor ([QMsm\\_ctor\(\)](#) constructor in this case). The superclass' constructor initializes this state machine to the top-most initial pseudostate declared in step [20].

[12] The class *constructor*, must call the constructors of its data members, if such constructors exist.

[20] Each state machine must have exactly one initial pseudo-state, which is a member function of the class as shown in the listing. By convention, the initial pseudo-state should be always called `initial`.

[30–34] States are represented as a group of member functions of the state machine class plus a [QMState](#) data structure. The member functions are as follows:

- [30] state-handler for all transitions (including internal transitions)
- [31] state action handler for state entry
- [32] state action handler for state exit
- [33] state action handler for nested initial transition within this state
- [34] state object that summarizes the information about this state (const in ROM)

**Note**

Only state actions actually used by the state are declared. For example, state `ToasterOven_toasting` does not have an exit action, so it does not define the `ToasterOven_toasting_x` action function.

All states are declared the same way regardless of their nesting level.

[40] The class might also provide any number of action functions that will be called in the state machine.

**Backward Traceability**

- [SAS\\_QP\\_FRM](#): QP Framework layer

**Forward Traceability (truncated to 2 level(s))****7.6.2.2 SDS\_QA\_QMsm\_top\_init**

Implementing the top-most initial pseudo-state.

**Description**

The top-most initial pseudostate is a member function of the state machine class (see [Figure SDS-SM \[1\]](#)). This pseudostate defines the top-most initial transition, which will be taken when the state machine is initialized (see [SDS\\_QP\\_QAsm](#) interface `init()` function).

### Example

The following example code shows a top-most initial pseudo-state function with various elements explained in the labeled annotations following the listing.

```
[1] QState ToasterOven_initial(ToasterOven * const me, void const * const par) {
[2]     me->setting = DEFAULT_SETTINGS;
me->temp = BSP(Temp());
// state history attributes
[3]     me->hist_doorClosed = &ToasterOven_off_s;

[4]     OS_FUN_DICTIONARY(&ToasterOven_doorClosed);
OS_FUN_DICTIONARY(&ToasterOven_off);
OS_FUN_DICTIONARY(&ToasterOven_heating);
OS_FUN_DICTIONARY(&ToasterOven_doorOpen);
OS_FUN_DICTIONARY(&ToasterOven_sm_heating_toasting);
OS_FUN_DICTIONARY(&ToasterOven_sm_heating_baking);
OS_FUN_DICTIONARY(&ToasterOven_sm_heating);

static struct {
    QMState const *target;
    QActionHandler act[3];
[5] } const tatbl_ = { // tran-action table
[6]     &ToasterOven_doorClosed_s, // target state
{
[7]         Q_ACTION_CAST(&ToasterOven_doorClosed_e), // entry
[8]         Q_ACTION_CAST(&ToasterOven_doorClosed_i), // initial tran.
[9]         Q_ACTION_NULL // zero terminator
}
};

[10] return QM_TRAN_INIT(&tatbl_);
}
```

[1] The top-most initial pseudostate function signature (see also [SDS\\_QA\\_QMsm\\_decl](#) [9])

[2] The top-most initial pseudostate initializes internal variables (see also [SDS\\_QA\\_QMsm\\_decl](#) [3])

[3] The top-most initial pseudostate initializes state histories for all states with history (see also [SDS\\_QA\\_QMsm\\_hist](#))

[4] If [QS](#) software tracing is used, the top-most initial pseudostate produces function dictionaries for all its state-handler functions

[5–9] The top-most initial pseudostate declares a static and const struct called "tran-action table", which specifies the sequence of actions to execute:

- [6] the target state of the initial transition
- [7] the entry action to that target state
- [8] the nested initial transition in that target state
- [9] the "zero terminator" that indicates the end of the "tran-action table"

### Note

The "tran-action table" speeds up the transition execution because all actions are preconfigured at compile-time rather than being discovered at runtime as in the case of the QHsm-based state machine implementation strategy. Being defined as both `static` and `const`, the "tran-action table" can be allocated in ROM and takes no precious RAM.

[10] The top-most initial pseudostate must end with the top-most initial transition, which is coded with the [QM\\_TRAN\\_INIT\(\)](#) macro. The argument of this macro is the "tran-action table" specified in steps [5–9] (see also [Figure SDS-SM](#) [1])

### Backward Traceability

- [SAS\\_QP\\_FRM](#): *QP Framework layer*

### Forward Traceability (truncated to 2 level(s))

### 7.6.2.3 SDS\_QA\_QMsm\_state

Implementing a state and its nesting.

#### Description

States in the QMsm-based implementation strategy are const data structures of type `QMState` that store:

1. superstate of a given state
2. state-handler function
3. state entry action function
4. state exit action function
5. state initial action function

The QP event processor uses this information to perform state transitions according to the hierarchical state machine semantics specified in [SRS\\_QP\\_SM\\_30](#). Specifically, the action functions are called through the "tran-action tables" to exit the previous state configuration and enter the next state configuration.

#### Example

The following example code shows two state data structures and the corresponding state-handler functions. The first one implements a top-level state. The second one implements a state that is explicitly nested in another state.

#### Note

The general structure of state-handler functions is the same regardless of their nesting level.

```
// top-level state
[1] QMState const ToasterOven_doorClosed_s = {
[2a]     QM_STATE_NULL, // superstate (top)
[3]     Q_STATE_CAST(&ToasterOven_doorClosed),
[4]     Q_ACTION_CAST(&ToasterOven_doorClosed_e),
[5]     Q_ACTION_CAST(&ToasterOven_doorClosed_x),
[6]     Q_ACTION_CAST(&ToasterOven_doorClosed_i)
};

[10] QState ToasterOven_doorClosed(ToasterOven * const me, QEvt const * const e) {
[11]     QState status_;
[12]     switch (e->sig) {
[13]         default: {
[14]             status_ = QM_SUPER();
[15]             break;
[16]         }
[17]     }
[18]     return status_;
}

// state nested in another state
[1] QMState const ToasterOven_off_s = {
[2b]     &ToasterOven_doorClosed_s, // superstate doorClosed
[3]     Q_STATE_CAST(&ToasterOven_off),
[4]     Q_ACTION_CAST(&ToasterOven_off_e),
[5]     Q_ACTION_NULL, // no exit action
[6]     Q_ACTION_NULL // no initial tran.
};

[10] QState ToasterOven_off(ToasterOven * const me, QEvt const * const e) {
[11]     QState status_;
[12]     switch (e->sig) {
[13]         default: {
[14]             status_ = QM_SUPER();
[15]             break;
[16]         }
[17]     }
[18]     return status_;
}
```

[1-6] The const state data struct for the ToasterOven state "doorClosed" is defined and initialized with:

- [2a] The superstate of this state ([QM\\_STATE\\_NULL](#) in case of a top-level state)
  - [2b] The superstate of this state (pointer to the state data struct in case of a state nested in another state)
  - [3] The state-handler function pointer to process application-defined events
  - [4] The action-handler function pointer to execute upon the entry to the state (might be [Q\\_ACTION\\_NULL](#))
  - [5] The action-handler function pointer to execute upon the exit from the state (might be [Q\\_ACTION\\_NULL](#))
  - [6] The action-handler function pointer to execute upon the nested initial transition in the state (might be [Q\\_ACTION\\_NULL](#))
- [10] The state-handler function to process application-defined events  
 [11] The local status\_ variable to return at the end of the state-handler function  
 [12] The state-handler function is structured as a `switch` statement that discriminates based on the signal of the current event `e->sig` (see [SDS\\_QP\\_QEvt](#)).  
 [13] The `switch` must always have the default case, which should be executed **without side effects**.  
 [14] The default case must set the `status_` to the macro [QM\\_SUPER\(\)](#).

#### Backward Traceability

- [SAS\\_QP\\_FRM](#): QP Framework layer

#### Forward Traceability (truncated to 2 level(s))

---

#### 7.6.2.4 SDS\_QA\_QMsm\_entry

Implementing state entry action.

##### Description

State entry action (see [Figure SDS-SM \[3\]](#)) is coded as a separate member function of the state machine class. The QP event processor executes this state-entry action function through the "tran-action table" when a state transition is taken. Please note that the state-entry action function has no access to the original event that triggered the state transition.

##### Example

The following example code shows a state-handler function with an entry action.

```
[1] QState ToasterOven_doorClosed_e(ToasterOven * const me) {
[2]     // any actions...
[3]     return QM_ENTRY(&ToasterOven_doorClosed_s);
}
```

- [1] The state-entry action function is a member of the state machine class.  
 [2] The state action function calls all the actions to be executed upon the entry to the state.  
 [3] The state-entry action function must return the [QM\\_ENTRY\(\)](#) macro, with the argument being the state struct corresponding to the state being entered.

#### Backward Traceability

- [SAS\\_QP\\_FRM](#): QP Framework layer

#### Forward Traceability (truncated to 2 level(s))

---

### 7.6.2.5 SDS\_QA\_QMsm\_exit

Implementing state exit action.

#### Description

State exit action (see [Figure SDS-SM \[3\]](#)) is coded as a separate member function of the state machine class. The QP event processor executes this state-exit action function through the "tran-action table" when a state transition is taken. Please note that the state-exit action function has no access to the original event that triggered the state transition.

#### Example

The following example code shows a state-handler function with an exit action.

```
[1] QState ToasterOven_doorClosed_x(ToasterOven * const me) {
[2]     // any actions...
[3]     me->hist_doorClosed = QMsm_stateObj(Q_MSM_UPCAST(me));
[4]     return QM_EXIT(&ToasterOven_doorClosed_s);
}
```

[1] The state-exit action function is a member of the state machine class.

[2] The state action function calls all the actions to be executed upon the exit from the state.

[3] If the state has a history transition, the state-exit action function must set the state history variable.

[4] The state-exit action function must return the [QM\\_EXIT\(\)](#) macro, with the argument being the state struct corresponding to the state being exited.

#### Backward Traceability

- [SAS\\_QP\\_FRM](#): QP Framework layer

#### Forward Traceability (truncated to 2 level(s))

---

### 7.6.2.6 SDS\_QA\_QMsm\_nest\_init

Implementing nested initial transition.

#### Description

Nested initial transition (see [Figure SDS-SM \[5\]](#)) is coded as a separate member function of the state machine class. The QP event processor executes this initial transition action function through the "tran-action table" when a state transition is taken. Please note that the initial transition function has no access to the original event that triggered the state transition.

#### Example

The following example code shows a initial transition function.

```
[1] QState ToasterOven_doorClosed_i(ToasterOven * const me) {
[2]     me->temp = BSP_temp();

[3]     static struct {
[4]         QMState const *target;
[5]         QActionHandler act[2];
[6]     } const tatbl_ = { // tran-action table
[7]         &ToasterOven_off_s, // target state
[8]         {
[9]             Q_ACTION_CAST(&ToasterOven_off_e), // entry
[10]            Q_ACTION_NULL // zero terminator
[11]         }
[12]     };
[13]     return QM_TRAN_INIT(&tatbl_);
}
```

[1] The initial-transition function has the usual signature of an action-handler.

[2] Any actions on this initial transition are executed

[3–6] The static and const struct called "tran-action table" specifies the sequence of actions to execute:

- [4] the target state of the initial transition
- [5] the entry action to that target state
- [6] the "zero terminator" that indicates the end of the "tran-action table"

[7] The initial transition function must end with the `QM_TRAN_INIT()` macro. The argument of this macro is the "tran-action table" specified in steps [3–6] (see also [Figure SDS-SM \[5\]](#)).

#### Backward Traceability

- [SAS\\_QP\\_FRM](#): QP Framework layer

#### Forward Traceability (truncated to 2 level(s))

---

### 7.6.2.7 SDS\_QA\_QMsm\_tran

Implementing state transition.

#### Description

A state transition (see [Figure SDS-SM \[6\]](#)) is coded as special case in the state-handler function.

#### Example

The following example code shows a state-handler function with a state transition.

```
QState ToasterOven_doorClosed(ToasterOven * const me, QEvt const * const e) {
    QState status_;
    switch (e->sig) {
        .
        .
[1]        case OPEN_SIG: {
            ToasterOven_actionA(me);
            static struct {
                QMState const *target;
                QActionHandler act[3];
[3]            } const ttabl_ = { // tran-action table
[4]                &ToasterOven_doorOpen_s, // target state
                {
[5]                    Q_ACTION_CAST(&ToasterOven_doorClosed_x), // exit
[6]                    Q_ACTION_CAST(&ToasterOven_doorOpen_e), // entry
[7]                    Q_ACTION_NULL // zero terminator
                }
            };
            status_ = QM_TRAN(&ttabl_);
            break;
        }
        .
        .
    }
    return status_;
}
```

[1] State with a transition needs a case-statement on the user-defined signal (transition trigger).

[7] Any actions on this initial transition are executed

[3–7] The static and const struct called "tran-action table" specifies the sequence of actions to execute:

- [4] the target state of the initial transition
- [5] the exit action from the source state
- [6] the entry action to that target state

- [7] the "zero terminator" that indicates the end of the "tran-action table"
- [8] The transition must be requested by means of the `QM_TRAN()` macro. The argument of this macro is the "tran-action table" specified in steps [3–7]

#### Backward Traceability

- `SAS_QP_FRM`: *QP Framework layer*

#### Forward Traceability (truncated to 2 level(s))

---

### 7.6.2.8 SDS\_QA\_QMsm\_intern

Implementing internal transition.

#### Description

An internal transition (see [Figure SDS-SM \[7\]](#)) is coded as special case-statement in the state-handler function.

#### Example

The following example code shows a state-handler function with an internal transition.

```
QState ToasterOven_off(ToasterOven * const me, QEvt const * const e) {
    QState status_;
    switch (e->sig) {
        . .
        [1]     case SETUP_SIG: {
            [2]         me->setup = Q_EVT_CAST(SetupEvt)->setup;
            [3]         status_ = QM_HANDLED();
            [4]         break;
        }
        . .
    }
    return status_;
}
```

[1] State with a transition needs a case-statement on the user-defined signal (transition trigger).

[2] Inside the case, you code all the actions to be called upon the transition.

[3] The action list must end with setting the status to the `QM_HANDLED()` macro, which informs the QP event processor that the *internal*/transition has been handled.

#### Backward Traceability

- `SAS_QP_FRM`: *QP Framework layer*

#### Forward Traceability (truncated to 2 level(s))

---

### 7.6.2.9 SDS\_QA\_QMsm\_choice

Implementing choice point and guard conditions.

#### Description

A choice point with guard conditions (see [Figure SDS-SM \[8\]](#)) is coded as special case for the user-defined signal (transition trigger).

### Example

The following example code shows a state-handler function with an internal transition.

```

QState ToasterOven_heating(ToasterOven * const me, QEvt const * const e) {
    QState status_;
    switch (e->sig) {
        .
        .
        [1]     case TIMEOUT_SIG: {
            [2]         me->temp = BSP_temp();
            [3]         if (me->temp >= LIMIT) {
            [4]             // any actions...
            [5]             static struct {
                QMState const *target;
                QActionHandler act[3];
            } const tatbl_ = { // tran-action table
                &ToasterOven_off_s, // target state
                {
                    Q_ACTION_CAST(&ToasterOven_heating_x), // exit
                    Q_ACTION_CAST(&ToasterOven_off_e), // entry
                    Q_ACTION_NULL // zero terminator
                }
            };
            status_ = QM_TRAN(&tatbl_);
        }
        [6]     else {
            [7]         // any actions...
            [8]         status_ = QM_HANDLED();
        }
        [9]     break;
    }
    .
    .
}
Q_UNUSED(me);
return status_;
}

```

[1] State-handler needs a case-statement on the user-defined signal (transition trigger).

[2] Inside the case, you code all the actions to be called before any guard conditions are evaluated.

[3] The guard condition is represented as a condition of the `if` statement.

[4] any actions to be executed if the guard evaluates to 'true'.

[5] as usual in all `QMsm` transitions, every transition needs a "tran-action table"

[6] state transition to a regular state is coded as usual with the `QM_TRAN()` macro with the argument being the "tran-action table".

[7] the complementary guard condition to all evaluated before is specified with the `else` statement.

### Note

A choice point might have many outgoing guards, each specified in `else if (guard)`

[8] any actions to be executed for this guard condition.

[9] The action list must end with setting the status to the `QM_HANDLED()` macro, which informs the QP event processor that the *internal* transition has been handled.

### Backward Traceability

- `SAS_QP_FRM`: *QP Framework layer*

### Forward Traceability (truncated to 2 level(s))

#### 7.6.2.10 SDS\_QA\_QMsm\_hist

Implementing state history.

### Description

State history (see [Figure SDS-SM \[11\]](#)) requires remembering the most recent active substate when the given state is exited, which is accomplished in the exit action from the state coded as already described in [SDS\\_QA\\_QMsm\\_exit](#).

### Example

The following example code shows an exit action that additionally saves the state history.

```
[1] QState ToasterOven_doorClosed_x(ToasterOven * const me) {
[2]     // any actions...
[3a]     // save deep history
[3b]     me->hist_doorClosed = QMsm_stateObj(Q_MSM_UPCAST(me));
[3b]     me->hist_doorClosed = QMsm_childStateObj(Q_MSM_UPCAST(me),
[3b]                                     &TstSM_doorClosed_s);
[4]     return QM_EXIT(&ToasterOven_doorClosed_s);
}
```

[1] The state-exit action function is a member of the state machine class.

[2] The state action function calls all the actions to be executed upon the exit from the state.

[3] If the state has a history transition, the state-exit action function must set the state history variable. The [QMsm](#) base class provides two member functions to obtain the current state:

- [QMsm\\_stateObj](#) to obtain the currently active state for **deep history**
- [QMsm\\_childStateObj\(\)](#) to obtain the immediate active child state of the given state for **shallow history**

[4] The state-exit action function must return the [QM\\_EXIT\(\)](#) macro, with the argument being the state struct corresponding to the state being entered.

### Backward Traceability

- [SAS\\_QP\\_FRM](#): *QP Framework layer*

### Forward Traceability (truncated to 2 level(s))

---

#### 7.6.2.11 SDS\_QA\_QMsm\_hist\_tran

Implementing transition to history.

### Description

Transition to state history (see [Figure SDS-SM \[12\]](#)) is coded similar as a regular state transition, but uses a different macro to take the transition [QM\\_TRAN\\_HIST\(\)](#).

### Example

The following example code shows a transition to state history.

```
QState ToasterOven_doorOpen(ToasterOven * const me, QEvt const * const e) {
    QState status_;
    switch (e->sig) {
        .
        .
        case CLOSE_SIG: {
            // any actions...
            static struct {
                QMState const *target;
                QActionHandler act[3];
            } const tabtbl_ = { // tran-action table
                &ToasterOven_doorClosed_s, // target state
                {
                    Q_ACTION_CAST(&ToasterOven_doorOpen_x), // exit

```

```
        Q_ACTION_CAST(&ToasterOven_doorClosed_e), // entry
        Q_ACTION_NULL // zero terminator
    }
};

[4]     status_ = QM_TRAN_HIST(me->hist_doorClosed, &tatbl_);
break;
}
. . .
}

return status_;
}
```

[1] State with a transition to history needs a case-statement for the user-defined signal (transition trigger).

[2] Inside the case, you code all the actions to be called upon the transition.

[3] The static and const struct called "tran-action table" specifies the sequence of actions to execute.

[4] The transition to history must be requested by means of the `QM_TRAN_HIST()` macro. The arguments of this macro are the history variable for the given state and the "tran-action table" specified in step [3]

#### Backward Traceability

- `SAS_QP_FRM`: *QP Framework layer*

#### Forward Traceability (truncated to 2 level(s))

---

## 7.7 Time Viewpoint

The **Time Viewpoint** focuses on the specific requirements and constraints of real-time event-driven systems, ensuring that they meet timing and performance criteria. This design viewpoint frames the following design concerns:

- time management mechanisms
- system clock tick rates
- concurrency
- performance

### 7.7.1 Time Event Life Cycle

#### 7.7.1.1 SDS\_QP\_TELC

Time Event Life Cycle

Model Kind

The time event life cycle is illustrated with UML state diagram.

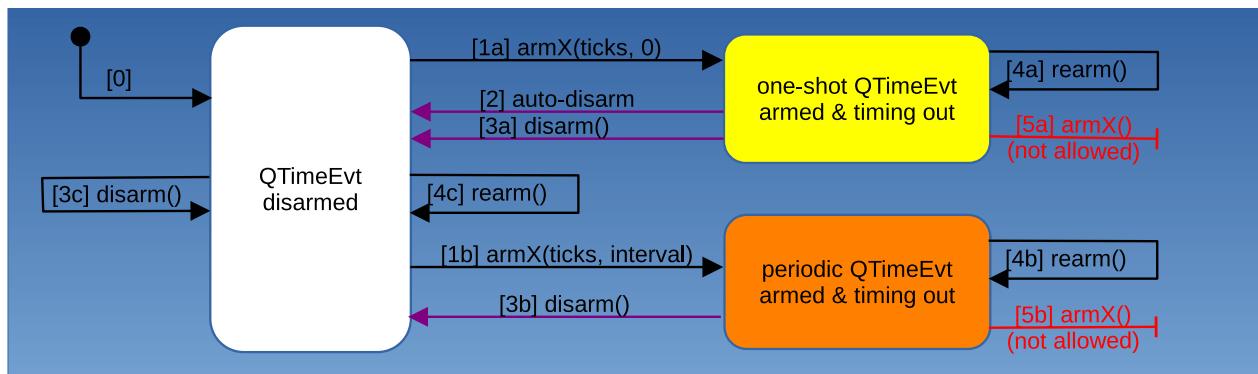


Figure SDS-TE-LIFE: Time Event Life Cycle

Figure SDS-TE-LIFE illustrates the time event life cycle:

- [0] Time Event constructed but disarmed.
- [1a] the `armX()` operation with the 'interval' argument of 0 arms a one-shot time event
- [1b] the `armX()` operation with the 'interval' argument of non-zero arms a periodic time event
- [2] a one-shot Time Event is automatically disarmed after it expires
- [3a] a one-shot Time Event can be disarmed while it is armed and still timing out (the `disarm()` operation returns 'true').
- [3b] a periodic Time Event can be disarmed when it is armed (the `disarm()` operation returns 'true')
- [3c] an already disarmed Time Event can be disarmed (the `disarm()` operation returns 'false')
- [4a] a one-shot Time Event can be rearmed while it is armed and still timing out (the `rearm()` operation returns 'true').
- [4b] a periodic Time Event can be rearmed while it is armed (the `rearm()` operation returns 'true').
- [4c] a disarmed Time Event can be rearmed (the `rearm()` operation returns 'false').
- [5a] arming an already armed one-shot Time Event is NOT allowed (QP framework asserts)
- [5b] arming an already armed periodic Time Event is NOT allowed (QP framework asserts)

Backward Traceability

Forward Traceability (truncated to 2 level(s))

---

## 7.8 Algorithm Viewpoint

The **Algorithm Viewpoint** focuses on the internal details and logic of design entities. This design viewpoint frames the following design concerns:

- analysis of algorithms in regard to their correctness under various scenarios
- analysis of algorithms in regard of their efficiency and time-space performance
- analysis of algorithms in regard of their testability

### 7.8.1 QHsm Implementation View

The [QHsm](#) class (see [Figure SDS-CLS \[13\]](#)) implements the state machine strategy optimized for "manual coding" (see [SRS\\_QP\\_SM\\_20](#)). The [QHsm Design View](#) describes how *QP Application* implements hierarchical state machines based on the [QHsm](#) class. This section describes the internal design of the [QHsm](#) class.

#### 7.8.1.1 SDS\_QP\_QHsm\_ctor

[QHsm](#) Constructor

Description

The [QHsm](#) constructor associates the state machine with its [top-most initial-transition](#). The top-most initial transition is not actually executed at this point.

Pseudocode

```
void QHsm_ctor(QHsm * const me, QStateHandler const initial);
```

---

#### 7.8.1.2 SDS\_QP\_QHsm\_init

[QHsm](#) Top-most Initial Transition

Description

The top-most initial transition in a hierarchical state machine might be complex because the UML semantics require "drilling" into the state hierarchy with any nested initial transitions encountered in the target states until the leaf state is reached. Unfortunately, this direction of navigation through the state hierarchy is opposite to the natural direction. As you recall, a [hierarchical state-handler function](#) provides the *superstate*, so it's easy to traverse the state hierarchy from substates to superstates. Although this order is very convenient for the efficient implementation of the most frequently used [dispatch operation](#), entering states is harder. The solution implemented in the [QHsm](#) class is to use an automatic array `path[]` to record the **exit path** from the target state of the initial transition source, without executing any actions. This is achieved by calling the state handlers with the reserved `QEP_EMPTY_SIG` signal, which causes every state handler to immediately return the superstate without executing any other actions. These superstates are saved in the `path[]` array until the current state is reached. The stored `path[]` array is subsequently played backward to *enter* the target substates (from superstates to substates), which is the exact reversed order in which they were visited.

**Pseudocode**

```
[1] void QHsm_init_(QHsm * const me, void const * const par) {
    // execute the top-most initial tran.
[2]     r = (*me->temp.fun)(me, par);

    // find all superstates of the target state and store it in array path[]
[3]     . . .

    // drill down into the state hierarchy with initial transitions...
[4]     QHsm_enter_target_(me, &path[0], ip);
}
```

---

**7.8.1.3 SDS\_QP\_QHsm\_dispatch****QHsm** Event Dispatching**Description****7.8.1.4 SDS\_QP\_QHsm\_tran-simple****QHsm** Simple Transition**Description****7.8.1.5 SDS\_QP\_QHsm\_tran-complex****QHsm** Complex Transition**Description****7.8.2 QMsm Implementation View**

The **QMsm** class (see [Figure SDS-CLS \[14\]](#)) derived from **QAsm** implements the state machine strategy optimized for "automatic code generation" (see [SRS\\_QP\\_SM\\_21](#)). The [QMsm Design View](#) describes how QP Applications implement hierarchical state machines *based* on the **QMsm** class. This section describes how the the **QMsm** class itself is **internally implemented**.

**7.8.2.1 SDS\_QP\_QMsm\_ctor****QMsm** Constructor**Description**

### 7.8.2.2 SDS\_QP\_QMsm\_init

QMsm Top-most Initial Transition

Description

---

### 7.8.2.3 SDS\_QP\_QMsm\_disp

QMsm Event Dispatching

Description

---

### 7.8.2.4 SDS\_QP\_QMsm\_tran

QMsm Transition

Description

---

### 7.8.2.5 SDS\_QP\_QMsm\_tat

QMsm Transition Action Tables

Description

---

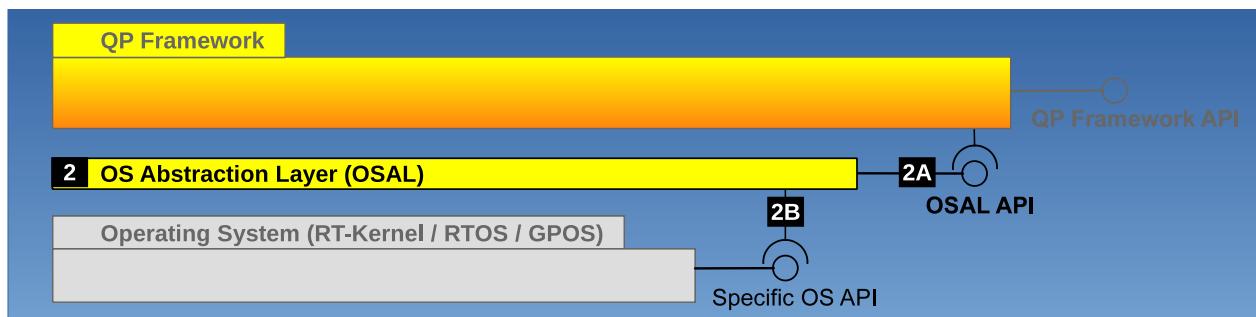
## 7.9 Interface Viewpoint

The **Interface Viewpoint** focuses on defining and describing the interfaces between QP Framework and the outside layers, such as the underlying operating system (OS Abstraction Layer) and the QP Application.

The **OS Abstraction Layer (OSAL)** shown in [Figure SDS-OSAL](#), insulates QP Framework from the underlying Operating System (OS), such as one of the built-in real-time kernels, 3rd-party RTOS, or a General-Purpose OS. The OSAL provides both the abstract *Operating System API* (OSAL API), see [Figure SDS-OSAL \[2A\]](#), and the *implementation* of this API for the specific OS ([Figure SDS-OSAL \[2B\]](#)). The main idea of an OSAL is that the *OSAL API* is fixed and is used inside the QP Framework source code, while the OSAL implementation (called the "QP port") depends on the specific underlying OS. In other words, there is a separate "QP port" for every supported real-time kernel, 3rd-party RTOS or GPOS. The main focus of this section is to describe the design of the *OSAL API*, with the specific examples are taken from various "QP ports" implementing the OSAL API.

### Remarks

The OSAL is an example of the "Facade" design pattern [[GoF:94](#)].



*Figure SDS-OSAL: QP Port implementing OS Abstraction Layer.*

### 7.9.1 Critical Section

**Critical section** is a non-blocking mutual exclusion mechanism that protects a sequence of instructions from preemption. QP Framework, just like any other system-level software, must internally use critical sections to guarantee *concurrency-safe* operation. The critical section implementation depends on the underlying OS, so the QP OSAL must provide an efficient critical section *abstraction* to insulate QP Framework from the specifics of the underlying OS. For example, in single-core embedded systems, critical sections are typically implemented by disabling interrupts upon the entry to the section and re-enabling them upon the exit. This is effective because in such systems interrupts are the only way preemption can be initiated (both interrupt preemption but also thread preemption in a real-time kernel). However, other systems (e.g., multicore) might use a different critical section mechanism (e.g. spin-lock). Some other systems (e.g., general purpose OS) might implement user-space critical sections with the traditional mutual-exclusion mechanism, such as a mutex. The critical section abstraction in QP OSAL must accommodate all such critical section implementations.

#### Note

The maximum time spent in a critical section directly affects the system's responsiveness to external events (interrupt latency and task-level response). Therefore, all critical sections inside QP Framework source code are carefully designed to be as short as possible and are of the same order as critical sections in any commercial RTOS. Of course, the length of critical sections depends on the actual critical section mechanism and the quality of the code generated by the compiler.

#### 7.9.1.1 SDS\_QP\_CRIT

##### Critical Section API

## Description

The critical section abstraction in QP OSAL consists of three parameter-less macros: critical-section status, critical-section-entry and critical-section-exit.

```
#define QF_CRIT_STAT    ...
#define QF_CRIT_ENTRY()   ...
#define QF_CRIT_EXIT()   ...
```

These macros support two main types of critical section implementations:

1. Simple critical section entry/exit (empty definition of the macro QF\_CRIT\_STAT);
2. Save-and-restore critical section status (not-empty definition of the macro QF\_CRIT\_STAT).

Internally, QP Framework uses the critical section macros in the following generic way:

```
. . . // outside critical section
[1] QF_CRIT_STAT
. . . // outside critical section
[2] QF_CRIT_ENTRY();
[3] . . . // inside critical section
[4] QF_CRIT_EXIT();
. . . // outside critical section
```

[1] If the macro QF\_CRIT\_STAT is not-empty, it allocates a local critical-section-status variable. Otherwise, if the macro is empty, the line does nothing.

[2] the macro QF\_CRIT\_ENTRY() enters critical section, which saves the critical section status into the status-variable (if it was defined at step [1])

[3] this code executes inside the critical section protection

[4] the macro QF\_CRIT\_EXIT() exits critical section, which restores the critical section status from the status variable (if it was defined at step [1])

## Examples

Example simple critical section macros without critical section status (ARM Cortex-M CPU):

```
#define QF_CRIT_STAT      // empty
#define QF_CRIT_ENTRY()   (asm volatile ("cpsid i"))
#define QF_CRIT_EXIT()   (asm volatile ("cpsie i"))
```

Example save-and-restore critical section status macros (MSP430 CPU):

```
#define QF_CRIT_STAT      unsigned short int_state_;
#define QF_CRIT_ENTRY() do {           \
    int_state_ = __get_interrupt_state(); \
    __disable_interrupt();             \
} while (false)
#define QF_CRIT_EXIT()  __set_interrupt_state(int_state_)
```

## Nesting Critical Sections

Some CPUs or operating systems require nesting of one critical section in another critical section. The "save-and-restore critical section status" method enables such critical section nesting in an explicit manner. But even the "simple critical section" method can be implemented in way that allows critical section nesting. For example, the critical section can use an internal up-down nesting counter, which will guarantee that only the last nesting level truly exits the critical section.

## Forward Traceability (truncated to 2 level(s))

### 7.9.2 Active Object OSAL

TBD...

# Chapter 8

## API Reference

### 8.1 Compile-Time Configuration

QP/C Framework supports compile-time configuration through the following mechanisms:

- command-line macros
- `qp_config.h` header file
- `qp_port.h` header file
- `qs_port.h` header file

### 8.2 Active Objects

QP/C Framework supports [Active Objects](#) through the following base classes.

`QActive` class (derived from `QAsm`)

- `QActive_ctor()`
- `QActive_start()`
- `QACTIVE_POST()`
- `QACTIVE_POST_X()`
- `QACTIVE_POST_LIFO()`
- `QActive_defer()`
- `QActive_recall()`
- `QActive_flushDeferred()`
- `QActive_stop()`

`QMActive` class (derived from `QActive`)

- `QMActive_ctor()`

**Publish-Subscribe**

- `QActive_psInit()`
- `QActive_subscribe()`
- `QActive_unsubscribe()`
- `QActive_unsubscribeAll()`
- `QACTIVE_PUBLISH()`

## 8.3 Events

QP/C Framework supports [events](#), through the following bases classes:  
**QEvt** class (see also [Event Management](#) and [Time Management](#))

- [QEvt](#)
- [QEVT\\_INITIALIZER\(\)](#) - initializer for immutable events
- [Q\\_EVT\\_CAST\(\)](#) - event down-casting

## 8.4 State Machines

QP/C Framework supports [event-driven, hierarchical state machines](#) through the following bases classes:  
**QAsm** class

- [QASM\\_INIT\(\)](#)
- [QASM\\_DISPATCH\(\)](#)
- [QASM\\_IS\\_IN\(\)](#)

**QHsm** class (derived from [QAsm](#))

- [QHsm\\_ctor\(\)](#)
- [QHsm\\_state\(\)](#)
- [QHsm\\_top\(\)](#)
- [Q\\_STATE\\_CAST\(\)](#)

**QMsm** class (derived from [QAsm](#))

- [QMsm\\_ctor\(\)](#)
- [QMsm\\_stateObj\(\)](#)

## 8.5 Mutable Event Management

- [QF\\_poolInit\(\)](#)
- [Q\\_NEW\(\)](#)
- [Q\\_NEW\\_X\(\)](#)
- [QEvt\\_init\(\)](#)
- [Q\\_NEW\\_REF\(\)](#)
- [Q\\_DELETE\\_REF\(\)](#)
- [QF\\_gc\(\)](#)

**QMPool** class (memory pool, also used for mutable events)

- [QMPool\\_init\(\)](#)
- [QMPool\\_get\(\)](#)
- [QMPool\\_put\(\)](#)

## 8.6 Time Management

[QTimeEvt class](#) (derived from [QEvt](#))

- [QTIMEEVNT\\_TICK\(\)](#)
- [QTimeEvt\\_ctorX\(\)](#)
- [QTimeEvt\\_armX\(\)](#)
- [QTimeEvt\\_disarm\(\)](#)
- [QTimeEvt\\_rearm\(\)](#)
- [QTimeEvt\\_currCtr\(\)](#)

[QTicker active object](#) (derived from [QTimeEvt](#))

## 8.7 Event Queues (raw thread-safe)

[QEQueue class](#)

- [QEQueue\\_init\(\)](#)
- [QEQueue\\_post\(\)](#)
- [QEQueue\\_postLIFO\(\)](#)
- [QEQueue\\_get\(\)](#)
- [QEQueue\\_getNFree\(\)](#)
- [QEQueue\\_getNMin\(\)](#)
- [QEQueue\\_isEmpty\(\)](#)

## 8.8 Software Tracing

[QS](#) is a software tracing system that enables developers to monitor live event-driven QP Applications with minimal target system resources and without stopping or significantly slowing down the code. [QS](#) is an ideal tool for testing, troubleshooting, and optimizing QP Applications. [QS](#) can even be used to support acceptance testing in product manufacturing.

### 8.8.1 QS-Transmit (QS-TX)

- [QS\\_INIT\(\)](#)
- [QS\\_initBuf\(\)](#)
- [QS\\_getByte\(\)](#)
- [QS\\_getBlock\(\)](#)
- [QS\\_onStartup\(\)](#)
- [QS\\_onCleanup\(\)](#)
- [QS\\_onFlush\(\)](#)
- [QS\\_onGetTime\(\)](#)

### QS Filters

- [QS\\_GLB\\_FILTER\(\)](#)
- [QS\\_LOC\\_FILTER\(\)](#)
- [QS\\_FILTER\\_AP\\_OBJ\(\)](#)

### QS Dictionaries

- [QS\\_SIG\\_DICTIONARY\(\)](#)
- [QS\\_OBJ\\_DICTIONARY\(\)](#)
- [QS\\_OBJ\\_ARR\\_DICTIONARY\(\)](#)
- [QS\\_FUN\\_DICTIONARY\(\)](#)
- [QS\\_USR\\_DICTIONARY\(\)](#)
- [QS\\_ENUM\\_DICTIONARY\(\)](#)

### QS Application-Specific Records

- [QS\\_BEGIN\\_ID\(\)](#)
- [QS\\_END\(\)](#)
- [QS\\_U8\(\) / QS\\_I8\(\)](#)
- [QS\\_U16\(\) / QS\\_I16\(\)](#)
- [QS\\_U32\(\) / QS\\_I32\(\)](#)
- [QS\\_ENUM\(\)](#)
- [QS\\_STR\(\)](#)
- [QS\\_MEM\(\)](#)
- ::QS\_USER enumeration

## 8.8.2 QS Receive-Channel (QS-RX)

- [QS\\_rxInitBuf\(\)](#)
- [QS\\_rxParse\(\)](#)
- [QS\\_onCommand\(\)](#)
- [QS\\_RX\\_PUT\(\)](#)

## 8.9 QV (Non-preemptive Kernel)

QV is a simple [non-preemptive kernel](#), which executes Active Objects one at a time, with priority-based scheduling performed before processing of each event. Due to naturally short duration of event processing in state machines, the simple [QV](#) kernel is often adequate for many real-time systems.

### 8.9.1 Kernel Initialization and Control

- `QF_run()`
- `QV_schedDisable()`
- `QV_schedEnable()`
- `QV_onIdle()`
- `QV_CPU_SLEEP()`

## 8.10 QK (Preemptive RTC Kernel)

QK is a lightweight preemptive, priority-based, run-to-completion (RTC) kernel designed specifically for executing event-driven active objects.

### 8.10.1 Kernel Initialization and Control

- `QF_run()`
- `QK_schedLock()`
- `QK_schedUnlock()`
- `QK_onIdle()`

#### QK Interrupt Management

- `QK_ISR_ENTRY()`
- `QK_ISR_EXIT()`

## 8.11 QXK (Dual-Mode Kernel)

QXK is a lightweight, preemptive, priority-based, dual-mode (run-to-completion/blocking) kernel that executes active objects like the **QK kernel**, but can also execute traditional **blocking** threads (extended threads). In this respect, **QXK** behaves exactly as a conventional **RTOS** (Real-Time Operating System). **QXK** has been designed specifically for mixing event-driven active objects with traditional blocking code, such as commercial middleware (TCP/IP stacks, UDP stacks, embedded file systems, etc.) or legacy software.

### 8.11.1 Kernel Initialization and Control

- `QF_run()`
- `QXK_onIdle()`
- `QXK_schedLock()`
- `QXK_schedUnlock()`
- `QXK_onIdle()`
- `QXK_current()`

#### QXK Interrupt Management

- `QXK_ISR_ENTRY()`

- [QXK\\_ISR\\_EXIT\(\)](#)

### **QXThread class**

- [QXThread\\_ctor\(\)](#)
- [QXThread\\_start\(\)](#)
- [QXTHREAD\\_POST\\_X\(\)](#)
- [Q\\_XTHREAD\\_CAST\(\)](#)
- [QXThread\\_delay\(\)](#)
- [QXThread\\_delayCancel\(\)](#)
- [QXThread\\_queueGet\(\)](#)

### **QXThread Message Queues**

- [QXTHREAD\\_POST\\_X\(\)](#) - posting messages to blocking threads
- [QACTIVE\\_POST\\_X\(\)](#) - posting events to Active Objects
- [QXThread\\_queueGet\(\)](#)

### **QXSemaphore class**

- [QXSemaphore\\_init\(\)](#)
- [QXSemaphore\\_wait\(\)](#)
- [QXSemaphore\\_tryWait\(\)](#)
- [QXSemaphore\\_signal\(\)](#)

### **QXMutex class**

- [QXMutex\\_init\(\)](#)
- [QXMutex\\_lock\(\)](#)
- [QXMutex\\_tryLock\(\)](#)
- [QXMutex\\_unlock\(\)](#)

### **QMPool class**

- [QMPool\\_init\(\)](#)
- [QMPool\\_get\(\)](#)
- [QMPool\\_put\(\)](#)

# Chapter 9

## Deprecated APIs

The following QP/C APIs are now deprecated:

### Member [char\\_t](#)

plain 'char' is no longer forbidden in MISRA-C:2023

### Member [Q\\_ALLEGGE \(expr\\_\)](#)

general purpose assertion without ID number that **always** evaluates the `expr_` expression. Instead of ID number, this macro is based on the standard `__LINE__` macro.

### Member [Q\\_ALLEGGE\\_ID \(id\\_, expr\\_\)](#)

#Q\_NASSERT preprocessor switch to disable QP assertions

general purpose assertion with user-specified ID number that **always** evaluates the `expr_` expression.

### Member [Q\\_ASSERT\\_COMPILE \(expr\\_\)](#)

Use [Q\\_ASSERT\\_STATIC\(\)](#) or better yet `_Static_assert ()` instead.

### Member [Q\\_onAssert \(module\\_, id\\_\)](#)

Assertion failure handler. Use [Q\\_onError\(\)](#) instead.

### Member [QACTIVE\\_START \(me\\_, prioSpec\\_, qSto\\_, qLen\\_, stkSto\\_, stkSize\\_, par\\_\)](#)

Macro for starting an Active Object. Use [QActive::QActive\\_start\(\)](#) instead.

### Member [QF::QF\\_psInit \(QSubscrList \\*const subscrSto, enum\\_t const maxSignal\)](#)

### Member [QF\\_getQueueMin \(prio\\_\)](#)

instead use: [QActive::QActive\\_getQueueMin\(\)](#)

### Member [QF\\_TICK \(sender\\_\)](#)

superseded by [QTIMEEV\\_TICK\(\)](#)

### Member [QF\\_TICK\\_X \(tickRate\\_, sender\\_\)](#)

superseded by [QTIMEEV\\_TICK\\_X\(\)](#)

**Member [QHSM\\_DISPATCH](#) (me\_, e\_, qslid\_)**

instead use: [QASM\\_DISPATCH\(\)](#)

**Member [QHSM\\_INIT](#) (me\_, par\_, qslid\_)**

instead use: [QASM\\_INIT\(\)](#)

**Member [QHsm\\_isIn](#) (me\_, state\_)**

instead use: [QASM\\_IS\\_IN\(\)](#)

**Member [QXTHREAD\\_START](#) (me\_, prioSpec\_, qSto\_, qLen\_, stkSto\_, stkSize\_, par\_)**

Macro for starting an eXtended Thread. Use QXThread::QXThread\_start() instead.

# Chapter 10

## Revision History

### 10.1 Version 8.0.3, 2025-04-07

#### QP/C source:

- in all built-in real-time kernels ([QV](#), [QK](#), and [QXK](#)) moved the [QF\\_onStartup\(\)](#) callback *after* the kernel initialization (such as `QV_START()`, `QK_START()`, and `QXK_START()`). This allows [QF\\_onStartup\(\)](#) to override the settings from the kernel initialization (e.g., the default interrupt priorities).
- replaced [QF\\_getQueueMin\(\)](#) with [QActive\\_getQueueMin\(\)](#) ([QF\\_getQueueMin\(\)](#) was added to the backward-compatibility APIs)
- updated licensing comments in the QP/C source code.
- removed explicit version number from the source code comments to reduce the churn when no changes are introduced at a given version.
- made minor changes in function argument names and `const` temporary variables for better MISRA:C-2023 compliance.

#### QP/C examples:

- updated FreeRTOS examples to the new FreeRTOS v11.2.0
- improved example `qpc/examples posix-win32/calculator` (Calculator without sub-machine and without repetitions in the operand state)
- improved examples `qpc/examples arm-cm/real-time_nucleo-c031c6` that demonstrate real-time performance of the [QV](#) and [QK](#) kernels

### 10.2 Version 8.0.2, 2025-01-20

This QP/C release marks the creation of the separate [SafeQP/C edition](#) focused on safety-critical applications requiring high integrity levels and specific Safety Functions. The Safety Functions causing overhead (e.g., MPU memory separation and duplicate inverse storage) have been **removed** from this QP/C version and moved to the "SafeQP/C" edition.

#### Note

QP/C 8.0.2 API remains backwards compatible with the previous releases 8.0.0/8.0.1. QP/C API is also shared with the SafeQP/C edition.

#### QP/C source:

- removed support for memory isolation with MPU (Memory Protection Unit)
- removed Safety Functions requiring redundancies, such as
  - Duplicate Inverse Storage
  - Fixed upper loop bounds

**QP/C examples:**

- improved the "Blinky" examples for ARM Cortex-M
- added new examples for STM32Cube (directory `qpc/examples/stm32cube`)

**Bug Fixes:**

- #378 QXK kernel intermittently performs incorrect context switch between extended and basic threads

**Remarks**

The [QXK source code](#) is no longer included with the open-source QP/C distribution, but is available to all commercial licensees and upon [evaluation request↑](#).

## **10.3 Version 8.0.1, 2024-12-17**

**Bug Fixes:**

- #369 QP ARM-CM port fails on ARM Cortex-M23
- #373 QP/C port to POSIX (with P-threads) missing `#include <pthread.h>`
- #372 QTimeEvt::noActive() nests critical sections
- #376 Incorrect critical section exit in QP ports to ARM Cortex-R, GNU-ARM

**QP/C ports:**

- as part of the fix for bug #369, the critical section policy in QP/C ports to ARM Cortex-M has been redesigned (native ports to [QV](#), [QK](#), and [QXK](#) kernels). Specifically, the PRIMASK/BASEPRI policy is no longer tied to the ARM architecture (ARMv6-M vs. ARMv7-M+). Instead, the policy is controlled by the `QF_USE_BASEPRI` macro in the `qf_config.h` header file. If this macro is defined, BASEPRI is used and the value of the `QF_USE_BASEPRI` macro determines the BASEPRI threshold for "kernel aware" ISRs. If `QF_USE_BASEPRI` is not defined, PRIMASK is used instead.
- modified critical section exit in QP/C port to ARM Cortex-R (`ports/arm-cr`), GNU-ARM (bug #376).
- modified QP/C port to POSIX (with P-threads) to include `<pthread.h>` header file.

**QP/C examples:**

- updated all examples to use the new `qp_config.h` header file (with `QF_USE_BASEPRI` and other structural improvements).
- modified QUTest examples (`examples/qutest/` directory) to pass command-line parameters to the test fixtures when compiled for the host. This is part of the bug fix #375 (QUTest script runner doesn't pass arguments to host executable).

## 10.4 Version 8.0.0, 2024-10-31

The main highlights of this milestone QP/C release are:

- Completed the implementation of all identified [Safety Requirements](#) for the functional safety of the QP/C Framework according to [IEC 61508-3:2010]. (This effort has been ongoing since QP/C release 7.0.0)
- Re-designed various functional areas to improve the *systematic software capability* of QP/C Framework to achieve SIL-3. The most important change here is **removal of sub-machines** and the related elements to reduce the cyclomatic complexity for testability, as identified in the [QP/C Software Hazard and Risk Analysis](#). (This change corresponds to the matching [QM version 7↑](#), which also removed the sub-machines feature.)
- Due to the growing non-compliance with the GPL as well as infringement on the [dual-licensing model of QP frameworks↑](#), the following QP/C components have been **removed from the open-source GPL distribution**:
  - [QS](#) target-resident software tracing component
  - [QXK](#) dual-mode kernel

### Attention

The complete QP/C source code will be provided to all **commercial QP/C licensees** ([upon request↑](#)) and will also be **readily available for evaluation** ([upon request↑](#)).

### QP/C source code:

- Simplified the [QMsm](#) implementation by removing the code for sub-machines, sub-machine states, entry-points, and exit-points.
- Re-designed and added the Fixed Upper Loop Bound monitoring to all not explicitly bounded loops in the [QHsm](#) and [QMsm](#) implementations.
- Re-designed transition-to-history handling in [QHsm](#) and [QMsm](#), and particularly [software tracing](#) of such transitions (to achieve identical trace behavior in [QHsm](#) and [QMsm](#) implementations).
- Re-designed self-monitoring of [QEvt](#) instances by replacing the Memory Marker with Duplicate Inverse Storage (DIS) for the volatile `QEvt::evtTag_` member. Made corresponding changes to verify the DIS invariants for [QEvt](#) instances throughout the code.
- Added Duplicate Inverse Storage (DIS) monitoring to the volatile members of [QEQueue](#) and active object queue. Added code to update and verify the DIS invariants with assertions.
- Added Duplicate Inverse Storage (DIS) monitoring to the volatile members of [QMPool](#). Added code to update and verify the DIS invariants with assertions.
- Added Duplicate Inverse Storage (DIS) monitoring to the volatile members of [QTimeEvt](#). Added code to update and verify the DIS invariants with assertions.
- Improved the code for the Safety Functions related to spatial isolation between software components by means of the MPU (Memory Protection Unit). Specifically, ensured that the mode changes between System and Application access rights are perfectly balanced. Also, added mode changes in the startup phase (changed for the ARMv8-M MPU that requires non-overlapping MPU regions).
- Re-designed the automatic initialization of dynamic events ("RAII" for dynamic events). Specifically, when the macro [QEVT\\_PAR\\_INIT](#) (previously macro [QEVT\\_DYNCTOR](#)) is defined, the dynamic allocation automatically calls the member function `QEvt_init()` (previously `QEvt_ctor()`) that must be provided in the event class (subclass of [QEvt](#)).
- Re-designed the target-info [QS](#) trace record (`::QS_TARGET_INFO`) to report both the QP version number and QP release date. This was necessary for the license management introduced in [QSpy 8.0.0↑](#).

- Removed the following QP/C source files from the open-source GPL distribution:

```

qpc
|
+---include
|   qs.h
|   qs_pkg.h
|   qxk.h
|
\---src
    |
    +---qs
        qs.c
        qs_64bit.c
        qs_fp.c
        qs_rx.c
        qutest.c
    |
    \---qxk
        qxk.c
        qxk_mutex.c
        qxk_sema.c
        qxk_xthr.c

```

#### QP/C ports:

- Changed all QP/C ports to mandate providing the `qp_config.h` header file in all QP/C applications. This avoids a whole class of potential defects caused by inconsistent compile-time options in different software builds. (NOTE: Compile-time configuration file `qp_config.h` has been introduced in the earlier QP/C release, but it was optional and controlled by the `QP_CONFIG` macro. Now `qp_config.h` is obligatory.

#### Attention

This change impacts **all existing QP/C applications**, which now must provide the `qp_config.h` file. The default `qp_config.h` header file can be copied from the `qpc/ports/config` directory.

- Fixed a bug in the `QK` and `QXK` ports to ARM Cortex-M to correctly handle configuration with an external IRQ instead of the default NMI.
- Removed the following QP/C port files from the open-source GPL distribution:

```

qpc
|
\---ports
    +---arm-cm
        +---qxk
            |   +---armclang
            |   |       qxk_port.c
            |   |       qp_port.h
            |   |       qs_port.h
            |
            |   +---config
            |   |       qp_config.h
            |
            |   +---gnu
            |   |       qxk_port.c
            |   |       qp_port.h
            |   |       qs_port.h
            |
            |   \---iar
            |       qxk_port.c
            |       qp_port.h
            |       qs_port.h
            |
            \---qutest
                qp_port.h
                qs_port.h
            |
            +---posix-qutest
                qp_port.h
                qs_port.h
                qutest_port.c
                safe_std.h
            |
            \---win32-qutest
                qp_port.h
                qs_port.h

```

```
qutest_port.c
safe_std.h
```

**QP/C examples:**

- Added `qp_config.h` files to all examples.
- Added examples for the STM32 NUCLEO-U545RE board (ARM Cortex-M33), which is now the primary embedded target for QP/C.
- Added examples of **integration testing** with QUTest in the directory `qpc/examples/qutest/integration_tests`.
- Added examples for **test automation** on the host as well as an embedded target (the STM32 NUCLEO-U545RE) in the directory `qpc/examples/qutest/auto_run`.
- Removed examples for the MPU and moved them to the **SafeQP Certification Kits**. These examples have been improved and extended for the ARMv8-M MPU available in the NUCLEO-U545RE board.

**Bug Fixes:**

- bug#361 `Q_NEW_X()` fails when `QEVT_DYNCTOR` is defined (feature re-designed)
- bug#357 `QXK callback QF_onContextSw()` problem

## 10.5 Version 7.3.4, 2024-03-21

**Bug Fixes:**

- bug#355 Assertions in QK and QV on MSP430

**QP/C source code:**

- modified `qv.c` (fix for bug#355)
- modified `qk.c` (fix for bug#355)

**QP/C ports:**

- removed the esp-idf port, which was NOT officially supported
- updated Cmake support (e.g., files CMakeLists.txt, etc.)

## 10.6 Version 7.3.3, 2024-03-01

The main highlights of this release are:

- cleaned up of numerous spelling mistakes and typos in QP/C source code and documentation
- separated the `qpc/examples` sub-directory as a `Git submodule qpc-examples` to allow external contributions to examples
- added new feature: Cmake support (thanks to `Stefan Schober`)
- bug fixes

**QP/C source code:**

- hardened the `QHsm` and `QMsm` implementations by adding explicit loop limits to prevent "malformed" state machines from crashing the event processor. Instead, added assertions if the loop limits are ever exceeded (see bug#351).

- changed semantics of QS::QS\_rxParse() to be called outside of critical section.
- added critical sections to QS-RX processing.
- added QS::QS\_setCurrObj() for QS software tracing (see bug#350)
- added Cmake support (e.g., files CMakeLists.txt, etc.)

#### QP/C ports:

- changed QUTest ports to General-Purpose OSes (posix-qutest and win32-qutest) NOT to use any console services (see bug#353).
- changed QP/C ports to General-Purpose OSes (posix, posix-qv, win32, and win32-qv) to provide console services only conditionally (when macro QF\_CONSOLE is defined)

#### QP/C examples:

- consistently changed all bsp.c files (Board Support Packages) to call QS\_rxParse() outside critical sections.
- modified examples for posix-win32 to define the macro QF\_CONSOLE when they actually use the console services.
- modified examples for MPU (examples/arm-cm/dpp\_ek-tm4c123gx1\_mpu and examples/arm-cm/dpp\_nucleo-c031c6\_mpu) to configure the MPU for the Idle thread, so that the QS-RX access is allowed.
- added example for Cmake: examples posix-win32-cmake/dpp

#### Bug Fixes:

- bug#353 win32-qutest port uses non-standard include 'conio.h'
- bug#352 Event posting via QS-RX stopped working
- bug#351 QP should not crash or lock-up for "malformed" state machines
- bug#350 QS\_setCurrObj() is missing

## 10.7 Version 7.3.2, 2023-12-13

This maintenance release adds a minor design change to the [Qasm](#) base class and corrects a few issues discovered in QP/C 7.3.1.

#### QP/C source code:

- added "virtual" function [QasmVtable::isIn](#) to the [Qasm](#) base class (see [QasmVtable](#))
- added macro [QASM\\_IS\\_IN\(\)](#) to "virtually" call the [isIn\(\)](#) operation in the *subclasses* of [Qasm](#)
- added implementation of the [QHsm\\_isIn\(\)](#) virtual function to the [QHsm](#) base class. In the process, fixed a bug in that implementation introduced by adding Duplicate Inverse Storage in the [Qasm::temp](#) attribute.
- added implementation of the [QMsm\\_isIn\(\)](#) virtual function to the [QMsm](#) base class
- deprecated the [QMsm\\_isInState\(\)](#) function, which should be now replaced with [QASM\\_IS\\_IN\(\)](#) "virtual" call
- changed the semantics of the [QS\\_onFlush\(\)](#) callback (software tracing), which is now NOT allowed to disable interrupts or use critical sections. This is necessary for avoiding nesting of QP critical sections, when [QS\\_onFlush\(\)](#) is used inside [Q\\_onError\(\)](#) handler (to trace the error).

#### Ports:

- removed critical sections from the [QS\\_onFlush\(\)](#) implementations in POSIX and Win32 ports (files `qs_port.c`).

#### Examples:

- removed interrupt disabling and critical sections from the [QS\\_onFlush\(\)](#) implementations in all BSP files (`bsp.c`).
- cleaned up the KEIL-uVision and IAR-EWARM project files that still contained old QP/C port header files (e.g., `qep_port.h`, `qf_port.h`, `qk_port.h`, `qv_port.h`, and `qxk_port.h`). These files (consolidated and renamed to [qp\\_port.h](#)) were not actually used to build the projects, but their presence was confusing.

## 10.8 Version 7.3.1, 2023-12-05

The focus of this release continues to be improving the support for **functional safety** standards. This release adds several mechanisms to the QP Functional Safety (FuSa) Subsystem to mitigate various potential hazards.

#### QP Functional Safety (FuSa) Subsystem:

- added assertions to critical sections in the QP/C (in ports to: ARM Cortex-M, POSIX, POSIX-QV, WIN32, and WIN32-QV) to ensure that:
  - critical sections inside QP never nest
  - all QP critical sections are "balanced" meaning that every entry to a critical section is matched by exactly one exit from critical section.
- added Duplicate Inverse Storage protection to all internal variables in the [QV](#) and [QK](#) kernels;
- added explicit numerical loop limits for internal loops that traverse linked lists (e.g., time-event lists) or bitmasks (e.g., subscribers in pub-sub event delivery)

#### [QV kernel:](#)

- Added functions to disable/enable the non-preemptive [QV](#) scheduler ([QV\\_schedDisable\(\)](#) and [QV\\_schedEnable\(\)](#)). These functions are intended for situations where a CPU-intensive AO does not wish to be activated until the scheduler is re-enabled (typically in the next clock tick). Such scheduling might improve the timeliness of higher-priority AOs.

#### Ports:

- Redesigned the critical section implementation in ports to: ARM Cortex-M, POSIX, POSIX-QV, WIN32, and WIN32-QV to be based on functions rather than inline macros. This was needed to add the assertions mentioned above. (NOTE this redesign has NO impact on the applications)
- Changed ports to POSIX (POSIX and POSIX-QV) to use only non-recursive mutex for critical sections. This is done after ensuring that the critical sections never nest, so the simple, fast and *portable* non-recursive mutex is sufficient.
- Added templates of the QP configuration file ([qp\\_config.h](#)) to the ARM Cortex-M ports (in the `qpc/ports/arm-cm/<qv|qk|qxk>/config` directories).
- Modified ports to 3rd-party RTOSes (embOS, FreeRTOS, ThreadX, uC-OS2, Zephyr) to allow setting the RTOS-native AO priority as the second parameter of the [Q\\_PRIO\(\)](#) macro.

#### Examples:

- Added "real-time" example `qpc/examples/arm-cm/real-time_nucleo-1053r8` that demonstrates various scheduling policies of periodic and sporadic tasks. This includes multi-stage tasks and time-triggered scheduling under the [QV](#) kernel.

## 10.9 Version 7.3.0, 2023-09-12

The main focus of this QP/C release is improving the support for **functional safety** standards, such as IEC 61508 for electrical systems, and related IEC 62304/FDA510(k) for medical devices, IEC 60335 for household appliances, DO-178B/C for airborne systems.

**QP Functional Safety (FuSa) Subsystem:** This release adds QP Functional Safety Subsystem (QP FuSa), which is an expansion of the assertion-based programming extensively used in the QP Frameworks from the beginning. The QP FuSa Subsystem consists of the following parts:

- Software assertions as a recommended technique (called Failure Assertion Programming (FAP) in IEC 61508)
- Software Self-Monitoring (SSM), which encompasses such techniques:
  - Duplicate Inverse Storage for critical variables
  - Memory Markers for critical objects (e.g., events)
  - **Memory Isolation by means of Memory Protection Unit (MPU)**

### Certification Kit

- Expanded traceability and improved the use of Unique Identifiers (UIDs)
- Completed MISRA-C:2023 Coding Standard Compliance documentation

### Source Code:

- As part of transitioning to C99, changed all comments from C-style /\* \*/ to C++ style //. C++ style comments are now part of the C99 standard and are allowed in MISRA-C:2012/2023. This brings more commonality between QP/C and QP/C++.
- Removed header file "qassert.h" and moved the assertions to the file "qsafe.h", which now is part of the QP Functional Safety (FuSa) Subsystem.
- Renamed `Q_onAssert()` callback to `Q_onError()` and changed its semantics, which now involves a critical section (interrupts disabled). Specifically, the `Q_onError()` is always invoked inside a critical section.
- Changed the macro for disabling the QP FuSa Subsystem from #Q\_NASSERT to `Q_UNSAFE` (disabling QP FuSa Subsystem is *NOT* recommended, especially in safety-related projects).
- Added the class `QAsm` (abstract state machine) as the abstract base class (ABC) for state machine implementations. This ABC removes the coupling between the QHsm-style and QMsm-style state machine implementation strategies.
- Redesigned `QEvt` class to support RAII (Resource Acquisition Is Initialization) for events, including dynamically allocated events
- Added a Memory Marker to all events (in the `QEvt.evtTag_` attribute).
- Added a `QEVT_INITIALIZER()` macro for initialization of immutable (const) events and encapsulate the Memory Marker.
- Added `QEvt` constructor for dynamic events (enabled by macro `#QEVT_DYNCTOR`).
- Modified macros `Q_NEW()` and `Q_NEW_X()` for the case when macro `#QEVT_DYNCTOR` is defined
- Re-designed the internals of the framework for Memory Isolation by dynamically adjusting memory settings (e.g., by dynamically reconfiguring the MPU). Among others, this required removing virtual functions for posting events from the `QActive` base class.
- Modified QP/C source code for better compliance with MISRA-C:2023.
- Added traceability links to the MISRA-C deviations in source code comments.

- Modified the behavior of [QXSemaphore](#) in the [QXK](#) kernel, which now behaves like semaphore in embOS/uC-OS2 rather than a semaphore in FreeRTOS.
- Added Duplicate Inverse Storage to the memory pools ([QMPool](#)). Specifically, all the pointers in the link-list of the free blocks in the pool are now guarded with Duplicate Inverse Storage. This doubles the minimum size of the free block from one to *two* pointers.
- Added Duplicate Inverse Storage to the subscriber lists ([QSubscrList](#)). This doubles the size of the size of [QSubscrList](#).
- Added Duplicate Inverse Storage to the ready-sets of the [QV](#), [QK](#), and [QXK](#) kernels.
- Introduced new macros for Memory Isolation:
  - [QF\\_MEM\\_ISOLATE](#)
  - [QF\\_MEM\\_SYS\(\)](#)
  - [QF\\_MEM\\_APP\(\)](#)

#### Ports:

- Redesigned and simplified the port structure. Consolidated multiple include files (e.g., `qep_port.h`, `qk_port.h`, etc.) with [qp\\_port.h](#). This reduced the number of header include file levels from 5 to 3.
- Added optional QP configuration file [qp\\_config.h](#), which can be used for compile-time configuration of the QP/C framework for specific application without changing the official port. (Requires command-line option [QP\\_CONFIG](#)).
- Replaced the macro `#QF_CRIT_STAT_TYPE` with [QF\\_CRIT\\_STAT](#) and changed the semantics of the macros for specifying critical sections in all existing QP ports. The macros [QF\\_CRIT\\_ENTRY\(\)](#)/[QF\\_CRIT\\_EXIT\(\)](#) no longer take the status parameter:
  - [QF\\_CRIT\\_STAT](#)
  - [QF\\_CRIT\\_ENTRY\(\)](#)
  - [QF\\_CRIT\\_EXIT\(\)](#)
- Modified the ARM Cortex-M ports (`qpc/ports/arm-cm`) to support Memory Isolation by means of the MPU (configurable by the macros [QF\\_MEM\\_ISOLATE](#), [QF\\_MEM\\_SYS\(\)](#), and [QF\\_MEM\\_APP\(\)](#)).

#### Examples:

- Re-designed the sample application code and QM models (for Blinky, DPP, etc.) to present a new "modern" way of structuring and organizing QP/C applications.
- Modified the BSPs (Board Support Packages) in examples (mostly for ARM Cortex-M) to define the function `assert_failed()`, which is now consistently called by fault handlers in the startup code.
- Added new examples for the affordable STM32C0 MCUs:
  - `qpcpp/examples/arm-cm/dpp_nucleo-c031c6`
- Added new examples for Memory Isolation with the MPU:
  - `qpc/examples/arm-cm/dpp_nucleo-c031c6_mpu` - use of MPU in ARM Cortex-M0+
  - `qpc/examples/arm-cm/dpp_ek-tm4c123gx1_mpu` - use of MPU in ARM Cortex-M4F
- Removed calls to [Q\\_onAssert\(\)](#) from all example startup code and replaced it with calls to `assert_failed()`. This is for consistency and interoperability with existing libraries (e.g., STM32Cube) and for adding flexibility in defining error handling for *hardware* exceptions.

**Note**

This change impacts most existing examples because it is now necessary to define `assert_failed()` in the applications (typically in the Board Support Package). The typical definition of `assert_failed()` is to call `Q_onError()`.

**Documentation:**

- Reorganized the QP/C documentation according to the new Documentation Management Plan.
- Removed the Doxygen documentation from the source code and placed it in the separate directory `qpc/doc`. This separation of concerns significantly reduces the need to change (and re-certify!) the QP/C source code just to update the documentation. This is part of the new Configuration Management Plan.

**Bug Fixes:**

- bug#343 FreeRTOS port uses `xQueueSendToBack()`/`xQueueSendToFront()` incorrectly
  - bug#342 Build fails for Cortex-M3 or `-M4+nofp` due to undefined compiler flag `__ARM_FP`
  - bug#338 QPC/Posix Critical section mutex is not recursive
- 

## 10.10 Version 7.2.1, 2023-01-15

**QP/C Source Code:**

- Changed the design of the `QXMutex` class to include `QActive` by composition rather than inheritance.

**Ports:**

- Fixed problems in `QK` ports for ARM Cortex-M: in case a regular IRQ is used for returning to the thread context. The port didn't handle correctly `QK_USE_IRQ_NUM` above 32.
- Fixed problems in `QXK` ports for ARM Cortex-M: in case a regular IRQ is used for returning to the thread context. The port didn't handle correctly `QK_USE_IRQ_NUM` above 32.

**Bug Fixes:**

- bug#328 Assertion `qxk:720` in QP/C/C++ 7.2.0

**Attention**

This bug in `QXK` affects releases QP/C 7.1.0 through 7.2.0

---

## 10.11 Version 7.2.0, 2023-01-06

**QP/C Source Code:**

- Added "enumeration dictionaries" for `QS` software tracing (see `QS_ENUM_DICTIONARY()`)
- Introduced common callback `QF_onContextSw()` used in all built-in kernels (`QV`, `QK`, `QXK`). This callback is enabled by the macro `QF_ON_CONTEXT_SW`.
- Removed callback `QK_onContextSW()` and macro `QK_ON_CONTEXT_SW` and replaced with `QF_onContextSw()`/`QF_ON_CONTEXT_SW`.
- Removed callback `QXK_onContextSW()` and macro `QXK_ON_CONTEXT_SW` and replaced with `QF_onContextSw()`/`QF_ON_CONTEXT_SW`.

- Added callback `QF_onContextSW()` and macro `QF_ON_CONTEXT_SW` to the `QV` kernel.
- added configuration macro `QS_CTR_SIZE` to determine the size of the type `::QSCtr` (see also [feature#195](#))
- Removed backwards-compatibility with QP 5.4.0 or older. This includes removing workarounds for `QFsm`, `QFsm_ctor()`, `_IGNORED()`.
- Modified `QMsm_childStateObj()` implementation to correctly handle submachine states (see also [bug#316](#))
- Added automatic generation of the `QS` function dictionary for the `QHsm_top()` state handler function. This happens only once on the first call to `QHSM_INIT()`.

**Note**

It is no longer necessary to generate `QS` function dictionary for `QHsm_top()` in the application (although it is still allowed, but wasteful).

**Ports:**

- `ports/arm-cm` added setup of the FPU (if configured) to all built-in kernels (`QV`, `QK`, `QXK`) and toolchains (ARMCLANG, GNU-ARM, IAR).
- `ports/arm-cm` all ports now clear the FPCA bit in the CONTROL register (if FPU configured) right before starting the interrupts. That potentially saves the stack space (MSP) for the automatically reserved FPU context.

**Note**

This change means that the applications no longer need to setup the FPU, which was done typically in the BSP.

- `ports posix` added configuration macro `QS_CTR_SIZE` 4U, to allow `QS` trace buffers > 64KB (see also [feature#195](#))
- `ports/win32` added configuration macro `QS_CTR_SIZE` 4U, to allow `QS` trace buffers > 64KB (see also [feature#195](#))
- `ports/esp-idf` fixed compilation error caused by obsolete macro `QF_PTR_RANGE_()`
- `ports/sample` added a sample QP/C port for reference and documentation of all available configuration options.
- `zephyr` updated `qf_port.h` to include `<zephyr/kernel.h>` rather than `<zephyr/zephyr.h>` (see also [bug#321](#))

**Examples:**

- examples for `QV`: added callback `QF_onContextSw()`
- examples for `QK`, `QXK`: replaced callbacks `QK_onContextSw()` / `QXK_onContextSw()` with `QF_onContextSw()`
- `examples/arm-cm` removed setting up the FPU from the BSPs (the FPU is now setup in the ports).
- `examples/arm-cm` added setting up the MPU (Memory Protection Unit) to catch `NULL-pointer dereferencing` and other incorrect memory access.

**Testing:**

- changed the QUTest assertion handler `Q_onAssert()` to reset the target rather than wait for user commands. This allows proper assertion handling even inside exception handlers, where the target really cannot handle interrupts (which were needed for reception of commands). This change matches the updated policy of **assertion failures** in [QUTest 7.2.0](#).

- examples for QUTest (examples/qutest and test directories): adjusted makefiles to invoke the qutest.py utility with the new command-line parameters.
- modified the test fixtures to use "enum dictionaries" for commands
- added example examples/qutest/start\_seq that demonstrates testing of start sequences.

#### Bug Fixes:

- bug#315 QSignal defined incorrectly when #Q\_SIGNAL\_SIZE==4U
- bug#316 Using History with Submachines does not work
- bug#321 zephyr port build error
- bug#325 QSpy shows Sig=NULL if #Q\_EVTCTOR is used

#### Feature Requests:

- feature#195 allow > 64K QS trace buffer on target. This is configurable by setting the macro QS\_CTR\_SIZE.

## 10.12 Version 7.1.3, 2022-11-18

#### Bug Fixes:

- bug#317 QK works incorrectly in QP/C/C++ 7.1.2

#### Source

- The file `qstamp.c` has been copied from the `include/` folder to `src/qs/`. This makes it easier to build SPY configuration. (The file `qstamp.c` has been left in the `include/` folder for backwards compatibility).

#### Ports

- This release removes the QP libraries from the **Windows** ports
- CMakeLists.txt file in the zephyr port has been modified to use `zephyr_library_named(qpc)` instead of `zephyr_library()`

#### Examples

- This release changes all examples for **Windows** (Makefiles for GCC and project files for Visual Studio) to build the QP Framework from sources rather than using QP libraries. This slightly extends the first build of the application, but avoids any problems with library incompatibilities (e.g., 32-bit/64-bit compilers used to build the application vs. the QP libraries).
- updated examples for emWin Embedded GUI (for emWin v6.28)
- updated examples for MSP430

#### Documentation

- Modified the `doxygen/` folder for generation of QM Manual in LaTeX (enables generation of PDF)

#### Licensing

- This release removes the `QM-EVAL-QPC.qlc` file from the `LICENSES` folder. This "QM License Certificate" is no longer needed for QM 5.2.3, where "QM License Certificates" are no longer needed for generating QMsm-style state machines.

## 10.13 Version 7.1.2, 2022-10-07

This release improves preemption-threshold scheduling (PTS) in [QK](#) kernel, especially the generation of the software tracing information about the scheduler activity. Also, PTS has been removed from the [QXK](#) kernel because of the inherent complexity of that kernel.

Additionally, the default for [QF\\_TIMEEVT\\_CTR\\_SIZE](#) has been increased from 2 to 4 bytes. This increases the default dynamic range for [QTimeEvt](#) counters to 32-bits.

### Source

- Modified QK\_sched\_() and QK\_activate\_()
- Modified QXK\_sched\_() and QXK\_activate\_()
- Added QXK\_contextSw() to handle context switching (generation of software tracing information and invoking of the QXK\_onContextSw() callback, if configured)
- increased default for [QF\\_TIMEEVT\\_CTR\\_SIZE](#) 4
- changed inclusion guards in the QP/C header files (e.g. QF\_H -> QP\_INC\_QF\_H\_) for compliance with the updated Quantum Leaps coding standard and to make the names more unique.

### Ports

- Modified [QXK](#) ports (assembly) to call QXK\_contextSw()
- Added port to RISC-V with the non-preemptive QV kernel

### Testing

- Added new directory `qpc/test/` for **system**-level testing of QP/C itself.
- Added new tests for [QK](#): `qpc/test/qk`
- Added new tests for [QXK](#): `qpc/test/qutest/qxk`

### Note

The `qpc/test/` directory is planned to contain a growing number of system-level tests, which are based on QUTest, but *without* the QP-stub. The tests take advantage of the new QUTest configuration, where the QP-stub is NOT included (because the actual QP Framework is linked). This configuration is activated by defining macro `Q_UTEST=0`.

### Attention

Most tests provided in the `qpc/test/` directory run only on embedded targets and cannot run on the host machine.

---

## 10.14 Version 7.1.0, 2022-08-30

This QP/C release introduces **preemption-threshold scheduling (PTS)** for the preemptive [QK](#) and [QXK](#) kernels. Specifically, it is now possible to limit preemption of a given QActive/QXThread by giving it both the QF-priority and preemption-threshold.

The following diagram shows the relationship between the "QF-priority" and "preemption-threshold" (see also [QPrioSpec](#)):

**Note**

For backwards-compatibility, [QPriorSpec](#) data type might contain only the "QF-priority" component (and the "preemption-threshold" component left at zero). In that case, the "preemption-threshold" will be assumed to be the same as the "QF-priority". This corresponds exactly to the previous semantics of AO priority.

**Source**

- modified the QP/C code for preemption-threshold
- improved [QS](#) tracing of Scheduler activities in [QK](#) and [QXK](#)
- improved implementation of [QXSemaphore](#) and [QXMutex](#) in [QXK](#)

**Ports**

- Modified [QK](#) and [QXK](#) ports to ARM Cortex-M for preemption-threshold and for handling the context-switch callback
- Updated Zephyr port and examples for Zephyr 3.1.99

**Examples**

- Added [Q\\_PRIO\(\)](#) macro for setting QF-priority and preemption-threshold to various examples (in calls to [QACTIVE\\_START\(\)](#))
- increased size of the main stack in most examples from 1024 to 2048 bytes. This is to prevent stack overflow that was observed with the lower limit.

**Testing**

- modified QUTest to also allow testing of the QP Framework itself.
- 

## 10.15 Version 7.0.2, 2022-08-12

**Bug Fixes:**

- [bug#312 "Bug in QK and QXK 7.0.1"](#)

**Ports** Modified [QK](#) and [QXK](#) ports to ARM Cortex-M

- [qpc/ports/arm-cm/qk](#)
- [qpc/ports/arm-cm/qxk](#)

**Modified Zephyr port**

- moved the port from the usual location in [qpc/ports/zephyr](#) up a level to [qpc/zephyr](#). This was done to make QPC into an external [Zephyr module](#), which has to comply with the Zephyr specification.

**Examples (ARM Cortex-M):**

- [qpc/examples/arm-cm/dpp\\_efm32-slstk3401a/qk](#) provided examples of configuring either NMI or IRQ for Cortex-M4 (ARMv7m)
- [qpc/examples/arm-cm/dpp\\_efm32-slstk3401a/qxk](#) provided examples of configuring either NMI or IRQ for Cortex-M4 (ARMv7m)
- [qpc/examples/arm-cm/dpp\\_nucleo-1053r8/qk](#) provided examples of configuring either NMI or IRQ for Cortex-M0+ (ARMv6m)
- [qpc/examples/arm-cm/dpp\\_nucleo-1053r8/qk](#) provided examples of configuring either NMI or IRQ for Cortex-M0+ (ARMv6m)

**Examples (Zephyr):**

- [qpc/examples/zephyr/blinky](#) - modified to use the new Zephyr port
  - [qpc/examples/zephyr/dpp](#) - modified to use the new Zephyr port and to demonstrate QSPY software tracing.
-

## 10.16 Version 7.0.1, 2022-07-31

This release is the first one that contains the complete [QM model](#) of the QP/C framework (in the file `qpc/qpc.qm`). This model is then used to **generate all QP/C source code**:

- `qpc/`
  - `include`
  - `src`

The QM model of QP/C explicitly captures the *logical design* of the framework, which is then mapped to the preexisting [physical design](#). The *logical design*, which consists of packages (some associated with namespaces), classes, types, and macros traces explicitly to the [Software Architecture Specification \(SAS\)](#) and is easier to manipulate and refactor.

### Source Code Changes:

- The whole QP/C source code is now generated from the `qpc.qm` model.
- Refactored classes with only static members ([QF](#), [QV](#), [QK](#), [QXK](#), [QS](#)) into "namespaces".
- Refactored:
  - `QF_TICK_X()` -> [QTIMEEVNT\\_TICK\(\)](#) (see also `QTimeEvt_tick_()`)
  - `QF_PUBLISH()` -> [QACTIVE\\_PUBLISH\(\)](#) (see also `QActive_publish_()`)
  - `QF_psInit()` -> `QActive_psInit()`

### Note

The old APIs ([QF\\_PUBLISH\(\)](#), [QF\\_TICK\\_X\(\)](#), `QF_psInit()`) are still provided, but are tagged as [deprecated](#).

- Moved package-scope header files `qf_pkg.h` and `qs_pkg.h` from the `qpc/src` directory to `qpc/include`. This allows the build process to specify only the `qpc/include` directory to build applications and QP/C source code. Previously, the build process had to specify additionally `qpc/src`, but this is no longer needed.

### Note

This change allowed all provided examples to be simplified by removing the `qpc/src` include directory from the various project files.

### Ports/Examples:

- Removed port and examples for the older ARM7/ARM9
- Added port to the [Zephyr RTOS](#)
- Added examples for the Zephyr RTOS
- redesigned QP/C [port to FreeRTOS](#), see `qpc/ports/freertos`
- slightly improved QP/C [port to ThreadX](#), see `qpc/ports/threadx`

**Improved Documentation:** The Doxygen documentation has been re-structured and improved. Also the more attractive styling of the HTML has been applied.

## 10.17 Version 7.0.0, 2022-04-30

This QP/C release changes the QP/C implementation from C89 to C99 and also improves compliance of the QP/C source code with MISRA:C-2012 (a.k.a. MISRA3). Also, major focus of this release is the adaptation of the code and documentation to better support safety certification (see [QP Certification Kit](#))

### Modified QP/C source code:

- updated QP/C source code for C99

### Note

The QP/C source code requires C99 minimum and no longer compiles as C89.

- made the QP/C source code compliant with MISRA:C-2012, C99 variant
- made the compliant with the updated [Quantum Leaps Embedded C/C++ Coding Style \(QL-C/C++:2022\)](#)
- added traceability links between QP/C source code and requirements, architecture, design, and MISRA deviation permits.
- modified the top-level file comments now have standard-compliant license specification (SPDX-License-Identifiers). This allows QP/C to be included in automatic generation of Software Bill Of Materials (SBOM).

### Modified QP/C ports:

- modified the [QK](#) and [QXK](#) ports to ARM Cortex-M to allow compile-time selection between using either NMI or an unused IRQ (via macro QK\_USE\_IRQ\_HANDLER) as a mechanism to return to the thread mode
- removed unused [ARM\\_ARC](#) symbol from Makefiles for GNU-ARM
- fixed the following bugs in [QXK](#) port to ARM Cortex-M
  - bug#305 Floating-point context corruption in QXK
  - bug#306 #QF\_ON\_CONTEXT\_SW option is not compatible with soft-float

### Removed QP/C ports:

- removed QP/C port to PIC24/dsPIC with XC16 (not compatible with C99)

### Modified GitHub repository:

- removed the `3rd_party` directory from the [qpc GitHub repository](#) to avoid tracking changes caused by updates to the 3rd-party components. (The QP/C releases still have the `3rd_party` directory, because it is needed for building the examples).

### Added/Modified examples:

- modified examples for ARM Cortex-M7 (to integrate better with STM32CubeH7)
- added examples for ARM Cortex-M33 (for STM32 NUCLEO-L552ZE and STM32CubeL5)
- added `qutest` examples for Unity (comparison between Unity and QUTest)

### Modified QP/C documentation:

- restructured the Doxygen documentation for traceability
- added the [QP/C Tutorial](#)
- added the QP Certification Kit

### Remarks

The [QP Certification Kit](#) is still work in progress in this release, but the remaining work is limited to documentation only and no code restructuring is anticipated for that.

## 10.18 Version 6.9.3, 2021-04-12

### Feature Requests:

- `feature#191 "Add QS local filtering to QF_PUBLISH()`

The feature has been implemented by means of the "sender" parameter of the `QF_PUBLISH()` macro, which must now be a pointer to a struct that has the `prio` member. This design enables applying `QS` local filter to the `prio` as the QS-Id. This works for publishing events from active objects. For publishing events outside active objects (e.g., from ISRs), the header file `qs.h` provides a type `QSpyId`, which should be used to define `QS` ID variables to be used as "sender" in `QF_PUBLISH()`.

### Note

This change only affects the Spy build configuration, where the `QS` software tracing instrumentation is enabled. The existing code could require modifications to provide "sender" pointer that can access `prio` directly. For example, AOs would need to call `QF_PUBLISH(e, &me->super)` instead of `QF_PUBLISH(e, me)`.

- `feature#185 "Add possibility of assigning name to AO threads in QP-RTOS ports"`

This feature request is now implemented uniformly in all QP-ports to 3rd-party RTOSes by means of the generic QP API `QActive_SetAttr()`. Specifically, the QP port will allow you to set the thread name by calling `QActive_SetAttr()` before calling `QACTIVE_START()`.

### Bug Fixes:

- `bug#292 "QP-C-C++ on 64 Bit Linux - Setting of Current-Obj - AO_Obj from QView raises segmentation fault"`
- `bug#293 "[Qpccp 6.9.2] test of assertion fail now"`
- `bug#298 "ARM Cortex-M Erratum 838869 Handled Inconsistently in Ports"`

### Source code changes:

- `qpc/include/qf.h` - `feature#191`
- `qpc/include/qs.h` - `feature#191`
- `qpc/src/qf/qf_ps.c` - `feature#191`
- `qpc/src/qs/qs_rx.c` - `feature#191` and `bug#292`

### Modified QP/C ports:

- `qpc/ports/arm-cm/` - fixed `bug#298`
- `qpc/ports/embos/` - implemented `feature#185`
- `qpc/ports/threadx/` - implemented `feature#185`
- `qpc/ports/uicos2/` - implemented `feature#185`
- `qpc/ports/win32/` - added QSpy64 (64-bit) build configuration to the Visual Studio project
- `qpc/ports/win32-qv/` - added QSpy64 (64-bit) build configuration to the Visual Studio project
- `qpc/ports/win32-quit/` - added "port" used for "QP Unit Internal Testing" (QUIT)

### Updated examples:

- all examples using `QF_PUBLISH()` - adjusted the `sender` parameter of the macro to have the `prio` member

- qpc/examples/arm-cm/ - fixed [bug#298](#) (added calls to QV|QK|QXK\_ARM\_ERRATUM\_838869())
- qpc/examples/qutest/self\_test/test/ - added examples of the Python include files (\*.pyi) for the new `include` command.
- qpc/examples/arm-cm/dpp\_efm32-s1stk3401a/win32-gui/ - corrected handling of user-drawn buttons in `bsp.c`
- qpc/examples/arm-cm/game\_efm32-s1stk3401a/win32-gui/ - corrected handling of user-drawn buttons in `bsp.c`

#### Updated 3rd-Party Components:

- qpc/3rd\_party/threadx to version 6.1.6 (latest open source version from [GitHub](#))

## 10.19 Version 6.9.2, 2021-01-18

The main purpose of this release is a redesign of the QS-RX (software tracing input) to implement the feature request #187:

- `feature#187 "Make the byte order in QS-RX buffer compatible with DMA"`

Additionally, as part of the re-design of the internal QS-RX implementation, this release fixes the bug #287 ("QS\_rxGetNfree() returns too low value", see the "Bug Fixes" section)

Additionally, this release introduces a new "virtual" function `getStateHandler()` in the `QHsm` base class. This is part of the fix for bug #290 ("QView query object state reports wrong state with QMsm strategy", see the "Bug Fixes" section)

#### Source code changes:

The introduction of the `getStateHandler()` virtual function involves the following code changes:

- [qpc/include/qep.h](#)
- [qpc/src/qf/qep\\_hsm.c](#)
- [qpc/src/qf/qep msm.c](#)
- [qpc/src/qf/qf\\_actq.c](#)
- [qpc/src/qf/qf\\_qact.c](#)
- [qpc/src/qf/qf\\_qmact.c](#)
- [qpc/src/qs/qs\\_rx.c](#)
- [qpc/src/qs/qutest.c](#)
- [qpc/src/qxk/qxk\\_xthr.c](#)
- [qpc/include/qs.h](#) file – added new API `QS_queryCurrObj()`

The redesign of QS-RX implementation involves the following code changes:

- [qpc/include/qs.h](#) – macro `QS_RX_PUT()` now returns `bool` (true if the byte was inserted into the QS-RX buffer and false otherwise)
- [qpc/src/qs/qs\\_rx.c](#) - changed several functions
- [qpc/src/qs/qs\\_fp.c](#) - improved implementation

#### Modified QP/C ports:

- qpc/ports posix / - modified to take advantage of the QS-RX direct byte ordering
- qpc/ports posix-qutest / - modified to take advantage of the QS-RX direct byte ordering
- qpc/ports posix-qv / - modified to take advantage of the QS-RX direct byte ordering
- qpc/ports win32 / - modified to take advantage of the QS-RX direct byte ordering
- qpc/ports win32-qutest / - modified to take advantage of the QS-RX direct byte ordering
- qpc/ports win32-qv / - modified to take advantage of the QS-RX direct byte ordering

#### **Added QP/C ports:**

- qpc/ports/freertos-esp32 / - added new *experimental* port to FreeRTOS-ESP32

#### **Updated examples:**

- qpc/examples/workstation/calc fixed minor bug in the calculator

#### **Bug Fixes:**

- bug#289 "QS\_OBJ\_PTR() macro does not compile on 64-bit machine"
- bug#288 "POSIX port does not compile in Spy build configuration"
- bug#287 "QS\_rxGetNfree() returns too low value"
- bug#285 "QView bug with querying the state of QMsm-type state machines"
- bug#284 "Typo in QM generated headers"

#### **Updated 3rd-Party Components:**

- qpc/3rd\_party/CMSIS to version 5.7.0
- qpc/3rd\_party/FreeRTOS to version 10.4.3 (202012LTS released December 2020)

## **10.20 Version 6.9.1, 2020-09-28**

The main purpose of this release is a redesign of the [QS Local Filter](#) (see also [feature request #127](#)). This new design supports filtering on **multiple** active objects (as well as other objects in the Target memory), as opposed to filtering just one such object at a time. The main use case for this redesign of QS Local Filter is an application, where some active objects are very "noisy", and would overwhelm your trace. The new QS Local Filter allows you to selectively silence the "noisy" active objects and let all the others through.

The new QS Local Filter is based on "QS-IDs" associated with various objects in the Target memory. The QS-IDs are small integer numbers, such as the unique priorities assigned to QP Active Objects, but there are more such QS-IDs, which you can assign to various other objects through the macro [QS\\_BEGIN\\_ID\(\)](#). Then, you can setup the QS Local Filter to trace only a specific QS-IDs or whole groups of QS-IDs by means of the macro [QS\\_LOC\\_FILTER\(\)](#) or remotely via the QS-RX channel.

#### **Source code changes:**

The redesign of the QS Local Filter impacts the QS trace instrumentation in QP/C: both the [pre-defined QS trace records](#) and the [application-specific trace records](#). The changes to the pre-defined records are confined to the QP/C source code and are transparent to the application developers. However, the changes to the application-specific trace records require adjusting existing applications as follows:

- use the new macro [QS\\_LOC\\_FILTER\(\)](#) to set/clear the QS Local Filter
- use the new macro [QS\\_BEGIN\\_ID\(\)](#) to define [application-specific trace records](#).

**Note**

The macro [QS\\_BEGIN\\_ID\(\)](#) assigns the specified QS-ID number to the app-specific record, which can be subsequently filtered by the new [QS](#) Local Filter. The old macro [QS\\_BEGIN\(\)](#), with the old Local Filter interface, is still available, but is [deprecated](#) and not recommended.

The following macros are [deprecated](#):

- macro [QS\\_FILTER\\_SM\\_OBJ\(\)](#) does nothing in QP/C 6.9.1
- macro [QS\\_FILTER\\_AO\\_OBJ\(\)](#) does nothing in QP/C 6.9.1
- macro [QS\\_FILTER\\_MP\\_OBJ\(\)](#) does nothing in QP/C 6.9.1
- macro [QS\\_FILTER\\_EQ\\_OBJ\(\)](#) does nothing in QP/C 6.9.1
- macro [QS\\_FILTER\\_TE\\_OBJ\(\)](#) does nothing in QP/C 6.9.1
- macro [QS\\_FILTER\\_AP\\_OBJ\(\)](#) still works for [QS\\_BEGIN\(\)](#)
- macro [QS\\_FILTER\\_ON\(\)](#) still works, but uses [QS\\_GLB\\_FILTER\(\)](#) internally
- macro [QS\\_FILTER\\_OFF\(\)](#) still works, but uses [QS\\_GLB\\_FILTER\(\)](#) internally
- macro [QS\\_BEGIN\(\)](#) still works, but uses the old "AP-OBJ" pointer

The following APIs have been [changed](#):

- [QHSM\\_INIT\(\)](#) now takes extra `qsId` parameter
- [QHSM\\_DISPATCH\(\)](#) now takes extra `qsId` parameter
- [QMPool\\_get\(\)](#) now takes extra `qsId` parameter
- [QMPool\\_put\(\)](#) now takes extra `qsId` parameter
- [QEQueue\\_post\(\)](#) now takes extra `qsId` parameter
- [QEQueue\\_postLIFO\(\)](#) now takes extra `qsId` parameter
- [QEQueue\\_get\(\)](#) now takes extra `qsId` parameter

**Note**

The API changes are [not backwards-compatible](#) and require adjusting existing QP/C applications.

Additionally, this release introduces the new pre-defined [QS](#) record #QS\_QF\_NEW\_ATTEMPT, which is generated when [Q\\_NEW\\_X\(\)](#) fails to allocate a dynamic event. Also, this release [reverses the order](#) of the pre-defined [QS](#) records #QS\_QF\_NEW and #QS\_QF\_MPOOL\_GET. This was necessary if the dynamic allocation is allowed to fail, because only *after* attempting to allocate a memory block, the QF\_newX\_() function can generate either #QS\_QF\_NEW or #QS\_QF\_NEW\_ATTEMPT.

**Note**

The reversal of #QS\_QF\_NEW and #QS\_QF\_MPOOL\_GET trace records has implications for the existing [QUTest test scripts](#), where the order of expectations for "QF-New" and "MP-Get" needs to be reversed as well.

Additionally, this release modifies the [QXK](#) source code, so that the [QActive.prio](#) attribute is the fixed priority assigned in [QActive\\_start\(\)](#), while [QActive.dynPrio](#) is the "dynamic" priority that can be changed when a QXK extended thread acquires a [mutex](#).

**Updated Ports:**

- all ARM Cortex-M ports (added workaround for the [ARM Erratum 838869](#))

- all **QK** ports (because of the changed QMPool\_get() / QMPool\_put() interface)
- all **QXK** ports (because of the changed QMPool\_get() / QMPool\_put() interface)
- embOS port (because of the changed QMPool\_get() / QMPool\_put() interface)
- FreeRTOS port (because of the changed QMPool\_get() / QMPool\_put() interface)
- ThreadX port (because of the changed QMPool\_get() / QMPool\_put() interface)
- uC/OS-II port (because of the changed QMPool\_get() / QMPool\_put() interface)
- posix, posix-qv (because of the changed QMPool\_get() / QMPool\_put() interface)
- win32, win32-qv (because of the changed QMPool\_get() / QMPool\_put() interface)

#### **Feature Requests:**

- `feature#181 "Use of QS_U64() on 32 bit machine"`

#### **Bug Fixes:**

- `bug#279 "Confusing documentation of QF_newX_()`
- `bug#280 "QXK_onContextSw() not called at the end of QXK_activate_()`

## 10.21 Version 6.9.0, 2020-08-21

The main purpose of this release is to adjust the QP/C RTEF to the changes and improvements introduced in [QTools 6.9.0](#). Specifically, QP/C now includes examples for the new [QView Visualization & Monitoring](#) as well as adjustments for the new version of [QUTest Unit Testing](#).

#### **Source code changes:**

Added new #QS\_QF\_RUN trace record to [QS software tracing](#), which is now generated in all QP/C ports upon the entry to QF\_run(). This trace record marks the end of the application startup, at which time all the [QS dictionaries](#) are typically produced.

Also, added the information about the target **endianness** to the #QS\_TARGET\_INFO [QS](#) trace record.

#### **Note**

The addition of the #QS\_QF\_RUN trace record affects only the [Spy build configuration](#) and has **no impact** on the Release or Debug build configurations.

#### **Updated Ports:**

- All QP/C ports to 3rd-party RTOSes (embOS, FreeRTOS, ThreadX, uC/OS-II) and OSes (POSIX, POSIX-QV, WIN32, WIN32-QV) have been updated to generate the #QS\_QF\_RUN trace record.
- The POSIX and POSIX-QV ports have been updated to add the call to `pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED)` (see also [bug#276](#))
- The MSP430 ports have been extended for the GNU-MSP430 compiler, so that they now work with IAR-MSP40, TI-MSP40 and GNU-MSP430 toolchains.

#### **Updated Examples:**

- All [QUTest](#) examples (`qpc\examples\qutest` directory) have been modified to use the new location of the `qutest.py` Python module. Also, all QUTest examples that use the `on_reset()` callback have been modified to call `expect_run()`.

**Note**

The #QS\_QF\_RUN record is now generated in `QUTest unit testing`, which requires adjustments in the existing `test scripts`. Specifically, the test scripts that provide their own `on_reset()` callback must now also call `expect_run()`.

- ARM Cortex-M examples for STM32 NUCLEO-L053RE  
(`qpc\examples\arm-cm\dpp_nucleo-l053r8`) and NUCLEO-L152RE  
(`qpc\examples\arm-cm\dpp_nucleo-l152re`) have been modified to support bi-directional QSPY communication. These examples now include the QView demos.
  - Added examples of new `Sequence Diagram Generation` in QSPY 6.9.0
  - The example projects for MSP430 now contain the `ccs-ti` and `ccs-gnu` directories, for the TI-MSP430 and GNU-MSP430 toolchains, respectively.

**Bug Fixes:**

- bug#276 "QP/C/C++ POSIX port doesn't set thread priorities correctly"
- 

## 10.22 Version 6.8.2, 2020-07-17

**Source code changes:**

- Changed definitions of `QMsm`, `QMActive`, and `QTicker` "classes" from using `typedef's` to using `struct's`. This means that `QMsm`, `QMActive`, and `QTicker` are now **distinct types** different from the respective base classes: `QHsm`, `QActive`, and `QActive`. Consequently, implicit conversions to the base class are *no longer performed*, which improves the type safety.

**Note**

This change might cause compiler warning in code that assumes equivalence between `QHsm` and `QMsm` or `QActive` and `QMActive`. Specifically, any "opaque" pointers to the state machine base class should be of type `QHsm *` (and *not* `QMsm *`). Similarly, any "opaque" pointers to the active object base class should be of type `QActive *` (and *not* `QMActive *`)

- Applied the `Doxygen` to indicate **inheritance** and to explicitly specify base classes for all derived classes (e.g., `QTimeEvt` **extends** `QEvt`).
- Fixed errors in the Doxygen documentation, such as: missing documentation for parameters, wrong parameter names, unresolved references, etc.
- Applied new, clearer styling to the Doxygen documentation.
- Changed the `QS` trace record name `QS_QF_ACTIVE_POST_FIFO` to `::QS_QF_ACTIVE_POST` and `QS_QF_EQUEUE_POST_FIFO` to `::QS_QF_EQUEUE_POST`. This refactoring now better matches the QP/C API `QACTIVE_POST()` and `QEQueue_post()`.

**Updated Ports:**

- enabled `QACTIVE_CAN_STOP` in the ports: Win32, Win32-QV, Win32-QUTEST, POSIX, POSIX-QV, and POSIX-QUTEST.
- `port to uC/OS-II` has been adapted to the new uC/OS-II v2.93.00 (recently released by Silicon Labs under the open source Apache 2.0 license).

**Updated Examples:**

- Modified `Makefiles` for the GNU-ARM projects to use `gcc` as the linker instead of `g++`

- Modified Makefiles for the workstation examples to add the `-no-pie` linker option only when `GCC_OLD` environment variable is NOT defined. This is to accommodate older POSIX platforms with older GCC distribution.
- Modified all ARM-KEIL uVision projects for ARM-CLANG to use the startup code in the ARM-ASM syntax.
- New uC/OS-II example uc-os2\_dpp\_nucleo-l053r8 (Cortex-M0+)
- Removed uC/OS-II example for the STM32-NUCLEO-L152RE (Cortex-M3)
- Added QM model files: `workstation/defer/defer.qm` and `workstation/reminder2/reminder2.qm`

#### **Updated QUTest examples to generate code-coverage information:**

- Modified Makefiles in QUTest examples (directory `examples\qutest`) to generate **code-coverage information**. Specifically, when the `GCOV` environment variable is defined, the Makefiles generate code-coverage information for `GCOV`.

#### Note

The GCOV code-coverage generation is currently available only on the hosts.

#### **Updated 3rd-Party Components:**

- CMSIS from 5.6.0 to 5.7.0
- uC/OS-II from 2.92.10 to 2.93.00 (open source Apache 2.0 license)

#### **Bug Fixes:**

- bug#267 "QP/C Spy build configuration fails on 64-bit target"

## **10.23 Version 6.8.1, 2020-04-04**

#### **Source code changes:**

Improved comments in the `QF_stop()` function. The comments now make it very clear that after calling `QF_stop()` the application must terminate and cannot continue. In particular, `QF_stop()` is **not** intended to be followed by a call to `QF_init()` to "resurrect" the application. The previous comments apparently were confusing and some developers attempted to "restart" a running application, which led to system crashes.

#### **Bug Fixes:**

- bug#261 `QTime_disarm()` clears the wrong flag
- bug#262 `Ticker0` does not work in QPC 6.8.0

Also, this release updates the QP/C ports and examples for workstations (Windows and POSIX) by consistently applying the "safe" versions of services from `<stdio.h>` and `<string.h>`. The "portable" versions of these services are defined as macros in the `safe_std.h` header file and include the following services:

- `MEMMOVE_S()` -> `memmove_s()`
- `STRCPY_S()` -> `strcpy_s()`
- `STRCPY_S()` -> `strncpy_s()`
- `STRCAT_S()` -> `strcat_s()`
- `SNPRINTF_S()` -> `_snprintf_s()`
- `PRINTF_S()` -> `printf_s()`

- FPRINTF\_S() -> fprintf\_s()
- FREAD\_S() -> fread\_s()
- FOPEN\_S() -> fopen\_s()
- LOCALTIME\_S() -> localtime\_s()

These "safe" functions are mapped to the best approximation of these services available on a given platform. For example, `STRCPY_S()` is mapped to `strncpy_s()` on Windows and `strncpy()` on POSIX (Linux/MacOS/etc.).

## 10.24 Version 6.8.0, 2020-03-21

### Source code changes:

- Fixed inconsistencies between QP API declarations in the .h files and definitions in the .c files, such as different parameter names in declarations and definitions (MISRA-C:2012).
- Removed a lot of excessive type casting of compile-time constants, such as `(uint8_t) 0`, which is now coded simply as `0U`
- Introduced `Q_NORETURN` macro for `Q_onAssert()`, which can (if defined) inform the compiler that `Q_onAssert()` does not return. This can potentially improve the code generation and can improve diagnostics generated by the compiler as well as static code analysis tools.
- Moved many qs facilities for internal use (only inside the QP/C source code) from qs.h to qs\_pkg.h. These are mostly facilities related to internal `QS` implementation as well as pre-defined `QS` trace records, which are only used inside QP and are not needed in the QP Applications.
- Removed all pre-conditions from `QActiveDummy_start_()` function in qutest.c. This is to allow starting dummy AOs the exact same way as the real counterparts (e.g., with stack storage, which `QActiveDummy` does not really need).

### Updated 3rd-Party Components:

- FreeRTOS from 10.2.1 to 10.3.0
- SEGGER embOS from 4.34.1 to 5.06.1
- SEGGER emWin from 5.32 to 6.10
- Removed TI-RTOS (both port and examples)

### Updated Examples:

Converted most of the examples for ARM-MDK from the no-longer maintained compiler-5 (RVDS) to the new ARM Compiler-6 (ARM-clang). The examples for ARM Compiler-6 are located in the `armclang` sub-directories.

### Note

The older ARM Compiler-5 is still supported, but will be phased out in the future. The only examples for ARM Compiler-5 are for the EK-TM4C123GXL (TivaC LanuchPad) board. These examples are located in the `arm` sub-directories.

---

## 10.25 Version 6.7.0, 2019-12-30

The main purpose of this release is providing improved compliance with MISRA-C:2012 (including MISRA-C:2012-Amendment-1) and also to provide support for the `PC-Lint-Plus` static analysis tool (see also feature request [#169](#)). Specifically, the QP/C source code and some examples have been adjusted to comply with MISRA-C:2012-Amendment-1 rules, with all deviations captured in the PC-Lint-Plus configuration files. These PC-Lint-Plus configuration files have been provided in the new "port" to PC-Lint-Plus in the directory `qpc/ports/lint-plus/`.

**Note**

The support for the older PC-Lint 9.x and the older MISRA-C:2004 has been dropped to avoid confusion and conflicts with the newer MISRA-C:2012 and the newer PC-Lint-Plus.

This release also includes the offline documentation for **this** particular version of QP/C (in the `html` / folder). To view the offline documentation, open the file `html/index.html` in your web browser. (The online HTML documentation for the **latest** version of QP/C remains located at: <https://www.state-machine.com/qpc/>)

The backwards-compatibility layer for QP 4.x has been removed. Among others, the macros `Q_ROM` and `Q_ROM_VAR` are no longer defined.

Also, this release updates the Windows-GUI examples with the QWIN Prototyping Toolkit to work with the latest Visual Studio 2019 and specifically with the Resource Editor now available. Specifically here, the Game-GUI example (`qpc\examples\arm-cm\game_efm32-s1stk3401a\win32-gui`) and the DPP-GUI example (`qpc\examples\arm-cm\dpp_efm32-s1stk3401a\win32-gui`) have been updated to build with the Visual Studio 2019.

**Bug Fixes:**

- bug#254 "Q\_NEW\_X\_FROM\_ISR for FreeRTOS port"
- bug#255 "IAR-ARM "Multi-file Compilation" fails for QK and QXK applications"

**10.26 Version 6.6.0, 2019-10-31**

The main purpose of this release is the change in distribution of the QP/C framework, which is now bundled together with the other QP Frameworks (QP/C++ and QP-nano) as well as QM, and the QTools collection into "QP-bundle". This "QP-bundle" provides a single, streamlined and simplified download and installation of all QP Frameworks and all the accompanying tools.

This release brings also the following changes:

- removed "://" (double slash) from all comments for compliance with the MISRA-C:2012 required Rule 3.1. (The "://" sequence was used in the URLs, such as "http://~~~").
- converted inclusion guards in header files to uppercase as per coding convention for macros.
- replaced "vtbl" with "vtable" in QP/C source code
- changed the type of the last parameter in `QHsm_init_()` and `QHSM_INIT()` from "QEvt const \*" to "void const \*" to allow passing arbitrary data during initialization.
- increased #QS\_USER from 70 to 100 and made the freed-up records reserved
- changed the QS\_Ux\_RECORDS groups to partition user trace records 100-124 into 5 groups of 5 records each.
- removed Tcl test scripts from QUTest examples

**10.27 Version 6.5.1, 2019-05-24****Bug Fixes:**

- bug#241 "QUTest produces superfluous Trg-Done QS\_RX\_EVENT after publish".
- bug#242 "QUTest fails to process events posted/published from user commands".

Also, this release extends the Makefiles in the `qpc\examples\qutest` directory to provide the debug target. Specifically, the `qpc\examples\qutest\blinky` example has been extended with projects to build/debug this example with Visual C++ (on the host) and with ARM-KEIL on embedded target (EK-TM4C123).

Finally, this release adds ports and examples for PIC24/dsPIC 16-bit MCUs with MPLAB-X/XC16 and PIC32 with MPLAB-X/XC32 toolchains.

## 10.28 Version 6.5.0, 2019-03-30

This QP/C release matches the [QM release 4.5.0](#), which introduced new implementation for QP/C++. Even though this has no impact on the QP/C state machine implementation with QM, this release is needed for compatibility checking between QP and QM.

For commonality with the new QP/C++ implementation, this release adds macros [Q\\_ACTION\\_NULL](#) and [QM\\_STATE\\_NULL](#), which are used in the code generated from QM 4.5.0. Also, this release comes with several models for **QM 4.5.0**.

Additionally, this release changes the API `QTimeEvt_ctr()` to `QTimeEvt_currCtr()`. This is to avoid name shadowing with the `ctr` variables, which is not compliant with MISRA.

Additionally, the examples for `calc1` and `calc1_sub` (with QM 4.5.0 models) have been extended to properly handle the operator precedence (multiplication and division have higher precedence than addition and subtraction).

## 10.29 Version 6.4.0, 2019-02-10

This release brings the following changes:

- removed `QActive_stop()`
- added assertions to [QHSM\\_INIT\(\)](#), [QACTIVE\\_START\(\)](#) and [QXTHREAD\\_START\(\)](#) to make sure the the virtual-pinter (vptr) is initialized

### Note

These additional assertions require `Q_DEFINE_THIS_FILE` or `Q_DEFINE_THIS_MODULE` at the top of the .c file that calls [QHSM\\_INIT\(\)](#), [QACTIVE\\_START\(\)](#) or [QXTHREAD\\_START\(\)](#).

- modified examples for ThreadX to define `BSP_TICKS_PER_SEC` as `TX_TIMER_TICKS_PER_SECOND` instead of a hard-coded value
- modified examples for uC/OS-II to define `BSP_TICKS_PER_SEC` as `OS_TICKS_PER_SEC` instead of a hard-coded value
- modified qassert.h to work correctly when assertions are disabled with `#Q_NASSERT` and also updated the QP/C source code to build correctly without assertions
- improved the [QXK](#) semaphores to handle correctly the maximum number of tokens
- reduced the size of [QPSet](#) in case [QF\\_MAX\\_ACTIVE](#) does not exceed 8 or 16.
- improved performance of the `QF_qlog2()` algorithm when [QF\\_MAX\\_ACTIVE](#) does not exceed 8 or 16.
- modified Makefiles for workstations (Windows and POSIX) to use compiler options `-std=c99` for C and `-std=c++11` for C++ as well as `-pedantic` option.
- fixed examples for MSP430 with CCS and added QUTest support for MSP430.

## 10.30 Version 6.3.8, 2018-12-31

The main purpose of this release is the update the Makefiles that use the [GNU-ARM Embedded Toolchain](#) to match the recent update in [QTools for Windows 6.3.8](#). Specifically, all Makefiles in the examples directory have been updated to use the GNU-ARM toolchain located in `qtools\gnu_arm-none-eabi` and use the tools prefix **arm-none-eabi-**.

### Note

The Makefiles that use the GNU-ARM included in this release require [QTools for Windows version 6.3.8 or newer](#) and will **NOT** work with the older QTools collection.

## 10.31 Version 6.3.7, 2018-11-20

The main goal of this release is to provide Python test scripts that match the newly re-designed Python support for the [QUTest unit testing harness](#) (see [QTools 6.3.7](#)). Specifically, the Python tests scripts (\*.py files) in the qpc/examples/qutest directory have been all upgraded to the new syntax and format. Also, all the makefiles in that directory have been modified to execute Python test scripts by default, and Tcl test scripts only when the argument `SCRIPTS=tcl` is specified.

### Note

The new scripting interface `qutest.py` is now the primary supported QUTest scripting interface. The older interfaces, such as TCL ("qutest.tcl") and "qspypy" are considered *obsolete* and are **not recommended** for writing new test scripts.

The second change in this release is updating the QP ports to Win32 and POSIX to eliminate the obsolete function `gethostbyname()` and replacing it with `getaddrinfo()`. This change has ripple effects on Windows, because it requires linking with "ws2\_32" library, instead of "wsock2". All affected Makefiles in the `qpc\examples\` directory have been updated to link the newer "ws2\_32" library.

Also, in this release, the prototypes of the internal functions `QActive_start_()`, `QActive_get_()`, `QActive_post_()` and `QActive_postLIFO_()` have been removed from the public [QF](#) interface (in `qf.h`) and moved to the [QF](#)-implementation. This is to prevent calling these internal functions directly in the application-level code. The only allowed use of these functions is through the macros `QACTIVE_START()`, `QACTIVE_POST()`, `QACTIVE_POST_X()` and `QACTIVE_POST_LIFO()`.

Finally, this release updates the internal implementation of [QXK](#) function `QXThread_post_()` for software tracing and testing with QUTest.

## 10.32 Version 6.3.6, 2018-10-20

This release brings important changes and improvements to the unit-testing support for [QUTest](#). Specifically, a new "dummy" active object class [QActiveDummy](#) for testing has been added. Instances of this "dummy" AO can be now used as test-doubles for active objects that are recipients of events directly posted by the active object under test (AOUT). This, in turn, eliminates the need to alter the existing event-posting implementation, so that more of the actual QP code can be used in the QP test-stub (`qutest.c`).

### Note

The QUTest projects that build QP from sources need to include the `qf_time.c` file.

As a consequence of the changes in the [QUTest](#) support, the [QUTest examples](#) have been re-designed and improved. Here, the most important changes include the new code organization, which reflects the customary separation of the code-under-test (CUT) from the code for testing. Also, the tests based on the ["TDD book" by James Grenning](#) have been replaced with fully functional tests based on the [Unity testing framework](#). This is to directly compare the traditional approach (Unity) with QUTest.

The next change related to unit testing is adding `::QS_RX_QUERY_CURR` facility to QS-RX (software tracing input channel) and the reply `::QS_QUERY_DATA` to the [QS](#) output channel. These two extensions allow you to query the status of the "current object" inside the target. The most important example is querying the current state-machine object (`::SM_OBJ`), which returns the **current state**.

### Note

The new "query" facilities in QP/C match the latest additions to the [QSPY host application](#).

Also this release adds [QS dictionary](#) generation for all registered event pools. The generated object dictionaries are `EvtPool1` for pool-1, `EvtPool2` for pool-2, etc.

**Note**

Applications no longer need to generate [QS](#) object dictionaries for the event pool storage buffers, such as `smlPoolSto[]`, etc.

Another big change in this release is re-designing of the [examples for workstations](#) (previously `win32`, `win32-qv`, `posix` and `posix-qv`). All these types of examples have been consolidated in the [examples/workstation](#) folder, where the provided Makefiles have been extended to be cross-platform so that they work on Windows, Linux, and MacOS without changes.

To facilitate the creation of truly portable, cross-platform examples, the existing [QP ports to Windows/POSIX](#) have been augmented with abstractions for portable console access. Also, the [QS software tracing](#) support via TCP/IP has been now added to the ports themselves. This means that applications no longer need to repeat the code for [QS](#) callbacks in the BSP implementation.

Finally, this release fixes a bug in the [POSIX-QV port](#), where the internal condition variable `QV_condVar_` has not been initialized.

## 10.33 Version 6.3.4, 2018-08-10

This release adds new API `QTimeEvt_wasDisarmed()` for checking the status of a [QTimeEvt](#) object after it has been disarmed. Specifically, the status of the last call to `QTimeEvt_disarm()` is kept inside the time event object and can be subsequently checked with the `QTimeEvt_wasDisarmed()` API. This new function is designed to be used directly as a guard condition on the timeout event, as described in the [PSiCC2 book](#), Section 7.7.3 "Arming and Disarming a Time Event" on page 359. The `QTimeEvt_wasDisarmed()` has a side effect of setting the "was disabled" status, so the guard evaluates to 'true' the next time it is checked.

## 10.34 Version 6.3.3a, 2018-07-16

This release adds Python test scripts to the QUTest examples (folder `qpc/examples/qutest`). Specifically, the makefiles have been augmented to accept symbol `SCRIPT=py`, in which case the Python test scripts (`*.py`) are used instead of the default Tcl test scripts (`*.tcl`).

**Remarks**

This release does not change any QP/C APIs, QP/C implementation, ports, or other examples.

## 10.35 Version 6.3.3, 2018-06-22

This release fixes the following bugs:

- [bug#216 "Generated code for QP version check violates MISRA-C Rule 10.1".](#)
- [bug#217 "GNU-ARM -fstack-protector incompatible with the QP ports to GNU-ARM".](#)

Also, this release demonstrates the new features of QM 4.3.0 in several example models (`qpc/examples/` directory). Finally, this release updates `3rd_party/CMSIS/Include` to the latest version from GitHub.

## 10.36 Version 6.3.2, 2018-06-20

This release fixes the following bugs:

- [bug#215 "QP is internally inconsistent in calling assertion macros".](#)

Also, this release improves the QUTest DPP example (directory `qpc/examples/qutest/dpp`) by demonstrating the proper use of `QS_TEST_PAUSE()` and the corresponding test scripts. This example now matches the [QUTest documentation of this feature](#).

Finally, this release modifies the QP/C ports to POSIX and POSIX-QV by allowing to configure the p-thread priority of the ticker thread. This is achieved by adding a `tickPri` parameter to the `QF_setTickRate()` function. (**NOTE** this modification will require changing existing QP Applications for POSIX or POSIX-QV that call `QF_setTickRate()`).

## 10.37 Version 6.3.1, 2018-05-24

This release migrates the `QUTest` examples to `QM 4.2.1`, which now can generate `QS_FUN_DICTIONARY()` records automatically. This release also adds a generic, simple blinky example for QUTest located in `examples/qutest/blinky`. Also, this release fixes a bug in the example `qutest/TDDbook_LedDriver` so that the code compiles cleanly with the updated header file "qassert.h".

## 10.38 Version 6.3.0, 2018-05-10

The main purpose of this release is fixing the sub-machine support in the QP/Spy build configuration. Specifically, this release fixes the following bug:

- `bug#213 "QP/C/C++ applications with submachines occasionally crash in Spy build configuration"`

### Attention

This release matches `QM 4.2.0`.

Additionally, the release contains some re-factoring of the QS-RX input channel.

## 10.39 Version 6.2.0, 2018-03-16

The main purpose of this release is extending the functionality of the `QUTest unit testing` for QP/C applications. Specifically, this release adds support for testing of **self-posting** of events in active objects, which is an essential element in the `Reminder` and `Deferred Event` design patterns. To implement this new feature, the QS-RX (QS receive channel) has been extended by a small scheduler that processes all secondary events generated by dispatching, posting, or publishing events (only active when the `Q_UTEST` macro is defined). Also, the implementation of the target resident QUTest components (files `src/qs/qutest.c` and `src/qf/qf_actq.c`) have been modified to allow posting of events during unit testing.

Additionally, the release adds standard QS trace records (the `qs.h` header file) for event deferring and recalling (`::QS_QF_ACTIVE_DEFER` and `::QS_QF_ACTIVE_RECALL`) as well as recall-attempt (`::QS_QF_ACTIVE_RECALL_ATTEMPT`). Also standard QS trace records have been added for creating/deleting new event references (`::QS_QF_NEW_REF` and `::QS_QF_DELETE_REF`, respectively). To make room for these new records, the following rarely-used records have been removed: `QS_QF_ACTIVE_ADD`, `QS_QF_ACTIVE_REMOVE`, `QS_QF_EQUEUE_INIT`, and `QS_QF_MPOOL_INIT`. The global filter settings in the `QS_filterOn()` and `QS_filterOff()` functions have been updated to the re-organized QS trace records.

### Note

Because of the changes in the standard QS trace records, this release requires the matching **QSPY 6.2.0** or later. Also, because of the changes this release might break some existing QUTest unit test scripts, which need to be re-adjusted to the new trace records.

Additionally, this release improves support for tracing and unit-testing embedded POSIX targets, such as embedded Linux, or OSes/RTOSes with the POSIX compatibility layer. Specifically, the POSIX ports (`qpc/ports posix/` and

`qpc/ports/posix-qutest/`) no longer produce libraries. Instead all examples for POSIX (`qpc/examples/posix/` and `qpc/examples/qutest`) build the QP/C framework directly from the sources, which promotes consistency in the toolchain and options used. For unit testing, the makefiles `posix.mak` have been added for projects `qpc/examples/qutest/dpp/` and `qpc/examples/qutest/self_test/`. These makefiles support remote unit testing of embedded POSIX targets over TCP/IP, where the POSIX target runs only the `test fixture`, but the host (e.g., Windows host) executes both the QSPY host application and runs the `QUTest scripts`. The previous makefiles for POSIX have been renamed to `posix_host.mak`, because they are intended to use POSIX as a *host*, as opposed to a *target*.

Also, this release adds new QUTest examples, which illustrate:

- event deferral (`qpc/examples/qutest/defer/`)
- dispatching/posting events with parameters (`qpc/examples/qutest/evt_par/`).

Additionally, this release updates the QP/C ports to `win32-qv` and `posix-qv` to allow a "tickless" mode, where the "tickerThread" is not created. This mode is set by configuring the system clock tick rate to 0 (`QF_setTickRate(0)`). Finally, this release phases out the `qp_port.h` header file. If any of your projects still includes this file, please replace it with the `qpc.h` header file.

## 10.40 Version 6.1.1, 2018-02-18

The main purpose of this release is adding the context-switch callbacks to the preemptive `QK` and `QXK` kernels. The `QK_onContextSw()` and `QXK_onContextSw()` callback functions provide a mechanism to perform additional custom operations when `QK/QXK` switches context from one thread to another. To avoid extra overhead when this functionality is not needed and for backwards-compatibility with the existing applications, the callbacks are enabled only when the macros `#QK_ON_CONTEXT_SW` (for `QK`) and `#QXK_ON_CONTEXT_SW` (for `QXK`) are defined. These macros can be defined either directly in command-line for the compiler, or in the `QK/QXK` port files (`qk_port.c` for `QK` and `qxk_port.c` for `QXK`). Examples for the context-switch callbacks have been provided for the NUCLEO-L053R8 (Cortex-M0+) and the NUCLEO-H743ZI (Cortex-M7).

Also, this release changes the ARM Cortex-M ports for the IAR-ARM toolchain in that it replaces the assembly modules with the equivalent C implementation. This change enables using the configuration macros `#QK_ON_CONTEXT_SW` (for `QK`) and `#QXK_ON_CONTEXT_SW` (for `QXK`) in the ports. All existing example projects for IAR-ARM have been updated to use the `q(x) k_port.c` files instead of `q(x) k_port.s` files.

Also, this release adds new project files for the Atollic TRUEStudio for STM32. The TRUEStudio projects (Eclipse) have been added for the NUCLEO-L053R8 (Cortex-M0+) and the NUCLEO-H743ZI (Cortex-M7).

Finally, this release updates the CMSIS to version 5.3.0 (see `qpc/3rd_party/CMSIS`)

## 10.41 Version 6.1.0, 2018-02-04

The main purpose of this release is adding the support for the `ARM Compiler 6 (ARM-Clang)`, which is a C/C++ toolchain for ARM processors based on the modern `Clang frontend` and the `LLVM framework`. This release adds the ARM-Clang ports and examples for all built-in kernels (`QV`, `QK`, and `QXK`) for the ARM Cortex-M CPU cores.

This release also adds support for the STM32H7 high-performance Cortex-M7 with the `double-precision FPU` (FPV5-DP-D16-M). Specifically, this release provides examples for the `NUCLEO-H743ZI board` (Cortex-M7 with FPV5-DP-D16-M). The examples for NUCLEO-H743ZI board include all built-in kernels with ARM-CLANG, ARM-KEIL, GNU-ARM, and IAR-ARM. Additionally the NUCLEO-H743ZI examples also include the QP FreeRTOS with ARM-KEIL, GNU-ARM, and IAR-ARM.

Also, this release changes the ARM Cortex-M ports for the GNU-ARM toolchain in that it replaces the assembly modules with the equivalent C implementation. This change enables using wider range of GNU-ARM toolchain distributions, such as GNU-ARM Linaro, which had trouble with the assembly modules, but compiles correctly the C modules. All related GNU-ARM example projects have been modified to use the new C-ports.

Also, this release updates the existing projects for the Code Composer Studio (CCS) to CCSv7. All existing CCS projects for the EK-TM4C123GXL (TivaC LaunchPad) and LaunchXL2-TMS57012 (Hercules) boards have been

upgraded and tested.

Also, all examples for MSP430 with CCS have been updated to CCSv7 and re-tested on the supported boards.

## 10.42 Version 6.0.4, 2018-01-10

The main purpose of this release is the provision of the official [QP/C port to FreeRTOS](#) (version 10). The QP/C port to FreeRTOS is completely generic and should work on any CPU supported by FreeRTOS. The port comes with the following examples (see `examples/freertos`):

- DPP on EK-TM4C123GXL (ARM Cortex-M4F) with ARM-KEIL, GNU-ARM and IAR-ARM toolchains.
- DPP on STM32F746G-Discovery (ARM Cortex-M7) with ARM-KEIL, GNU-ARM and IAR-ARM toolchains.

This release also replaces assembly with C implementation in the ARM-KEIL ports or the [QK](#) and [QXK](#) kernels to Cortex-M to take advantage of the `__asm` functions. (NOTE: this change has impact on the existing QP/C applications that use the ARM-KEIL toolchain and the [QK](#) or [QXK](#) kernels.)

Additionally, this release fixes some problems with the native examples for STM32F4 and STM32F7 boards.

Additionally, this release adds a generic function `QActive_SetAttr()` to set thread attributes in various QP ports to 3rd-party RTOSes. This function is then used in the [embOS port](#) and the [uC/OS-II port](#).

Finally, this release fixes the following bug in the [ThreadX port](#):

- `bug#197 "Problems with scheduler locking in QP/C/C++ ports to ThreadX"`

## 10.43 Version 6.0.3, 2017-12-12

Changes since version 6.0.1:

- fixed `bug#193 "QXK: context switch from extended to basic thread fails"`
- replaced attributes of [QActive](#) and [QEQueue](#) classes from type `uint_fastX_t` to `uintX_t`. For the ARM Cortex-M port, these changes reduce the memory footprint of each [QActive](#) instance from 50 to 38 bytes (25% improvement).
- improved the performance of the ARM Cortex-M0/M0+ ports, which do not support the CLZ (count leading zeros) instruction. All QP ports to Cortex-M0/M0+ (qv, qk and qxk) now contain hand-optimized assembly implementation of the LOG2 (log-base-2) function. The implementation executes only in 14 machine instructions and uses only 16-byte look-up-table (LUT). This replaces LOG2 implementation based on a 256-byte LUT.
- changed the interrupt disabling policy in ARM Cortex-M ports back to "unconditional interrupt disabling", which is simpler and faster than "saving-and-restoring interrupt status".
- added a port to ARM Cortex-M with the LLVM compiler (currently only for the dual-mode [QXK](#) kernel). Unlike the other ARM Cortex-M ports, the port to LLVM uses the "saving-and-restoring interrupt status" interrupt disabling policy (explicit request from a commercial customer).
- removed ARM Cortex-M port and examples for the TI-ARM compiler. (Projects for the TI CCS can use the GNU-ARM toolchain and the existing GNU-ARM ports).
- Added the DPP-COMP example (Dining Philosophers Problem) with the "Orthogonal Component" state pattern located in `qpc/examples/win32/dpp-comp`. This example demonstrates also a partitioned QM model into external packages for the Container (Table active object) and the Components (Philo active objects).

### Note

This release does not change any of the QP/C APIs.

## 10.44 Version 6.0.1, 2017-11-10

The main focus of this release is to fix the remaining problems with transitions out of eExit-Points in sub-machines. Specifically, this release modifies the QMsm-based state machine implementation strategy (file [src/qf/qep msm.c](#) to properly handle transitions from eExit-Points to Entry-Points and from eExit-Points to History connectors in sub-machines. These changes are part of fixing the following bugs reported for QM:

- [bug#190 "Exit-Point segment targeting History doesn't work"](#)
- [bug#189 "Exit-Point segment targeting an Entry-Point to sub-machine state doesn't work"](#)

Additionally, this release fixes the following bug in transitions to "shallow history":

- [bug#191 "Transition to shallow history in QMsm causes assertion qep msm#810"](#)

The bug#191 is fixed by modifying the function `QMsm_childStateObj_()` in [qep msm.c](#) to return the parent state in the corner case when the current state is the parent state.

### Note

This QP/C 6.0.1 release is the minimum version required in QM 4.1.0. This is because QM 4.1.0 assumes the modified QMsm-state machine implementation strategy in order to properly handle the various transitions out of eExit-Points in sub-machine states.

Additionally, this release changes the [QXK](#) implementation related to the [bug#186 "QXK: Extended thread context switch causes assertion in PendSV\\_Handler"](#). Specifically, the case of context switching away and back to the same thread (which can arise under specific interrupt preemption scenarios) is now handled as a simple return from PendSV. The [QXK](#) scheduler has been modified to set the "next" thread pointer to NULL when it detects switching back to the "current" thread.

## 10.45 Version 6.0.0, 2017-10-13

This release fixes two bugs found in the [QXK](#) kernel:

- [bug#185 "QXK: PendSV\\_Handler uses inconsistent stack frames for saving and restoring AO for Cortex-M0\(+\)"](#)
- [bug#186 "QXK: Extended thread context switch causes assertion in PendSV\\_Handler"](#)

Additionally, this release includes a fix for the bug found in ARM Cortex-M0 port with the GNU-ARM compiler:

- [bug#189 "QS\\_END\(\) leaves all interrupts disabled with GNU-ARM"](#)

This specific problem observed in QSpy turned out to be caused by a bug in the GNU-ARM compiler itself. This problem affected the ARMv6-M architecture (Cortex-M0/M0+/M1) and manifested itself in generation of incorrect code for the QP critical section at certain gcc optimization levels (such as -O). This bug was first discovered and filed as [bug#184](#). The bug affected the GNU-ARM ports to all built-in kernels [QV](#), [QK](#), and [QXK](#).

### Attention

This release no longer contains the directory `qpc/source`, which was scheduled to be phased out in QP5. In QP6 the source code is found only in the `qpc/src` directory.

## 10.46 Version 5.9.9, 2017-09-29

This release implements the feature request [#132 "Extend the QXK mutex to support also simple operation without the priority-ceiling protocol"](#).

**Note**

This release is completely backwards-compatible with the [previous QP/C release 5.9.8](#).

Specifically, the [QXMutex](#) class has been extended as follows:

- a mutex initialized with `ceiling==0` (`QXMutex_init(0)`) means "no ceiling", so such mutex will NOT use the priority-ceiling protocol.
- in this case the mutex will NOT require a unique priority level
- in this case the mutex will not change (boost) the priority of the holding thread in the `QXMutex_lock()` operation.
- in this case the mutex will still support nesting of locks (as before), up to 255 levels of nesting.
- when initialized with `ceiling>0`, [QXMutex](#) WILL use the priority-ceiling protocol, as before. It will require that the ceiling priority be unique and not used by any other thread or mutex. In other words, the previous functionality remains unchanged.

## 10.47 Version 5.9.8, 2017-09-15

This release fixes the [QXK kernel bug#182 "Inconsistent QXThread\\_post\\_\(\) behavior with respect to the 'margin' parameter"](#).

Also, the pre-condition assertion in the function `QF_newRef_()` (file `src/qf/qf_dyn.c`) has been modified to allow creating event references only for dynamic events (`e->poolId_ == 0`).

Improved comments in the source code.

Modified ARM Cortex-M examples with the GNU-ARM toolset to be consistent with QP/C++ (support for RTTI and C++ Exception handling), as follows:

- removed definitions of `_init()` and `_fini()` from the GNU-ARM startup code for all supported boards in the `3rd_party` directory
- commented out the call to `__libc_init_array()` in the GNU-ARM startup code for all supported boards in the `3rd_party` directory (because it requires `_init()`)
- Modified all GNU-ARM linker scripts (.ld files) to add the following symbols:

```
__exidx_start = .;
.ARM.exidx : { *(.ARM.exidx* .gnu.linkonce.armexidx.*) } >RAM
__exidx_end = .;
```

## 10.48 Version 5.9.7, 2017-08-18

The main focus of this release are new requested features for the [dual-mode QXK kernel](#):

- [feature#129 "Allow blocking while holding a mutex in QXK"](#); and
- [feature#130 "Allow QXK mutex locks to nest while acquired by the same thread"](#); and
- [feature#131 "Add a non-blocking "tryLock\(\)" operation to the QXK mutex"](#).

Additionally, this release adds also the non-blocking `QXSemaphore_tryWait()` operation.

In the process of re-implementing the [QXMutex](#) class, the previous non-blocking priority ceiling mutex has been replaced with an equivalent selective [QXK](#) scheduler locking up to the specified ceiling priority. Specifically, this feature has been implemented with two new operations `QXK_schedLock()` and `QXK_schedUnlock()`.

For consistency, the non-blocking mutex of the [QK kernel](#) has been also replaced by the operations `QK_schedLock()` and `QK_schedUnlock()`.

All related [QXK](#) and [QK](#) examples have been updated to use the selective scheduler locking instead of the mutex. The new blocking [QXK](#) mutex has been demonstrated in the following updated examples:

- arm-cm\_dpp\_efm32-slstk3401a
- arm-cm\_dpp\_ek-tm4c123gxl

#### Attention

The changes to [QXMutex](#) in [QXK](#) and the now obsolete QMutex in [QK](#) **break backwards-compatibility** with the existing code that relies these features. In the existing code, the mutexes should be replaced with selective scheduler locking.

Also, the blocking APIs with timeouts, such as `QXMutex_lock()`, `QXSemaphore_wait()`, `QXThread_delay()`, and `QXThread_queueGet()` no longer require the last `tickRate` parameter. This also **breaks backward-compatibility** with the existing code that uses these operations.

This release also updates CMSIS to version 5.1.0 (3rd\_party/CMSIS).

This release also adds the [API Reference](#) section to the QP/C documentation.

Finally, this release fixes the following bug:

- [bug#181 "Win32-qv port and QF\\_run "idle" time"](#)

## 10.49 Version 5.9.6, 2017-08-04

The main focus of this release are improvements to the "dual-mode" [QXK](#) kernel. Specifically, this release implements the [feature request #128 "QP Semaphore Max Value Setting"](#) for [QXK](#). This feature changes the [QXK](#) function `QXSemaphore_init()`, which now takes additional parameter `count`. This parameter specifies the maximum allowed count for the semaphore (e.g., `count` of 1 makes the semaphore a binary-semaphore).

#### Note

This change breaks backwards-compatibility with the existing code that uses the `QXSemaphore_init()` function.

This release also fixes a bug inside the assertions in `QXSemaphore_signal()`, where the check for the extended-thread is performed.

Also, this release adds assertions to the [QXK](#) code, which ensure that any available blocking calls can only be made from extended-threads and not from basic-threads (active objects) or ISRs.

Also, this release adds protection in the IRQ priorities initialization in [QK/QV/QXK](#) for ARM Cortex-M3/M4/M7, so that the number of IRQs is extracted from bits 0-2 of the ICTR register (INTLINESNUM).

Finally, this release consistently changes all example projects (for all toolchains) to use the `src` directory for QP/C source code, instead of the `source` directory. The `source/` directory is now truly obsolete, but is still provided in this release for backwards compatibility with user projects.

## 10.50 Version 5.9.5, 2017-07-20

This release changes the macro `QXTHREAD_START()` in the [QXK](#) kernel so that it can be used only with [QXThread](#) pointers and not [QActive](#) pointers.

#### Note

The change of `QXTHREAD_START()` has impact on existing [QXK](#) applications, because the calls like `QXTHREAD_START (&XT_Test1->super ~~~)` need to be changed to `QXTHREAD_START (XT_Test1 ~~~)`, where `XT_Test1` is a pointer to `QXThread *`.

This release fixes the following bugs:

- [bug#178 "GNU-ARM compiler reports "Error: unaligned opcodes~~~" in startup code for QP/C/C++/nano examples". The bug fix entails modifying the startup code for the GNU-ARM compiler in the 3rd\\_party directory. Specifically, the proper alignment directives have been added to the inline assembly in the exception handlers.](#)

- bug#179 "Assertion ID 210 fires when signaling on a QXK semaphore"

This release fixes the naming problem of the startup code for the STM32F7-Discovery board (in the `3rd_party/stm32f7-discovery/gnu/` and `arm/` directories), where the startup code was renamed from `startup_stm32f4xx.c/s` to `startup_stm32f746xx.c/s`. The change has been also made in the example projects for the STM32F7-Discovery board (for ARM-KEIL and GNU-ARM toolsets).

## 10.51 Version 5.9.4, 2017-07-07

This release adds Thread-Local Storage (TLS) feature for the dual-mode `QXK` kernel (see `srs-qp_qxk_tls`).

## 10.52 Version 5.9.3, 2017-06-19

This release implements the feature request #126 "Allow non-asserting event allocation for zero-margin allocations". Specifically, calling `Q_NEW_X()` or `QACTIVE_POST_X()` with the margin argument of zero will no longer assert if the allocation/posting fails.

## 10.53 Version 5.9.2, 2017-06-05

This release adapts the Makefiles for GNU-ARM to the new location of the GNU-ARM toolset, which is now included in the QTools Collection (v 5.9.1) for Windows.

Also, this release improves the flash loading scripts for the JLink hardware debugger (for GNU-ARM projects for the EFM32-SLSTK3401A board). Specifically, the JLink configuration file for flash download is generated by the flash batch script based on the command-line parameter (the binary file to load into the flash). This eliminates the need to manually maintain JLink configuration files.

Also, this release adds bi-directional QP/Spy to the embOS example project for the STM32F4-Discovery board.

Also, this release adds GNU-ARM port to uC/OS-II and adds GNU-ARM example project for the EK-TM4C123GXL board.

Finally, this release implements the feature request #125 "Include QPC Demo application for STM32F4 processor without RTOS" (see <https://sourceforge.net/p/qpc/feature-requests/125/>). The DPP demo for the STM32F4-Discovery board has been added in the directory: `qpc/examples/arm-cm/dpp_stm32f4-discovery`. This demo includes `QV`, `QK` and `QXK` kernels and ARM-Keil, GNU-ARM, and IAR-ARM toolsets. The demos support bi-directional QP/Spy.

## 10.54 Version 5.9.1, 2017-05-26

This release fixes the following bug:

- bug#169 "Submachine-state eXit Point does not work correctly"

### Note

The bug only affects sub-machines and does not affect any other aspects of QMsm-style state machines.

Also, this release changes the organization of the QP/C source code to make it more friendly for the Eclipse CDT, as proposed in the feature request #123 "Eclipse-friendly source directory structure for QP/C/C++". Specifically, the QP/C source code is now provided in the `qpc/src/` directory with the following structure:

```
qpc/
+-source/ - existing source directory with "flat" structure
 |       (for backwards-compatibility)
 +-src/   - new source directory grouped by functionality
   +-qf/   - core framework (QEP + QF)
```

```

+-qk/      - QK kernel
+-qv/      - QV kernel (only one file qv.c)
+-qxk/      - QXK kernel
+-qs/      - QS software tracing
+-qf_pkg.h
+-qs_pkg.h
+-qxk_pkg.h

```

**Note**

The original `qpc/source` directory is still provided for backwards compatibility with the existing QP/C projects. This directory will be phased out in the future QP/C releases. Please use the new source code structure provided in the `qpc/src` directory.

## 10.55 Version 5.9.0, 2017-05-19

The main purpose of this milestone QP/C release is to provide support for the powerful **Unit Testing Framework** called **QUTest** (*pronounced cutest*). QUTest is the fundamental tooling for Test-Driven Development (TDD) of QP/C applications, which is a highly recommended best-practice. This release introduces changes in the QS-RX (receive) channel and adds several new callbacks.

**Note**

The signature of the `QS_onCommand()` has changed and the function now takes 3 arbitrary 32-bit parameters instead of one. This introduces backwards-incompatibility with previous code that used `QS_onCommand()`.

This release also changes the critical section for QP/C ports to ARM Cortex-M in that the policy of "save and restore interrupt status" is used. This policy permits nesting of critical sections, which was requested by customers. Additionally, this release changes the selective interrupt disabling for ARM Cortex-M3/4/7 (with the BASEPRI register) to address the hardware problem on ARM Cortex-M7 core r0p1 ([SDEN-1068427](#), [erratum 837070](#)). The QP ports to ARM Cortex-M3/4/7 now implement the workaround recommended by ARM, which is to surround MSR BASEPRI with the "CPSID i"/"CPSIE i" pair. This workaround works also for Cortex-M3/M4 cores.

New ports:

- `ports/win32-qutest` folder contains port to QUTest for Windows
- `ports posix-qutest` folder contains port to QUTest for POSIX (Linux)
- `ports/arm-cm/qutest` folder contains port to QUTest for ARM Cortex-M

New examples:

- `examples/qutest/dpp` folder contains the QUTest DPP test for various platforms (Win32, EFM32-SLSTK3401A and EK-TM4C123GXL)
- `examples/qutest/qhsmtst` folder contains the QUTest test for the QHsmTst state machine (structural test)
- `examples/qutest/qmsmtst` folder contains the QUTest test for the QMsmTst state machine (structural test)
- `examples/qutest/self_test` folder contains the QUTest self-test for various features of QUTest
- `examples/qutest/TDDbook_Flash` folder contains the QUTest of a flash memory driver from Chapter 10 of the "TDD-book" by James Grenning.
- `examples/qutest/TDDbook_LedDriver` folder contains the QUTest of a flash memory driver from Chapters 3&4 of the "TDD-book" by James Grenning.
- `examples/qutest/TDDbook_Sprintf` folder contains the QUTest of a flash memory driver from Chapter 1 of the "TDD-book" by James Grenning.

Updates of 3rd\_party software:

- the 3rd\_party/CMSIS folder has been updated to CMSIS 5.0.2.
- the 3rd\_party/embOS folder has been updated to embOS 4.34.1

Finally, this release fixes the following bugs:

- bug#162 "QF critical sections require modification for M7 core"

## 10.56 Version 5.8.2, 2017-02-08

This release adds examples for the ARM Cortex-M7 CPU. Specifically, the release contains the standard [Dining Philosophers Problem \(DPP\)](#) examples for the STM32F746G-Discovery board, all built-in kernels ([QV](#), [QK](#), and [QXK](#)), and ARM-KEIL, IAR EWARM, GNU-ARM toolsets.

To provide examples for STM32F746G-Discovery board, the release now provides the folder `3rd_party/stm32f7-discovery` with the support code for the STM32F7xx MCUs, which contains parts of STM32CubeF7 library.

Also, the `3rd_party/CMSIS` folder now provides the new CMSIS V5.0.1.

Finally, this release fixes the following bugs:

- bug#159 QP/C/C++ Win32 ports don't work on all x86 CPUs
- bug#157 In QPC ucosii port, conversion of AO's priority to OS task priority is incorrect.
- bug#152 Typo (qpc/ports/arm7-9/qk/gnu/qk\_port.s:42) prevents compilation

## 10.57 Version 5.8.1, 2016-12-16

This release is in response to a recent finding that many QP users of the ports to ARM Cortex-M3/M4 forget to explicitly set their interrupt priorities, as described in the AppNote "[Setting ARM Cortex-M Interrupt Priorities in QP 5.x](#)".

Specifically, this release improves safety of QP ports to ARM Cortex-M3/M4, by initializing the interrupt priorities to a safe default in a generic, portable way. This QP port includes such a fix for QV/QK/QXK ports to ARM Cortex-M3/M4. Additionally, this release introduces the new [QTicker](#) class, which is an efficient active object specialized to process [QF](#) system clock tick at a specified tick frequency [0..([QF\\_MAX\\_TICK\\_RATE](#) - 1)]. Placing system clock tick processing in an active object allows you to remove the non-deterministic [QF\\_TICK\\_X\(\)](#) processing from the interrupt level and move it into the thread-level, where you can prioritize it as low as you wish.

Changes in detail:

- modified the [QV](#), [QK](#), and [QXK](#) source code to call `QV_init()`, `QK_init()`, and `QXK_init()`, respectively.
- modified the ARM Cortex-M ports of [QV](#), [QK](#), and [QXK](#) to initialize priorities all exceptions and IRQs to the safe value `#QF_BASEPRI`.

### Note

The [QV](#) port now has a new `qv_port.c` module that needs to be added to the build.

- added declaration of the [QTicker](#) class to `qf.h`
- added implementation of the [QTicker](#) class to `qf_actq.c`
- modified the following examples to demonstrate the use of the [QTicker](#):
  - `qpc/examples/arm-cm/dpp_efm32-slstk3401a/qk`
  - `qpc/examples/arm-cm/game_efm32-slstk3401a/qv`
  - `qpc/examples/arm-cm/game_efm32-slstk3401a/qk`
- added the header file `cmsis_ccs.h` to `qpc/3rd_party/CMSIS/Include` directory (used in the examples for the Code Composer Studio). The file has been dropped during the upgrade to CMSIS 5.0.1, because it is not part of the standard distribution.

## 10.58 Version 5.8.0, 2016-11-30

The main purpose of this milestone QP/C release is to finally provide the baseline framework fully compatible with the upcoming QM 4.0.0.

This release changes the class hierarchy so that [QHsm](#) becomes the base class of [QMsm](#) and [QActive](#). Also, [QActive](#) becomes the base class of [QMActive](#), which reverses the changes introduced in version 4.1.

The modified class hierarchy better reflects the fact that [QHsm](#) state machine implementation strategy is simpler and supports less functionality than the more advanced [QMsm](#) strategy. For example, only the [QMsm](#) class fully supports sub-machines and sub-machine states that are the main feature of QM 4.x. This clean progression of supported functionality from subclasses to superclasses allows QM to easier check and enforce that advanced features are not generated for subclasses that don't have the required capabilities. (With previous class hierarchy with [QMsm](#) as the base class all subclasses, including [QHsm](#), would technically inherit the advanced functionality, which is not the case).

### Note

All changes in QP/C 5.8.0 remain transparent for the existing QP/C applications, because of the provided backwards compatibility layer in [qpc.h](#).

Also, this release changes the implementation of the [QV](#), [QK](#), and [QXK](#) kernels in that the ready-set representing active threads is cleared only *after* completion of the RTC-step, not when the last event is removed from the corresponding event queue. In case of the [QXK](#) kernel this change fixes the high-priority [bug#147](#). But even in case of the [QV](#) and [QK](#) kernels, where this behavior didn't lead to any bugs, the policy better reflects the semantics of the ready-set.

This release also updates the CMSIS interface included in the 3rd\_party/CMSIS folder to the latest CMSIS-5.

All examples and QM models have been updated to the new upcoming QM 4.0.0. All these models require QM 4.x.

Finally, the complete list of bugs fixed in this release is as follows:

- [bug#147](#) "QXK: PendSV\_error is triggered on special conditions"
- [bug#146](#) "Misra-C 2004 warning for rule 8.3 in qxk.c"
- [bug#144](#) "Obsolete Win32 API in qwin\_gui.c"
- [bug#143](#) "QACTIVE\_POST\_LIFO() on initial transition asserts on QXK"
- [bug#124](#) "Windows port now cause memory leakage"

## 10.59 Version 5.7.4, 2016-11-04

This release fixes the following bugs:

- [bug#145](#) [QF\\_PUBLISH\(\)](#) leaks events that have no subscribers
- [bug#144](#) Obsolete Win32 API in qwin\_gui.c
- [bug#143](#) [QACTIVE\\_POST\\_LIFO\(\)](#) on initial transition asserts on [QXK](#)

## 10.60 Version 5.7.3, 2016-10-07

This release adds QP ports to the TI-RTOS kernel (SYS/BIOS) with TI-CCS and IAR EWARM toolsets. Examples are provided for the EK-TM4C123GXL (TivaC LaunchPad) in the directory:

[qpc/examples/ti-rtos/arm-cm/dpp\\_ek-tm4c123gxl](#)

NOTE: The examples require a separate installation of the TI-RTOS (file [tirtos\\_tivac\\_setupwin32\\_2\\_16\\_01\\_14.exe](#))  
Also, this release fixes the following bugs:

- [bug#140](#) (PendSV\_Handler() exception stacked PC not halfword aligned).
- [bug#142](#) (PendSV\_restore\_ex may not be able to enable interrupt before returning to task).

## 10.61 Version 5.7.2, 2016-09-30

This is the first production release of the "dual-mode" **QXK** kernel. "Dual-mode" **QXK** means that **QXK** supports both basic-threads (BC1 class from the OSEK/VDX RTOS specification) as well as extended-threads (EC1 class from the OSEK/VDX RTOS specification). In other words, **QXK** executes active objects (basic threads) like the **QK** kernel using the single stack (Main Stack on ARM Cortex-M), but can also execute traditional *blocking* threads (extended threads). Only the extended threads (**QXThread** class) need their private stack spaces and the overhead of the full context switch. The basic threads (**QMActive** and **QActive** classes) run efficiently using the main stack with much lower context switch overhead.

The **QXK** examples have been updated for more thorough demonstration of the **QXK** features. The **QXK** examples are available in the following directories: dpp\_efm32-slstk3401a, dpp\_ek-tm4c123gxl, and dpp\_nucleo-l053r8.

This release fixes several issues in **QXK** 5.7.1-beta with handling timeouts while blocking in extended-threads, such as timed blocking on event queues and semaphores.

This release also changes the internal **QK** implementation to match the terminology applied in the **QXK** kernel (e.g., **QK\_sched\_()** has been renamed to **QK\_activate\_()** and **QK\_schedPrio\_()** to **QK\_sched\_()**). These changes fall into the category of refactoring and have no impact on the API or performance.

Finally, this release improves the implementation of scheduler locking in publish-subscribe event delivery.

## 10.62 Version 5.7.0, 2016-08-31

This release adds support for sub-machines and sub-machine states for reusing pieces of state machines (an advanced UML concept) to the QMsm-state machine implementation strategy. This feature is to match the upcoming QM 4.0.0.

Also, this release adds support for the ARM Cortex-R processor. Specifically, the release contains a generic port to ARM Cortex-R with the IAR and TI-CCS toolsets and examples for the TI Hercules TMS570LS12x safety MCU (LAUNCHPADXL2-TMS57012).

Also, this release changes once more the **QK** port to ARM Cortex-M, to reduce the interrupt latency. This has been achieved by shortening the critical section in the PendSV exception.

Also, this release changes slightly the **QXK** port to ARM Cortex-M, where again the critical section in PendSV has been slightly shortened.

Finally, this release replaces all absolute paths with relative paths in all CCS-Eclipse project files (for TivaC, Hercules, and MSP430).

Changes in detail:

1. Modified `qep msm.c` to correct the support for sub-machines and sub-machine states
2. Added new port to ARM Cortex-R in the directory `ports/arm-cr`
3. Added examples for ARM Cortex-R in the directory `examples/arm-cr`
4. Modified the ARM Cortex-M **QK** ports (ARM-KEIL, GNU, IAR, and TI)
5. Modified the ARM Cortex-M **QXK** ports (ARM-KEIL, GNU, IAR, and TI)

## 10.63 Version 5.6.5, 2016-06-06

This release adds support for the new board: EFM32-SLSTK3401A (Pearl Gecko Starter Kit from Silicon Labs). This board replaces the Stellaris EK-LM3S811 board, which has been discontinued. (The Stellaris EK-LM3S811 board had been used in the "Fly 'n' Shoot" game example accompanying the PSiCC2 book).

This release also introduces a new version of the QWIN GUI Toolkit in the Windows prototypes for the "Fly 'n' Shoot" game and the DPP-GUI version (see <https://www.state-machine.com/products/qtools/#QWin>). Additionally, this release also includes the QP/C integration with the emWin embedded GUI from SEGGER, which is also the same product as uC/GUI distributed by Micrium (exa\_emwin).

Finally, this release comes with updated project files for TI Code Composer Studio (both for ARM Cortex-M and for MSP430).

This release fixes the following bugs:

- bug#130 (POSIX port stop->start leads to reuse of destroyed mutex).
- bug#131 (QF\_newRef\_increments reference counter without QF\_CRIT\_ENTRY\_()).

## 10.64 Version 5.6.4, 2016-04-25

This release fixes a serious Bug #128 ( <https://sourceforge.net/p/qpc/bugs/128> ) in the **QK** port to ARM Cortex-M introduced back in QP 5.6.1

## 10.65 Version 5.6.3, 2016-04-12

This release fixes a serious Bug #126 ( <https://sourceforge.net/p/qpc/bugs/126> ) in the **QK** preemptive scheduler introduced in QP 5.6.2.

## 10.66 Version 5.6.2, 2016-03-31

The main purpose of this release is to introduce *atomic event multicasting*, meaning that event publishing to all subscribers is now protected from preemption. This eliminates potential for re-ordering of events under preemptive kernels (such as **QK**, **QXK**, or 3rd-party RTOSes), when events are published from low-priority AOs and some higher-priority subscribers can preempt multicasting and post/publish events of their own (before the original event is posted to all subscribers).

The atomic event multicasting is implemented by means of selective scheduler locking—very much like a priority-ceiling mutex. During event multicasting the scheduler gets locked, but only up to the highest-priority subscriber to a given event. The whole point here is that active objects with priorities above such "priority ceiling" are *not* affected. Please see the discussion thread:

<https://sourceforge.net/p/qpc/discussion/668726/thread/c186bf45>

This release also changes the implementation of the priority-ceiling mutex in the preemptive built-in kernels: **QK** and **QXK**. Specifically, the implementation now re-uses the selective scheduler locking mechanism. In this new implementation, the **QXMutex** of the **QXK** kernel is much more efficient and lightweight, but it *cannot block* while holding a mutex.

Finally, this release changes the QP ports to 3rd-party RTOSes by performing any RTOS operations (like posting events to message queues) outside critical sections. Also the ports have been augmented to support scheduler locking (this feature depends on what's available in the specific RTOSes).

Changes in detail:

1. Added scheduler locking to QF\_publish\_() in **qf\_ps.c**. This feature is added in a portable way, via macros **#QF\_SCHED\_STAT\_TYPE\_**, **QF\_SCHED\_LOCK\_()** and **QF\_SCHED\_UNLOCK\_()**, which need to be implemented in every QP port.
2. Modified **QV** kernel to provide (dummy) implementation of selective scheduler locking.
3. Modified **QK** kernel to implement selective scheduler locking via modified priority-ceiling mutex **QMutex**.
4. Modified **QXK** kernel to implement selective scheduler locking via modified priority-ceiling mutex **QXMutex**.
5. Modified embOS port to provide (global) scheduler locking, which affects all priorities, because that's all embOS supports. Also, modified the embOS port to perform event posting outside the **QF** critical section.
6. Modified uC/OS-II port to provide (global) scheduler locking, which affects all priorities, because that's all uC/OS-II supports. Also, modified the uC/OS-II port to perform event posting outside the **QF** critical section.
7. Modified ThreadX port to provide selective scheduler locking, by means of "priority-threshold" available in ThreadX. Also, modified the ThreadX port to perform event posting outside the **QF** critical section.

8. Changed the ThreadX example to run on ARM Cortex-M4 board (STM32DiscoveryF4), instead of Win32 emulation (see qpc/examples/threadx/arm-cm/dpp\_stm32f429-discovery).
9. Modified the Win32 port to provide (global) scheduler locking, which is implemented by Win32 critical section.
10. Fixed Bug#122 (QP didn't initiate some internal variables)  
<https://sourceforge.net/p/qpc/bugs/122/> by adding explicit clearing of all QP variables in QF\_init().
11. Modified the POSIX port to dummy-out scheduler locking. This means that this port currently does NOT lock scheduler around event publishing. (At this point it is not clear how to implement POSIX scheduler locking in a portable way.)
12. Modified **QK** and **QXK** examples in qpc/examples/arm-cm/dpp\_ek-tm4c123gxl to demonstrate the usage of the new priority-ceiling mutexes.
13. Fixed the 3rd-party file startup\_stm32l32l1xx.c to include exceptions for Cortex-M3 (MemManage\_Handler, BusFault\_Handler, and UsageFault\_Handler).
14. Updated the 3rd-party files for the EK-TM4C123GXL board (TivaC LaunchPad).
15. Modified Makefiles for the EK-TM4C123GXL board with GNU-ARM toolset to define the symbol TARGET\_IS\_TM4C123\_RB1 for compatibility with the updated 3rd-party files.
16. Implemented Feature Request #110 as well as the duplicate Request #62 by adding function QActive\_flushDeferred()

## 10.67 Version 5.6.1, 2016-01-01

This release is the first official (production) release of the new blocking **QXK** kernel.

Changes in detail:

1. Added error directives to source files from different built-in kernels (**QV**, **QK**, and **QXK**) to generate meaningful error messages when these files are mixed in one project. For example, a project based on **QK** will report errors when source files for **QV** or **QXK** are included in it.
2. Corrected example projects for the ARM Cortex-M with TI/CCS toolset

## 10.68 Version 5.6.0-beta, 2015-12-24

The main purpose of this *beta* release is to introduce a new component of the QP/C framework called **QXK** ("eXtended Quantum Kernel"). **QXK** is a small, preemptive, priority-based, **blocking** kernel that provides most features you might expect of a traditional blocking RTOS kernel.

**QXK** has been designed specifically for applications that need to mix event-driven active objects with traditional blocking code, such as commercial middleware (TCP/IP stacks, UDP stacks, embedded file systems, etc.) or legacy software. The **QXK** kernel is integrated tightly and optimally with the rest of the QP. It reuses all mechanisms already provided in QP, thus avoiding any code duplication, inefficient layers of indirection, and additional licensing costs, which are inevitable when using 3rd-party RTOS kernels to run QP/C applications.

**Note**

The [QXK](#) documentation is available in the QP/C Reference Manual at [Preemptive Dual-Mode Kernel](#)

Additionally, this release removes the macros Q\_ROM, Q\_ROM\_BYTEx, and Q\_ROM\_VAR from the QP/C code. These macros have been necessary for odd Harvard-architecture 8-bit CPUs (such as AVR, 8051) to place constant data in ROM. As QP/C stopped supporting those CPUs, the non-standard extensions could be removed from the QP/C code base.

Additionally, this release re-designs the priority-ceiling mutex in the [QK](#) kernel, which now works the same as the mutex of the new [QXK](#) kernel. Also, the [QK](#) ports to ARM Cortex-M no longer need or use the SVC\_Handler (Supervisor Call). This is done to make the [QK](#) ports compatible with various "hypervisors" (such as mbed uVisor or Nordic SoftDevice), which use the SVC exception.

Finally, this release modifies the GNU-ARM ports of [QK](#) for ARM Cortex-M, to use the \_\_ARM\_ARCH macro to distinguish among different architectures (ARCHv6 vs ARCHv7).

Changes in detail:

1. Added new header files for [QXK](#): qxk.h, and qxthread.h.
2. Added new source files for [QXK](#): qxk.c, qxk\_mutex.c, qxk\_pkg.h, qxk\_sema.c, qxk\_xthr.c.
3. Added [QXK](#) ports to ARM Cortex-M for ARM-KEIL, GNU-ARM, IAR, and TI-ARM toolsets (see [Preemptive "Dual-Mode" QXK Kernel](#))
4. Added [QXK](#) examples for ARM Cortex-M (in arm-cm\_dpp\_ek-tm4c123gxl and arm-cm\_dpp\_nucleo-l053r8) for all supported toolsets.
5. Removed Q\_ROM, Q\_ROM\_BYTEx, and Q\_ROM\_VAR from the QP/C code.
6. Added Q\_ROM, Q\_ROM\_BYTEx to the compatibility-layer in [qpc.h](#).
7. Removed ports and examples for the following 3rd-party RTOSes: CMSIS-RTX and FreeRTOS, as [QXK](#) provided all the features found in those kernels and is recommended over those kernels.
8. Removed AVR ports and examples.
9. Re-designed the [QK](#) priority-mutex in files [qk.h](#) and [qk\\_mutex.c](#).
10. Provided [QK](#) mutex examples in arm-cm\_dpp\_ek-tm4c123gxl and arm-cm\_dpp\_nucleo-l053r8.
11. Updated Makefiles for GNU-ARM to use the \_\_ARM\_ARCH macro for defining the ARM architecture.
12. Updated CMSIS from 4.2 to 4.3 in [qpc](#)/3rd-party/CMSIS

## 10.69 Version 5.5.1, 2015-10-05

The main focus of this release is to improve the AAPCS compliance of the ARM Cortex-M port to the [QK](#) preemptive kernel. Specifically, the PendSV handler in assembly did not always maintain the 8-byte stack alignment, which is required by AAPCS. This version corrects the stack misalignment in the [qk\\_port.s](#) files for all supported ARM compilers (ARM-Keil, GNU, IAR, and TI CCS). All these ports should also be ready for ARM Cortex-M7.

Also, this release adds support for the TI CCS ARM compiler. Specifically, a new ARM Cortex-M ports have been added (in directories [qpc/ports/arm-cm/qk/ti/](#) and [qpc/ports/arm-cm/qk/ti/](#)) and TI CCS example projects have been provided (in directories [qpc/examples/arm-cm/dpp\\_ek-tm4c123gxl/qk/ti/](#) and [qpc/examples/arm-cm/dpp\\_ek-tm4c123gxl/qv/ti/](#)).

Finally, this release corrects a bug in the DPP example for EK-TM4C123GXL with the [QV](#) non-preemptive kernel. Specifically, the file [qpc/examples/arm-cm/dpp\\_ek-tm4c123gxl/qv/bsp.c](#) did not re-enable interrupts in the [QV\\_onIdle\(\)](#) callback.

## 10.70 Version 5.5.0, 2015-09-04

The main purpose of this release is the extension of the [QS](#) software tracing system to bi-directional communication with embedded Targets. Specifically, the QS-RX (receive channel for [QS](#)) has been added with the following capabilities:

1. Set global [QS](#) filters inside the Target
2. Set local [QS](#) filters inside the Target
3. Inject an arbitrary event to the Target (direct post or publish)
4. Execute a user-defined callback function inside the Target with arguments supplied from QSPY
5. Peek data inside the Target and send to QSPY
6. Poke data (supplied from QSPY) into the Target
7. Execute clock tick inside the Target
8. Request target information (version, all sizes of objects, build time-stamp)
9. Remotely reset of the Target

This QP/C version complements the recent release of Qtools 5.5.0, where the [QSPY host application](#) has been extended with a UDP socket, which is open for communication with various Front-Ends (GUI-based or headless). An example Front-End written in Tcl/Tk called "QspyView" has been developed to demonstrate all the features. The example application located in the directory qpc/examples/arm-cm/dpp\_ek-tm4c123gxl/qspy contains customization of the "qspyview" script for the DPP application. Please refer to the documentation of this example (arm-cm\_dpp\_ek-tm4c123gxl) for more information.

Finally, this release adds a state machine operation for implementing the shallow history mechanism. The operation is called "childState", because it computes a child state of a given parent, such that the child belongs to the same state hierarchy as the current state.

Changes in detail:

1. Modified the [QS](#) software tracing component to add new functionality, such as the QS-RX input channel. Also added new trace records.
2. Added file "qstamp.c" (in the qpc/include/ folder) to provide time-stamp of the application build.
3. Added function QMsm\_childStateObj() to the [QMsm](#) class and QHsm\_childState() to the [QHsm](#) class. These functions have been added to support the shallow-history mechanism.
4. Modified all example projects (qpc/examples/ folder) to include the "qstamp.c" file and force its re-compilation for each new build, so that every build has an up-to-date and unique time stamp.
5. Extended the DPP on TivaC LauchPad example (directory qpc/examples/arm-cm/dpp\_ek-tm4c123gxl/) to demonstrate QS-RX ([QS](#) receive channel).
6. Provided example of customizing the "QspyView" Tcl/Tk script for the DPP application in the directory qpc/examples/arm-cm/dpp\_ek-tm4c123gxl/qspy/
7. Modified all examples (qpc/examples/ folder) to call the [QS\\_ASSERTION\(\)](#) macro to the [Q\\_onAssert\(\)](#) callback function.
8. Modified the startup code (in the qpc/3rd\_party/ folder) for ARM Cortex-M to invoke the [Q\\_onAssert\(\)](#) callback from the assert\_failure() exception handler. This is to allow application-level code to define [Q\\_onAssert\(\)](#) for each specific project.
9. Replaced deprecated registers in TM4C (TivaC) projects (SYSCTL->RCGCGPIO rather than the deprecated SYSCTL->RCGC2).

## 10.71 Version 5.4.2, 2015-06-04

The main focus of this release is to improve the support for "dual targeting" of QP/C applications, which is developing of deeply embedded code as much as possible on the desktop OS, such as Windows. Experience shows that "dual targeting" dramatically improves productivity of embedded systems developers, perhaps more than any other technique.

This release makes it possible to use exactly the **same** application code, main function, and the Board Support Package interface (bsp.h) on both deeply embedded target and on Windows. The only differences between these targets can be completely encapsulated in the Board Support Package implementation (bsp.c).

The support for "dual targeting" in this QP/C release works both for Win32 console and Win32 GUI applications. The Win32-GUI support enables developers to easily emulate the front-panels of the embedded devices, with LCD-screens (graphical and segmented), LEDs, buttons, switches, sliders, etc.

Changes in detail:

1. Modified the QP/C ports to Windows (both [Win32 API \(Multithreaded\)](#) and [Win32-QV \(Single Threaded\)](#)) so that they support both Win32 console and Win32-GUI applications. The newly introduced pre-processor `#WIN32_GUI` macro is now required to use the Win32-GUI facilities.
2. Added portable "safe" macros from `<stdio.h>` and `<string.h>` to the QP/C ports to Windows. These macros encapsulate the differences between Microsoft Visual C++ and other compilers (such as MinGW).
3. Simplified the structure of the QP/C Windows ports by eliminating one level of directories for the compilers used. Both VC++ and MinGW builds can now be run in the same port directory.
4. Modified the `QF_stop()` function in the QP/C port to [Win32-QV \(Single Threaded\)](#), so that it unblocks the `QV` event-loop and thus lets the application terminate.
5. Modified all examples for Windows to use the new port structure.
6. Improved all Makefiles (for the MinGW toolset) in all Windows examples, to make them easier to adapt to custom applications, both Win32 console and Win32 GUI.
7. Moved several examples from the `examples/win32/` and `examples/win32-qv` directories to `examples/arm-cm/` directory with native embedded examples for ARM Cortex-M. This co-location of the Win32 emulation with the embedded code running on the actual board demonstrates better the "dual targeting" development approach.
8. Updated all Windows examples to the latest QP API by compiling the code with the macro `QP_API_VERSION` set to 9999 (latest API without backwards compatibility)
9. Improved the PC-Lint support for checking the application-level code located in `examples/arm-cm/dpp_ek-tm4c123gx1/lint`

## 10.72 Version 5.4.1, 2015-05-14

This release changes the active object class hierarchy so that [QMActive](#) is now more fundamental and is the base class for [QActive](#). (Previously [QMActive](#) was a subclass of [QActive](#)). The newly added documentation section about QP/C Design shows the current class hierarchy.

### Note

Because the types [QMActive](#) and [QActive](#) are equivalent in QP/C, this change has minimal impact on the applications, but it is now more correct to use [QMActive](#) as the base class for all "opaque" active object pointers.

Also, this release brings several cosmetic improvements:

1. All QM models included in examples have been modified to use the [QMActive](#) "opaque" pointers.

2. All QM models have been saved with QM 3.3.0, which means that they will not open with QM 3.2.x or earlier QM versions.
3. The ROM-able QP version string QP\_versionStr[] has been added and used consistently in the macros QEP\_getVersion(), QF\_getVersion(), QK\_getVersion(), QV\_getVersion(), and QS\_getVersion() macros.
4. The `qpc/ports/arm-cm/qk/gnu/qk_port.s` ARM Cortex-M port to [QK](#) with GNU has been modified to use the CMSIS-compliant symbol `__FPU_PRESENT` instead of the `FPU_VFP_V4_SP_D16` symbol.
5. All Makefiles for the GNU toolset have been cleaned up, whereas any / (back-slash) characters in the paths have been replaced with / (forward-slash) characters. Also all these Makefiles have been updated to provide the `__FPU_PRESENT` to C and assembler when the hardware FPU is used.
6. The file display driver for the EK-LM2S811 board located at `qpc/3rd_party/ek-lm3s811/display96x16x1.c` has been modified to fix the problem with incorrect hardware delay with the GNU compiler at higher levels of optimization. The in-line assembly for the GNU compiler has been updated such that the delay loop cannot be "optimized away".
7. Several README files have been updated.

## 10.73 Version 5.4.0, 2015-04-26

This release changes the basic philosophy of distributing the QP Frameworks by **combining** the "QP/C Baseline Code" with all currently available "QP/C Development Kits" (QDK/C). This is done to eliminate any potential mistakes in downloading and installing separate pieces of code.

Additionally, this release changes the basic philosophy of building your embedded applications with the QP/C framework. Starting with this release, all [examples](#) for embedded boards include the QP/C framework as **source code** within the projects, instead of statically linking with a QP/C library. (**NOTE:** It is still possible to use QP/C as a library, but you need to build such libraries yourself, as they are no longer provided in the QP/C distribution.)

The move to building QP/C from sources ensures the consistent toolset version and compiler options applied to the application code as well as the QP/C framework code. (**NOTE:** The QP/C examples for "big operating systems", like Windows/POSIX, still use QP/C as a pre-compiled library that is statically linked with the application code.)

### Note

Even though the QP/C source has been re-packaged in this release, there are no API changes to the code, so it remains **backwards compatible** with the existing applications. (Except the build process, which builds QP/C from sources rather than linking to the QP/C library.)

The two changes in basic approach to distributing and building the framework have also the following ripple effects:

1. The QP/C source code has been simplified and has been re-packaged into a much smaller number of source files. The whole QP/C source code now resides in the single `source/` folder. Additionally, the source code files have now the **read-only** protection to prevent inadvertent changes to the QP/C source code that is part of your projects.
2. It is no longer necessary to define the **QP/C environment variable** to build the QP/C examples. All directories and files referenced by example projects are **relative** to the project folder. This change reflects the fact that most development tools add source files to the project using relative paths (and now the projects contain QP/C source code, not just the QP library).
3. The `QP/C ports/` folder has been reorganized to contain all currently available QP/C ports. The ports are organized into three categories: [native QP/C ports](#) ("bare-metal"), [ports to 3rd-party RTOSes](#), and [ports to big operating systems](#) (Windows and Linux). (**NOTE:** the ports are now documented in the this **QP/C Reference Manual**. Each port sub-directory contains a `README` link to the corresponding page in the online documentation)

4. The QP/C `exa/` folder has been reorganized to reduce the repetitions and contains all currently available QP/C examples. The folder includes four categories of examples: [native QP/C examples](#) ("bare-metal"), [examples for 3rd-party RTOSes](#), [examples for big operating systems](#) (Windows and Linux), and examples for 3rd-party Middleware. As mentioned before, all example projects for embedded systems use QP/C as source code and not as a library. The `examples/` folder has been expanded to contain all currently available QP/C examples, many of them are new in this release. (**NOTE:** the currently available examples are now documented in the **QP/C Reference Manual**. Each example sub-directory contains a README link to the corresponding page in the online documentation)
5. A new `3rd_party/` folder created to contain the Third-Party code used in the QP/C ports and examples, such as MCU register files, low-level startup code, device drivers, etc. The `3rd_party/` folder avoids the need to repeat such code in every project. Also, the separation of the Third-Party components helps to clearly indicate code that comes from various sources, and to which Quantum Leaps, LLC expressly makes **no claims of ownership**. The Third-Party software components included in this "3rd\_party" folder are licensed under a variety of different licensing terms that are defined by the respective owners of this software and are spelled out in the README.txt or LICENSE.txt files included in the respective sub-folders.
6. This release also comes with the much expanded online **QP/C Reference Manual**, which is cross-linked with the ports and examples.

Changes in detail:

1. Renamed the "Vanilla" kernel to the [QV non-preemptive kernel](#) for symmetry with the [QK preemptive kernel](#). Renamed `QF_onIdle()` callback to `QV_onIdle()`.
2. Removed class `QFsm` (which is now deprecated). Legacy state machines coded in the "QFsm-style" will continue to work, but will use the [QHsm](#) implementation internally. There is no longer any efficiency advantage in using the "QFsm-style" state machines.
3. Applied a slight performance improvement to the ARM Cortex-M port to the [QK](#) preemptive kernel. The [QK](#) port now checks for ISR context by looking at the IPSR register, instead of incrementing and decrementing the `QF_intNest_up-down` counter.
4. Updated ARM Cortex-M examples and provided new examples for NXP mbed-LPC1768, and STM32 NUCLEO-L053R8, and NUCLEO-L152RE boards. All examples now use the latest CMSIS (V4.3.0). All ARM Cortex-M examples are provided for the ARM-KEIL, GNU-ARM, and IAR-ARM toolsets.
5. Added the native port and examples to the classic ARM7/9 with AT91SAM7S-EK board and the IAR-ARM toolset.
6. Added the native port and examples to the AVR (AVRmega) with GNU-AVR and IAR-AVR toolsets. The examples are provided for the Arduino-UNO board.
7. Added the native [port](#) and examples to MSP430 with TI CCS-430 and IAR-430 toolsets. The examples are provided for the MSP430 LaunchPad board (MSP-EXP430F5529LP for the "classic" MSP430 and "extended" MSP430X, respectively).
8. Added port to CMSIS-RTOS RTX. Examples are available for TI EK-TM4C123GLX, STM32 NUCLEO-L053R8, and NUCLEO-L152RE boards with ARM-KEIL, GNU-ARM, and IAR-ARM toolsets.
9. Updated port to [embOS](#). Examples are available for STM32 STM32F4-Discovery board with IAR-ARM toolset.
10. Updated port to [FreeRTOS](#) for the latest version 8.2.1. Examples are available for TI EK-TM4C123GLX board with GNU-ARM and IAR-ARM toolsets.
11. Added [port to Thread-X](#). Example is available for the Thread-X demo with Visual Studio on Windows.
12. Updated port to [uC/OS-II](#) for the latest version v2.92. Examples are available for TI EK-TM4C123GLX and STM32 NUCLEO-L152RE boards with ARM-KEIL and IAR-ARM toolsets.

13. Updated [port to Win32](#) (Windows). Modified the port to apply a generous "fudge factor" in over-sizing QP event queues and event pools, to minimize the risk of overflowing queues/pools due to non-deterministic Windows behavior.
14. Added new [port to Win32-QV](#) (Windows with non-preemptive QV scheduler, previously known as Win32-1T).
15. Updated the lwIP-QP example for EK-LM3S6965 board.

## 10.74 Version 5.3.1, 2014-09-19

### Note

QP/C 5.3.1 remains backwards-compatible with all QP/C ports and applications

This release fixes the following bugs:

1. QMsm\_isInState() returns invalid result (bug #105)
2. QP/C syntax error in qf\_pkg.h (bug #104)
3. QF\_gc() doc typo (bug #102)
4. POSIX-port Makefile error (bug #65)

Additionally, this release improves the uC/OS-II port in that it is now generic and applicable for any CPU, for which uC/OS-II port exists. Specifically, all references to DOS or x86 have been removed from the QP port and any CPU-specific dependencies have been placed in the separate part of the port.

Finally, this release improves the "QP/C Reference Manual" generated by Doxygen and available both inside the QP/C baseline distribution (qpc.chm file) and online at: <https://www.state-machine.com/qpc>

## 10.75 Version 5.3.0, 2014-03-31

This release adds the "transition to history" (deep history) feature to both [QHsm](#) and [QMsm](#) state machines and their subclasses. This QP/C release matches the new QM modeling tool version 3.1.0, which now supports the "transition to history" connector and the corresponding code generation for transitions to history.

### Note

QP/C 5.3.0 remains backwards-compatible with QP/C applications developed for QP/C 4.x and QP/5.x. However, any QM models created for the previous QP/C versions require re-generating the code with QM 3.1.0.

This release adds new [QS](#) (Quantum Spy) instrumentation for tracing transitions to history as well as entry and exit points in submachines. All these features require the matching QSPY host application included in Qtools 5.3.0.

Additionally, the [QMsm](#) state machine has been extended to add implementation of the reusable submachine states and submachines with entry points and exit points. The reusable submachines in QP/C 5.3.0 lay the groundwork for providing reusable submachine states and submachine diagrams in the next upcoming QM version.

This release also goes several steps towards compliance with the new MISRA-C:2012 rules. For example, unused tag declarations have been removed (MISRA-C:2012 Rule 2.4), the C99 standard Boolean data type in <stdbool.h> has been added instead of uint8\_t for stricter type analysis, and the C99 data types uint\_fast8\_t and uint\_fast16\_t are used instead of the non-standard uint\_t.

Finally, this QP/C release brings deep changes in the source code comments and the doxygen documentation generated from the source code. All comments have now more consistent structure, and every function is now documented in the implementation file (.c file), whereas the interface (.h files) contain only the brief descriptions of the functions. This re-structuring of documentation is performed as part of the validation and verification effort that has begun to provide a certification package for QP/C for safety standards, such as IEC 61508 and ISO 62304 (FDA 510(k)).

Changes in detail:

1. Moved detailed documentation of functions from the header files (.h) to implementation files (.c).
2. Removed the header file "qevt.h" and merged its contents into "qep.h"
3. Added macros: trace records QS\_QEP\_TRAN\_HIST, QS\_QEP\_TRAN\_EP, and QS\_QEP\_TRAN\_XP to "qs.h"
4. Added macros: [Q\\_TRAN\\_HIST\(\)](#), [QM\\_TRAN\\_HIST\(\)](#), [QM\\_TRAN\\_EP\(\)](#), [QM\\_TRAN\\_XP\(\)](#), and [QM\\_SUPER\\_SUB\(\)](#) to "qep.h"
5. Added attributes entryAction and initAction to the [QMState](#) struct in "qep.h" (needed for transition to history).
6. Added attribute act to the QMArr union in "qep.h" (needed for transitions to entry point in submachine states).
7. Changed return type to bool in functions [QHsm\\_isIn\(\)](#), [QMsmVtbl.post\(\)](#), [QActive\\_post\\_\(\)](#), [QEQueue\\_post\(\)](#), [QActive\\_defer\(\)](#), [QTimeEvt\\_rearm\(\)](#), [QTimeEvt\\_disarm\(\)](#), [QF\\_noTimeEvtsActiveX\(\)](#).
8. Changed the [QState](#) return type from action/state handler functions to `uint_fast8_t`.
9. Changed the prio attribute of [QActive](#) to `uint_fast8_t`.
10. Changed the type of prio argument to `uint_fast8_t` and qlen/stkSize to `uint_fast16_t` in the signature of [QActiveVtbl.start](#) function pointer and [QActive\\_start\\_\(\)](#) implementation.
11. Changed the type of the tickRate argument in [QTimeEvt\\_ctorX\(\)](#) and [QF\\_tickX\\_\(\)](#), and [QF\\_noTimeEvtsActiveX\(\)](#) to `uint_fast8_t`.
12. Changed the type of the poolSize argument in [QF\\_poolInit\(\)](#) to `uint_fast16_t`.
13. Changed arguments evtSize and margin in [QF\\_newX\\_\(\)](#) to `uint_fast16_t`.
14. Changed attribute bits in [QPSet8](#) as well as bytes and bits[] in [QPSet64](#) to `uint_fast8_t`.
15. Changed the [QEQueueCtr](#) event queue counter type to `uint_fast8_t`.
16. Changed type of arguments qLen and margin in [QEQueue\\_init\(\)](#) and [QEQueue/QActive\\_post\(\)](#) to `uint_fast16_t`.
17. Changed the return type from [QK\\_schedPrio\\_\(\)](#) (priority) as well as the p argument in [QK\\_sched\\_\(\)](#) and [QK\\_schedExt\\_\(\)](#) to `uint_fast8_t`
18. Added function [QMsm\\_isInState\(\)](#) to "qep.h" and its implementation file [qmsm\\_in.c](#). This function tests whether the [QMsm](#) state machine (or its subclasses like [QMAactive](#)) "is in" the given state.
19. Updated all make scripts for QP/C ports to include the new [qmsm\\_in.c](#) in the QP/C library builds.

## 10.76 Version 5.2.1, 2014-01-06

This release fixes two bugs.

1. In file [qmsm\\_dis.c](#) added saving of the action-table into a temporary variable *before* exiting the current state to the transition source. Also, changed the signature of the [QMsm\\_tran\\_\(\)](#) helper function to take the action table as parameter. NOTE: This bug only affected the Spy configuration and because of this escaped regression testing. The internal testing process have been updated to test all build configurations: Debug, Release, and Spy.
2. In file [qs\\_mem.c](#) fixed an error in accounting used bytes in the [QS](#) trace buffer.

## 10.77 Version 5.2.0, 2013-12-26

This release matches the new QM 3.0.0, for which it provides model examples based on the new QMsm/QMActive classes. This, in turn demonstrates the new state machine code generation that QM3 was specifically designed to do. This release also provides consistent API for late-binding ("virtual" functions) introduced in QP 5.0.0, as opposed to using regular linking (early-binding) for direct function calls, such as QHsm\_dispatch(). A clearly separated API compatibility layer is provided, whereas you can configure a level of backwards compatibility by means of the `QP_API_VERSION` macro. This facilitates migrating existing QP applications to the newer API.

An cyclomatic complexity (McCabe V(G)) analysis of this version has been performed and the maximum V(G) complexity per function has been reduced to 15 by breaking up the QHsm\_dispatch\_() function. The code metrics report, including cyclomatic complexity by function as well as other standard code metrics (e.g., lines of code), is now included in the "QP/C Reference Manual", see <https://www.state-machine.com/qpc/metrics.html>. Also, in this release all internal QP data that were previously uninitialized are now explicitly initialized to zero. In other words, this release no longer assumes that all uninitialized data (global and static inside functions) is implicitly initialized to zero before the control is transferred to main(). This is a requirement of the C Standard, but some embedded startup code fails to do this.

Finally, this release demonstrates safer stack allocation and safer exception handlers in all ARM Cortex-M examples. The techniques are described in the Embedded.com article "Are We Shooting Ourselves in the Foot with Stack Overflow?".

Changes in detail:

1. In file qep.h renamed the implementation functions, such as QHsm\_init() and QHsm\_dispatch() to QHsm\_init\_() and QHsm\_dispatch\_() (note the underscore, which means that the functions should not be called directly by the application code). The only correct way of calling the functions is through the macros `QMSM_INIT()` and `QMSM_DISPATCH()`, respectively. The latter macros implement late-binding ("virtual" functions in C).
2. In file qf.h renamed the implementation functions, such as QActive\_start(), QActive\_post() and QActive\_postLIFO() to QActive\_start\_(), QActive\_post\_(), and QActive\_postLIFO\_, respectively (note the underscore, which means that the functions should not be called directly by the application code). The only correct way of calling the functions is through the macros `QACTIVE_START()`, `QACTIVE_POST()`, and `QACTIVE_POST_LIFO()`, respectively. The latter macros implement late-binding ("virtual" functions in C).
3. for backwards compatibility, in file `qp_port.h` defined "API Compatibility Layer", which is controlled by the macro `QP_API_VERSION`. For example, specifying `QP_API_VERSION=500` chooses API compatible with QP version 5.0.0 or newer, but excludes APIs that became deprecated in the earlier versions. If the macro `QP_API_VERSION` is not defined by the user (typically on the command line for the compiler), the default value of 0 is assumed. This default means maximum backwards compatibility (from version 0.0.0). On the other hand, higher values of `QP_API_VERSION` mean less backwards compatibility. For example `QP_API_VERSION=9999` will specify compatibility only with the latest version of QP. The API Compatibility Layer for `QP_API_VERSION < 500` provides macros: `QHsm_init()`, `QHsm_dispatch()`, `QActive_start()`, `QActive_post()` and `QActive_postLIFO()`. These macros resolve to `QMSM_INIT()` and `QMSM_DISPATCH()`, `QACTIVE_START()`, `QACTIVE_POST()` and `QACTIVE_POST_LIFO()` respectively, so that calls based on the older API also use late-binging.
4. In file `qhsm_dis.c`, broken up the function `QHsm_dispatch()` into two functions `QHsm_dispatch_()` and `QHsm_tran_()`. This has reduced the cyclomatic complexity from 25 for the original function, to 11 and 15 for `QHsm_dispatch_()` and `QHsm_tran_()`, respectively.
5. In file `qmsm_dis.c`, broken up the function `QMsm_dispatch()` into two functions `QMsm_dispatch_()` and `QMsm_tran_()`. This has reduced the cyclomatic complexity from 15 for the original function, to 9 and 7 for `QMsm_dispatch_()` and `QMsm_tran_()`, respectively.
6. In file `qf_act.c` added the function `QF_bzero()`, and in files `qv.c` and `qk.c` added calls to `QF_bzero()` to explicitly clear the uninitialized data. Also added calls to `QF_bzero()` inside `qf_psini.c`.
7. Updated all examples for ARM Cortex-M to use safer stack allocation and safer exception handlers in all ARM Cortex-M examples, as described in the Embedded.com article "Are We Shooting Ourselves in the Foot with Stack Overflow?".

## 10.78 Version 5.1.1, 2013-10-10

This release fixes reversal of logic in the QF\_noTimeEvtsActiveX() function as well as sleep mode transition in the ARM Cortex-M3/M4 ports to the non-preemptive **QV** kernel. Also, the native QP event queue implementation has been changed to count the extra "front-event" location into the number of free entries, which fixes the problem of defer queues of depth 1. Finally, the release restores the support for linting (with PC-Lint) of the QP/C applications for ARM Cortex-M (with IAR and GNU compilers).

Changes in detail:

1. In file qf\_tick.c reversed the logic inside QF\_noTimeEvtsActiveX()
2. Modified free entry accounting (nFree) in the files: qeq\_init.c, qeq\_fifo.c, qeq\_get.c, and qeq\_lifo.c.
3. Modified free entry accounting (nFree) in the files: qa\_init.c, qa\_fifo.c, qa\_get\_.c, and qa\_lifo.c.
4. Introduced new macro QF\_CPU\_SLEEP() in the ARM Cortex-M **QV** ports.
5. Changed Board Support Package files (bsp.c) in the ARM Cortex-M **QV** examples.
6. Modified the CMSIS-compliant startup code in all ARM Cortex-M **QV** examples.
7. Modified the application examples with PC-Lint (qpc/examples/arm-cm/qk/gnu/dpp-qk\_ek-lm3s811-lint and qpc/examples/arm-cm/qk/iar/dpp-qk\_ek-lm3s811-lint). Updated lint files for the latest PC-Lint

## 10.79 Version 5.1.0, 2013-09-23

This release brings significant improvements to the **QS** software tracing implementation and also brings important changes the ARM Cortex-M port.

### Note

**QP/C 5.1.0 requires changing the interrupt priority setting in the existing ARM Cortex-M applications.** Specifically, you need to set the interrupt priorities equal or lower than **QF\_AWARE\_ISR\_CMSIS\_PRI** constant provided in the qf\_port.h header file.

Changes to the **QS** software tracing component in detail:

1. Optimized the internal **QS** implementation of all functions that insert trace data into the trace buffer. The general idea of the optimization is to extensively use automatic variables instead of global variables (such as buffer head and tail indexes, the running checksum, etc.). For the modern CPUs (such as ARM) this resulting machine code performs most operations in registers, instead of constantly updating the memory through the expensive load/store instructions. The time savings through avoiding load/store instructions are significant, even after taking the performance hit from loading the registers from the globals in the beginning of each function and storing the final register values into the globals at the end.
2. Reduced the **QS** code size by using loops instead of unrolled-loops as before. This reduced the **QS** component size from over 4KB to 1.7KB (for ARM Cortex-M3/M4, IAR compiler).
3. Modified the make scripts for building QP libraries to use higher-level optimization for the **QS** software tracing functions in the SPY build configuration. This brings additional 20-50% speed improvement, depending on the compiler and optimization options used. Please note that only the **QS** component is built with high-optimization. The QEP, **QF**, and **QK** components in the SPY configuration are still built with low-optimization level, so that the application can be conveniently debugged.
4. Reduced the number of **QS** global filters from 256 to 124 (0x7C). This enables the code to avoid escaping the trace record numbers (because they cannot overlap the special flag byte 0x7E or the escape byte 0x7D) and also speeds up the QS\_filterOff(QS\_ALL\_RECORDS) function, which is useful for stopping the trace quickly to avoid overwriting some interesting data with the new data.

5. An empty **QS** record and the QS\_RESET record are now inserted automatically into the trace buffer in the function `QS_initBuf()`. The empty **QS** record/QS\_RESET pair provides a clean start of a session and allows the QSPY host application to re-synch with the data stream, even if the last **QS** record of a previous session is incomplete. This ability is very helpful for re-setting the target while collecting a trace.

Overall, lab tests for ARM Cortex-M4 with the IAR compiler show that the processing time of the `QS_u32_()` function (the one frequently used to store pointers and timestamps) dropped from 233 CPU cycles for QP 5.0 with low-level optimization to just 76 cycles for QP 5.1 with high-level of optimization. At the same time, the code size of this function dropped from 876 bytes to 274 bytes.

Changes to the QP ports to ARM Cortex-M in detail:

QP 5.1.0 never completely disables interrupts in the ARM Cortex-M3/M4 cores, even inside the critical sections. On Cortex-M3/M4 (ARMv7-M architectures), the QP port disables interrupts selectively using the BASEPRI register. (NOTE: The BASEPRI register is not implemented in the ARMv6-M architecture (Cortex-M0/M0+), so Cortex-M0/M0+ need to use the PRIMASK register to disable interrupts globally).

This new policy of disabling interrupts divides interrupts into "kernel-unaware" interrupts, which are never disabled, and "kernel-aware" interrupts, which are disabled in the QP critical sections. Only "kernel-aware" interrupts are allowed to call QP services. "Kernel-unaware" interrupts are *NOT* allowed to call any QP services and they can communicate with QP only by triggering a "kernel-aware" interrupt (which can post or publish events).

As mentioned above, all QP ports to ARM Cortex-M included in the QP 5.1.0 Baseline Code provide the constant `QF_AWARE_ISR_CMSIS_PRI`, which must be used to offset the "kernel-aware" interrupt priorities.

All example projects for ARM Cortex-M included in the QP 5.1.0 Baseline Code demonstrate the recommended way of assigning interrupt priorities in your applications. The initialization consist of two steps:

1. you enumerate the "kernel-unaware" and "kernel-aware" interrupt priorities (whereas you offset the "kernel-aware" priorities by the constant `QF_AWARE_ISR_CMSIS_PRI`) and
2. you assign the priorities to *ALL* interrupts by calling the `NVIC_SetPriority()` CMSIS function.

#### Note

Leaving the interrupt priority at the default value of zero (the highest priority) is most likely **incorrect**, because the "kernel-unaware" interrupts **cannot** call any QP services.

For more information, please read the short Application Note "Setting ARM Cortex-M Interrupt Priorities in QP 5.1" available at:

[https://www.state-machine.com/doc/AN\\_ARM-Cortex-M\\_Interrupt-Priorities.pdf](https://www.state-machine.com/doc/AN_ARM-Cortex-M_Interrupt-Priorities.pdf)

## 10.80 Version 5.0.0, 2013-09-10

#### Note

QP/C 5.0.0 remains **backwards-compatible** with the existing QP/C 4.x applications.

The main purpose of this milestone QP/C release is to enable the QM modeling tool to generate a new type of state machine code (requires QM version 3.0.0, which is still in development as of this writing).

This new type of state machine implementation in QP/C 5 is based on the new `QMsm` class, which takes advantage of the QM tool as an advanced "state machine compiler". QM can perform optimizations that were not possible with the C pre-processor alone. Specifically, the QM can easily determine the LCA (Least-Common-Ancestor) state for every transition and it can generate the complete transition-sequences (sequences of exit/entry/initial actions) at code-generation time. The resulting code can be still highly human-readable, but it will no longer be human-maintainable. The lab tests indicate that the new "housekeeping" code for executing hierarchical state machines can be about twice as fast as the previous code based on the `QHsm` class. Additionally, the new code requires less run-time support (smaller event processor) and uses 70% less of stack space in the call to the `QMsm_dispatch()` operation than `QHsm_dispatch()`.

The next big feature introduced in QP/C 5 is polymorphism ("virtual" functions) for basic operations, such as state machine `init()` and `dispatch()` and active object `start()`, `post()`, and `postLIFO()` perations. Making these functions "virtual"

means that all these operations can be re-defined in sub-classes of state machines and active objects. This, in turn, allows a single application to use a mix of state machine classes derived from the new [QMsm](#) base class with state machines derived from the [QHsm](#) base class, each one using a different state machine implementation strategy. Additionally, the virtual post() operation could be very useful for implementing various Proxy active objects (e.g., for active object event posting across networks).

Another big feature introduced in QP/C 5 are the multiple, independent system clock tick rates for time events. The number of system tick rates can be now configured in the QP/C ports. For example, a digital watch can use a "fast" clock tick rate of 100Hz and a "slow" clock tick rate of only 1Hz. These clock tick rates can be managed independently, so for example, the fast clock tick rate can be shut down in the absence of time events assigned to this rate. This feature allows the applications to implement sophisticated power-saving policies.

As yet another important feature, QP/C adds a new "extended" API for non-asserting event allocation and posting. This feature is intended for situations, where an application is hammered with external events that at times arrive too fast for processing, but that can be ignored under the overload conditions. In those cases firing an assertion inside the framework is undesirable. The non-asserting API allows a designer to request a safety-margin when allocating or posting an event. The event is not allocated/posted if the safety margin cannot be satisfied at the time of the call. On the other hand, the safety margin allows the application to still use the regular (asserting) event allocation and posting, because the event pools and event queues are guaranteed to maintain a minimal margin for safe operation.

Finally, QP/C adds a number of smaller features and improvements, summarized in the following detailed list of changes:

1. Added the new [QMsm](#) "class" to qep.h. Changed the inheritance tree by deriving [QHsm](#) and [QFsm](#) from the [QMsm](#) base class. Added virtual table structures for [QMsm](#), [QHsm](#), and [QFsm](#) (polymorphism).
  - added macro [QMSM\\_INIT\(\)](#) to polymorphically call the state machine initialization implementation in the [QMsm](#) base class and all subclasses.
  - added macro [QMSM\\_DISPATCH\(\)](#) to polymorphically call the state machine event dispatching implementation in the [QMsm](#) base class and all subclasses.
2. Added new source files [qmsm\\_ini.c](#) and [qmsm\\_dis.c](#) to the QEP. These files implement the [QMsm\\_init\(\)](#) and [QMsm\\_dispatch\(\)](#) functions, respectively.
3. Added the new "QMActive" "class" to qf.h. Extended the inheritance tree to derive [QMActive](#) from [QActive](#). Added virtual table structures for [QMActive](#) and [QActvie](#) (polymorphism).
  - modified macro [QACTIVE\\_POST\(\)](#) to polymorphically call the direct event posting to an active object.
  - modified macro [QACTIVE\\_POST\\_LIFO\(\)](#) to polymorphically call the post-LIFO (self-posting) to an active object.
  - modified macro [QACTIVE\\_START\(\)](#) to polymorphically call the starting of an active object.
4. Added the multiple system clock tick rates feature in qf.h:
  - added new configuration macro [QF\\_MAX\\_TICK\\_RATE](#), which specifies the number of clock tick rates. This macro is to be defined in the [QF](#) ports (in the [qf\\_port.h](#) header file). If the macro is undefined, the default value is 1 (one clock tick rate).
  - renamed and re-implemented the [QF\\_tick\(\)](#) function as the "extended" [QF\\_tickX\(\)](#) function with the argument 'tickRate' for processing time events allocated to different clock rates. The application must call [QF\\_tickX\(0\)](#), [QF\\_tickX\(1\)](#), ~~~~ at the specified tick rates from ISRs or tasks.
  - added an "extended" time event constructor [QTimeEvt\\_ctorX\(\)](#), which assigns a time event to a specific tick rate as well as specific active object.
  - renamed and re-implemented the internal function [QTimeEvt\\_arm\\_\(\)](#) to a public function [QTimeEvt\\_armX\(\)](#) for arming time events initialized with the "extended" constructor. The [QTimeEvt\\_armX\(\)](#) function is the new recommended API for arming time events, both one-shot and periodic.
  - re-implemented [QTimeEvt\\_disarm\(\)](#) and [QTimeEvt\\_rarm\(\)](#).
  - renamed [QF\\_noTimeEvtsActive\(\)](#) to the "extended" version [QF\\_noTimeEvtsActiveX\(\)](#), which checks time events assigned to the given tick rate.

5. Added the new non-asserting API to qf.h:

- renamed internal function QF\_new\_() to QF\_newX\_(), the latter one taking the argument 'margin' for allocating events. The function returns NULL if the event pool has less free events than the specified margin. The function asserts if the margin is zero and the event can't be allocated.
- added function QActive\_post() to post an event to the given active object. The function does not post the event if the target event queue has less free slots than the specified margin. The function asserts if the margin is zero and the event can't be posted.
- added "extended" macro QF\_NEW\_X() for allocating events with a margin.
- added "extended" macro [QACTIVE\\_POST\\_X\(\)](#) for posting events with a margin.

6. Modified the QActive\_defer() function to return the status of the defer operation (true==success), instead of asserting when the defer queue overflows.

7. Modified [QS](#) (Quantum Spy) software tracing implementation:

- added additional tick rate byte to the trace records QS\_QF\_TICK and QS\_QFF\_TIMEEVT\_\*.
- added new trace records QS\_QF\_ACTIVE\_POST\_ATTEMPT, QS\_QF\_EQUEUE\_POST\_ATTEMPT, and QS\_QF\_MPOOL\_GET\_ATTEMPT for the "extended" non-asserting event allocation and posting.
- added new trace records QS\_TEST\_RUN and QS\_TEST\_FAIL for future support for unit testing.
- added new [QS](#) source file qs\_dict.c with functions QS\_\*\_dict() to generate various dictionary entries. Changed the macros QS\_\*\_DICTIONARY() to call these functions. This was done to significantly reduce the amount of tracing code needed to send the dictionaries from applications.
- grouped together the various [QS](#) variables (such as filters, trace buffer indexes, etc.) in a single struct, which results in a more efficient code for various [QS](#) operations.

8. Changed the structure of the ARM Cortex-M ports

- renamed the sub-directory for ARM Cortex-M ports and examples from "arm-cortex" to "arm-cm". This is done to avoid confusion with other ARM Cortex variants, such as Cortex-A/R, which very different from Cortex-M.
- removed the CMSIS (Cortex Microcontroller Software Interface Standard) directories from the Cortex-M examples and moved it to the common location in the QPC%/ports/arm-cm/cmsis/ directory. Upgraded the CMSIS to the latest version 3.20.
- added the ARM Cortex-M ports and examples with Keil/ARM MDK to the QP Baseline Code.
- upgraded ARM Cortex-M ports with IAR to the latest IAR EWARM 6.60
- upgraded ARM Cortex-M ports with Sourcery CodeBench to the latest version 2013.05-53.

9. Added the requested simple "Blinky" example for Windows and ARM Cortex-M (with the GNU, IAR, and Keil toolsets).

- Added "Getting Started with QP/C" guide based on the Blinky example.

10. Updated the Doxygen documentation (QP/C Reference Manual)

- updated the QP/C tutorial
- updated and added documentation and code samples
- added search box and tree-view panel to the HTML documentation

## 10.81 Version 4.5.04, Feb 08, 2013

The main purpose of this release is adding support for the ARM Cortex-M4F processors with the hardware Floating-Point Unit (FPU). The QP/C ports to Cortex-M4F take full advantage of the "lazy stacking" feature of the FPU registers, and by doing so offer the most efficient preemptive multitasking on this processor.

### Note

QP/C Version 4.5.04 preserves full compatibility with QM 2.2.03 and all existing QDKs for QP/C 4.5.xx.

Changes in detail:

1. Added ports and examples for ARM Cortex-M4F with the EK-LM4F120XL board (Stellaris Launchpad).
2. Added the macro [QF\\_LOG2\(\)](#), which can be re-implemented in the QP ports, if the CPU supports special instructions, such as CLZ (count leading zeros in Cortex-M3/M4). If the macro is not defined in the QP port, the default implementation uses a lookup table.
3. Updated all ARM Cortex-M ports and examples to the latest IAR EWARM 6.50 and Sourcery CodeBench 2012.09-85.
4. Updated App Notes "QP and ARM Cortex-M with IAR" and "QP and ARM Cortex-M with GNU".
5. Updated the PC-Lint support files (include/lib-qpc.Int, au-misra2.Int) to the latest PC-Lint 9.00j.
6. Updated the Application Note: "QP/C MISRA-C:2004 Compliance Matrix".
7. Spell-checked the comments in all QP/C source files and removed several typos.
8. Removed the Qt ports and examples from the QP/C Baseline Code and moved them to the separate QDK/C-Qt.

## 10.82 Version 4.5.03, Nov 27, 2012

This release changes the directory structure of QP ports to various operating systems, such as POSIX (Linux, BSD, etc.), Win32 (Windows), Qt, etc. The OS ports are moved from the ports/80x86/ directory one level up to ports/. Also, the OS examples are moved from the examples/80x86/ directory one level up to examples/.

## 10.83 Version 4.5.02, Aug 04, 2012

The main purpose of this release is better, more comprehensive support for (rapid) prototyping of embedded QP Applications on the desktop with the Win32 API and with Qt. Among others, this release adds a complete toolkit for creating realistic embedded front panels with pure Win32-GUI API and free Visual C++ Express and ResEdit. An extensive Application Note "QP and Win32" is included in this release.

This release simplifies the QP ports to desktop OSs, such as Windows (Win32), Linux, BSD, Mac OSX (POSIX) and combines 32-bit and 64-bit ports in one with conditional compilation.

This release also adds an option for using initialization for dynamically allocated events. When the configuration macro Q\_EVT\_CTOR is defined, the [Q\\_NEW\(\)](#) macro becomes variadic and takes the arguments for the event initialization. This generally allows creating dynamic events "on-the-fly" without a temporary pointer to the event. This QP configuration is demonstrated only in the QP port to Qt, but can be used in any other port.

**Note**

The event initialization feature is NOT backward-compatible with the existing applications.

This release also adds a new macro QF\_MPOOL\_EL, which is intended for allocating properly aligned storage for memory pools and event pools.

All QP ports included in this release use only a single QP library, rather than separate libraries for QEP, [QF](#), [QK](#), and [QS](#).

Finally, this release adds QM models, created with the new QM 2.2.00 to most of the examples. The code generated by this new QM version complies with MISRA-C:2004 rules.

Changes in detail:

1. Modified QP port to Win32 and used the free Visual C++ Express 2010 with Platform SDK rather than Visual C++ Pro 2008. Renamed the port directory from vc2008/ to vc/. Provided a completely revised App Note "QP and Win32".
2. Eliminated QP port to Win32 with one thread (Win32-1T).
3. Consolidated all QP ports to POSIX OSs (Linux, Linux64, Mac OSX) into a single port to POSIX and placed it in the directory posix/.
4. Renamed the port directory qt\_1t/ to qt/.
5. Added event initialization to qevt.h (controlled by the configuration macro Q\_EVT\_CTOR).
6. Added new variadic version of the [Q\\_NEW\(\)](#) macro in qf.h when Q\_EVT\_CTOR is defined.
7. Added macro QF\_MPOOL\_EL to [qmpool.h](#). Modified all examples to demonstrate the use of this macro to allocate properly aligned storage for event pools.
8. Added new typedef '[enum\\_t](#)' and modified signatures of functions taking event signals from [QSignal](#) to [enum\\_t](#). This was done to significantly reduce the number of casts necessary when enumerated signals were passed to QP functions.
9. Modified all QP ports distributed in the QP/C baseline code to generate only a single QP library, rather than separate libraries for QEP, [QF](#), [QK](#), and [QS](#). This includes all QP ports to the desktop (ports/80x86/ directory) and ARM Cortex-M ports (ports/arm-cortex/ directory).
10. Modified all examples to link only one QP library.
11. Added QM models to most examples and used the automatically generated code from the models instead of the hand-written code.
12. Modified Qt ports to use the new "event constructors" and modified examples for Qt to demonstrate this feature.
13. Added .ui files to the Qt examples for generating UIs graphically with the Qt Designer tool. Revised and updated the App Note "QP and Qt".
14. Added new macro [QS\\_USR\\_DICTIONARY\(\)](#) to [QS](#) for providing symbolic names for user-defined trace records
15. Added new macro [QS\\_RESET\(\)](#) to [QS](#) for telling the QSPY application when the target resets. This allows QSPY to reset its internal state.

## 10.84 Version 4.5.01, Jun 14, 2012

The main purpose of this minor release is providing improved MISRA-compliant state machine implementation. Specifically, a new macro [Q\\_UNHANDLED\(\)](#) has been added for a situation when a guard condition evaluates to FALSE, but the state model does not prescribe the explicit [else] case for this guard. In this case, the state handler can return [Q\\_UNHANDLED\(\)](#), which will cause the QEP event processor to propagate the event to the superstate, which is what UML semantics prescribes.

**Note**

These changes to the QEP event processor are completely backwards-compatible. All state handler functions coded the old way will continue to handle the guard conditions correctly and in accordance with the UML specification. The new [Q\\_UNHANDLED\(\)](#) macro is necessary only for MISRA-compliant state handler coding, which will be applied in the upcoming release of the QM modeling and code generation tool.

Changes in detail:

1. Added macro [Q\\_UNHANDLED\(\)](#) and return value [Q\\_RET\\_UNHANDLED](#) in [qep.h](#).
2. Modified [qhsms\\_dis.c](#) to handle the [Q\\_RET\\_UNHANDLED](#) return value.
3. Updated the QP/C MISRA-C:2004 compliance matrix to include the new MISRA-compliant way of coding guard conditions.
4. Modified [qs.h](#) and [qs\\_dummy.h](#) to add new trace record type [QS\\_QEP\\_UNHANDLED](#), which is generated when the state handler returns [Q\\_RET\\_UNHANDLED](#).
5. Modified [qs.h](#) and [qs\\_dummy.h](#) to add the User record dictionary trace record and macro [QS\\_USR\\_DICTIONARY\(\)](#).

**Note**

This new trace record requires the updated QSPY 4.5.01.

1. Corrected [qfsm\\_dis.c](#), which did not generate [QS](#) trace records for entry and exit from non-hierarchical states.
2. Updated the IAR ARM compiler used in the ARM Cortex-M examples to the latest version IAR EWARM 6.40.
3. Modified the Qt port not to define the [QPApp::onClockTick\(\)](#) slot function, but instead to allow defining this slot function in the BSP of the application.

## 10.85 Version 4.5.00, May 29, 2012

The main purpose of this release is to improve host-based development of QP applications, which is critical for Test-Driven Development (TDD). Among others, this release provides integration between QP and the popular Qt GUI framework, which allows developers to easily build host-based simulations of the embedded systems with the realistic user interfaces.

This release also simplifies implementing transitions to history, which is a preparation to providing this feature in the QM modeling tool.

Changes in detail:

1. Renamed the event class from [QEEvent](#) to [QEvt](#) to avoid a name conflict with the Qt framework. Also, for consistency, renamed the file [qevent.h](#) to [qevt.h](#) and the macro [Q\\_EVENT\\_CAST\(\)](#) to [Q\\_EVT\\_CAST\(\)](#).

**Note**

To minimize impact of this change on the existing QP ports and applications, the name [QEEvent](#) is provided as well, but this can be suppressed by defining the configuration macro [Q\\_NQEVENT](#) in [qep\\_port.h](#).

1. Changed the design of [QF\\_tick\(\)](#) (file [qf\\_tick.c](#)) to better support calling this function from low-priority tasks (as opposed to interrupts and highest-priority tasks), which often happens when QP is executed on the desktop operating systems. In this design only [QF\\_tick\(\)](#) can remove time events from the active linked list, so no unexpected changes to the list structure are eliminated.
2. Simplified the [QTimeEvt](#) class by removing the 'prev' link pointer, as the new design no longer needs a bi-directional list. These changes impact the files: [qte\\_\\*.c](#).
3. Added return value to [QF\\_run\(\)](#) to allow transfer of the exit status to the desktop operating systems.

**Note**

This modification has impact on most QP/C ports, because the QF\_run() function must now return a int16\_t value.

1. Eliminated the 'running' member of the [QActive](#) class, which has been used only in the QP ports to "big" OSes such as Linux or Windows.
2. Added member 'temp' to the [QHsm](#) and QFsm base classes to prevent clobbering the current state (the 'state' member) during transitions. This change allows keeping the current state unchanged during the entire transition chain, which in turn allows easy and generic access to the state information to store the state history in the exit actions from states. Additional bonus of this re-design is the opportunity of testing for stable state configuration in assertions added to the qhsm\_\*.c and qfsm\_\*.c files.
3. Added the QHsm\_state() and QFsm\_state() accessor macros.
4. Modified the "Transition to History" pattern implementation to use the simplified technique of obtaining the current state in the exit action from the superstate rather than all the exit actions from the substates. Modified the "State-Local Storage" (SLS) pattern as well, because it was using the transition to history constructs.
5. Re-designed the implementation of the QSPY host application, so that it can be conveniently included as part of the QP library. This allows direct [QS](#) tracing output to the screen for QP Applications running on the desktop.

**Note**

This change is part of the Qtools release 4.5.00.

1. Modified the QP ports to Win32\_1t (both the MinGW and VC2008) to output [QS](#) trace data directly to the stdout via the QSPY host-application interface. Modified the DPP examples for Win32\_1T to demonstrate the direct [QS](#) output to the screen.
2. Added QP port to Qt\_1t (Qt with one thread) and two example applications (DPP and PELICAN crossing).
3. Added GNU compiler option -pthread to QP ports for POSIX with P-threads, including QP ports and examples for Linux and Mac OS X.

## **10.86 Version 4.4.01, Mar 23, 2012**

The release fixes a bug in Q-SPY software tracing, which caused the linking error: "QS\_SIG\_() not defined". This release also includes a few cosmetic changes, which the Microchip C18 compiler didn't like.

1. Moved QS\_SIG\_() definition from qep.h to qs.h
2. Changed (QEvent \*)0 to (QEvent const \*)0 in source files qeq\_get.c, qeq\_init.c, and qa\_get\_.c.

## **10.87 Version 4.4.00, Jan 30, 2012**

The main purpose of this release is MISRA-C:2004 compliance, strong-type checking compliance, update of PC-Lint option files and tests, and general cleanup.

1. Moved the [qp\\_port.h](#) header file from the port directories to the qcp/include/ directory. Also, moved the inclusion of the [QS](#) (Spy) header files ([qs\\_port.h](#)/[qs\\_dummy.h](#)) from qep.h, qf.h, and [qk.h](#) headers to [qp\\_port.h](#). These structural changes were made to reduce the number of preprocessor #if nesting levels below 8, which is the ANSI-C limit. This was done to comply with the MISRA-C rule 1.1 (all code shall conform to ANSI/ISO C).

**Note**

This modifications have impact on most QP/C ports, because the `qp_port.h` header file must be removed from the port.

1. Added the PC-Lint option files `std.lnt` and `lib-qpc.lnt` to the `qcp/include/` directory.
2. Cleaned the whole QP/C code from lint comments. All PC-Lint options have been moved to PC-Lint option files.
3. Modified QP assertion macro `Q_DEFINE_THIS_MODULE()` to avoid using the `#` operator (MISRA rule 19.13). This macro now requires the argument enclosed in double quotes "".

**Note**

This modification has impact on some QP/C ports.

1. Added typedefs for `char`, `int_t`, `float32_t`, and `float64_t` to `event.h` header file for compliance with MISRA-C:2004 rules 6.1 and 6.3.
2. Added macros `Q_STATE_CAST()` and `Q_EVENT_CAST()` to `qep.h` to encapsulate deviation from MISRA-C rule 11.4.
3. Added macro `Q_UINT2PTR_CAST()` to encapsulate casting unsigned integers to pointers, which deviates from MISRA-C rule 11.3. This macro has been added for *application-level* code.
4. Updated ARM Cortex-M examples with the latest CMSIS v3.0, which complies with more MISRA-C:2004 rules.
5. Added DPP examples for MISRA-C:2004 compliant applications (for IAR-ARM and GNU-ARM).
6. Added testing of PC-Lint option files against the MISRA-C Exemplar Suite.
7. Updated ARM-Cortex-M3 port with GNU to the latest Sourcery CodeBench 2011.09-60.
8. Added QP/C port to Win32-1t and examples (Windows with 1 thread). This port is useful for testing embedded QP/C applications on windows.
9. Added documentation to QP/C distribution in the directory `qpc/doc/`, with the following Application notes: "MISRA-C:2004 Compliance Matrix", "Quantum Leaps Coding Standard", "QP and ARM Cortex-M, and QP and Windows",

## 10.88 Version 4.3.00, Nov 01, 2011

1. This release changes the names of critical section macros and introduces macros for unconditional interrupt disabling/enabling. This is done to simplify and speed up the built-in `QV` and `QK` kernels, which no longer are dependent on the interrupt locking policy.

**Note**

The change in handling the critical section in the `QV` and `QK` kernels can break QP ports, which use the "save and restore interrupt lock" policy, because all such ports must also define unconditional interrupt disabling and enabling.

1. This release changes the partitioning of the `QK` scheduler. Specifically, the `QK` scheduler is now divided between two functions `QK_schedPrio_()` and `QK_sched_()`, to calculate the highest-priority task ready to run and to perform scheduling, respectively. The function `QK_schedPrio_()` is useful to determine if scheduling is even necessary.
2. Updated all QP ports to comply with the new critical section names and policies.
3. Modified the ARM Cortex-M port `qk_port.s` to take advantage of the new structure of the `QK` scheduler.
4. Upgraded the examples for ARM Cortex with IAR EWARM to the latest IAR EWARM version 6.30.
5. Upgraded the examples for ARM Cortex with GNU (CodeSourcery) to the latest Sourcery CodeBench 2011.07-60.

## 10.89 Version 4.2.04, Sep 24, 2011

The main purpose of this release is to provide a bug fix for the QK port to ARM Cortex processors. The bug fix addresses a very rare and undocumented behavior of late-arrival of an interrupt while entering the PendSV exception. In this case the PENDSVSET bit in the NVIC-ICSR register is *not* cleared when finally PendSV is entered, so the PendSV exception is entered in a *different* state when it is entered via the late-arrival mechanism versus the normal activation of the exception through tail-chaining. The consequence of this undocumented and inconsistent hardware behavior, PendSV could be re-entered again before the SVCall exception cleans up the stack. The bug fix is implemented in the qk\_port.s file and consists of clearing the PENDSVSET bit programmatically inside PendSV\_Handler.

## 10.90 Version 4.2.02, Sep 08, 2011

1. The main purpose of this release is to repackage the default QP/C distribution to contain the single root directory `qpc/` in the archive. That way, unzipping the archive will produce only one directory (`qpc/`), which can be then changed by the user.
2. This release also changes the ARM Cortex QP ports with GNU. The suffix `_cs` has been added to all QP libraries generated by the Code Sourcery toolset (now Mentor Sourcery CodeBench). This is to distinguish libraries generated by different GNU-toolchains (such as CodeRed, Attolic, DevKit ARM, etc.)

## 10.91 Version 4.2.01, Aug 13, 2011

1. Modified file qassert.h to add assertion macros `Q_ASSERT_ID`, `Q_REQUIRE_ID`, `Q_ENSURE_ID`, `Q_INVARIANT_ID`, and `Q_ERROR_ID`, which are better suited for unit testing, because they avoid the volatility of line numbers for identifying assertions.
2. Added QP port and examples for Mac OS X on 80x86.

## 10.92 Version 4.2.00, Jul 14, 2011

The goal of this milestone release is to extend the number of event pools (theoretically up to 255 pools) instead of just three event pools available up til now. Also, this release adds several improvements to the QS/QSPY software tracing, such as adding sender information to event posting so that sequence diagrams could be easily and automatically reconstructed from tracing data. Also, the tracing now supports 64-bit targets, such as embedded Linux 64-bit. Finally, this milestone release migrates the examples to use the environment variable QPC instead of relying on the relative path to the QP/C framework. This allows easier adaptation of the examples for real projects, which don't really belong to the examples directory.

The changes in detail are:

1. Changed the QEvent base struct (file qevent.h). The private member `'dynamic_'` has been replaced by two members `'poolId_'` and `'refCtr_'`.
2. Added configuration macro `QF_MAX_EPOOL` (file qf.h) to define the maximum number of event pools in the QP Application (default to 3). The maximum theoretical number of this macro is 255.
3. Made algorithmic changes in the QF source code related to the change of storing the event pool-IDs and reference counters inside QEvent.
4. Changed the default signal size (macro `Q_SIGNAL_SIZE` in the file qevent.h) from 1 to 2 bytes.
5. Changed the signature of `QActive_recall()` to return `uint8_t` instead of `QEvent*`, which this could encourage incorrect usage (processing of the event at the point of recalling it). Now, the function only returns 1 (TRUE) if the event was recalled and 0 (FALSE) if the event was not recalled.

6. Added the function QTimeEvt\_ctr() and new source file qte\_ctr.c. The function returns the counter of a time event, which allows using a time event for measuring the time.
7. Added new **QF** macros **QF\_TICK**, **QF\_PUBLISH**, and **QACTIVE\_POST** in file qf.h to provide sender of the events for software tracing.
8. Added new **QS** macros (files qs.h and **qs\_dummy.h**) for handling 64-bit integers.
9. Added the functions QS\_u64() and QS\_u64\_() and new source file qs\_u64.c.
10. Added the **QS** macro #QS\_U32\_HEX\_T for hexadecimal formatting of integer numbers in the user-defined trace records.
11. Added the new port linux64 for 64-bit Linux. Also added the corresponding examples for 64-bit Linux.
12. Adapted the QSPY host application for 64-bit pointer sizes and the changed layout of trace records that contain event information (such as PoolID and RefCtr). Also added the backwards-compatibility option (-v) for switching the tool to the previous data format.
13. Removed the tools directory from the QPC distribution and moved the QSPY host application to the QTOOLS distribution, which now also contains the GNU make tools for Windows.
14. Modified the make files and project files to use the environment variable QPC instead of relying on the relative path to the QP/C framework.
15. Upgraded the examples for ARM Cortex with IAR EWARM to the latest IAR EWARM 6.20.

## 10.93 Version 4.1.07, Feb 28, 2011

The goal of this release is to improve the ease of experimenting with QP/C on the desktop. This release adds support for Windows (Win32) to the baseline code. Two most popular compilers for Windows are supported: Microsoft Visual Studio and MinGW (GNU). The support for Linux has been improved by including pre-built QP/C libraries and improving makefiles for Eclipse compatibility.

The changes in detail are:

1. Added Win32 port with the Visual C++ 2008 (ports/80x86/win32/vc2008). This directory contains the Visual Studio solution all\_qp.sln for building all QP/C libraries from the IDE. Three build configurations (Debug, Release, and Spy) are supported.
2. Added Win32 port with the MinGW compiler (ports/80x86/win32/mingw). This directory contains the Makefile for building all QP/C libraries. Three build configurations (dbg, rel, and spy) are supported.

### Note

the Makefile assumes that the MinGW/bin directory is added to the PATH.

1. Added Win32 examples for Visual C++ 2008 (examples/80x86/win32/ vc2008/dpp and examples/80x86/win32/ vc2008/qhsmtst). Visual Studio solutions are provided for all build configurations.
2. Added Win32 examples for MinGW (examples/80x86/win32/mingw/dpp and examples/80x86/win32/mingw/qhsmtst). Eclipse-compatible makefiles are provided for all build configurations.  
NOTE: the Makefiles assume that the MinGW/bin directory is added to the PATH.
3. Removed memory alignment correction in the file qmp\_init.c. This correction required casting of pointers to integers and was problematic on 64-bit targets (such as 64-bit Linux).
4. Upgraded the examples for ARM Cortex with CodeSourcery to the latest Sourcery G++ 2011.02-2.

## 10.94 Version 4.1.06, Jan 07, 2011

1. Made cosmetic improvements to the example QM models of the "Fly 'n' Shoot" game.
2. Made improvements in make.bat files for building the examples for DOS/Open Watcom to run better in DosBox on Linux.
3. Upgraded the examples for ARM Cortex with IAR to the latest IAR EWARM version 6.10.
4. Upgraded the examples for ARM Cortex with CodeSourcery to the latest Sourcery G++ 2010.09-66.

## 10.95 Version 4.1.05, Nov 01, 2010

This release adds examples for the QM (QP Modeler) graphical modeling and code generation tool. The examples are based on the "Fly 'n' Shoot" game described in the QP/C Tutorial and in Chapter 1 of the PSiCC2 book.

Specifically, the directory `<qpc>/examples/80x86/dos/watcom/1/game-qm/` contains the "Fly 'n' Shoot" game model file "game.qm". This model, when opened in the QM tool contains all state machine diagrams and generates code into the subdirectory `qm_code/`. This code can then be built and executed on any 80x86 machine (newer versions of Windows or Linux require the DOSbox application, see <http://www.dosbox.com>).

The directory `<qpc>/examples/arm-cortex/vanilla/iar/game-ev-lm3s811-qm/` contains the version of the game for the EV-LM3S811 ARM Cortex-M3 board. This directory contains the model file "game.qm", which is actually identical as the model in the DOS version. The LM3S811 version needs to be compiled with the IAR compiler and executed on the EV-LM3S811 board.

Additionally, the QP/C baseline code has been slightly modified for better conformance to the MISRA C 2004 rules and the latest PC-Lint 9.x.

## 10.96 Version 4.1.04, Mar 16, 2010

This release adds compatibility of all examples for DOS with the DOSBox emulator (<http://www.dosbox.com>) that recreates a MS-DOS compatible environment on all versions of Windows, including 64-bit Windows that don't run 16-bit DOS applications anymore.

Also, this release includes QP ports and examples for EV-LM3S811 board with the GNU-based Code Sourcery G++ toolset. Support for Sourcery G++ provides a very cost-effective option for developing QP Applications for ARM Cortex MCUs.

Finally, this release improves the Cortex Microcontroller Software Interface Standard (CMSIS) for the whole family of the Stellaris LM3Sxxx MCUs. The improvement extends the CMSIS from Sandstorm to Fury, DustDevil, and Tempest Stellaris families.

## 10.97 Version 4.1.03, Jan 21, 2010

This release is concerned with the ARM Cortex ports and examples. Specifically, this release contains the following improvements:

1. Unified source code for ARM Cortex-M3 and the new ARM Cortex-M0 cores, including the code for the preemptive `QK` kernel.
2. Compliance with the Cortex Microcontroller Software Interface Standard (CMSIS) in all ARM Cortex examples.
3. Backward-compatible support for the new LM3S811 EVAL Rev C board with different OLED display than previous revisions. (NOTE: The OSRAM 96x16x1 OLED used in REV A-B boards has been replaced RITEK 96x16x1 OLED used in Rev C.)

In the process of making the examples CMSIS-compliant, the dependency on the Luminary Micro driver library (`driverlib.a`) has been completely removed.

Additionally, the screen saver of the "Fly 'n' Shoot" game has been improved to periodically switch off the power of the OLED display, which better protects the display from burn-in. The affected file is tunnel.c.

Finally, this release introduces the QP\_VERSION macro, which identifies the QP version. Otherwise, this maintenance release does not change the QP/C API in any way, so the release has NO IMPACT on the QP/C applications except for the ARM Cortex ports and applications.

## 10.98 Version 4.1.02, Jan 14, 2010

The purpose of this minor maintenance release is the change in the directory structure for the ARM Cortex ports and examples. As new ARM Cortex cores are becoming available, the old port name "cortex-m3" could be misleading, because it actually applies to wider range of ARM Cortex cores. Consequently, all ARM Cortex ports and examples are hosted in the directory tree called "arm-cortex".

This maintenance release does not change the QP/C API in any way, so the release has NO IMPACT on the QP/C applications except for the ARM Cortex ports and applications.

## 10.99 Version 4.1.01, Nov 05, 2009

The main purpose of this release is to replace the Turbo C++ 1.01 toolset with the Open Watcom C/C++ toolset, because Turbo C++ 1.01 is no longer available for a free download. In contrast, Open Watcom is distributed under an OSI-certified open source license, which permits free commercial and non-commercial use. Open Watcom can be downloaded from [www.openwatcom.org](http://www.openwatcom.org).

All 80x86/DOS, 80x86/qk, and 80x86/ucoS2 ports and examples for Turbo C++ 1.01 have been replaced with ports and examples for Open Watcom. The make.bat scripts are provided to build the QP/C libraries and examples.

In the process of converting the examples to Open Watcom two new examples have been added to the standard QP/C distribution. The Calc2 example located in <qp>/examples/80x86/dos/watcom/l/calc2 shows how to derive state machine classes with QP 4.x. The SLS example located in <qp>/examples/80x86/dos/watcom/l/sls shows the implementation of the new State-Local Storage state design pattern.

## 10.100 Version 4.1.00, Oct 09, 2009

The release brings a number of improvements to QP/C and updates the QP/C ARM Cortex-M3 examples for the EK-LM3S811 board to the latest IAR EWARM 5.40.

This maintenance release does not change the QP/C API in any way, so the release has NO IMPACT on the QP/C applications.

The main changes in QP v4.1.00 with respect to earlier versions are as follows:

- in qs.h added a new trace record QS\_QEP\_DISPATCH logged when an event is dispatched to a state machine. This timestamped trace record marks the beginning of an RTC step. The end of the RTC step is marked by the existing timestamped trace records QS\_QEP\_INTERN\_TRAN, QS\_QEP\_INIT\_TRAN, and QS\_QEP\_IGNORED, depending on how the event is handled. The new QS\_QEP\_DISPATCH record facilitates measurement of the RTC step lengths.
- in qhsm\_dis.c added generation of the QS\_QEP\_DISPATCH trace record.
- in the tools/qspy/ sub-directory added output of the new trace record to the Q-SPY host application.
- in the tools/qspy/matlab/ sub-directory added processing of the new trace record to the qspy.m script.
- in qpset.h changed the implementation of the Priority Set. In particular, the QPSet64 now derives from QPSet8, which enables a common way of testing for non-empty set (e.g., useful in assembly). Also, the findMax() functions in QPSet8 and QPSet64 now can work with an empty set, in which case they return 0.
- in qk\_sched.c simplified the QK\_schedule\_() function to skip the testing of the ready-set for non-empty condition. Such test is no longer necessary. The test can still be performed outside the QK\_schedule\_() function (e.g., in assembly) to avoid calling the scheduler if the ready set is empty.

- in qk\_ext.c simplified the QK\_scheduleExt\_() function in the same way as QK\_schedule\_().
- modified make.bat files for building QP libraries in the ports/ directory to use the externally defined environment variables for the location of the toolchains. The defaults are applied only when the environment variable is not defined. This enables greater flexibility in installing the development tools in different directories than those chosen by Quantum Leaps.
- modified the ARM Cortex-M3 examples for the new IAR EWARM 5.40.
- modified slightly the Calculator example to allow extensibility.
- in the ARM Cortex-M3 port file qk\_port.s added explicit testing of the QF\_readySet\_set for empty condition. This test allows avoiding calling the QK scheduler and two context-switches if the ready-set is empty.
- in the game example moved setting up the QS filters from main.c to bsp.c.

## 10.101 Version 4.0.04, Apr 09, 2009

The maintenance release provides a fix for the compile-time assertions, which did not work correctly for the GNU compiler family. Also, the ARM Cortex-M3 examples have been recompiled with the newer IAR EWARM v5.30. This maintenance release does not change the QP/C API in any way, so the release has NO IMPACT on the QP/C applications.

The main changes in QP v4.0.04 with respect to earlier version are as follows:

- in qassert.h file the Q\_ASSERT\_COMPILE macro has been modified to render a negative array dimension when the asserted condition is not TRUE.

## 10.102 Version 4.0.03, Dec 27, 2008

The main purpose of this release is to fix a bug in the QK preemptive kernel, which occurs only when the advanced QK features are used. Specifically, the QK priority-ceiling mutex could interfere with QK thread-local storage (TLS) or QK extended context switch. When the QK mutex is in use, the TLS or the extended context for this task could get saved to an incorrect priority level.

The release 4.0.03 fixes the bug by strictly preserving the semantics of QK\_currPrio\_ variable. The mutex locking now uses a different variable QK\_ceilingPrio\_, which represents the ceiling-priority locked by the mutex. The QK scheduler and extended scheduler now perform an additional check to make sure that only tasks with priorities above the ceiling can run. To avoid that additional overhead, the user can define the macro QK\_NO\_MUTEX, which eliminates the QK mutex API and eliminates the additional tests in the QK schedulers.

This maintenance release does not change the QP/C API in any way, so the release has NO IMPACT on the QP/C applications.

The main changes in QP v4.0.03 with respect to earlier version are as follows:

- in qk.h file made the QK mutex API only visible when the macro QK\_NO\_MUTEX is *not* defined.
- in qk\_pkg.h file, added the QK\_ceilingPrio\_ external declaration when the macro QK\_NO\_MUTEX is not defined.
- in qk\_mutex.c file, changed priority-ceiling mutex implementation to use the QK\_ceilingPrio\_ instead of QK\_currPrio\_. Also, added compiler error when the macro QK\_NO\_MUTEX is defined this and this file is included in the build.
- in qk\_sched.c file added testing priority against the QK\_ceilingPrio\_, when the macro QK\_NO\_MUTEX is not defined.
- in qk\_ext.c file added testing priority against the QK\_ceilingPrio\_, when the macro QK\_NO\_MUTEX is not defined.

## 10.103 Version 4.0.02, Nov 15, 2008

This maintenance release does not change the QP/C API in any way, so the release has NO IMPACT on the QP/C applications.

The main changes in QP v4.0.02 with respect to earlier version are as follows:

- in qep.h file, added comments to macros [Q\\_TRAN\(\)](#) and [Q\\_SUPER\(\)](#) to suppress the PC-lint warning about using the comma-operator (MISRA rule 42).
- in qhsm\_in.c file, fixed a bug in the [QHsm\\_isIn\(\)](#) function.
- fixed a bug in tunnel.c file ("Fly 'n' Shoot" game). The constant event HIT\_WALL was not declared static.

## 10.104 Version 4.0.01, June 09, 2008

This maintenance release is made to allow using [QS](#) software tracing with the GNU compiler for AVR (WinAVR). Specifically, the output of the strings residing in ROM has been fixed.

This maintenance release does not change the QP/C API in any way, so the release has NO IMPACT on the QP/C applications.

The main changes in QP v4.0.01 with respect to earlier version are as follows:

- in qs\_.c file, updated the function QS\_str\_ROM\_().
- in qs\_str.c file, updated the function QS\_str\_ROM().
- in qvanilla.c file, function QF\_run(), declared the temporary variables as static to save stack space, because QF\_run() never returns and is not reentrant.

## 10.105 Version 4.0.00, Apr 07, 2008

This milestone release is made for the book /ref PSiCC2. The book describes in great detail this new release. The older "QP Programmer's Manual" is now phased out and is replaced with this hyper-linked /ref main\_page "QP/C Reference Manual", which provides very detailed, easily searchable reference to the software. The book /ref PSiCC2 provides in-depth discussion of the relevant concepts as well as the design study of QP v4.0.

The main changes in QP v4.0 with respect to earlier versions are as follows:

- the coding techniques for hierarchical state machines (HSMs) and the simpler finite state machines (FSMs) have changed. While the changes are quite simple, the backward compatibility with QEP 3.x has been broken, meaning that some manual changes to the state machines implemented with earlier versions are necessary. Please refer to the "QP/C Tutorial" Section /ref coding\_hsm for more information about coding state machines with QEP 4.x.
  - The main change is the signature of a state-handler function, which now returns simply a byte. This return type (typedef'd as [QState](#)) is the status of the event-handling that the state handler conveys to the QEP event processor.
  - The macro [Q\\_TRAN\(\)](#) must now always follow the return statement.
  - The new macro [Q\\_SUPER\(\)](#) designates the superstate of the given state. Again, this macro must follow the return statement.
  - Then two new macros [Q\\_HANDLED\(\)](#) and [#Q\\_IGNORED\(\)](#) have been added to return the status of event handled and event ignored, respectively.
- all callback functions are now consistently called /c #XXX\_onYYY():
  - Q\_assert\_handler() is now [Q\\_onAssert\(\)](#)
  - QF\_start() is now QF\_onStartup()

- QF\_cleanup() is now QF\_onCleanup()
- the new header file qevent.h has been broken off the qep.h header file. qevent.h contains the QEvent class and other basic facilities used in the whole QP. This new file allows easier replacement of the entire QEP event processor with custom event processors, if you wish do so.
- the macro QEP\_SIGNAL\_SIZE is renamed to [Q\\_SIGNAL\\_SIZE](#).
- the data type QSTATE is now deprecated. Please use [QState](#).
- the "protected" in C don't no longer have the trailing underscore. For example, QHsm\_ctor\_() is replaced with #QHsm\_ctor(), etc.
- the QF\_FSM\_ACTIVE macro is now deprecated. Instead, you have the family of macros QF\_ACTIVE\_SUPER\_, QF\_ACTIVECTOR\_, QF\_ACTIVE\_INIT\_, QF\_ACTIVE\_DISPATCH\_, QF\_ACTIVE\_STATE\_, which allow replacing the base class for active objects in [QF](#). By default, these macros are defined to use the [QHsm](#) class from the QEP hierarchical event processor, but you can replace the event processor, if you wish.
- the internal macro QACTIVE\_OSOBJECT\_WAIT\_() is now [QACTIVE\\_EQUEUE\\_WAIT\\_\(\)](#).
- the internal macro QACTIVE\_OSOBJECT\_SIGNAL\_() is now [QACTIVE\\_EQUEUE\\_SIGNAL\\_\(\)](#).
- the internal macro QACTIVE\_OSOBJECT\_ONIDLE\_() is now QACTIVE\_EQUEUE\_ONEMPTY\_().
- the data members [QActive.osObject](#) and [QActive.thread](#) are now present only if the macros [QACTIVE\\_OS\\_OBJ\\_TYPE](#) and [QACTIVE\\_THREAD\\_TYPE](#) are defined.
- the [QPSet](#) class has been renamed to QPSet64.
- the QPSet\_hasElements() has been renamed QPSet64\_notEmpty()
- the [QS](#) software tracing is now better integrated with all QP components. You no longer need to explicitly include [qs\\_port.h](#), because it is automatically included when you define the macro [Q\\_SPY](#). Also the file [qs\\_dummy.h](#) is included automatically when the macro [Q\\_SPY](#) is **not** defined.
- the new header file qvanilla.h now replaces the file qsched.h.
- the file qa\_ctor.c is now obsolete.
- the macros QF\_SCHED\_LOCK() and QF\_SCHED\_UNLOCK() are now obsolete.
- the native [QF](#) event queues (both the active object event queues and the "raw" thread-safe queues) are slightly more efficient by counting down the head and tail pointers rather than up. This leads to wrap-around at zero, which is easier (faster) to test than any other wrap-around point. Also the native [QF](#) event queues maintain the minimum of the free events in the queue rather the maximum of used events.
- the data member of QEQueue.nTot class is removed.
- the QF\_publish() function has been re-written so that [QF](#) no does **not need to lock the scheduler**. The QF\_publish() function posts events to active objects with scheduler unlocked starting from the highest-priority active objects. However, the event is protected from inadvertent recycling by incrementing its reference counter before the publish operation. After the event is posted to all subscribers, the garbage collector QF\_gc() is called to decrement the reference counter and recycle the event, if necessary.
- the qf\_run.c file is obsolete. The QF\_run() function for the non-preemptive "vanilla" kernel is now implemented in the file qvanilla.c.
- the QF\_tick() function has been rewritten to allow calling QF\_tick() from the task context as well as from the interrupt context. The nested critical section around QF\_tick() is no longer needed when it is called from the task level. Among others, this re-design **eliminates the need for the recursive mutex** in the POSIX [QF](#) port.

- the QMPool\_init() function has been re-designed to optimally align the memory buffer in a portable and platform-independent way. This should bring some performance improvements on some CPUs (e.g., 80x86).
- the extended **QK** scheduler has been re-designed to save stack space. The extended context (e.g., coprocessor registers) are no longer saved on the precious stack, but rather in the active object.
- a bug has been fixed in handling of Thread-Local Storage (TLS) in the **QK** scheduler and extended scheduler.
- the -q (quiet) flag has been added to the QSPY host application.
- the support for two new compilers for Windows has been added for the QSPY host application. The application can now be build with the MinGW GNU compiler for Windows as well as the Microsoft Visual C++ 2005.
- the QP port to Linux has been improved by eliminating the need for recursive P-Thread mutex.
- the QP port to MicroC/OS-II has been upgraded to the latest version 2.86.
- all examples in the standard QP distribution have been cleaned up and updated to the latest QP API changes.
- all examples that use **QF** now contain the **QS** software tracing support.

# Chapter 11

## Hierarchical Index

### 11.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

QActiveDummy . . . . .	327
QAsm . . . . .	327
QActive . . . . .	311
QMActive . . . . .	360
QTicker . . . . .	382
QHsm . . . . .	350
QMsm . . . . .	368
QAsmAttr . . . . .	329
QAsmVtable . . . . .	330
QEQueue . . . . .	331
QEvt . . . . .	337
QTimeEvt . . . . .	385
QF . . . . .	341
QF_Attr . . . . .	349
QFreeBlock . . . . .	350
QHsmDummy . . . . .	356
QK . . . . .	357
QK_Attr . . . . .	359
QMPool . . . . .	363
QMState . . . . .	376
QMTranActTable . . . . .	377
QPSet . . . . .	378
QS . . . . .	380
QS_Filter . . . . .	380
QS_TProbe . . . . .	380
QSpyId . . . . .	381
QSubscrList . . . . .	381
QV . . . . .	394
QV_Attr . . . . .	396
QXK . . . . .	397
QXK_Attr . . . . .	397
QXMutex . . . . .	397
QXSemaphore . . . . .	398
QXThread . . . . .	399



# Chapter 12

## Class Index

### 12.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

<a href="#">QActive</a>	Active object class (based on the <a href="#">QHsm</a> implementation strategy) . . . . .	311
<a href="#">QActiveDummy</a>	Dummy Active Object for testing . . . . .	327
<a href="#">QAsm</a>	Abstract State Machine class (state machine interface) . . . . .	327
<a href="#">QAsmAttr</a>	Attribute of for the <a href="#">QAsm</a> class (Abstract State Machine) . . . . .	329
<a href="#">QAsmVtable</a>	Virtual table for the <a href="#">QAsm</a> class . . . . .	330
<a href="#">QEQueue</a>	Native <a href="#">QF</a> Event Queue . . . . .	331
<a href="#">QEvt</a>	Event class . . . . .	337
<a href="#">QF</a>	<a href="#">QF</a> Active Object Framework ( <a href="#">QF</a> namespace emulated as a "class" in C) . . . . .	341
<a href="#">QF_Attr</a>	Private attributes of the <a href="#">QF</a> framework . . . . .	349
<a href="#">QFreeBlock</a>	Structure representing a free block in <a href="#">QMPool</a> . . . . .	350
<a href="#">QHsm</a>	Hierarchical State Machine class (QHsm-style state machine implementation strategy) . . . . .	350
<a href="#">QHsmDummy</a>	Dummy abstract-state machine class for testing . . . . .	356
<a href="#">QK</a>	<a href="#">QK</a> preemptive non-blocking kernel ( <a href="#">QK</a> namespace emulated as a "class" in C) . . . . .	357
<a href="#">QK_Attr</a>	Private attributes of the <a href="#">QK</a> kernel . . . . .	359
<a href="#">QMActive</a>	Active object class (based on <a href="#">QMsm</a> implementation strategy) . . . . .	360
<a href="#">QMPool</a>	Native <a href="#">QF</a> Memory Pool . . . . .	363
<a href="#">QMsm</a>	Hierarchical State Machine class (QMsm-style state machine implementation strategy) . . . . .	368
<a href="#">QMState</a>	State object for the <a href="#">QMsm</a> class (QM State Machine) . . . . .	376

<a href="#">QMTranActTable</a>	Transition-Action Table for the <a href="#">QMsm</a> State Machine . . . . .	377
<a href="#">QPSet</a>	Set of Active Objects of up to <a href="#">QF_MAX_ACTIVE</a> elements . . . . .	378
<a href="#">QS</a>	Software tracing instrumentation, target-resident component ( <a href="#">QS</a> namespace emulated as a "class" in C) . . . . .	380
<a href="#">QS_Filter</a>	<a href="#">QS</a> type for output filters (global and local) . . . . .	380
<a href="#">QS_TProbe</a>	QUTest Test-Probe attributes . . . . .	380
<a href="#">QSpyId</a>	<a href="#">QS</a> ID type for applying local filtering . . . . .	381
<a href="#">QSubscrList</a>	Subscriber List (for publish-subscribe) . . . . .	381
<a href="#">QTicker</a>	"Ticker" Active Object class . . . . .	382
<a href="#">QTimeEvt</a>	Time Event class . . . . .	385
<a href="#">QV</a>	<a href="#">QV</a> non-preemptive kernel ( <a href="#">QV</a> namespace emulated as a "class" in C) . . . . .	394
<a href="#">QV_Attr</a>	Private attributes of the <a href="#">QV</a> kernel . . . . .	396
<a href="#">QXK</a>	<a href="#">QXK</a> dual-mode kernel ( <a href="#">QXK</a> namespace emulated as a "class" in C) . . . . .	397
<a href="#">QXK_Attr</a>	Private attributes of the <a href="#">QXK</a> kernel . . . . .	397
<a href="#">QXMutex</a>	Blocking Mutex of the <a href="#">QXK</a> preemptive kernel . . . . .	397
<a href="#">QXSemaphore</a>	Counting Semaphore of the <a href="#">QXK</a> preemptive kernel . . . . .	398
<a href="#">QXThread</a>	EXtended (blocking) thread of the <a href="#">QXK</a> preemptive kernel . . . . .	399

# Chapter 13

## File Index

### 13.1 File List

Here is a list of all files with brief descriptions:

<a href="#">qp_config.h</a>	Sample QP/C configuration file . . . . .	403
<a href="#">qp_port.h</a>	Sample QP/C port . . . . .	411
<a href="#">qs_port.h</a>	Sample QS/C port . . . . .	415
<a href="#">dequeue.h</a>	QP native platform-independent thread-safe event queue interface . . . . .	417
<a href="#">qk.h</a>	QK/C (preemptive non-blocking kernel) platform-independent public interface . . . . .	418
<a href="#">qmpool.h</a>	QP native platform-independent memory pool <a href="#">QMPool</a> interface . . . . .	420
<a href="#">qp.h</a>	QP/C platform-independent public interface . . . . .	422
<a href="#">qp_pkg.h</a>	Internal (package scope) QP/C interface . . . . .	449
<a href="#">qpc.h</a>	QP/C interface including the backwards-compatibility layer . . . . .	451
<a href="#">qs_dummy.h</a>	QS/C dummy public interface . . . . .	455
<a href="#">qsafe.h</a>	QP Functional Safety (FuSa) Subsystem . . . . .	469
<a href="#">qstamp.h</a>	Date/time for time-stamping the QP builds (used in <a href="#">QS</a> software tracing) . . . . .	480
<a href="#">qv.h</a>	QV/C (non-preemptive kernel) platform-independent public interface . . . . .	480
<a href="#">qep_hsm.c</a>	<a href="#">QHsm</a> class implementation . . . . .	482
<a href="#">qep_msm.c</a>	<a href="#">QMsm</a> class implementation . . . . .	483
<a href="#">qf_act.c</a>	Defines Active Object management internal data . . . . .	483
<a href="#">qf_actq.c</a>	Active Object queue operations based on the <a href="#">QEQueue</a> class . . . . .	484
<a href="#">qf_defer.c</a>	Event deferral facilities . . . . .	484

qf_dyn.c	Dynamic event management facilities . . . . .	485
qf_mem.c	QMPool implementation . . . . .	485
qf_ps.c	Publish-subscribe facilities . . . . .	486
qf_qact.c	Active Object management facilities . . . . .	486
qf_qeq.c	QEQueue implementation . . . . .	486
qf_qmact.c	QMActive implementation . . . . .	487
qf_time.c	QTimeEvt implementation . . . . .	487
qk.c	Preemptive non-blocking QK kernel implementation . . . . .	487
qstamp.c	QS time stamp . . . . .	487
qv.c	Non-preemptive non-blocking QV kernel implementation . . . . .	488

# Chapter 14

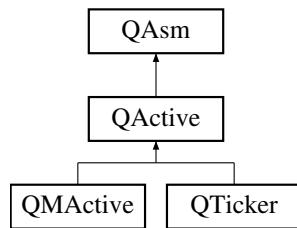
## Class Documentation

### 14.1 QActive Class Reference

Active object class (based on the `QHsm` implementation strategy)

```
#include "qp.h"
```

Inheritance diagram for QActive:



#### Public Member Functions

- void `QActive_SetAttr` (`QActive *const me`, `uint32_t attr1`, `void const *attr2`)
- void `QActive_Start` (`QActive *const me`, `QPrioSpec const prioSpec`, `QEvtPtr *const qSto`, `uint_fast16_t const qLen`, `void *const stkSto`, `uint_fast16_t const stkSize`, `void const *const par`)

*Starts execution of an active object and registers the object with the framework.*

#### Static Public Member Functions

- void `QActive_PslInit` (`QSubscrList *const subscrSto`, `enum_t const maxSignal`)  
*Publish event to all subscribers of a given signal e->sig*
- `uint_fast16_t QActive_GetQueueMin` (`uint_fast8_t const prio`)  
*This function returns the minimum of free entries of the given event queue.*

#### Protected Member Functions

- void `QActive_Ctor` (`QActive *const me`, `QStateHandler const initial`)  
*QActive constructor (abstract base class)*
- void `QActive_Stop` (`QActive *const me`)  
*Stops execution of an active object and removes it from the framework's supervision.*
- void `QActive_Subscribe` (`QActive const *const me`, `enum_t const sig`)  
*Subscribes for delivery of signal sig to the active object.*
- void `QActive_Unsubscribe` (`QActive const *const me`, `enum_t const sig`)

- void **QActive\_unsubscribeAll** (**QActive** const \*const me)
 

*Unsubscribes from the delivery of signal `sig` to the active object.*
- bool **QActive\_defer** (**QActive** const \*const me, struct **QEQueue** \*const eq, **QEvt** const \*const e)
 

*Defer an event to a given separate event queue.*
- bool **QActive\_recall** (**QActive** \*const me, struct **QEQueue** \*const eq)
 

*Recall a deferred event from a given event queue.*
- uint\_fast16\_t **QActive\_flushDeferred** (**QActive** const \*const me, struct **QEQueue** \*const eq, uint\_fast16\_t const num)
 

*Flush the specified number of events from the deferred queue `eq`*

### Protected Attributes

- **QAsm super**
- uint8\_t **prio**

*QF-priority [1..**QF\_MAX\_ACTIVE**] of this AO.*
- uint8\_t **pthre**

*Preemption-threshold [1..**QF\_MAX\_ACTIVE**] of this AO.*
- **QACTIVE\_THREAD\_TYPE** **thread**

*Port-dependent representation of the thread of the active object.*
- **QACTIVE\_OS\_OBJ\_TYPE** **osObject**

*Port-dependent per-thread object.*
- **QACTIVE\_EQUEUE\_TYPE** **eQueue**

*Port-dependent event-queue type (often **QEQueue**)*

### Protected Attributes inherited from **QAsm**

- struct **QAsmVtable** const \* **vptr**

*Virtual pointer inherited by all **QAsm** subclasses (see also **SAS\_QP\_OO**)*
- union **QAsmAttr** **state**

*Current state (pointer to the current state-handler function)*
- union **QAsmAttr** **temp**

*Temporary storage for target/act-table etc.*

### Private Member Functions

- void **QActive\_register\_** (**QActive** \*const me)
 

*Register this active object to be managed by the framework.*
- void **QActive\_unregister\_** (**QActive** \*const me)
 

*Un-register the active object from the framework.*
- bool **QActive\_post\_** (**QActive** \*const me, **QEvt** const \*const e, uint\_fast16\_t const margin, void const \*const sender)
 

*Posts an event `e` directly to the event queue of the active object using the First-In-First-Out (FIFO) policy.*
- void **QActive\_postLIFO\_** (**QActive** \*const me, **QEvt** const \*const e)
 

*Posts an event `e` directly to the event queue of the active object using the Last-In-First-Out (LIFO) policy.*
- **QEvt** const \* **QActive\_get\_** (**QActive** \*const me)
 

*Get an event from the event queue of an active object.*
- void **QActive\_evtLoop\_** (**QActive** \*const me)
 

*Event loop thread routine for executing an active object `act` (defined some in QP ports)*

### Static Private Member Functions

- void [QActive\\_publish\\_](#) (QEvt const \*const e, void const \*const sender, uint\_fast8\_t const qslid)  
*Publish event to all subscribers of a given signal e->sig*
- static void [QActive\\_postFIFO\\_](#) (QActive \*const me, QEvt const \*const e, void const \*const sender)

### Private Attributes

- [QActive \\* QActive\\_registry\\_](#) [QF\_MAX\_ACTIVE+1U]  
*Static (one per-class) array of registered active objects.*
- [QSubscrList \\* QActive\\_subscrList\\_](#)  
*Static (one per-class) pointer to all subscriber AOs for a given event signal.*
- [enum\\_t QActive\\_maxPubSignal\\_](#)  
*Static (one per-class) maximum published signal (the size of the subscrList\_ array)*

### Additional Inherited Members

#### Static Protected Member Functions inherited from [QAsm](#)

- static void [QAsm\\_ctor](#) (QAsm \*const me)  
*Constructor of the [QAsm](#) base class.*

### 14.1.1 Detailed Description

Active object class (based on the [QHsm](#) implementation strategy)

#### Details

[Active Objects](#) (a.k.a., Actors) are autonomous software objects, each possessing an [event queue](#), [execution context](#), and an internal [state machine](#). All interactions with an Active Object occur by asynchronous posting of events, which are processed one at a time (to completion) in the execution context of the Active Object. As long as there is no sharing of data or resources among Active Objects (or any other concurrent entities), there are no concurrency hazards.

The [QActive](#) class represents an Active Object that uses the QHsm-style implementation strategy for state machines. This strategy is tailored to manual coding, but it is also supported by the QM modeling tool.

#### Note

[QActive](#) is not intended to be instantiated directly, but rather serves as the abstract base class for derivation of active objects in the applications.

#### See also

- [QHsm](#)
- [QMActive](#)

#### Backward Traceability

- [SDS\\_QP\\_QF](#): *QF Active Object Framework*
- [SDS\\_QP\\_QActive](#): *QActive Active Object class.*
- [QHsm](#): *Hierarchical State Machine class (QHsm-style state machine implementation strategy)*

## Usage

The following example illustrates how to derive an active object from `QActive`.

```
typedef struct {
    QActive super;      // inherit QActive

    QTimeEvt timeEvt; // to timeout the blinking
} Blinky;
.

void Blinky_ctor(Blinky * const me) {
    // constructor of the superclass <---
    QActive_ctor(&me->super, Q_STATE_CAST(&Blinky_initial));

    // constructor(s) of the members
    QTimeEvt_ctorX(&me->timeEvt, &me->super, TIMEOUT_SIG, 0U);
}
```

## 14.1.2 Member Function Documentation

### 14.1.2.1 QActive\_ctor()

```
void QActive_ctor (
    QActive *const me,
    QStateHandler const initial) [protected]
```

`QActive` constructor (abstract base class)

#### Parameters

in,out	me	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	initial	pointer to the top-most initial state-handler function in the derived active object

#### Backward Traceability

- `QActive`: Active object class (based on the `QHsm` implementation strategy)

### 14.1.2.2 QActive\_setAttr()

```
void QActive_setAttr (
    QActive *const me,
    uint32_t attr1,
    void const * attr2)
```

Generic setting of additional attributes (defined in some QP ports)

#### Backward Traceability

- `QActive`

### 14.1.2.3 QActive\_start()

```
void QActive_start (
    QActive *const me,
    QPrioSpec const prioSpec,
    QEvtPtr *const qSto,
    uint_fast16_t const qLen,
    void *const stkSto,
    uint_fast16_t const stkSize,
    void const *const par)
```

Starts execution of an active object and registers the object with the framework.

#### Details

Starts execution of the AO and registers the AO with the framework.

### Parameters

in, out	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>prioSpec</i>	priority specification for the AO (See <a href="#">QPrioSpec</a> )
in	<i>qSto</i>	pointer to the storage for the ring buffer of the event queue
in	<i>qLen</i>	length of the event queue [# <a href="#">QEvtPtr</a> pointers]
in	<i>stkSto</i>	pointer to the stack storage (might be NULL)
in	<i>stkSize</i>	stack size [bytes]
in	<i>par</i>	pointer to an extra parameter (might be NULL)

### Backward Traceability

- [QActive](#): Active object class (based on the [QHsm](#) implementation strategy)
- [DVR\\_QP\\_MC4\\_R08\\_13](#): Rule 8.13(Advisory): A pointer should point to a const-qualified type whenever possible

### Usage

The following example shows starting an AO when a per-task stack is NOT needed:

```
#include "qpc.h"

int main() {
    QF_init(); // initialize the framework and the underlying RT kernel
    BSP_init(); // initialize the Board Support Package
    ...
    // instantiate and start the active objects...
    Blinky_ctor();
    static QEvtPtr l_blinkyQSto[10]; // Event queue storage for Blinky
    QActive_start(AO_Blinky, // AO pointer to start
                  1U, // unique QP priority of the AO
                  l_blinkyQSto, // storage for the AO's queue
                  Q_DIM(l_blinkyQSto), // length of the queue [entries]
                  (void *)0, // stack storage (not used in QK)
                  0U, // stack size [bytes] (not used in QK)
                  (void *)0); // initialization parameter (or 0)
    ...
    return QF_run(); // run the QF application
}
```

#### 14.1.2.4 QActive\_stop()

```
void QActive_stop (
    QActive *const me) [protected]
```

Stops execution of an active object and removes it from the framework's supervision.

### Parameters

in, out	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
---------	-----------	---

### Attention

[QActive\\_stop\(\)](#) must be called only from the AO that is about to stop its execution. By that time, any pointers or references to the AO are considered invalid (dangling) and it becomes illegal for the rest of the application to post events to the AO.

### Backward Traceability

- [QActive](#): Active object class (based on the [QHsm](#) implementation strategy)

#### 14.1.2.5 QActive\_register\_()

```
void QActive_register_(
    QActive *const me) [private]
```

Register this active object to be managed by the framework.

##### Details

This function adds a given active object to the active objects managed by the **QF** framework. It should not be called by the application directly, only through the function `QActive::start()`.

##### Parameters

in, out	me	current instance pointer (see <a href="#">SAS_QP_OO</a> )
---------	----	---

**Precondition** qf\_qact:100

- the "QF-priority" of the AO must be in range (must be set before calling [QActive\\_register\\_\(\)](#))
- the "QF-priority" must not be already in use (unique priority)
- the "QF-priority" must not exceed the "preemption-threshold"

**Postcondition** qf\_qact:190

- the preceding pre-thre must not exceed the preemption-threshold
- the preemption-threshold must not exceed the next preemption-threshold

##### Backward Traceability

- [QActive: Active object class \(based on the QHsm implementation strategy\)](#)

#### 14.1.2.6 QActive\_unregister\_()

```
void QActive_unregister_(
    QActive *const me) [private]
```

Un-register the active object from the framework.

##### Details

This function un-registers a given active object from the active objects managed by the **QF** framework. It should not be called by the QP ports.

##### Parameters

in	me	pointer to the active object to remove from the framework.
----	----	--

**Precondition** qf\_qact:200

- the priority of the active object must not be zero and cannot exceed the maximum [QF\\_MAX\\_ACTIVE](#)
- the priority of the AO must be already registered.

##### Note

The active object that is removed from the framework can no longer participate in any event exchange.

##### Backward Traceability

- [QActive: Active object class \(based on the QHsm implementation strategy\)](#)

### 14.1.2.7 QActive\_post\_()

```
bool QActive_post_ (
    QActive *const me,
    QEvt const *const e,
    uint_fast16_t const margin,
    void const *const sender) [private]
```

Posts an event *e* directly to the event queue of the active object using the First-In-First-Out (FIFO) policy.

#### Details

Direct event posting is the simplest asynchronous communication method available in [QF](#). Should be called only through the macro [QACTIVE\\_POST\(\)](#).

#### Parameters

in,out	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>e</i>	pointer to the event to be posted
in	<i>margin</i>	number of required free slots in the queue after posting the event or <a href="#">QF_NO_MARGIN</a> .
in	<i>sender</i>	pointer to a sender object (used in <a href="#">QS</a> only)

#### Returns

'true' (success) if the posting succeeded (with the provided margin) and 'false' (failure) when the posting fails.

#### Precondition [qf\\_actq:102](#)

- the event pointer must be valid
- check internal event integrity (QP FuSa Subsystem)

#### Postcondition [qf\\_actq:110](#)

- the event must be posted if (*margin* == [QF\\_NO\\_MARGIN](#))

#### Attention

For *margin* == [QF\\_NO\\_MARGIN](#), this function will assert internally if the event posting fails. In that case, it is unnecessary to check the return value from this function.

#### Note

This function should be called only through the macro [QACTIVE\\_POST\(\)](#). The reason is that the last parameter (*sender*) is needed only for [software tracing](#), and typically is only defined when software tracing is enabled. When software tracing is disabled, the macro [QACTIVE\\_POST\(\)](#) does not use the *sender* parameter, so it might not be defined.

#### Remarks

This function might be implemented differently in various QP/C ports. The provided implementation assumes that the [QEQueue](#) class is used for the [QActive](#) event queue.

#### Backward Traceability

- [QActive](#): Active object class (based on the [QHsm](#) implementation strategy)
- DVP\_QS\_MC4\_R15\_05: Rule 15.5(Advisory): A function should have a single point of exit at the end

## Usage

```

extern QActive * const AO_Table;

QState Philo_hungry(Philo * const me, QEvt const * const e) {
    QState status;
    switch (e->sig) {
        case Q_ENTRY_SIG: {
            TableEvt *pe = Q_NEW(TableEvt, HUNGRY_SIG); // dynamic alloc
            pe->philNum = me->num;
            QACTIVE_POST(AO_Table, &pe->super, me); // <===
            status = Q_HANDLED();
            break;
        }
        .
        .
        default: {
            status = Q_SUPER(&QHsm_top);
            break;
        }
    }
    return status;
}

```

### 14.1.2.8 QActive\_postLIFO\_()

```

void QActive_postLIFO_ (
    QActive *const me,
    QEvt const *const e) [private]

```

Posts an event *e* directly to the event queue of the active object using the Last-In-First-Out (LIFO) policy.

#### Details

The LIFO policy should be used only for self-posting and with caution because it alters order of events in the queue.

#### Parameters

in, out	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>e</i>	pointer to the event to be posted

#### Precondition *qf\_actq*:200

- for the [QXK](#) kernel, postLIFO\_() cannot be called from an extended thread

#### Precondition *qf\_actq*:201

- the queue must be able to accept the event (cannot overflow)

#### Backward Traceability

- [QActive](#): Active object class (based on the [QHsm](#) implementation strategy)
- DVP\_QS\_MC4\_R15\_05: Rule 15.5(Advisory): A function should have a single point of exit at the end

#### Note

This function might be implemented differently in various QP/C ports. The provided implementation assumes that the [QEQueue](#) class is used for the [QActive](#) event queue.

#### See also

- [QActive\\_post\\_\(\)](#)

### 14.1.2.9 QActive\_get\_()

```
QEvt const * QActive_get_ (
    QActive *const me) [private]
```

Get an event from the event queue of an active object.

#### Details

This function is only for internal use inside the QP Framework and QP Application should not call this function (there is never a need for it). The behavior of this function depends on the kernel used in the [QF](#) port. For built-in kernels ([QV](#) or [QK](#)) the function can be called only when the queue is not empty, so it doesn't block. For a blocking kernel/OS the function can block and wait for delivery of an event.

#### Parameters

in, out	me	current instance pointer (see <a href="#">SAS_QP_OO</a> )
---------	----	---

#### Returns

A pointer to the received event. The returned pointer is guaranteed to be valid (can't be NULL).

#### Note

This function might be implemented differently in various QP/C ports. The provided implementation assumes that the [QEQueue](#) class is used for the [QActive](#) event queue.

#### Backward Traceability

- [QActive](#): Active object class (based on the [QHsm](#) implementation strategy)

### 14.1.2.10 QActive\_psInit()

```
void QActive_psInit (
    QSubscrList *const subscrSto,
    enum_t const maxSignal) [static]
```

Publish event to all subscribers of a given signal `e->sig`

#### Details

This function posts (using the FIFO policy) the event `e` to **all** active objects that have subscribed to the signal `e->sig`, which is called *multicasting*. The multicasting performed in this function is very efficient based on reference-counting inside the published event ("zero-copy" event multicasting). This function is designed to be callable from any part of the system, including ISRs, device drivers, and active objects.

#### Note

To avoid any unexpected re-ordering of events posted into AO queues, the event multicasting is performed with scheduler **locked**. However, the scheduler is locked only up to the priority level of the highest-priority subscriber, so any AOs of even higher priority, which did not subscribe to this event are *not* affected.

#### Backward Traceability

- [SDS\\_QA\\_START](#): QA Application startup sequence
- [QActive](#): Active object class (based on the [QHsm](#) implementation strategy)

#### 14.1.2.11 QActive\_publish\_()

```
void QActive_publish_ (
    QEvt const *const e,
    void const *const sender,
    uint_fast8_t const qsId) [static], [private]
```

Publish event to all subscribers of a given signal `e->sig`

##### Details

This function posts (using the FIFO policy) the event `e` to **all** active objects that have subscribed to the signal `e->sig`, which is called *multicasting*. The multicasting performed in this function is very efficient based on reference-counting inside the published event ("zero-copy" event multicasting). This function is designed to be callable from any part of the system, including ISRs, device drivers, and active objects.

**Precondition** `qf_ps >= 200`

- the published signal must be within the configured range

**Precondition** `qf_ps >= 202`

- check the integrity of the subscriber set (QP FuSa Subsystem)

##### Note

To avoid any unexpected re-ordering of events posted into AO queues, the event multicasting is performed with scheduler **locked**. However, the scheduler is locked only up to the priority level of the highest-priority subscriber, so any AOs of even higher priority, which did not subscribe to this event are *not* affected.

##### Attention

This operation is typically used via macro `QACTIVE_PUBLISH()`. This is because the last parameter `qsId` might be defined/provided only in the Spy build configuration (see [Q\\_SPY](#)). The macro ignores the `qsId` parameter outside the Spy configuration, so the same code builds correctly in all configurations.

##### Backward Traceability

- [QActive](#): Active object class (based on the [QHsm](#) implementation strategy)

##### Usage

```
QState Philo_eating(Philo * const me, QEvt const * const e) {
    QState status_;
    switch (e->sig) {
        .
        .
        case Q_EXIT_SIG: {
            TableEvt const *pe = Q_NEW(TableEvt, DONE_SIG, me->id);
            QACTIVE_PUBLISH(&me->super, pe, me); // <===
            status_ = Q_RET_HANDLED;
            break;
        }
        .
        .
    }
    return status_;
}
```

#### 14.1.2.12 QActive\_getQueueMin()

```
uint_fast16_t QActive_getQueueMin (
    uint_fast8_t const prio) [static]
```

This function returns the minimum of free entries of the given event queue.

**Details**

Queries the minimum of free ever present in the given event queue of an active object with priority `prio`, since the active object was started.

**Note**

This function is available only when the native `QF` event queue implementation is used. Requesting the queue minimum of an unused priority level raises an assertion in the `QF`. (A priority level becomes used in `QF` after the call to the `QActive_register_()` function.)

**Parameters**

<code>in</code>	<code>prio</code>	Priority of the active object, whose queue is queried
-----------------	-------------------	---

**Returns**

The minimum of free ever present in the given event queue of an active object with priority `prio`, since the active object was started.

**Backward Traceability**

- `SDS_QP_QF: QF Active Object Framework`

**14.1.2.13 `QActive_subscribe()`**

```
void QActive_subscribe (
    QActive const *const me,
    enum_t const sig) [protected]
```

Subscribes for delivery of signal `sig` to the active object.

**Details**

This function is part of the Publish-Subscribe event delivery mechanism available in `QF`. Subscribing to an event means that the framework will start posting all published events with a given signal `sig` to the event queue of the active object.

**Parameters**

<code>in,out</code>	<code>me</code>	current instance pointer (see <code>SAS_QP_OO</code> )
<code>in</code>	<code>sig</code>	event signal to subscribe

**Precondition** `qf_ps:300`

- signal must be in range of subscribe signals
- subscriber AO priority must be in range
- the AO must be registered (started)

**Precondition** `qf_ps:302`

- check the integrity of the subscriber set (QP FuSa Subsystem)

## Backward Traceability

- **QActive**: Active object class (based on the *QHsm* implementation strategy)

## Usage

The following example shows how the Table active object subscribes to three signals in the initial transition:

```
QState Table_initial(Table * const me, QEvt const * const e) {
    Q_UNUSED_PAR(e);

    // subscribe to event signals...
    QActive_subscribe(&me->super, (enum_t)HUNGRY_SIG);
    QActive_subscribe(&me->super, (enum_t)DONE_SIG);
    QActive_subscribe(&me->super, (enum_t)TERMINATE_SIG);

    for (uint8_t n = 0U; n < N; ++n) {
        me->fork[n] = FREE;
        me->isHungry[n] = false;
    }
    return Q_TRAN(&Table_serving);
}
```

### 14.1.2.14 QActive\_unsubscribe()

```
void QActive_unsubscribe (
    QActive const *const me,
    enum_t const sig) [protected]
```

Unsubscribes from the delivery of signal *sig* to the active object.

#### Details

This function is part of the Publish-Subscribe event delivery mechanism available in **QF**. Unsubscribing from an event means that the framework will stop posting published events with a given signal *sig* to the event queue of the active object.

#### Parameters

in, out	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>sig</i>	event signal to unsubscribe

**Precondition** *qf\_ps*: 400

- signal must be in range of subscribe signals
- subscriber AO priority must be in range
- the AO must be registered (started)

**Precondition** *qf\_ps*: 402

- check the integrity of the subscriber set (QP FuSa Subsystem)

#### Note

Due to the latency of event queues, an active object should NOT assume that a given signal *sig* will never be dispatched to the state machine of the active object after un-subscribing from that signal. The event might be already in the queue, or just about to be posted and the un-subscribe operation will not flush such events.

Un-subscribing from a signal that has never been subscribed in the first place is considered an error and **QF** will raise an assertion.

## Backward Traceability

- **QActive**: Active object class (based on the *QHsm* implementation strategy)

**14.1.2.15 QActive\_unsubscribeAll()**

```
void QActive_unsubscribeAll (
    QActive const *const me) [protected]
```

Unsubscribes from the delivery of all signals to the active object.

**Details**

This function is part of the Publish-Subscribe event delivery mechanism available in [QF](#). Un-subscribing from all events means that the framework will stop posting any published events to the event queue of the active object.

**Parameters**

in, out	me	current instance pointer (see <a href="#">SAS_QP_OO</a> )
---------	----	---

**Precondition** qf\_ps: 500

- subscriber AO priority must be in range
- the AO must be registered (started)

**Note**

Due to the latency of event queues, an active object should NOT assume that no events will ever be dispatched to the state machine of the active object after un-subscribing from all events. The events might be already in the queue, or just about to be posted and the un-subscribe operation will not flush such events. Also, the alternative event-delivery mechanisms, such as direct event posting or time events, can be still delivered to the event queue of the active object.

**Backward Traceability**

- [QActive](#): Active object class (based on the [QHsm](#) implementation strategy)

**14.1.2.16 QActive\_defer()**

```
bool QActive_defer (
    QActive const *const me,
    struct QEQueue *const eq,
    QEvt const *const e) [protected]
```

Defer an event to a given separate event queue.

**Details**

This function is part of the event deferral support. An active object uses this function to defer an event *e* to the QF-supported native event queue *eq*. [QF](#) correctly accounts for another outstanding reference to the event and will not recycle the event at the end of the RTC step. Later, the active object might recall one event at a time from the event queue.

**Remarks**

An active object can use multiple event queues to defer events of different kinds.

**Parameters**

in, out	me	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	eq	pointer to a "raw" thread-safe queue to recall an event from.
in	e	pointer to the event to be deferred

**Returns**

'true' (success) when the event could be deferred and 'false' (failure) if event deferral failed due to overflowing the queue.

**Backward Traceability**

- [QActive](#): Active object class (based on the [QHsm](#) implementation strategy)

**14.1.2.17 QActive\_recall()**

```
bool QActive_recall (
    QActive *const me,
    struct QEQueue *const eq) [protected]
```

Recall a deferred event from a given event queue.

**Details**

This function is part of the event deferral support. An active object uses this function to recall a deferred event from a given [QF](#) event queue. Recalling an event means that it is removed from the deferred event queue `eq` and posted (LIFO) to the event queue of the active object.

**Parameters**

<code>in, out</code>	<code>me</code>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
<code>in</code>	<code>eq</code>	pointer to a "raw" thread-safe queue to recall an event from.

**Returns**

'true' if an event has been recalled and 'false' if not.

**Note**

An active object can use multiple event queues to defer events of different kinds.

**Backward Traceability**

- [QActive](#)

**14.1.2.18 QActive\_flushDeferred()**

```
uint_fast16_t QActive_flushDeferred (
    QActive const *const me,
    struct QEQueue *const eq,
    uint_fast16_t const num) [protected]
```

Flush the specified number of events from the deferred queue `eq`

**Details**

This function is part of the event deferral support. An active object can use this function to flush a given [QF](#) event queue. The function makes sure that the events are not leaked.

**Parameters**

<code>in, out</code>	<code>me</code>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
<code>in</code>	<code>eq</code>	pointer to a "raw" thread-safe queue to flush.
<code>in</code>	<code>num</code>	number of events to flush (note can be a big number, like 0xFFFFU to flush all events)

**Returns**

the number of events actually flushed from the queue.

**Backward Traceability**

- [QActive](#)

**14.1.2.19 QActive\_evtLoop\_()**

```
void QActive_evtLoop_ (
```

```
    QActive *const me) [private]
```

Event loop thread routine for executing an active object `act` (defined some in QP ports)

**Backward Traceability**

- [QActive](#): Active object class (based on the [QHsm](#) implementation strategy)

**14.1.2.20 QActive\_postFIFO\_()**

```
static void QActive_postFIFO_ (
```

```
    QActive *const me,
```

```
    QEvt const *const e,
```

```
    void const *const sender) [static], [private]
```

**14.1.3 Member Data Documentation****14.1.3.1 super**

[QAsm](#) `super` [protected]

**14.1.3.2 prio**

`uint8_t prio` [protected]

QF-priority [1..[QF\\_MAX\\_ACTIVE](#)] of this AO.

**See also**

- [QPrioSpec](#)

**Backward Traceability**

- [QActive](#): Active object class (based on the [QHsm](#) implementation strategy)

**14.1.3.3 pthre**

`uint8_t pthre` [protected]

Preemption-threshold [1..[QF\\_MAX\\_ACTIVE](#)] of this AO.

**See also**

- [QPrioSpec](#)

**Backward Traceability**

- [QActive](#): Active object class (based on the [QHsm](#) implementation strategy)

#### 14.1.3.4 thread

`QACTIVE_THREAD_TYPE` `thread` [protected]

Port-dependent representation of the thread of the active object.

##### Details

This data might be used in various ways, depending on the `QF` port. In some ports `me->thread` is used to store the thread handle. In other ports `me->thread` can be a pointer to the Thread-Local-Storage (TLS).

##### Backward Traceability

- `QActive`: Active object class (based on the `QHsm` implementation strategy)

#### 14.1.3.5 osObject

`QACTIVE_OS_OBJ_TYPE` `osObject` [protected]

Port-dependent per-thread object.

##### Details

This data might be used in various ways, depending on the `QF` port. In some ports `me->osObject` is used to block the calling thread when the native `QF` queue is empty. In other `QF` ports the OS-dependent object might be used differently.

#### 14.1.3.6 eQueue

`QACTIVE_EQUEUE_TYPE` `eQueue` [protected]

Port-dependent event-queue type (often `QEQueue`)

##### Details

The type of the queue depends on the underlying operating system or a kernel. Many kernels support "message queues" that can be adapted to deliver `QF` events to the active object. Alternatively, `QF` provides a native event queue implementation that can be used as well.

##### Note

The native `QF` event queue is configured by defining the macro `QACTIVE_EQUEUE_TYPE` as `QEQueue`.

##### Backward Traceability

- `QActive`: Active object class (based on the `QHsm` implementation strategy)

#### 14.1.3.7 QActive\_registry\_

`QActive *` `QActive_registry_` [private]

Static (one per-class) array of registered active objects.

#### 14.1.3.8 QActive\_subscrList\_

`QSubscrList*` `QActive_subscrList_` [private]

Static (one per-class) pointer to all subscriber AOs for a given event signal.

##### Backward Traceability

- `QActive`: Active object class (based on the `QHsm` implementation strategy)

### 14.1.3.9 QActive\_maxPubSignal\_

`enum_t QActive_maxPubSignal_ [private]`

Static (one per-class) maximum published signal (the size of the `subscrList_` array)

The documentation for this class was generated from the following files:

- [qp.h](#)
- [qf\\_act.c](#)
- [qf\\_actq.c](#)
- [qf\\_defer.c](#)
- [qf\\_ps.c](#)
- [qf\\_qact.c](#)
- [qv.c](#)
- [qk.c](#)
- [qp.dox](#)

## 14.2 QActiveDummy Class Reference

Dummy Active Object for testing.

### 14.2.1 Detailed Description

Dummy Active Object for testing.

#### Details

`QActiveDummy` is a test double for the role of collaborating Active Objects in QUTest unit testing.

The documentation for this class was generated from the following file:

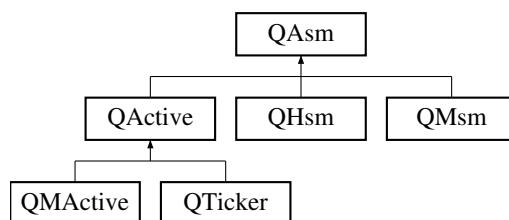
- [qs.dox](#)

## 14.3 QAsm Class Reference

Abstract State Machine class (state machine interface)

`#include "qp.h"`

Inheritance diagram for QAsm:



### Static Protected Member Functions

- static void `QAsm_ctor (QAsm *const me)`

*Constructor of the `QAsm` base class.*

## Protected Attributes

- struct [QAsmVtable](#) const \* [vptr](#)  
*Virtual pointer inherited by all [QAsm](#) subclasses (see also [SAS\\_QP\\_OO](#))*
- union [QAsmAttr](#) [state](#)  
*Current state (pointer to the current state-handler function)*
- union [QAsmAttr](#) [temp](#)  
*Temporary storage for target/act-table etc.*

### 14.3.1 Detailed Description

Abstract State Machine class (state machine interface)

#### Backward Traceability

- [SDS\\_QP\\_QEP](#): *QEP Event Processor*
- [SDS\\_QP\\_QAsm](#): *QAsm Abstract state machine class.*

### 14.3.2 Member Function Documentation

#### 14.3.2.1 [QAsm\\_ctor\(\)](#)

```
static void QAsm_ctor (
    QAsm *const me) [inline], [static], [protected]
```

Constructor of the [QAsm](#) base class.

#### Details

The constructor initializes the [QAsm::vptr](#) and clears the internal attributes. The constructor is "protected" because it is only intended to be invoked from the subclasses of the abstract base class [QAsm](#).

#### Parameters

in, out	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
---------	-----------	---

#### Backward Traceability

- [QAsm](#): *Abstract State Machine class (state machine interface)*

### 14.3.3 Member Data Documentation

#### 14.3.3.1 [vptr](#)

```
struct QAsmVtable const* vptr [protected]
```

Virtual pointer inherited by all [QAsm](#) subclasses (see also [SAS\\_QP\\_OO](#))

#### Backward Traceability

- [QAsm](#): *Abstract State Machine class (state machine interface)*

#### 14.3.3.2 [state](#)

```
union QAsmAttr state [protected]
```

Current state (pointer to the current state-handler function)

### 14.3.3.3 temp

```
union QAsmAttr temp [protected]
Temporary storage for target/act-table etc.
```

#### Backward Traceability

- [QAsm](#): Abstract State Machine class (state machine interface)

#### Details

The `temp` data member is used for passing information from the QP Application to the "event processor" of QP. The `temp` member is also used as the Duplicate Inverse Storage of the `state` attribute in between transitions (part of QP Functional Safety (FuSa) Subsystem).

The documentation for this class was generated from the following files:

- [qp.h](#)
- [qp.dox](#)

## 14.4 QAsmAttr Union Reference

Attribute of for the [QAsm](#) class (Abstract State Machine)

```
#include "qp.h"
```

#### Private Attributes

- [QStateHandler](#) `fun`
- [QActionHandler](#) `act`
- [QXThreadHandler](#) `thr`
- [QMTranActTable](#) `const * tatbl`
- struct [QMState](#) `const * obj`

### 14.4.1 Detailed Description

Attribute of for the [QAsm](#) class (Abstract State Machine)

#### Details

This union represents possible values stored in the 'state' and 'temp' attributes of the [QAsm](#) class.

#### Backward Traceability

- DVP\_QP\_MC4\_R19\_02: Rule 19.2(Advisory): The `union` keyword should not be used

### 14.4.2 Member Data Documentation

#### 14.4.2.1 fun

```
QStateHandler fun [private]
```

#### 14.4.2.2 act

```
QActionHandler act [private]
```

#### 14.4.2.3 thr

```
QXThreadHandler thr [private]
```

#### 14.4.2.4 tatbl

```
QMTranActTable const* tatbl [private]
```

#### 14.4.2.5 obj

```
struct QMState const* obj [private]
```

The documentation for this union was generated from the following file:

- qp.h

## 14.5 QAsmVtable Struct Reference

Virtual table for the [QAsm](#) class.

```
#include "qp.h"
```

### Public Attributes

- void(\* [init](#) )(QAsm \*const me, void const \*const e, uint\_fast8\_t const qsId)  
*Virtual function to take the top-most initial transition in the state machine.*
- void(\* [dispatch](#) )(QAsm \*const me, QEvt const \*const e, uint\_fast8\_t const qsId)  
*Virtual function to dispatch an event to the state machine.*
- bool(\* [isIn](#) )(QAsm \*const me, QStateHandler const stateHndl)  
*Virtual function to check whether the state machine is in a given state.*
- QStateHandler(\* [getStateHandler](#) )(QAsm \*const me)  
*Virtual function to get the current state handler of the state machine.*

### 14.5.1 Detailed Description

Virtual table for the [QAsm](#) class.

#### Backward Traceability

- [SRS\\_QP\\_SM\\_10](#): *QP Framework shall support multiple and interchangeable State Machine Implementation Strategies*

## 14.5.2 Member Data Documentation

### 14.5.2.1 init

```
void(* QAsmVtable::init) (QAsm *const me, void const *const e, uint_fast8_t const qsId)
```

Virtual function to take the top-most initial transition in the state machine.

### 14.5.2.2 dispatch

```
void(* QAsmVtable::dispatch) (QAsm *const me, QEvt const *const e, uint_fast8_t const qsId)
```

Virtual function to dispatch an event to the state machine.

### 14.5.2.3 isIn

```
bool(* QAsmVtable::isIn) (QAsm *const me, QStateHandler const stateHndl)
```

Virtual function to check whether the state machine is in a given state.

#### 14.5.2.4 getStateHandler

`QStateHandler(* QAsmVtable::getStateHandler) (QAsm *const me)`

Virtual function to get the current state handler of the state machine.

The documentation for this struct was generated from the following files:

- `qp.h`
- `qp.dox`

## 14.6 QEQueue Class Reference

Native `QF` Event Queue.

```
#include "qqueue.h"
```

### Public Member Functions

- void `QEQueue_init` (`QEQueue` \*const me, struct `QEvt` const \*\*const qSto, `uint_fast16_t` const qLen)  
*Initialize the native `QF` event queue.*
- bool `QEQueue_post` (`QEQueue` \*const me, struct `QEvt` const \*const e, `uint_fast16_t` const margin, `uint_fast8_t` const qslid)  
*Post an event to the "raw" thread-safe event queue (FIFO)*
- void `QEQueue_postLIFO` (`QEQueue` \*const me, struct `QEvt` const \*const e, `uint_fast8_t` const qslid)  
*Post an event to the "raw" thread-safe event queue (LIFO)*
- struct `QEvt` const \* `QEQueue_get` (`QEQueue` \*const me, `uint_fast8_t` const qslid)  
*Obtain an event from the "raw" thread-safe queue.*

### Static Public Member Functions

- static `QEQueueCtr` `QEQueue_getNFree` (`QEQueue` const \*const me)  
*Obtain the number of free entries still available in the queue.*
- static `QEQueueCtr` `QEQueue_getNMin` (`QEQueue` const \*const me)  
*Obtain the minimum number of free entries ever in the queue (a.k.a. "low-watermark")*
- static bool `QEQueue_isEmpty` (`QEQueue` const \*const me)  
*Find out if the queue is empty.*

### Private Attributes

- struct `QEvt` const \*volatile `frontEvt`  
*Pointer to event at the front of the queue.*
- struct `QEvt` const \*\* `ring`  
*Pointer to the start of the ring buffer.*
- `QEQueueCtr` `end`  
*Offset of the end of the ring buffer from the start of the buffer.*
- `QEQueueCtr` volatile `head`  
*Offset to where next event will be inserted into the buffer.*
- `QEQueueCtr` volatile `tail`  
*Offset of where next event will be extracted from the buffer.*
- `QEQueueCtr` volatile `nFree`  
*Number of free events in the ring buffer.*
- `QEQueueCtr` `nMin`  
*Minimum number of free events ever in the ring buffer.*

### 14.6.1 Detailed Description

Native QF Event Queue.

#### Details

This class describes the native QF event queue, which can be used as the event queue for active objects, or as a simple "raw" event queue for thread-safe event passing among non-framework entities, such as ISRs, device drivers, or other third-party components.

The native QF event queue is configured by defining the macro `QACTIVE_EQUEUE_TYPE` as `QEQueue` in the specific QF port header file.

The `QEQueue` structure contains only data members for managing an event queue, but does not contain the storage for the queue buffer, which must be provided externally during the queue initialization.

The event queue can store only event pointers, not the whole events. The internal implementation uses the standard ring-buffer plus one external location that optimizes the queue operation for the most frequent case of empty queue. The `QEQueue` structure is used with two sets of functions. One set is for the active object event queue, which might need to block the active object task when the event queue is empty and might need to unblock it when events are posted to the queue. The interface for the native active object event queue consists of the following functions: `QActive_post()`, `QActive_postLIFO()`, and `QActive_get()`.

The other set of functions, uses `QEQueue` as a simple "raw" event queue to pass events between entities other than active objects, such as ISRs. The "raw" event queue is not capable of blocking on the `get()` operation, but is still thread-safe because it uses QF critical section to protect its integrity. The interface for the "raw" thread-safe queue consists of the following functions: `QEQueue_post()`, `QEQueue_postLIFO()`, and `QEQueue_get()`. Additionally the function `QEQueue_init()` is used to initialize the queue.

Most event queue operations (both the active object queues and the "raw" queues) internally use the QF critical section. You should be careful not to invoke those operations from other critical sections when nesting of critical sections is not supported.

#### See also

`QEQueue` for the description of the data members

### 14.6.2 Member Function Documentation

#### 14.6.2.1 QEQueue\_init()

```
void QEQueue_init (
    QEQueue *const me,
    struct QEvt const **const qSto,
    uint_fast16_t const qLen)
```

Initialize the native QF event queue.

#### Details

Initialize the event queue by giving it the storage for the ring buffer.

#### Parameters

in,out	<code>me</code>	current instance pointer (see <code>SAS_QP_OO</code> )
in	<code>qSto</code>	an array of <code>QEvtPtr</code> to serve as the ring buffer for the event queue
in	<code>qLen</code>	the length of the <code>qSto</code> buffer (in <code>QEvt</code> pointers)

#### Note

The actual capacity of the queue is `qLen + 1`, because of the extra location forntEvt.

This function is also used to initialize the event queues of active objects in the built-in `QV` and `QK` kernels, as well as other QP ports to OSes/RTOSes that do provide a suitable message queue.

### 14.6.2.2 QEQueue\_post()

```
bool QEQueue_post (
    QEQueue *const me,
    struct QEvt const *const e,
    uint_fast16_t const margin,
    uint_fast8_t const qsId)
```

Post an event to the "raw" thread-safe event queue (FIFO)

#### Details

Post an event to the "raw" thread-safe event queue using the First-In-First-Out (FIFO) order.

#### Parameters

in, out	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>e</i>	pointer to the event to be posted to the queue
in	<i>margin</i>	number of required free slots in the queue after posting the event. The special value <a href="#">QF_NO_MARGIN</a> means that this function will assert if posting
in	<i>qsId</i>	QS-id of this state machine (for <a href="#">QS</a> local filter)

#### Returns

'true' (success) when the posting succeeded with the provided margin and 'false' (failure) when the posting fails.

#### Precondition [qf\\_qeq:200](#)

- event must be valid

#### Note

The [QF\\_NO\\_MARGIN](#) value of the *margin* parameter is special and denotes situation when the post() operation is assumed to succeed (event delivery guarantee). An assertion fires, when the event cannot be delivered in this case.

This function can be called from any task context or ISR context.

#### See also

- [QEQueue\\_postLIFO\(\)](#)
- [QEQueue\\_get\(\)](#)

### 14.6.2.3 QEQueue\_postLIFO()

```
void QEQueue_postLIFO (
    QEQueue *const me,
    struct QEvt const *const e,
    uint_fast8_t const qsId)
```

Post an event to the "raw" thread-safe event queue (LIFO)

#### Details

Post an event to the "raw" thread-safe event queue using the Last-In-First-Out (LIFO) order.

#### Parameters

in, out	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>e</i>	pointer to the event to be posted to the queue
in	<i>qsId</i>	QS-id of this state machine (for <a href="#">QS</a> local filter)

**Precondition** `qf_qeq <= 300`

- the queue must be able to accept the event (cannot overflow)

#### Attention

The LIFO policy should be used only with great **caution**, because it alters the order of events in the queue.

#### Note

This function can be called from any task context or ISR context.

this function is used for the "raw" thread-safe queues and **not** for the queues of active objects.

#### See also

- [QEQueue\\_post\(\)](#)
- [QEQueue\\_get\(\)](#)
- [QActive\\_defer\(\)](#)

#### 14.6.2.4 QEQueue\_get()

```
struct QEvt const * QEQueue_get (
    QEQueue *const me,
    uint_fast8_t const qsId)
```

Obtain an event from the "raw" thread-safe queue.

#### Details

Retrieves an event from the front of the "raw" thread-safe queue and returns a pointer to this event to the caller.

#### Parameters

in, out	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>qsId</i>	QS-id of this state machine (for <a href="#">QS</a> local filter)

#### Returns

pointer to event at the front of the queue, if the queue is not empty and NULL if the queue is empty.

#### Note

This function is used for the "raw" thread-safe queues and **not** for the queues of active objects.

#### See also

- [QEQueue\\_post\(\)](#)
- [QEQueue\\_postLIFO\(\)](#)
- [QActive\\_recall\(\)](#)

#### 14.6.2.5 QEQueue\_getNFree()

```
static QEQueueCtr QEQueue_getNFree (
    QEQueue const *const me) [inline], [static]
```

Obtain the number of free entries still available in the queue.

##### Details

This operation needs to be used with caution because the number of free entries can change unexpectedly. The main intent for using this operation is in conjunction with event deferral. In this case the queue is accessed only from a single thread (by a single AO), so the number of free entries cannot change unexpectedly.

##### Parameters

in	me	current instance pointer (see <a href="#">SAS_QP_OO</a> )
----	----	---

##### Returns

the current number of free slots in the queue.

#### 14.6.2.6 QEQueue\_getNMin()

```
static QEQueueCtr QEQueue_getNMin (
    QEQueue const *const me) [inline], [static]
```

Obtain the minimum number of free entries ever in the queue (a.k.a. "low-watermark")

##### Details

This operation needs to be used with caution because the "low-watermark" can change unexpectedly. The main intent for using this operation is to get an idea of queue usage to size the queue adequately.

##### Parameters

in	me	current instance pointer (see <a href="#">SAS_QP_OO</a> )
----	----	---

##### Returns

the minimum number of free entries ever in the queue since init.

#### 14.6.2.7 QEQueue\_isEmpty()

```
static bool QEQueue_isEmpty (
    QEQueue const *const me) [inline], [static]
```

Find out if the queue is empty.

##### Details

This operation needs to be used with caution because the queue status can change unexpectedly. The main intent for using this operation is in conjunction with event deferral. In this case the queue is accessed only from a single thread (by a single AO), so no other entity can post events to the queue.

##### Parameters

in, out	me	current instance pointer (see <a href="#">SAS_QP_OO</a> )
---------	----	---

##### Returns

'true' if the queue is current empty and 'false' otherwise.

### 14.6.3 Member Data Documentation

#### 14.6.3.1 frontEvt

`struct QEvt const* volatile frontEvt [private]`

Pointer to event at the front of the queue.

##### Details

All incoming and outgoing events pass through the frontEvt location. When the queue is empty (which is most of the time), the extra frontEvt location allows to bypass the ring buffer altogether, greatly optimizing the performance of the queue. Only bursts of events engage the ring buffer.

The additional role of this attribute is to indicate the empty status of the queue. The queue is empty when frontEvt is NULL.

#### 14.6.3.2 ring

`struct QEvt const* * ring [private]`

Pointer to the start of the ring buffer.

#### 14.6.3.3 end

`QEQueueCtr end [private]`

Offset of the end of the ring buffer from the start of the buffer.

#### 14.6.3.4 head

`QEQueueCtr volatile head [private]`

Offset to where next event will be inserted into the buffer.

#### 14.6.3.5 tail

`QEQueueCtr volatile tail [private]`

Offset of where next event will be extracted from the buffer.

#### 14.6.3.6 nFree

`QEQueueCtr volatile nFree [private]`

Number of free events in the ring buffer.

#### 14.6.3.7 nMin

`QEQueueCtr nMin [private]`

Minimum number of free events ever in the ring buffer.

##### Details

This attribute remembers the low-watermark of the ring buffer, which provides a valuable information for sizing event queues.

##### See also

- [QF\\_getQueueMargin\(\)](#)

The documentation for this class was generated from the following files:

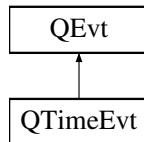
- [queue.h](#)
- [qf\\_qeq.c](#)
- [queue.dox](#)

## 14.7 QEvt Class Reference

Event class.

```
#include "qp.h"
```

Inheritance diagram for QEvt:



### Static Public Member Functions

- static void [QEvt\\_ctor](#) (QEvt \*const me, enum\_t const sig)
- static QEvt \* [QEvt\\_init](#) (QEvt \*const me, uint8\_t const dummy)

*Event without parameters initialization.*

### Public Attributes

- [QSignal sig](#)

*Signal of the event (see [Event Signal](#))*

### Static Private Member Functions

- static void [QEvt\\_refCtr\\_inc\\_](#) (QEvt const \*const me)
- static void [QEvt\\_refCtr\\_dec\\_](#) (QEvt const \*const me)

### Private Attributes

- uint8\_t [poolNum\\_](#)

*Event pool number of this event.*

- uint8\_t volatile [refCtr\\_](#)

*Event reference counter.*

### 14.7.1 Detailed Description

Event class.

#### Details

Class [QEvt](#) represents both immutable and mutable [QP events](#). It can be instantiated directly (concrete class), in which case it represents events *without parameters*. QP/C Applications can also inherit and extend QP::QEvt to add custom event parameters (see also [SDS\\_QP\\_QEvt](#)).

#### Attention

Event parameters must be included in the event instance *directly* (as opposed to being referenced by pointers or references). Therefore, the [QEvt](#) subclasses are restricted to have both "standard layout" and be "trivially copyable". In QP/C this means that [QEvt](#) subclasses should:

- contain **NO pointers** (including NO virtual pointer); The only exception is passing an *opaque* pointer to Active Object intended as the recipient of future events.
- contain NO virtual functions (consequence of no virtual pointer);

- contain NO access specifiers (protected, and private);
- contain NO non-trivial copy or move constructor;
- contain NO non-trivial copy or move assignment operator;

#### Backward Traceability

- `SDS_QP_QEvt`: *QEvt event class.*

#### Forward Traceability

- `QEvt::sig`: *Signal of the event (see Event Signal)*
- `QEvt::refCtr_`: *Event reference counter.*
- `QEvt::poolNum_`: *Event pool number of this event*
- `QEvt::sig_dis`: *Duplicate Inverse Storage for QEvt::sig*
- `QEvt::poolNum_dis`: *Duplicate Inverse Storage for QEvt::poolNum\_*
- `QEvt::refCtr_dis`: *Duplicate Inverse Storage for QEvt::refCtr\_*
- `QEvt::QEvt_init()`: *Event without parameters initialization*

#### Usage

The following example illustrates how to add **event parameter(s)** by inheriting the `QEvt` class. Please note that the `QEvt` member `super` is defined as the FIRST member of the derived class:

```
typedef struct {
    QEvt super; // <== inherit QEvt
    // event parameters follow...
    uint8_t keyId; // ID of the key pressed
} KeypressEvt;
```

The following examples illustrate recommended ways to instantiate event objects based on the **RAll** principle (Resource Acquisition Is Initialization). Please see also `QEVT_INITIALIZER()`.

```
// immutable event without parameters
static QEvt const myEvt = QEVT_INITIALIZER(MY_SIG);
// post, publish, or dispatch the immutable event...

// immutable event with parameters
static KeypressEvt const keyEvt = {
    QEVT_INITIALIZER(KEYPRESS_SIG),
    .keyId = 12U
}
// post, publish, or dispatch the immutable event...

// mutable event without parameters (e.g., automatic object inside a function)
QEvt myEvt = QEVT_INITIALIZER(sig); // sig is a variable
// dispatch the event (asynchronous posting or publishing probably WRONG!)

// mutable event with parameters (e.g., automatic object inside a function)
KeypressEvt keyEvt = { // sig and keyId are variables
    QEVT_INITIALIZER(sig),
    .keyId = keyId
};
// dispatch the event (asynchronous posting or publishing probably WRONG!)
```

#### Note

Please see macros `Q_NEW()` or `Q_NEW_X()` for code examples of allocating dynamic **mutable** events.

### 14.7.2 Member Function Documentation

#### 14.7.2.1 QEvt\_ctor()

```
static void QEvt_ctor (
    QEvt *const me,
    enum_t const sig) [inline], [static]
```

### 14.7.2.2 QEvt\_init()

```
static QEvt * QEvt_init (
    QEvt *const me,
    uint8_t const dummy) [inline], [static]
```

Event without parameters initialization.

#### Details

The main purpose of the [QEvt](#) initialization is to support dynamic allocation of events without parameters with the macros [Q\\_NEW\(\)](#) or [Q\\_NEW\\_X\(\)](#).

#### Remarks

The [QEvt\\_init\(\)](#) can be also used to initialize event instances (or the portion inherited from [QEvt](#) in events with parameters), but this is not recommended. Instead, the recommended ways of instantiating event objects use the **RAll** principle (Resource Acquisition Is Initialization) and are based on the [QEVT\\_INITIALIZER\(\)](#) macro, as illustrated in the [QEvt](#) class.

#### Parameters

in, out	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>dummy</i>	shall be set to <a href="#">QEVT_DYNAMIC</a> for dynamic events

#### Returns

Pointer to the initialized event (the *me* pointer, see [SAS\\_QP\\_OO](#)).

#### Backward Traceability

- [SRS\\_QP\\_EVT\\_40](#): *QP Framework may be compile-time configurable to allow customized constructor and other custom operations on event instances*
- [QEvt](#): *Event class*

#### Usage

The following example illustrates the use of the [QEvt\\_init\(\)](#) directly (not recommended because not based on **RAll**):

```
// mutable event without parameters (type QEvt)
QEvt myEvt; // <== event instance (e.g., on the stack)
QEvt_init(&myEvt, MY_SIG); // <== initialize the event (not recommended)
// dispatch the event (asynchronous posting or publishing probably WRONG!)

// mutable event with parameters (type KeypressEvt derived from QEvt)
KeypressEvt keypressEvt; // <== event instance (e.g., on the stack)
QEvt_init(&keypressEvt.super, KEYPRESS_SIG); // <== initialize the .super member
keypressEvt.keyCode = 23U; // initialize the event parameter
// dispatch keypressEvt...
```

The following example illustrates the use of the [QEvt\\_init\(\)](#) implicitly (called by the macro [Q\\_NEW\(\)](#)) when the configuration macro [QEVT\\_PAR\\_INIT](#) is defined:

```
// variadic version of Q_NEW() (QEVT_DYNAMIC passed to QEvt_init())
QEvt const *pe = Q_NEW(QEvt, MY_SIG, QEVT_DYNAMIC);
// post or publish the event...
```

#### 14.7.2.3 QEvt\_refCtr\_inc\_()

```
static void QEvt_refCtr_inc_ (
    QEvt const *const me) [inline], [static], [private]
```

Internal function to increment the refCtr of a const event.

##### Details

This function requires "casting `const` away" from the event pointer, which violates MISRA-C:2023 Rule 11.8.  
This function encapsulates this violation.

##### Backward Traceability

- DVR\_QP\_MC4\_R11\_08

#### 14.7.2.4 QEvt\_refCtr\_dec\_()

```
static void QEvt_refCtr_dec_ (
    QEvt const *const me) [inline], [static], [private]
```

Internal function to decrement the refCtr of a const event.

##### Details

This function requires "casting `const` away" from the event pointer, which violates MISRA-C:2023 Rule 11.8.  
This function encapsulates this violation.

##### Backward Traceability

- DVR\_QP\_MC4\_R11\_08

### 14.7.3 Member Data Documentation

#### 14.7.3.1 sig

`QSignal` sig

Signal of the event (see [Event Signal](#))

##### Backward Traceability

- [QEvt: Event class](#)
- [SRS\\_QP\\_EVT\\_20: Each event instance shall contain the event Signal](#)

#### 14.7.3.2 poolNum\_

`uint8_t` poolNum\_ [private]

Event pool number of this event.

##### Details

The 8-bit poolNum\_ member stores the event-pool number. For mutable events, the event-pool number is in the range [1..QF\\_MAX\\_EPOOL](#). For immutable (static) events and for time events (instances [QTimeEvt](#)), the event-pool number is 0.

##### Backward Traceability

- [QEvt: Event class](#)
- [SRS\\_QP\\_EVT\\_31: Event abstraction may contain other data items for internal event management inside QP Framework](#)
- [SRS\\_QP\\_EVT\\_30: QP Framework shall allow Application to create event instances with Parameters defined by the Application](#)

### 14.7.3.3 refCtr\_

`uint8_t volatile refCtr_ [private]`

Event reference counter.

#### Details

For mutable events (events from event pools), the 8-bit member `QEvt::refCtr_` holds the reference count, which must be in the range 0..2\*`QF_MAX_ACTIVE`. For immutable (static) events and for time events (instances `QTimeEvt`), the member `QEvt::refCtr_` is not used for reference counting and instead is used as an indicator of the event origin, which is initialized in the macro `QEVT_INITIALIZER()`.

#### Backward Traceability

- [QEvt: Event class](#)
- [SRS\\_QP\\_EVT\\_31: Event abstraction may contain other data items for internal event management inside QP Framework](#)
- [SRS\\_QP\\_EVT\\_30: QP Framework shall allow Application to create event instances with Parameters defined by the Application](#)

The documentation for this class was generated from the following files:

- [qp.h](#)
- [qp\\_pkg.h](#)
- [qp.dox](#)
- [qp\\_pkg.dox](#)

## 14.8 QF Class Reference

QF Active Object Framework ([QF](#) namespace emulated as a "class" in C)

### Static Public Member Functions

- void [QF\\_init](#) (void)  
*QF initialization.*
- void [QF\\_stop](#) (void)  
*Invoked by the application layer to stop the QF framework and return control to the OS/Kernel (used in some QF ports)*
- [int\\_t QF\\_run](#) (void)  
*Transfers control to QF to run the application.*
- void [QF\\_onStartup](#) (void)  
*Startup QF callback.*
- void [QF\\_onCleanup](#) (void)  
*Cleanup QF callback.*
- void [QF\\_onContextSw](#) ([QActive](#) \*prev, [QActive](#) \*next)  
*QF context switch callback used in built-in kernels (QV/QK/QXK)*
- void [QF\\_poolInit](#) (void \*const poolSto, `uint_fast32_t` const poolSize, `uint_fast16_t` const evtSize)  
*Event pool initialization for dynamic allocation of events.*
- `uint_fast16_t` [QF\\_poolGetMaxBlockSize](#) (void)  
*Obtain the block size of any registered event pools.*
- `uint_fast16_t` [QF\\_getPoolMin](#) (`uint_fast8_t` const poolNum)  
*Obtain the minimum of free entries of the given event pool.*
- void [QF\\_gc](#) ([QEvt](#) const \*const e)  
*Recycle a mutable (mutable) event.*
- void [QF\\_gcFromISR](#) ([QEvt](#) const \*const e)
- static void [QF\\_psInit](#) ([QSubscrList](#) \*const subscrSto, `enum_t` const maxSignal)

## Static Private Member Functions

- `QEvt * QF_newX_(uint_fast16_t const evtSize, uint_fast16_t const margin, enum_t const sig)`  
*Internal QF implementation of creating new mutable (dynamic) event.*
- `QEvt const * QF_newRef_(QEvt const *const e, void const *const evtRef)`  
*Internal QF implementation of creating new event reference.*
- `void QF_deleteRef_(void const *const evtRef)`  
*Internal QF implementation of deleting event reference.*
- `void QF_bzero_(void *const start, uint_fast16_t const len)`

## Static Private Attributes

- `QF_Attr QF_priv_`

### 14.8.1 Detailed Description

QF Active Object Framework (QF namespace emulated as a "class" in C)

### 14.8.2 Member Function Documentation

#### 14.8.2.1 QF\_init()

```
void QF_init (
    void ) [static]
```

QF initialization.

##### Details

Initializes QF and must be called exactly once before any other QF function. Typically, `QF_init()` is called from `main()` even before initializing the Board Support Package (BSP).

##### Note

`QF_init()` clears the internal QF variables, so that the framework can start correctly even if the startup code fails to clear the uninitialized data (as is required by the C Standard).

##### Backward Traceability

- `SDS_QP_QF: QF Active Object Framework`
- `SDS_QA_START: QA Application startup sequence`
- `DVR_QP_MC4_R11_08: Rule 11.8(Required): A cast shall not remove any 'const' or 'volatile' qualification from the type pointer to by a pointer`

#### 14.8.2.2 QF\_stop()

```
void QF_stop (
    void ) [static]
```

Invoked by the application layer to stop the QF framework and return control to the OS/Kernel (used in some QF ports)

##### Details

This function stops the QF application. After calling this function, QF attempts to gracefully stop the application. This graceful shutdown might take some time to complete. The typical use of this function is for terminating the QF application to return back to the operating system or for handling fatal errors that require shutting down (and possibly re-setting) the system.

**Attention**

After calling [QF\\_stop\(\)](#) the application must terminate and cannot continue. In particular, [QF\\_stop\(\)](#) is **not** intended to be followed by a call to [QF\\_init\(\)](#) to "resurrect" the application.

**Backward Traceability**

- [SDS\\_QP\\_QF](#): *QF Active Object Framework*

**14.8.2.3 QF\_run()**

```
int_t QF_run (
    void ) [static]
```

Transfers control to [QF](#) to run the application.

**Details**

[QF\\_run\(\)](#) is typically called from your startup code after you initialize the [QF](#) and start at least one active object with [QActive\\_start\(\)](#).

**Returns**

Typically in embedded systems [QF\\_run\(\)](#) does not return, but in case [QF](#) runs on top of a General-Purpose OS (GPOS), [QF\\_run\(\)](#) returns the error status with value 0 representing success.

**Backward Traceability**

- [SDS\\_QP\\_QF](#): *QF Active Object Framework*
- [SDS\\_QA\\_START](#): *QA Application startup sequence*

**14.8.2.4 QF\_onStartup()**

```
void QF_onStartup (
    void ) [static]
```

Startup [QF](#) callback.

**Details**

The purpose of the [QF\\_onStartup\(\)](#) callback is to configure and enable hardware interrupts. The callback is invoked from [QF\\_run\(\)](#), right before starting the underlying real-time kernel. By that time, the application is considered ready to receive and service interrupts.

This function is application-specific and is not implemented in [QF](#), but rather in the Board Support Package (BSP) for the given application.

**Backward Traceability**

- [SDS\\_QP\\_QF](#): *QF Active Object Framework*
- [SDS\\_QA\\_START](#): *QA Application startup sequence*

**14.8.2.5 QF\_onCleanup()**

```
void QF_onCleanup (
    void ) [static]
```

Cleanup [QF](#) callback.

## Details

`QF_onCleanup()` is called in some `QF` ports before `QF` returns to the underlying real-time kernel or operating system.

This function is strongly platform-specific and is not implemented in the `QF`, but either in the `QF` port or in the Board Support Package (BSP) for the given application. Some `QF` ports might not require implementing `QF_onCleanup()` at all, because many embedded applications don't have anything to exit to.

## Backward Traceability

- `SDS_QP_QF: QF Active Object Framework`

### 14.8.2.6 `QF_onContextSw()`

```
void QF_onContextSw (
    QActive * prev,
    QActive * next) [static]
```

`QF` context switch callback used in built-in kernels (QV/QK/QXK)

## Details

This callback function provides a mechanism to perform additional custom operations when one of the built-in kernels switches context from one thread to another.

## Parameters

in	<code>prev</code>	pointer to the previous thread (active object) ( <code>prev==0</code> means that <code>prev</code> was the idle loop)
in	<code>next</code>	pointer to the next thread (active object) ( <code>next==0</code> means that <code>next</code> is the idle loop)

## Attention

`QF_onContextSw()` is invoked with interrupts **disabled** and must also return with interrupts **disabled**.

## Backward Traceability

- `SDS_QP_QF: QF Active Object Framework`

## Usage

This callback is enabled by defining the macro `QF_ON_CONTEXT_SW`.

```
#ifdef QF_ON_CONTEXT_SW
// NOTE: the context-switch callback is called with interrupts DISABLED
void QF_onContextSw(QActive *prev, QActive *next) {
    if (next != (QActive *)0) {
        MPU->CTRL = 0U; // disable the MPU

        MPU_Region const * const region = (MPU_Region const *)next->thread;
        MPU->RBAR = region[0].RBAR;
        MPU->RASR = region[0].RASR;
        MPU->RBAR = region[1].RBAR;
        MPU->RASR = region[1].RASR;
        MPU->RBAR = region[2].RBAR;
        MPU->RASR = region[2].RASR;

        MPU->CTRL = MPU_CTRL_ENABLE_Msk           // enable the MPU
                  | MPU_CTRL_PRIVDEFENA_Msk; // enable background region
        __ISB();
        __DSB();
    }
}
#endif // QF_ON_CONTEXT_SW
```

### 14.8.2.7 QF\_poolInit()

```
void QF_poolInit (
    void *const poolSto,
    uint_fast32_t const poolSize,
    uint_fast16_t const evtSize) [static]
```

Event pool initialization for dynamic allocation of events.

#### Details

This function initializes one event pool at a time and must be called exactly once for each event pool before the pool can be used.

#### Parameters

in	<i>poolSto</i>	pointer to the storage for the event pool
in	<i>poolSize</i>	size of the storage for the pool in bytes
in	<i>evtSize</i>	the block-size of the pool in bytes, which determines the maximum size of events that can be allocated from the pool.

**Precondition** `qf_dyn >= 200`

- the number of event-pools initialized so far must not exceed the maximum [QF\\_MAX\\_EPOOL](#)

**Precondition** `qf_dyn >= 201`

- except the first event-pool 0, the event-size of the previously initialized event pool must not exceed the next event size.

You might initialize many event pools by making many consecutive calls to the [QF\\_poolInit\(\)](#) function. However, for the simplicity of the internal implementation, you must initialize event pools in the **ascending order** of the event size.

#### Remarks

Many RTOSes provide fixed block-size heaps, a.k.a. memory pools that can be adapted for [QF](#) event-pools. In case the pools from the RTOS are not used, [QF](#) provides a native memory pool implementation ([QMPool](#)). The macro [QF\\_EPOOL\\_TYPE](#) determines the type of event pool used by a particular [QF](#) port.

#### Note

The actual number of events available in the pool might be actually less than  $(\text{poolSize} / \text{evtSize})$  due to the internal alignment of the blocks that the pool might perform. You can always check the capacity of the pool by calling [QF\\_getPoolMin\(\)](#).

The dynamic allocation of events is optional, meaning that you might choose not to use mutable events. In that case calling [QF\\_poolInit\(\)](#) and using up memory for the memory blocks is unnecessary.

#### Backward Traceability

- [SDS\\_QP\\_QF](#): *QF Active Object Framework*
- [SDS\\_QA\\_START](#): *QA Application startup sequence*

### 14.8.2.8 QF\_poolGetMaxBlockSize()

```
uint_fast16_t QF_poolGetMaxBlockSize (
    void) [static]
```

Obtain the block size of any registered event pools.

#### Details

Obtain the block size of any registered event pools

#### 14.8.2.9 QF\_getPoolMin()

```
uint_fast16_t QF_getPoolMin (
    uint_fast8_t const poolNum) [static]
```

Obtain the minimum of free entries of the given event pool.

##### Details

This function obtains the minimum number of free blocks in the given event pool since this pool has been initialized by a call to [QF\\_poolInit\(\)](#).

##### Parameters

in	<i>poolNum</i>	event pool ID in the range 1..max_pool, where max_pool is the number of event pools initialized with the function <a href="#">QF_poolInit()</a> .
----	----------------	---

##### Returns

the minimum number of unused blocks in the given event pool.

##### Precondition

- the poolNum must be in range

##### Backward Traceability

- [SDS\\_QP\\_QF: QF Active Object Framework](#)

#### 14.8.2.10 QF\_newX\_()

```
QEvt * QF_newX_ (
    uint_fast16_t const evtSize,
    uint_fast16_t const margin,
    enum_t const sig) [static], [private]
```

Internal [QF](#) implementation of creating new mutable (dynamic) event.

##### Details

Allocates an event dynamically from one of the [QF](#) event pools.

##### Parameters

in	<i>evtSize</i>	the size (in bytes) of the event to allocate
in	<i>margin</i>	the number of un-allocated events still available in a given event pool after the allocation completes. The special value <a href="#">QF_NO_MARGIN</a> means that this function will assert if allocation fails.
in	<i>sig</i>	the signal to be assigned to the allocated event

##### Returns

Pointer to the newly allocated event. This pointer can be NULL only if margin != [QF\\_NO\\_MARGIN](#) and the event cannot be allocated with the specified margin still available in the given pool.

**Precondition** `qf_dyn >= 300`

- the event size must fit one of the initialized event pools

#### Note

The internal `QF` function `QF_newX_()` raises an assertion when the `margin` parameter is `QF_NO_MARGIN` and allocation of the event turns out to be impossible due to event pool depletion, or incorrect (too big) size of the requested event.

The application code should not call this function directly. The only allowed use is thorough the macros `Q_NEW()` or `Q_NEW_X()`.

#### Backward Traceability

- `SDS_QP_QF: QF Active Object Framework`

### 14.8.2.11 `QF_gc()`

```
void QF_gc (
    QEvt const *const e) [static]
```

Recycle a mutable (mutable) event.

#### Details

This function implements a simple garbage collector for the mutable events. Only mutable events are candidates for recycling. (A mutable event is one that is allocated from an event-pool, which is determined as non-zero `QEvt_getPoolId_(e)`) Next, the function decrements the reference counter of the event (`e->refCtr_`), and recycles the event only if the counter drops to zero (meaning that no more references are outstanding for this event). The mutable event is recycled by returning it to the pool from which it was originally allocated.

#### Parameters

in	e	pointer to the event to recycle
----	---	---------------------------------

#### Note

`QF` invokes the garbage collector at all appropriate contexts, when an event can become garbage (automatic garbage collection), so the application code should have no need to call `QF_gc()` directly. The `QF_gc()` function is exposed only for special cases when your application sends mutable events to the "raw" thread-safe queues (see `QEQueue`). Such queues are processed outside of `QF` and the automatic garbage collection is **NOT** performed for these events. In this case you need to call `QF_gc()` explicitly.

#### Backward Traceability

- `SDS_QP_QF: QF Active Object Framework`
- `DVR_QP_MC4_R11_08: Rule 11.8(Required): A cast shall not remove any 'const' or 'volatile' qualification from the type pointer to by a pointer`

### 14.8.2.12 `QF_newRef_()`

```
QEvt const * QF_newRef_ (
    QEvt const *const e,
    void const *const evtRef) [static], [private]
```

Internal `QF` implementation of creating new event reference.

#### Details

Creates and returns a new reference to the current event `e`

**Parameters**

in	<i>e</i>	pointer to the current event
in	<i>evtRef</i>	the event reference

**Returns**

The newly created reference to the event *e*

**Precondition** *qf\_dyn*: 500

- the event must be from a pool (mutable event)
- the provided event reference must not be already in use

**Note**

The application code should not call this function directly. The only allowed use is thorough the macro [Q\\_NEW\\_REF\(\)](#).

**Backward Traceability**

- [SDS\\_QP\\_QF](#): QF Active Object Framework

**14.8.2.13 QF\_deleteRef\_()**

```
void QF_deleteRef_
    void const *const evtRef) [static], [private]
```

Internal QF implementation of deleting event reference.

**Details**

Deletes an existing reference to the event *e*

**Parameters**

in	<i>evtRef</i>	the event reference
----	---------------	---------------------

**Note**

The application code should not call this function directly. The only allowed use is thorough the macro [Q\\_DELETE\\_REF\(\)](#).

**Backward Traceability**

- [SDS\\_QP\\_QF](#): QF Active Object Framework
- [DVR\\_QP\\_MC4\\_R11\\_05](#): Rule 11.5(Advisory): A conversion should not be performed from pointer to void into pointer to object

**14.8.2.14 QF\_gcFromISR()**

```
void QF_gcFromISR (
    QEvt const *const e) [static]
```

**14.8.2.15 QF\_bzero\_()**

```
void QF_bzero_ (
    void *const start,
    uint_fast16_t const len) [static], [private]
```

**14.8.2.16 QF\_psInit()**

```
static void QF_psInit (
    QSubscrList *const subscrSto,
    enum_t const maxSignal) [inline], [static]
```

**Deprecated****14.8.3 Member Data Documentation****14.8.3.1 QF\_priv\_**

`QF_Attr` `QF_priv_` [static], [private]

The documentation for this class was generated from the following files:

- [qp.dox](#)
- [qp\\_pkg.h](#)
- [qp.h](#)
- [qpc.h](#)
- [qf\\_act.c](#)
- [qf\\_dyn.c](#)
- [qv.c](#)
- [qk.c](#)

**14.9 QF\_Attr Class Reference**

Private attributes of the `QF` framework.

```
#include "qp_pkg.h"
```

**Private Attributes**

- `QF_EPOOL_TYPE_ePool_[QF_MAX_EPOOL]`
- `uint_fast8_t maxPool_`

**14.9.1 Detailed Description**

Private attributes of the `QF` framework.

**14.9.2 Member Data Documentation****14.9.2.1 ePool\_**

`QF_EPOOL_TYPE_ePool_[QF_MAX_EPOOL]` [private]

Array of event pools sized for the maximum allowed number of pools

### 14.9.2.2 maxPool\_

`uint_fast8_t maxPool_ [private]`

Number of event pools managed by the [QF Framework](#)

The documentation for this class was generated from the following files:

- [qp\\_pkg.h](#)
- [qp\\_pkg.dox](#)

## 14.10 QFreeBlock Struct Reference

Structure representing a free block in [QMPool](#).

### 14.10.1 Detailed Description

Structure representing a free block in [QMPool](#).

The documentation for this struct was generated from the following file:

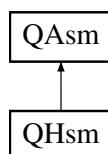
- [qmpool.dox](#)

## 14.11 QHsm Class Reference

Hierarchical State Machine class (QHsm-style state machine implementation strategy)

`#include "qp.h"`

Inheritance diagram for QHsm:



### Public Member Functions

- [QStateHandler QHsm\\_childState \(QHsm \\*const me, QStateHandler const parentHndl\)](#)  
*Obtain the current active child state of a given parent in [QHsm](#).*

### Static Public Member Functions

- static [QStateHandler QHsm\\_state \(QHsm const \\*const me\)](#)  
*Obtain the current active state from a HSM (read only)*

### Protected Member Functions

- void [QHsm\\_ctor \(QHsm \\*const me, QStateHandler const initial\)](#)  
*Constructor of the [QHsm](#) base class.*
- [QState QHsm\\_top \(QHsm const \\*const me, QEvt const \\*const e\)](#)

### Protected Attributes

- [QAsm super](#)

## Protected Attributes inherited from [QAsm](#)

- struct [QAsmVtable](#) const \* [vptr](#)  
*Virtual pointer inherited by all [QAsm](#) subclasses (see also [SAS\\_QP\\_OO](#))*
- union [QAsmAttr](#) [state](#)  
*Current state (pointer to the current state-handler function)*
- union [QAsmAttr](#) [temp](#)  
*Temporary storage for target/act-table etc.*

## Private Member Functions

- void [QHsm\\_init\\_](#) ([QAsm](#) \*const me, void const \*const e, uint\_fast8\_t const qslid)  
*Implementation of the top-most initial transition in [QHsm](#).*
- void [QHsm\\_dispatch\\_](#) ([QAsm](#) \*const me, [QEvt](#) const \*const e, uint\_fast8\_t const qslid)  
*Implementation of dispatching events to a [QHsm](#).*
- bool [QHsm\\_isIn\\_](#) ([QAsm](#) \*const me, [QStateHandler](#) const stateHndl)  
*Check if a given state is part of the current active state configuration.*
- [QStateHandler](#) [QHsm\\_getStateHandler\\_](#) ([QAsm](#) \*const me)  
*Implementation of getting the state handler in a [QHsm](#) subclass.*

## Static Private Member Functions

- static int\_fast8\_t [QHsm\\_tran\\_simple\\_](#) ([QAsm](#) \*const me, [QStateHandler](#) \*const path, uint\_fast8\_t const qslid)
- static int\_fast8\_t [QHsm\\_tran\\_complex\\_](#) ([QAsm](#) \*const me, [QStateHandler](#) \*const path, uint\_fast8\_t const qslid)
- static void [QHsm\\_enter\\_target\\_](#) ([QAsm](#) \*const me, [QStateHandler](#) \*const path, int\_fast8\_t const depth, uint\_fast8\_t const qslid)

## Additional Inherited Members

### Static Protected Member Functions inherited from [QAsm](#)

- static void [QAsm\\_ctor](#) ([QAsm](#) \*const me)  
*Constructor of the [QAsm](#) base class.*

## 14.11.1 Detailed Description

Hierarchical State Machine class (QHsm-style state machine implementation strategy)

### Details

[QHsm](#) represents a Hierarchical State Machine (HSM) with full support for hierarchical nesting of states, entry/exit actions, initial transitions, and transitions to history in any composite state. This class is designed for ease of manual coding of HSMs in C, but it is also supported by the QM modeling tool.

### Note

[QHsm](#) is not intended to be instantiated directly, but rather serves as the abstract base class for derivation of state machines in the QP Application.

### Backward Traceability

- [SRS\\_QP\\_SM\\_00](#): *QP Framework shall provide support for hierarchical state machines both for Active Objects and for passive event-driven objects in the Application*
- [SRS\\_QP\\_SM\\_10](#): *QP Framework shall support multiple and interchangeable State Machine Implementation Strategies*
- [SDS\\_QP\\_QEP](#): *QEP Event Processor*
- [SDS\\_QP\\_QHsm](#): *QHsm State machine class.*

### Usage

The following example illustrates how to derive a state machine class from [QHsm](#). Please note that the [QHsm](#) member `super` is defined as the FIRST member of the derived class.

```
typedef struct {
    QHsm super; // <== inherit QHsm

    double operand1;
    double operand2;
    char display[DISP_WIDTH + 1];
    uint8_t len;
    uint8_t opKey;
} Calc;

void Calc_ctor(void) {
    QHsm_ctor(&me->super, Q_STATE_CAST(&Calc_initial)); // ctor of the superclass
    .
}
```

## 14.11.2 Member Function Documentation

### 14.11.2.1 QHsm\_ctor()

```
void QHsm_ctor (
    QHsm *const me,
    QStateHandler const initial) [protected]
```

Constructor of the [QHsm](#) base class.

#### Details

The constructor initializes the [QHsm::state](#). The constructor is "protected" because it is only intended to be invoked from the subclasses of the abstract base class [QAsm](#).

#### Parameters

in, out	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>initial</i>	initial pseudostate of the instantiated state machine

### Backward Traceability

- [QHsm](#): *Hierarchical State Machine class (QHsm-style state machine implementation strategy)*

### 14.11.2.2 QHsm\_init\_()

```
void QHsm_init_ (
    QAsm *const me,
    void const *const e,
    uint_fast8_t const qsId) [private]
```

Implementation of the top-most initial transition in [QHsm](#).

#### Details

Synchronously executes the top-most initial transition in a state machine.

#### Parameters

in, out	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>e</i>	pointer to an extra parameter (might be NULL)
in	<i>qsId</i>	QS-id of this state machine (for <a href="#">QS</a> local filter)

**Precondition** `qep_msm:200`

- the virtual pointer must be initialized,
- the top-most initial transition must be initialized,
- the initial transition must not be taken yet.

#### Note

This function should be called only via the virtual table (see [QASM\\_INIT\(\)](#)) and should NOT be called directly in the applications.

#### Backward Traceability

- [QHsm](#): Hierarchical State Machine class (*QHsm-style state machine implementation strategy*)

### 14.11.2.3 QHsm\_dispatch\_()

```
void QHsm_dispatch_ (
    QAsm *const me,
    QEvt const *const e,
    uint_fast8_t const qsId) [private]
```

Implementation of dispatching events to a [QHsm](#).

#### Details

Synchronously dispatches an event for processing to a state machine. The processing of an event represents one run-to-completion (RTC) step.

#### Parameters

in, out	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>e</i>	pointer to the event to be dispatched to the MSM
in	<i>qsId</i>	QS-id of this state machine (for <a href="#">QS</a> local filter)

**Precondition** `qep_msm:302`

- current state must be initialized
- check the internal integrity (Software Self-Monitoring (SSM))

#### Note

This function should be called only via the virtual table (see [QASM\\_DISPATCH\(\)](#)) and should NOT be called directly in the applications.

#### Backward Traceability

- [QHsm](#): Hierarchical State Machine class (*QHsm-style state machine implementation strategy*)

#### 14.11.2.4 QHsm\_isIn\_()

```
bool QHsm_isIn_ (
    QAsm *const me,
    QStateHandler const stateHndl) [private]
```

Check if a given state is part of the current active state configuration.

##### Details

Please note that for a HSM, to "be in a state" means also to be in a superstate of the state.

##### Parameters

in	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>state</i>	pointer to the state-handler function to be tested

##### Returns

'true' if the HSM "is in" the *state* and 'false' otherwise

##### Note

This function should be called only via the virtual table (see [QASM\\_IS\\_IN\(\)](#)) and should NOT be called directly in the applications.

##### Attention

This function should be only called only when the state machine is in a "stable state configuration".

##### Precondition *qep\_hsm*: 602

- internal integrity check (Software Self-Monitoring (SSM))

##### Backward Traceability

- [SRS\\_QP\\_SM\\_25](#): All State Machine Implementation Strategies provided by QP Framework might supply a method for checking if a state machine is in a given state
- [QHsm](#): Hierarchical State Machine class (QHsm-style state machine implementation strategy)

#### 14.11.2.5 QHsm\_getStateHandler\_()

```
QStateHandler QHsm_getStateHandler_ (
    QAsm *const me) [private]
```

Implementation of getting the state handler in a [QHsm](#) subclass.

##### Parameters

in, out	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
---------	-----------	---

##### Note

This function is only called internally via the virtual table

**14.11.2.6 QHsm\_top()**

```
QState QHsm_top (
    QHsm const *const me,
    QEvt const *const e) [protected]
```

**14.11.2.7 QHsm\_state()**

```
static QStateHandler QHsm_state (
    QHsm const *const me) [inline], [static]
```

Obtain the current active state from a HSM (read only)

**Parameters**

in	me	current instance pointer (see <a href="#">SAS_QP_OO</a> )
----	----	---

**Returns**

the current active state-handler

**Note**

This function is used for state history (deep history) in the auto-generated code by the QM modeling tool.

**Backward Traceability**

- **QHsm:** Hierarchical State Machine class (QHsm-style state machine implementation strategy)

**14.11.2.8 QHsm\_childState()**

```
QStateHandler QHsm_childState (
    QHsm *const me,
    QStateHandler const parent)
```

Obtain the current active child state of a given parent in [QHsm](#).

**Details**

Finds the child state of the given `parent`, such that this child state is an ancestor of the currently active state. The main purpose of this function is to support \*\*shallow history\* transitions in state machines derived from [QHsm](#).

**Parameters**

in	me	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	parent	pointer to the state-handler

**Returns**

the child of a given parent state-handler, which is an ancestor of the currently active state. For the corner case when the currently active state is the given `parent` state, function returns the `parent` state.

**Postcondition** `qep msm: 890`

- the child must be found

**Note**

This function is used in QM for auto-generating code for state history (shallow history)

**Backward Traceability**

- [QHsm](#): *Hierarchical State Machine class (QHsm-style state machine implementation strategy)*

**14.11.2.9 QHsm\_tran\_simple\_()**

```
static int_fast8_t QHsm_tran_simple_
    (QAsm *const me,
     QStateHandler *const path,
     uint_fast8_t const qsId) [static], [private]
```

**14.11.2.10 QHsm\_tran\_complex\_()**

```
static int_fast8_t QHsm_tran_complex_
    (QAsm *const me,
     QStateHandler *const path,
     uint_fast8_t const qsId) [static], [private]
```

**14.11.2.11 QHsm\_enter\_target\_()**

```
static void QHsm_enter_target_
    (QAsm *const me,
     QStateHandler *const path,
     int_fast8_t const depth,
     uint_fast8_t const qsId) [static], [private]
```

**14.11.3 Member Data Documentation****14.11.3.1 super**

[QAsm](#) super [protected]

The documentation for this class was generated from the following files:

- [qp.h](#)
- [qep\\_hsm.c](#)
- [qp.dox](#)

**14.12 QHsmDummy Class Reference**

Dummy abstract-state machine class for testing.

**14.12.1 Detailed Description**

Dummy abstract-state machine class for testing.

**Details**

[QHsmDummy](#) is a test double for the role of "Orthogonal Components" state machine objects in QUTest unit testing.

The documentation for this class was generated from the following file:

- [qs.dox](#)

## 14.13 QK Class Reference

QK preemptive non-blocking kernel (QK namespace emulated as a "class" in C)  
`#include "qk.h"`

### Static Public Member Functions

- `QSchedStatus QK_schedLock (uint_fast8_t const ceiling)`
- `void QK_schedUnlock (QSchedStatus const prevCeil)`
- `void QK_onIdle (void)`

### Static Private Member Functions

- `uint_fast8_t QK_sched_ (void)`
- `uint_fast8_t QK_sched_act_ (QActive const *const act, uint_fast8_t const pthre_in)`
- `void QK_activate_ (void)`

### Private Attributes

- `QK_Attr QK_priv_`

#### 14.13.1 Detailed Description

QK preemptive non-blocking kernel (QK namespace emulated as a "class" in C)

#### 14.13.2 Member Function Documentation

##### 14.13.2.1 QK\_sched\_()

```
uint_fast8_t QK_sched_ (
    void ) [static], [private]
```

QK scheduler finds the highest-priority AO ready to run

The QK scheduler finds out the priority of the highest-priority AO that (1) has events to process and (2) has priority that is above the current priority.

#### Returns

The QF-priority of the next active object to activate, or zero if no activation of AO is needed.

#### Precondition qk : 400

- check the internal integrity (duplicate inverse storage)

#### Attention

`QK_sched_()` must be always called with interrupts **disabled** and returns with interrupts **disabled**.

##### 14.13.2.2 QK\_sched\_act\_()

```
uint_fast8_t QK_sched_act_ (
    QActive const *const act,
    uint_fast8_t const pthre_in) [static], [private]
```

### 14.13.2.3 QK\_activate\_()

```
void QK_activate_
    void) [static], [private]
```

**QK** activator activates the next active object. The activated AO preempts the currently executing AOs.

**QK\_activate\_()** activates ready-to run AOs that are above the initial preemption-threshold.

**Precondition** qk : 500

- QK\_attr\_.actPrio and QK\_attr\_.nextPrio must be in range

#### Attention

**QK\_activate\_()** must be always called with interrupts **disabled** and returns with interrupts **disabled**.

### 14.13.2.4 QK\_schedLock()

```
QSchedStatus QK_schedLock (
    uint_fast8_t const ceiling) [static]
```

**QK** selective scheduler lock

This function locks the **QK** scheduler to the specified ceiling.

#### Parameters

in	ceiling	preemption ceiling to which the <b>QK</b> scheduler needs to be locked
----	---------	--

#### Returns

The previous **QK** Scheduler lock status, which is to be used to unlock the scheduler by restoring its previous lock status in **QK\_schedUnlock()**.

**Precondition** qk : 100

- The **QK** scheduler lock cannot be called from an ISR

#### Note

**QK\_schedLock()** must be always followed by the corresponding **QK\_schedUnlock()**.

#### See also

[QK\\_schedUnlock\(\)](#)

#### Usage

The following example shows how to lock and unlock the **QK** scheduler:

```
uint32_t BSP_random(void) {
    uint32_rnd;

    QSchedStatus lockStat = QK_schedLock(N_PHILO); // <== N_PHILO ceiling
    . . . // access/manipulate the shared random seed resource
    QK_schedUnlock(lockStat); // <== unlock

    return rnd;
}
```

### 14.13.2.5 QK\_schedUnlock()

```
void QK_schedUnlock (
    QSchedStatus const stat) [static]
```

**QK** selective scheduler unlock

This function unlocks the **QK** scheduler to the previous status.

**Parameters**

in	stat	previous QK Scheduler lock status returned from <a href="#">QK_schedLock()</a>
----	------	--

**Precondition** qk : 200

- the QK scheduler cannot be unlocked: from the ISR context
- the current lock ceiling must be greater than the previous

**Note**

[QK\\_schedUnlock\(\)](#) must always follow the corresponding [QK\\_schedLock\(\)](#).

**See also**

[QK\\_schedLock\(\)](#)

**Usage**

The following example shows how to lock and unlock the QK scheduler:

```
uint32_t BSP_random(void) {
    uint32_rnd;

    QSchedStatus lockStat = QK_schedLock(N_PHILO); // <== N_PHILO ceiling
    . . . // access/manipulate the shared random seed resource
    QK_schedUnlock(lockStat); // <== unlock

    return rnd;
}
```

**14.13.2.6 QK\_onIdle()**

```
void QK_onIdle (
    void ) [static]
```

QK idle callback (customized in BSPs for QK)

[QK\\_onIdle\(\)](#) is called continuously by the QK idle loop. This callback gives the application an opportunity to enter a power-saving CPU mode, or perform some other idle processing.

**Note**

[QK\\_onIdle\(\)](#) is invoked with interrupts enabled and must also return with interrupts enabled.

**14.13.3 Member Data Documentation****14.13.3.1 QK\_priv\_**

[QK\\_Attr](#) QK\_priv\_ [private]

The documentation for this class was generated from the following files:

- [qk.h](#)
- [qk.c](#)
- [qk.dox](#)

**14.14 QK\_Attr Class Reference**

Private attributes of the QK kernel.

```
#include "qk.h"
```

## Public Attributes

- `QPSet readySet`
- `uint_fast8_t actPrio`
- `uint_fast8_t nextPrio`
- `uint_fast8_t actThre`
- `uint_fast8_t lockCeil`
- `uint_fast8_t intNest`

### 14.14.1 Detailed Description

Private attributes of the `QK` kernel.

### 14.14.2 Member Data Documentation

#### 14.14.2.1 readySet

`QPSet readySet`

Set of active-objects/threads that are ready to run in the `QK` kernel

#### 14.14.2.2 actPrio

`uint_fast8_t actPrio`

Priority of the currently active AO

#### 14.14.2.3 nextPrio

`uint_fast8_t nextPrio`

Next AO priority scheduled by `QK`

#### 14.14.2.4 actThre

`uint_fast8_t actThre`

Preemption threshold of the currently active AO

#### 14.14.2.5 lockCeil

`uint_fast8_t lockCeil`

Scheduler lock-ceiling (0 if scheduler unlocked)

#### 14.14.2.6 intNest

`uint_fast8_t intNest`

Up-down counter indicating current interrupt nesting (used in some `QK` ports)

The documentation for this class was generated from the following files:

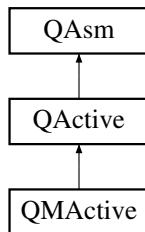
- `qk.h`
- `qk.dox`

## 14.15 QMActive Class Reference

Active object class (based on `QMsm` implementation strategy)

`#include "qp.h"`

Inheritance diagram for QMActive:



### Protected Member Functions

- void `QMActive_ctor` (`QMActive` \*const me, `QStateHandler` const initial)  
*Constructor of `QMActive` class.*

### Protected Member Functions inherited from `QActive`

- void `QActive_ctor` (`QActive` \*const me, `QStateHandler` const initial)  
*`QActive` constructor (abstract base class)*
- void `QActive_stop` (`QActive` \*const me)  
*Stops execution of an active object and removes it from the framework's supervision.*
- void `QActive_subscribe` (`QActive` const \*const me, `enum_t` const sig)  
*Subscribes for delivery of signal `sig` to the active object.*
- void `QActive_unsubscribe` (`QActive` const \*const me, `enum_t` const sig)  
*Unsubscribes from the delivery of signal `sig` to the active object.*
- void `QActive_unsubscribeAll` (`QActive` const \*const me)  
*Unsubscribes from the delivery of all signals to the active object.*
- bool `QActive_defer` (`QActive` const \*const me, struct `QEQueue` \*const eq, `QEvt` const \*const e)  
*Defer an event to a given separate event queue.*
- bool `QActive_recall` (`QActive` \*const me, struct `QEQueue` \*const eq)  
*Recall a deferred event from a given event queue.*
- `uint_fast16_t` `QActive_flushDeferred` (`QActive` const \*const me, struct `QEQueue` \*const eq, `uint_fast16_t` const num)  
*Flush the specified number of events from the deferred queue `eq`*

### Protected Attributes

- `QActive` super

### Protected Attributes inherited from `QActive`

- `QAsm` super
- `uint8_t` `prio`  
*QF-priority [1..`QF_MAX_ACTIVE`] of this AO.*
- `uint8_t` `pthre`  
*Preemption-threshold [1..`QF_MAX_ACTIVE`] of this AO.*
- `QACTIVE_THREAD_TYPE` `thread`  
*Port-dependent representation of the thread of the active object.*
- `QACTIVE_OS_OBJ_TYPE` `osObject`  
*Port-dependent per-thread object.*
- `QACTIVE_EQUEUE_TYPE` `eQueue`  
*Port-dependent event-queue type (often `QEQueue`)*

## Protected Attributes inherited from [QAsm](#)

- struct [QAsmVtable](#) const \* [vptr](#)  
*Virtual pointer inherited by all [QAsm](#) subclasses (see also [SAS\\_QP\\_OO](#))*
- union [QAsmAttr state](#)  
*Current state (pointer to the current state-handler function)*
- union [QAsmAttr temp](#)  
*Temporary storage for target/act-table etc.*

## Additional Inherited Members

### Public Member Functions inherited from [QActive](#)

- void [QActive\\_setAttr](#) ([QActive](#) \*const me, uint32\_t attr1, void const \*attr2)
- void [QActive\\_start](#) ([QActive](#) \*const me, [QPrioSpec](#) const prioSpec, [QEvtPtr](#) \*const qSto, uint\_fast16\_t const qLen, void \*const stkSto, uint\_fast16\_t const stkSize, void const \*const par)  
*Starts execution of an active object and registers the object with the framework.*

### Static Public Member Functions inherited from [QActive](#)

- void [QActive\\_psInit](#) ([QSubscrList](#) \*const subscrSto, [enum\\_t](#) const maxSignal)  
*Publish event to all subscribers of a given signal e->sig*
- uint\_fast16\_t [QActive\\_getQueueMin](#) (uint\_fast8\_t const prio)  
*This function returns the minimum of free entries of the given event queue.*

### Static Protected Member Functions inherited from [QAsm](#)

- static void [QAsm\\_ctor](#) ([QAsm](#) \*const me)  
*Constructor of the [QAsm](#) base class.*

#### 14.15.1 Detailed Description

Active object class (based on [QMsm](#) implementation strategy)

##### Details

[QMActive](#) represents an active object that uses the [QMsm](#) style state machine implementation strategy. This strategy requires the use of the QM modeling tool to generate state machine code automatically, but the code is faster than in the [QHsm](#) style implementation strategy and needs less run-time support (smaller event-processor).

##### Note

[QMActive](#) is not intended to be instantiated directly, but rather serves as the base class for derivation of active objects in the application.

##### Backward Traceability

- [SRS\\_QP\\_SM\\_21](#): *QP Framework should provide a State Machine Implementation Strategy optimized for "automatic code generation"*
- [SDS\\_QP\\_QF](#): *QF Active Object Framework*
- [SDS\\_QP\\_QMActive](#)
- [QMsm](#): *Hierarchical State Machine class (QMsm-style state machine implementation strategy)*

## Usage

The following example illustrates how to derive an active object from `QMActive`. Please note that the `QActive` member `super` is defined as the **first** member of the derived struct (see [SAS\\_QP\\_OO](#)). Please also note the call to the `QMActive_ctor()` in the `Blinky` subclass' constructor.

```
typedef struct {
    QMActive super; // <== inherit QMActive

    QTimeEvt timeEvt; // to timeout the blinking
} Blinky;
. . .

void Blinky_ctor(Blinky * const me) {
    // constructor of the superclass <===
    QMActive_ctor(&me->super, Q_STATE_CAST(&Blinky_initial));

    // constructor(s) of the members
    QTimeEvt_ctorX(&me->timeEvt, &me->super, TIMEOUT_SIG, 0U);
}
```

## 14.15.2 Member Function Documentation

### 14.15.2.1 QMActive\_ctor()

```
void QMActive_ctor (
    QMActive *const me,
    QStateHandler const initial) [protected]
```

Constructor of `QMActive` class.

#### Details

Performs the first step of active object initialization by assigning the virtual pointer and calling the superclass constructor.

#### Parameters

in,out	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>initial</i>	pointer to the event to be dispatched to the MSM

#### Note

Must be called only ONCE before `QASM_INIT()`.

#### Backward Traceability

- `QMActive`: Active object class (based on `QMsm` implementation strategy)

## 14.15.3 Member Data Documentation

### 14.15.3.1 super

```
QActive super [protected]
```

The documentation for this class was generated from the following files:

- `qp.h`
- `qf_qmact.c`
- `qp.dox`

## 14.16 QMPool Class Reference

Native QF Memory Pool.

```
#include "qmpool.h"
```

## Public Member Functions

- void `QMPool_init` (`QMPool` \*const `me`, `void` \*const `poolSto`, `uint_fast32_t` const `poolSize`, `uint_fast16_t` const `blockSize`)  
*Initializes the native QF memory pool.*
- `void` \* `QMPool_get` (`QMPool` \*const `me`, `uint_fast16_t` const `margin`, `uint_fast8_t` const `qsId`)  
*Obtain a memory block from a memory pool.*
- `void` `QMPool_put` (`QMPool` \*const `me`, `void` \*const `block`, `uint_fast8_t` const `qsId`)  
*Recycles a memory block back to a memory pool.*

## Private Attributes

- `void` \*\* `start`  
*Start of the memory managed by this memory pool.*
- `void` \*\* `end`  
*End of the memory managed by this memory pool.*
- `void` \*\*volatile `freeHead`
- `QMPoolSize` `blockSize`  
*Memory block size [bytes] held by this fixed-size pool.*
- `QMPoolCtr` `nTot`  
*Total number of memory blocks in this pool.*
- `QMPoolCtr` volatile `nFree`  
*Number of free memory blocks remaining in the pool at this point.*
- `QMPoolCtr` `nMin`  
*Minimum number of free blocks ever present in this pool.*

### 14.16.1 Detailed Description

Native QF Memory Pool.

#### Details

A fixed block-size memory pool is a very fast and efficient data structure for dynamic allocation of fixed block-size chunks of memory. A memory pool offers fast and deterministic allocation and recycling of memory blocks and is not subject to fragmentation.

The `QMPool` class describes the native QF memory pool, which can be used as the event pool for mutable event allocation, or as a fast, deterministic fixed block-size heap for any other objects in your application.

#### Note

`QMPool` contains only data members for managing a memory pool, but does not contain the pool storage, which must be provided externally during the pool initialization.

### 14.16.2 Member Function Documentation

#### 14.16.2.1 `QMPool_init()`

```
void QMPool_init (
    QMPool *const me,
    void *const poolSto,
    uint_fast32_t const poolSize,
    uint_fast16_t const blockSize)
```

Initializes the native QF memory pool.

**Details**

Initialize a fixed block-size memory pool by providing it with the pool memory to manage, size of this memory, and the block size.

#### Parameters

in, out	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>poolSto</i>	pointer to the memory buffer for pool storage
in	<i>poolSize</i>	size of the storage buffer in bytes
in	<i>blockSize</i>	fixed-size of the memory blocks in bytes

**Precondition** `qf_mem >= 100`

- the memory block must be valid
- the *poolSize* must fit at least one free block
- the *blockSize* must not be too close to the top of the dynamic range

#### Attention

The caller of `QMPool::init()` must make sure that the `poolSto` pointer is properly **aligned**. In particular, it must be possible to efficiently store a pointer at the location pointed to by `poolSto`. Internally, the `QMPool_init()` function rounds up the block size `blockSize` so that it can fit an integer number of pointers. This is done to achieve proper alignment of the blocks within the pool.

#### Note

Due to the rounding of block size the actual capacity of the pool might be less than (`poolSize / blockSize`). You can check the capacity of the pool by calling the `QF_getPoolMin()` function.

This function uses a potentially long critical section, because it is intended to be called only during the initialization of the system, when timing is not critical.

Many [QF](#) ports use memory pools to implement the event pools.

#### Backward Traceability

- [DVR\\_QP\\_MC4\\_R11\\_05: Rule 11.5\(Advisory\): A conversion should not be performed from pointer to void into pointer to object](#)

#### 14.16.2.2 `QMPool_get()`

```
void * QMPool_get (
    QMPool *const me,
    uint_fast16_t const margin,
    uint_fast8_t const qslId)
```

Obtain a memory block from a memory pool.

#### Details

The function allocates a memory block from the pool and returns a pointer to the block back to the caller.

#### Parameters

in, out	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>margin</i>	the minimum number of unused blocks still available in the pool after the allocation.
in	<i>qslId</i>	QS-id of this state machine (for <a href="#">QS</a> local filter)

**Returns**

A pointer to a memory block or NULL if no more blocks are available in the memory pool.

**Note**

This function can be called from any task level or ISR level.

The memory pool `me` must be initialized before any events can be requested from it. Also, the `QMPool_get()` function uses internally a `QF` critical section, so you should be careful not to call it from within a critical section when nesting of critical section is not supported.

**Attention**

An allocated block must be later returned back to the **same** pool from which it has been allocated.

**Backward Traceability**

- DVR\_QP\_MC4\_R18\_03: *Rule 18.3(Required): The relation operators shall not be applied to objects of pointer type except where they point into the same array*
- DVR\_QP\_MC4\_R11\_05: *Rule 11.5(Advisory): A conversion should not be performed from pointer to void into pointer to object*

**14.16.2.3 QMPool\_put()**

```
void QMPool_put (
    QMPool *const me,
    void *const block,
    uint_fast8_t const qsId)
```

Recycles a memory block back to a memory pool.

**Details**

Recycle a memory block to the fixed block-size memory pool.

**Parameters**

in,out	<code>me</code>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<code>block</code>	pointer to the memory block that is being recycled
in	<code>qsId</code>	QS-id of this state machine (for <code>QS</code> local filter)

**Precondition** `qf_mem:200`

- the number of free blocks cannot exceed the total # blocks
- the block pointer must be in range for this pool.

**Attention**

The recycled block must be allocated from the **same** memory pool to which it is returned.

**Note**

This function can be called from any task level or ISR level.

**Backward Traceability**

- DVR\_QP\_MC4\_R11\_05: *Rule 11.5(Advisory): A conversion should not be performed from pointer to void into pointer to object*
- DVR\_QP\_MC4\_R18\_03: *Rule 18.3(Required): The relation operators shall not be applied to objects of pointer type except where they point into the same array*

### 14.16.3 Member Data Documentation

#### 14.16.3.1 start

`void* * start [private]`  
Start of the memory managed by this memory pool.

#### 14.16.3.2 end

`void* * end [private]`  
End of the memory managed by this memory pool.

#### 14.16.3.3 freeHead

`void* * volatile freeHead [private]`

#### 14.16.3.4 blockSize

`QMPoolSize blockSize [private]`  
Memory block size [bytes] held by this fixed-size pool.

#### 14.16.3.5 nTot

`QMPoolCtr nTot [private]`  
Total number of memory blocks in this pool.

#### 14.16.3.6 nFree

`QMPoolCtr volatile nFree [private]`  
Number of free memory blocks remaining in the pool at this point.

#### 14.16.3.7 nMin

`QMPoolCtr nMin [private]`  
Minimum number of free blocks ever present in this pool.

#### Details

This attribute remembers the low watermark of the pool, which provides a valuable information for sizing event pools. (

#### See also

`QF_getPoolMin()`.

The documentation for this class was generated from the following files:

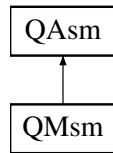
- [qmpool.h](#)
- [qf\\_mem.c](#)
- [qmpool.dox](#)

## 14.17 QMsm Class Reference

Hierarchical State Machine class (QMsm-style state machine implementation strategy)

`#include "qp.h"`

Inheritance diagram for QMsm:



## Public Member Functions

- `QStateHandler QMsm_getStateHandler_ (QAsm *const me)`  
*Implementation of getting the state handler in a `QMsm` subclass.*
- `QMState const * QMsm_childStateObj_ (QMsm const *const me, QMState const *const parent)`  
*Obtain the current active child state of a given parent in `QMsm`.*

## Static Public Member Functions

- static `QMState const * QMsm_stateObj_ (QMsm const *const me)`  
*Obtain the current state from a MSM (read only)*

## Protected Member Functions

- void `QMsm_ctor_ (QMsm *const me, QStateHandler const initial)`  
*Constructor of `QMsm`.*

## Protected Attributes

- `QAsm super`

## Protected Attributes inherited from `QAsm`

- struct `QAsmVtable const * vptr`  
*Virtual pointer inherited by all `QAsm` subclasses (see also `SAS_QP_OO`)*
- union `QAsmAttr state`  
*Current state (pointer to the current state-handler function)*
- union `QAsmAttr temp`  
*Temporary storage for target/act-table etc.*

## Private Member Functions

- void `QMsm_init_ (QAsm *const me, void const *const e, uint_fast8_t const qslid)`  
*Implementation of the top-most initial transition in `QMsm`.*
- void `QMsm_dispatch_ (QAsm *const me, QEvt const *const e, uint_fast8_t const qslid)`  
*Implementation of dispatching events to a `QMsm`.*
- bool `QMsm_isIn_ (QAsm *const me, QStateHandler const stateHndl)`  
*Tests if a given state is part of the current active state configuration.*

## Static Private Member Functions

- static `QState QMsm_execTabl_ (QAsm *const me, QMTranActTable const *const tabl, uint_fast8_t const qslid)`  
*Execute transition-action table.*
- static void `QMsm_exitToTransSource_ (QAsm *const me, QMState const *const cs, QMState const *const ts, uint_fast8_t const qslid)`

*Exit the current state up to the explicit transition source.*

- static `QState QMsm_enterHistory_ (QAsm *const me, QMState const *const hist, uint_fast8_t const qsId)`  
*Enter history of a composite state.*

### Additional Inherited Members

#### Static Protected Member Functions inherited from `QAsm`

- static void `QAsm_ctor (QAsm *const me)`  
*Constructor of the `QAsm` base class.*

### 14.17.1 Detailed Description

Hierarchical State Machine class (QMsm-style state machine implementation strategy)

#### Details

`QMsm` (QM State Machine) provides a more efficient state machine implementation strategy than `QHsm`, but requires the use of the QM modeling tool, but are the fastest and need the least run-time support (the smallest event-processor taking up the least code space).

#### Note

`QMsm` is not intended to be instantiated directly, but rather serves as the abstract base class for derivation of state machines in the application code.

#### Backward Traceability

- `SRS_QP_SM_21`: *QP Framework should provide a State Machine Implementation Strategy optimized for "automatic code generation"*
- `SDS_QP_QEP`: *QEP Event Processor*
- `SDS_QP_QMsm`: *QMsm State machine class.*

#### Usage

The following example illustrates how to derive a state machine class from `QMsm`. Please note that the `QMsm` member `super` is defined as the *first* member of the derived struct. Also, note that the derived class constructor calls the constructor of the base class `QMsm_ctor()`.

```
typedef struct {
    QMsm super; // <== inherit QMsm

    // private state histories
    QStateHandler hist_doorClosed;
    QStateHandler hist_heating;
} ToasterOven;

void ToasterOven_ctor(ToasterOven * const me) {
    QMsm_ctor(&me->super, Q_STATE_CAST(&ToasterOven_initial));
    ...
}
```

### 14.17.2 Member Function Documentation

#### 14.17.2.1 `QMsm_ctor()`

```
void QMsm_ctor (
    QMsm *const me,
    QStateHandler const initial) [protected]
```

Constructor of `QMsm`.

### Details

Performs the first step of [QMsm](#) initialization by assigning the initial pseudostate to the currently active state of the state machine.

### Parameters

<i>in, out</i>	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
<i>in</i>	<i>initial</i>	the top-most initial transition for the MSM.

### Note

Must be called only ONCE before [QASM\\_INIT\(\)](#).

### Backward Traceability

- [QMsm](#): *Hierarchical State Machine class (QMsm-style state machine implementation strategy)*

### Usage

The following example illustrates how to invoke [QMsm\\_ctor\(\)](#) in the "constructor" of a derived state machine:

```
void Calc_ctor(Calc const me) {
    // call superclass' ctor
    QMsm_ctor(&me->super, Q_STATE_CAST(&Calc_initial));

    me->operand1 = 0.0;
    me->operand2 = 0.0;
    me->len      = 0U;
    me->opKey    = 0U;
}
```

## 14.17.2.2 QMsm\_init\_()

```
void QMsm_init_(
    QAsm *const me,
    void const *const e,
    uint_fast8_t const qsId) [private]
```

Implementation of the top-most initial transition in [QMsm](#).

### Details

Synchronously executes the top-most initial transition in a state machine.

### Parameters

<i>in, out</i>	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
<i>in</i>	<i>e</i>	pointer to an extra parameter (might be NULL)
<i>in</i>	<i>qsId</i>	QS-id of this state machine (for <a href="#">QS</a> local filter)

**Precondition** `qep msm:200`

- the virtual pointer must be initialized,
- the top-most initial transition must be initialized,
- the initial transition must not be taken yet.

**Note**

This function should be called only via the virtual table (see [QASM\\_INIT\(\)](#)) and should NOT be called directly in the applications.

**Backward Traceability**

- [QMsm](#): *Hierarchical State Machine class (QMsm-style state machine implementation strategy)*

**14.17.2.3 QMsm\_dispatch\_()**

```
void QMsm_dispatch_()
    QAsm *const me,
    QEvt const *const e,
    uint_fast8_t const qsId) [private]
```

Implementation of dispatching events to a [QMsm](#).

**Details**

Synchronously dispatches an event for processing to a state machine. The processing of an event represents one run-to-completion (RTC) step.

**Parameters**

in,out	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>e</i>	pointer to the event to be dispatched to the MSM
in	<i>qsId</i>	QS-id of this state machine (for <a href="#">QS</a> local filter)

**Precondition** qep\_msm: 302

- current state must be initialized
- check the internal integrity (Software Self-Monitoring (SSM))

**Note**

This function should be called only via the virtual table (see [QASM\\_DISPATCH\(\)](#)) and should NOT be called directly in the applications.

**Backward Traceability**

- [QMsm](#): *Hierarchical State Machine class (QMsm-style state machine implementation strategy)*

**14.17.2.4 QMsm\_isIn\_()**

```
bool QMsm_isIn_
    QAsm *const me,
    QStateHandler const state) [private]
```

Tests if a given state is part of the current active state configuration.

**Details**

Please note that for a MSM, to "be in a state" means also to be in a superstate of the state.

**Parameters**

in	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>state</i>	pointer to the state-handler function to be tested

**Returns**

'true' if the MSM "is in" the *state* and 'false' otherwise

**Note**

This function should be called only via the virtual table (see [QASM\\_IS\\_IN\(\)](#)) and should NOT be called directly in the applications.

**Attention**

This function must be called only on a state machine that is in the "stable state configuration". Among others, this means that the state machine cannot call it in the middle of its own transition.

**Backward Traceability**

- [SRS\\_QP\\_SM\\_25](#): All State Machine Implementation Strategies provided by QP Framework might supply a method for checking if a state machine is in a given state
- [QMsm](#): Hierarchical State Machine class (QMsm-style state machine implementation strategy)

**14.17.2.5 QMsm\_getStateHandler\_()**

```
QStateHandler QMsm_getStateHandler_(
    QAsm *const me)
```

Implementation of getting the state handler in a [QMsm](#) subclass.

**Parameters**

in,out	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
--------	-----------	---

**Note**

This function is only called internally via the virtual table

**Backward Traceability**

- [QMsm](#): Hierarchical State Machine class (QMsm-style state machine implementation strategy)

**14.17.2.6 QMsm\_stateObj()**

```
static QMState const * QMsm_stateObj (
    QMsm const *const me) [inline], [static]
```

Obtain the current state from a MSM (read only)

**Parameters**

in	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
----	-----------	---

**Returns**

the current state object

**Note**

This function is used in QM for auto-generating code for state history (deep history)

**Backward Traceability**

- [QMsm: Hierarchical State Machine class \(QMsm-style state machine implementation strategy\)](#)

**14.17.2.7 QMsm\_childStateObj()**

```
QMstate const * QMsm_childStateObj (
    QMsm const *const me,
    QMState const *const parent)
```

Obtain the current active child state of a given parent in [QMsm](#).

**Details**

Finds the child state of the given parent, such that this child state is an ancestor of the currently active state. The main purpose of this function is to support \*\*shallow history\* transitions in state machines derived from [QMsm](#).

**Parameters**

in	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>parent</i>	pointer to the state-handler object

**Returns**

the child of a given parent state, which is an ancestor of the currently active state. For the corner case when the currently active state is the given parent state, function returns the parent state.

**Postcondition** qep\_msm:890

- the child must be found

**Note**

This function is used in the QM modeling tool for auto-generating code for state history (shallow history)

**Backward Traceability**

- [QMsm: Hierarchical State Machine class \(QMsm-style state machine implementation strategy\)](#)

**14.17.2.8 QMsm\_execTatbl\_()**

```
static QState QMsm_execTatbl_ (
    QAsm *const me,
    QMTranActTable const *const tatbl,
    uint_fast8_t const qsId) [static], [private]
```

Execute transition-action table.

**Details**

Helper function to execute transition sequence in a transition-action table.

**Parameters**

in, out	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>tatbl</i>	pointer to the transition-action table
in	<i>qsld</i>	QS-id of this state machine (for <a href="#">QS</a> local filter)

**Returns**

status of the last action from the transition-action table.

**Precondition** `qep_msm:400`

- provided state table cannot be NULL

**Note**

This function is for internal use inside the QEP event processor and should **\*\*not\*\*** be called directly from the applications.

**Backward Traceability**

- [QMsm: Hierarchical State Machine class \(QMsm-style state machine implementation strategy\)](#)

**14.17.2.9 QMsm\_exitToTranSource\_()**

```
static void QMsm_exitToTranSource_ (
    QAsm *const me,
    QMState const *const cs,
    QMState const *const ts,
    uint_fast8_t const qsId) [static], [private]
```

Exit the current state up to the explicit transition source.

**Details**

Static helper function to exit the current state configuration to the transition source, which in a hierarchical state machine might be a superstate of the current state.

**Parameters**

in, out	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>cs</i>	pointer to the current state
in	<i>ts</i>	pointer to the transition source state
in	<i>qsld</i>	QS-id of this state machine (for <a href="#">QS</a> local filter)

**Backward Traceability**

- [QMsm: Hierarchical State Machine class \(QMsm-style state machine implementation strategy\)](#)

#### 14.17.2.10 QMsm\_enterHistory\_()

```
static QState QMsm_enterHistory_ (
    QAsm *const me,
    QMState const *const hist,
    uint_fast8_t const qsId) [static], [private]
```

Enter history of a composite state.

##### Details

Static helper function to execute the segment of transition to history after entering the composite state and

##### Parameters

in, out	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>hist</i>	pointer to the history substate
in	<i>qsId</i>	QS-id of this state machine (for <a href="#">QS</a> local filter)

##### Returns

[Q\\_RET\\_TRAN\\_INIT](#), if an initial transition has been executed in the last entered state or [Q\\_RET\\_NULL](#) if no such transition was taken.

##### Backward Traceability

- [QMsm](#): Hierarchical State Machine class (QMsm-style state machine implementation strategy)

### 14.17.3 Member Data Documentation

#### 14.17.3.1 super

[QAsm](#) *super* [protected]

The documentation for this class was generated from the following files:

- [qp.h](#)
- [qep\\_msm.c](#)
- [qp.dox](#)

## 14.18 QMState Struct Reference

State object for the [QMsm](#) class (QM State Machine)

```
#include "qp.h"
```

### Private Attributes

- struct [QMState](#) const \* *superstate*
- [QStateHandler](#) const *stateHandler*
- [QActionHandler](#) const *entryAction*
- [QActionHandler](#) const *exitAction*
- [QActionHandler](#) const *initAction*

### 14.18.1 Detailed Description

State object for the [QMsm](#) class (QM State Machine)

#### Details

This class groups together the attributes of a [QMsm](#) state, such as the parent state (state nesting), the associated state handler function and the exit action handler function. These attributes are used inside the [QMsm::QMsm\\_dispatch\\_\(\)](#) and [QMsm::QMsm\\_init\\_\(\)](#) implementations.

#### Backward Traceability

- [SRS\\_QP\\_SM\\_21](#): *QP Framework should provide a State Machine Implementation Strategy optimized for "automatic code generation"*

#### Attention

The [QMState](#) class is only intended for the QM code generator and should not be used in hand-crafted code.

### 14.18.2 Member Data Documentation

#### 14.18.2.1 superstate

```
struct QMState const* superstate [private]
```

#### 14.18.2.2 stateHandler

```
QStateHandler const stateHandler [private]
```

#### 14.18.2.3 entryAction

```
QActionHandler const entryAction [private]
```

#### 14.18.2.4 exitAction

```
QActionHandler const exitAction [private]
```

#### 14.18.2.5 initAction

```
QActionHandler const initAction [private]
```

The documentation for this struct was generated from the following file:

- [qp.h](#)

## 14.19 QMTranActTable Struct Reference

Transition-Action Table for the [QMsm](#) State Machine.

```
#include "qp.h"
```

#### Private Attributes

- [QMState](#) const \* target
- [QActionHandler](#) const act [1]

### 14.19.1 Detailed Description

Transition-Action Table for the [QMsm](#) State Machine.

## 14.19.2 Member Data Documentation

### 14.19.2.1 target

```
QMState const* target [private]
```

### 14.19.2.2 act

```
QActionHandler const act[1] [private]
```

The documentation for this struct was generated from the following file:

- [qp.h](#)

## 14.20 QPSet Class Reference

Set of Active Objects of up to [QF\\_MAX\\_ACTIVE](#) elements.

```
#include "qp.h"
```

### Static Public Member Functions

- static void [QPSet\\_setEmpty](#) (QPSet \*const me)  
*Make the priority set empty.*
- static bool [QPSet\\_isEmpty](#) (QPSet const \*const me)  
*Find out whether the priority-set is empty.*
- static bool [QPSet\\_notEmpty](#) (QPSet const \*const me)  
*Find out whether the priority-set is NOT empty.*
- static bool [QPSet\\_hasElement](#) (QPSet const \*const me, uint\_fast8\_t const n)  
*Find out whether the priority-set has element n*
- static void [QPSet\\_insert](#) (QPSet \*const me, uint\_fast8\_t const n)  
*Insert element n into the priority-set (n = 1..QF\_MAX\_ACTIVE)*
- static void [QPSet\\_remove](#) (QPSet \*const me, uint\_fast8\_t const n)  
*Remove element n from the priority-set (n = 1..QF\_MAX\_ACTIVE)*
- static uint\_fast8\_t [QPSet\\_findMax](#) (QPSet const \*const me)  
*Find the maximum element in the set—returns zero if the set is empty.*

### Private Attributes

- [QPSetBits bits](#) [((QF\_MAX\_ACTIVE+(8U \*sizeof(QPSetBits))) - 1U)/(8U \*sizeof(QPSetBits))]  
*Bitmask with a bit for each element.*

## 14.20.1 Detailed Description

Set of Active Objects of up to [QF\\_MAX\\_ACTIVE](#) elements.

### Details

[QPSet](#) represents the set of Active Objects in the priority order. The set is capable of storing up to [QF\\_MAX\\_ACTIVE](#) elements.

### Backward Traceability

- [SDS\\_QP\\_QF: QF Active Object Framework](#)

## 14.20.2 Member Function Documentation

### 14.20.2.1 QPSet\_setEmpty()

```
static void QPSet_setEmpty (
    QPSet *const me) [inline], [static]
```

Make the priority set empty.

### 14.20.2.2 QPSet\_isEmpty()

```
static bool QPSet_isEmpty (
    QPSet const *const me) [inline], [static]
```

Find out whether the priority-set is empty.

#### Returns

'true' if the priority set is empty.

### 14.20.2.3 QPSet\_notEmpty()

```
static bool QPSet_notEmpty (
    QPSet const *const me) [inline], [static]
```

Find out whether the priority-set is NOT empty.

#### Returns

'true' if the priority set is NOT empty.

### 14.20.2.4 QPSet\_hasElement()

```
static bool QPSet_hasElement (
    QPSet const *const me,
    uint_fast8_t const n) [inline], [static]
```

Find out whether the priority-set has element n

#### Returns

'true' if the priority set has the element n.

### 14.20.2.5 QPSet\_insert()

```
static void QPSet_insert (
    QPSet *const me,
    uint_fast8_t const n) [inline], [static]
```

Insert element n into the priority-set (n = 1..QF\_MAX\_ACTIVE)

### 14.20.2.6 QPSet\_remove()

```
static void QPSet_remove (
    QPSet *const me,
    uint_fast8_t const n) [inline], [static]
```

Remove element n from the priority-set (n = 1..QF\_MAX\_ACTIVE)

### 14.20.2.7 QPSet\_findMax()

```
static uint_fast8_t QPSet_findMax (
    QPSet const *const me) [inline], [static]
```

Find the maximum element in the set—returns zero if the set is empty.

**Returns**

The current maximum element n.

### 14.20.3 Member Data Documentation

#### 14.20.3.1 bits

```
QPSetBits bits[((QF_MAX_ACTIVE+(8U *sizeof(QPSetBits))) - 1U)/(8U *sizeof(QPSetBits))] [private]
```

Bitmask with a bit for each element.

The documentation for this class was generated from the following files:

- [qp.h](#)
- [qp.dox](#)

## 14.21 QS Class Reference

Software tracing instrumentation, target-resident component ([QS](#) namespace emulated as a "class" in C)

### 14.21.1 Detailed Description

Software tracing instrumentation, target-resident component ([QS](#) namespace emulated as a "class" in C)

**Details**

This namespace groups together QP/Spy services for target-resident software tracing instrumentation.

The documentation for this class was generated from the following file:

- [qs.dox](#)

## 14.22 QS\_Filter Struct Reference

[QS](#) type for output filters (global and local)

### 14.22.1 Detailed Description

[QS](#) type for output filters (global and local)

The documentation for this struct was generated from the following file:

- [qs.dox](#)

## 14.23 QS\_TProbe Struct Reference

QUTest Test-Probe attributes.

```
#include "qs_dummy.h"
```

**Public Attributes**

- QSFun [addr](#)
- uint32\_t [data](#)
- uint8\_t [idx](#)

### 14.23.1 Detailed Description

QUTest Test-Probe attributes.

#### Details

Test-Probe allows dynamic modification of the behavior of the CUT (Code Under Test).

### 14.23.2 Member Data Documentation

#### 14.23.2.1 addr

```
QSFun QS_TProbe::addr
```

#### 14.23.2.2 data

```
uint32_t QS_TProbe::data
```

#### 14.23.2.3 idx

```
uint8_t QS_TProbe::idx
```

The documentation for this struct was generated from the following file:

- [qs\\_dummy.h](#)

## 14.24 QSpyId Struct Reference

QS ID type for applying local filtering.

### 14.24.1 Detailed Description

QS ID type for applying local filtering.

The documentation for this struct was generated from the following file:

- [qs.dox](#)

## 14.25 QSubscrList Struct Reference

Subscriber List (for publish-subscribe)

```
#include "qp.h"
```

#### Private Attributes

- [QPSet set](#)

*The set of AOs that subscribed to a given event signal.*

### 14.25.1 Detailed Description

Subscriber List (for publish-subscribe)

#### Details

This data type represents a set of Active Objects that subscribe to a given signal. The set is represented as priority-set, where each bit corresponds to the unique QF-priority of an AO (see [QPrioSpec](#)).

#### Backward Traceability

- [SDS\\_QP\\_QF: QF Active Object Framework](#)

## 14.25.2 Member Data Documentation

### 14.25.2.1 set

`QPSet set [private]`

The set of AOs that subscribed to a given event signal.

The documentation for this struct was generated from the following files:

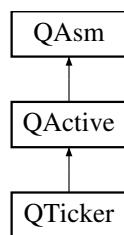
- [qp.h](#)
- [qp.dox](#)

## 14.26 QTicker Class Reference

"Ticker" Active Object class

`#include "qp.h"`

Inheritance diagram for QTicker:



### Public Member Functions

- void [QTicker\\_ctor](#) (`QTicker *const me, uint_fast8_t const tickRate`)

*Constructor of the `QTicker` Active Object class.*

### Public Member Functions inherited from [QActive](#)

- void [QActive\\_setAttr](#) (`QActive *const me, uint32_t attr1, void const *attr2`)
- void [QActive\\_start](#) (`QActive *const me, QPrioSpec const prioSpec, QEvtPtr *const qSto, uint_fast16_t const qLen, void *const stkSto, uint_fast16_t const stkSize, void const *const par`)

*Starts execution of an active object and registers the object with the framework.*

### Protected Attributes

- [QActive super](#)

### Protected Attributes inherited from [QActive](#)

- [QAsm super](#)
- `uint8_t prio`  
*QF-priority [1..[QF\\_MAX\\_ACTIVE](#)] of this AO.*
- `uint8_t pthre`  
*Preemption-threshold [1..[QF\\_MAX\\_ACTIVE](#)] of this AO.*
- `QACTIVE_THREAD_TYPE thread`  
*Port-dependent representation of the thread of the active object.*
- `QACTIVE_OS_OBJ_TYPE osObject`  
*Port-dependent per-thread object.*

- `QACTIVE_EQUEUE_TYPE eQueue`  
*Port-dependent event-queue type (often QEQueue)*

## Protected Attributes inherited from QAsm

- struct `QAsmVtable const * vptr`  
*Virtual pointer inherited by all QAsm subclasses (see also SAS\_QP\_OO)*
- union `QAsmAttr state`  
*Current state (pointer to the current state-handler function)*
- union `QAsmAttr temp`  
*Temporary storage for target/act-table etc.*

## Private Member Functions

- void `QTicker_init_ (QAsm *const me, void const *const par, uint_fast8_t const qsld)`
- void `QTicker_dispatch_ (QAsm *const me, QEvt const *const e, uint_fast8_t const qsld)`
- void `QTicker_trig_ (QActive *const me, void const *const sender)`  
*Asynchronously trigger the QTicker active object to perform tick processing.*

## Additional Inherited Members

### Static Public Member Functions inherited from QActive

- void `QActive_psInit (QSubscrList *const subscrSto, enum_t const maxSignal)`  
*Publish event to all subscribers of a given signal e->sig*
- uint\_fast16\_t `QActive_getQueueMin (uint_fast8_t const prio)`  
*This function returns the minimum of free entries of the given event queue.*

### Protected Member Functions inherited from QActive

- void `QActive_ctor (QActive *const me, QStateHandler const initial)`  
*QActive constructor (abstract base class)*
- void `QActive_stop (QActive *const me)`  
*Stops execution of an active object and removes it from the framework's supervision.*
- void `QActive_subscribe (QActive const *const me, enum_t const sig)`  
*Subscribes for delivery of signal sig to the active object.*
- void `QActive_unsubscribe (QActive const *const me, enum_t const sig)`  
*Unsubscribes from the delivery of signal sig to the active object.*
- void `QActive_unsubscribeAll (QActive const *const me)`  
*Unsubscribes from the delivery of all signals to the active object.*
- bool `QActive_defer (QActive const *const me, struct QEQueue *const eq, QEvt const *const e)`  
*Defer an event to a given separate event queue.*
- bool `QActive_recall (QActive *const me, struct QEQueue *const eq)`  
*Recall a deferred event from a given event queue.*
- uint\_fast16\_t `QActive_flushDeferred (QActive const *const me, struct QEQueue *const eq, uint_fast16_t const num)`  
*Flush the specified number of events from the deferred queue eq*

## Static Protected Member Functions inherited from [QAsm](#)

- static void [QAsm\\_ctor](#) ([QAsm](#) \*const me)

*Constructor of the [QAsm](#) base class.*

### 14.26.1 Detailed Description

"Ticker" Active Object class

#### Details

[QTicker](#) is a [QActive](#) subclass specialized to process [QF](#) system clock tick at a specified tick rate. Placing system clock tick processing in an active object allows you to remove the non-deterministic [QTIMEEV\\_TICK\\_X\(\)](#) processing from the interrupt level and move it into the thread-level, where you can prioritize it as low as you wish.

#### Backward Traceability

- [SDS\\_QP\\_QF: QF Active Object Framework](#)

#### Usage

The following example illustrates use of [QTicker](#) active objects.

#### Attention

The [QTicker](#) active objects must be started **without** any queue buffer and **without** any stack (see [QActive\\_start\(\)](#) calls in the code below).

```
// bsp.h -----
// opaque pointers to the ticker active objects in the application
extern QActive * const the_Ticker0; // "ticker" pointer for clock rate 0
extern QActive * const the_Ticker1; // "ticker" pointer for clock rate 1

// bsp.c -----
static QTicker l_ticker0; // "ticker" instance for clock rate 0
QActive * const the_Ticker0 = &l_ticker0.super;

static QTicker l_ticker1; // "ticker" instance for clock rate 1
QActive * const the_Ticker1 = &l_ticker1.super;

// clock tick ISR for tick rate 0
void SysTick_Handler(void) {
    .
    .
    QTICKER_TRIG(the_Ticker0, &qs_tick0_id);
    .
}

// clock tick ISR for tick rate 1
void Timer0A_IRQHandler(void) {
    .
    .
    QTICKER_TRIG(the_Ticker1, &qs_tick1_id);
    .
}

// main.c -----
main () {
    QTicker_ctor(&l_ticker0, 0U); // active object for tick rate 0
    QActive_start(the_Ticker0,
                  1U, // priority
                  0, 0, 0, 0, 0); // must be always 0 for ticker AO

    QTicker_ctor(&l_ticker1, 1U); // active object for tick rate 1
    QActive_start(the_Ticker1,
                  2U, // priority
                  0, 0, 0, 0, 0); // must be always 0 for ticker AO
    .
}
```

## 14.26.2 Member Function Documentation

### 14.26.2.1 QTicker\_ctor()

```
void QTicker_ctor (
    QTicker *const me,
    uint_fast8_t const tickRate)
```

Constructor of the [QTicker](#) Active Object class.

#### Backward Traceability

- [QTicker](#): "Ticker" Active Object class

### 14.26.2.2 QTicker\_init\_()

```
void QTicker_init_ (
    QAsm *const me,
    void const *const par,
    uint_fast8_t const qsId) [private]
```

### 14.26.2.3 QTicker\_dispatch\_()

```
void QTicker_dispatch_ (
    QAsm *const me,
    QEvt const *const e,
    uint_fast8_t const qsId) [private]
```

### 14.26.2.4 QTicker\_trig\_()

```
void QTicker_trig_ (
    QActive *const me,
    void const *const sender) [private]
```

Asynchronously trigger the [QTicker](#) active object to perform tick processing.

#### Details

This function only triggers the given "ticker" AO. The actual clock-tick processing happens in the thread context of the "ticker" AO, running at the configured priority level.

#### Backward Traceability

- [QTicker](#): "Ticker" Active Object class

## 14.26.3 Member Data Documentation

### 14.26.3.1 super

[QActive](#) super [protected]

The documentation for this class was generated from the following files:

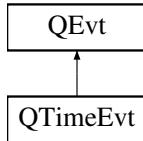
- [qp.h](#)
- [qf\\_actq.c](#)
- [qp.dox](#)

## 14.27 QTimeEvt Class Reference

Time Event class.

```
#include "qp.h"
```

Inheritance diagram for QTimeEvt:



## Public Member Functions

- void `QTimeEvt_ctorX` (`QTimeEvt *const me, QActive *const act, enum_t const sig, uint_fast8_t const tickRate)`  
*The "extended" constructor to initialize a Time Event.*
- void `QTimeEvt_armX` (`QTimeEvt *const me, uint32_t const nTicks, uint32_t const interval)`  
*Arm a time event (extended version for one shot or periodic time event)*
- bool `QTimeEvt_disarm` (`QTimeEvt *const me)`  
*Disarm a time event.*
- bool `QTimeEvt_rearm` (`QTimeEvt *const me, uint32_t const nTicks)`  
*Rearm a time event.*
- bool `QTimeEvt_wasDisarmed` (`QTimeEvt *const me)`  
*Check the "was disarmed" status of a time event.*
- `QTimeEvtCtr QTimeEvt_currCtr` (`QTimeEvt const *const me)`  
*Get the current value of the down-counter of a time event.*

## Static Public Member Functions

- bool `QTimeEvt_noActive` (`uint_fast8_t const tickRate)`  
*Check if any time events are active at a given clock tick rate.*

## Static Public Member Functions inherited from `QEvt`

- static void `QEvt_ctor` (`QEvt *const me, enum_t const sig)`
- static `QEvt * QEvt_init` (`QEvt *const me, uint8_t const dummy)`  
*Event without parameters initialization.*

## Protected Attributes

- `QEvt super`

## Private Member Functions

- `QTimeEvt * QTimeEvt_expire_` (`QTimeEvt *const me, QTimeEvt *const prev_link, QActive const *const act, uint_fast8_t const tickRate)`

## Static Private Member Functions

- void `QTimeEvt_init` (`void)`
- void `QTimeEvt_tick_` (`uint_fast8_t const tickRate, void const *const sender)`  
*Processes all armed time events at every clock tick.*
- void `QTimeEvt_tick1_` (`uint_fast8_t const tickRate, void const *const sender)`  
*Processes one clock tick for QUTest.*

## Private Attributes

- struct [QTimeEvt](#) \*volatile [next](#)  
*Link to the next time event in the list.*
- void \* [act](#)  
*Active object that receives the time events.*
- [QTimeEvtCtr](#) volatile [ctr](#)  
*Down-counter of the time event.*
- [QTimeEvtCtr](#) [interval](#)  
*Interval for periodic time event (zero for one-shot time event)*
- uint8\_t [tickRate](#)
- uint8\_t [flags](#)
- [QTimeEvt](#) [QTimeEvt\\_timeEvtHead\\_](#) [[QF\\_MAX\\_TICK\\_RATE](#)]  
*Static array of heads of linked lists of time events (one for every clock tick rate)*

## Additional Inherited Members

### Public Attributes inherited from [QEvt](#)

- [QSignal](#) [sig](#)  
*Signal of the event (see [Event Signal](#))*

## 14.27.1 Detailed Description

Time Event class.

### Details

Time events are special [QF](#) events equipped with the notion of time passage. The basic usage model of the time events is as follows. An active object allocates one or more [QTimeEvt](#) objects (provides the storage for them). When the active object needs to arrange for a timeout, it arms one of its time events to fire either just once (one-shot) or periodically. Each time event times out independently from the others, so a [QF](#) application can make multiple parallel timeout requests (from the same or different active objects). When [QF](#) detects that the appropriate moment has arrived, it inserts the time event directly into the recipient's event queue. The recipient then processes the time event just like any other event.

Time events, as any other [QF](#) events derive from the [QEvt](#) base class. Typically, you will use a time event as-is, but you can also further derive more specialized time events from it by adding some more data members and/or specialized functions that operate on the specialized time events.

Internally, the armed time events are organized into linked lists—one list for every supported ticking rate. These linked lists are scanned in every invocation of the [QTIMEEVNT\\_TICK\\_X\(\)](#) macro. Only armed (timing out) time events are in the list, so only armed time events consume CPU cycles.

### Remarks

[QF](#) manages the time events in the [QTIMEEVNT\\_TICK\\_X\(\)](#) macro, which must be called periodically, from the clock tick ISR or from other periodic source. [QTIMEEVNT\\_TICK\\_X\(\)](#) caYou might also use the special [QTicker](#) active object.

### Note

Even though [QTimeEvt](#) is a subclass of [QEvt](#), [QTimeEvt](#) instances can NOT be allocated dynamically from event pools. In other words, it is illegal to allocate [QTimeEvt](#) instances with the [Q\\_NEW\(\)](#) or [Q\\_NEW\\_X\(\)](#) macros.

### Backward Traceability

- [SDS\\_QP\\_QTimeEvt](#): *QTimeEvt time event class.*

## 14.27.2 Member Function Documentation

### 14.27.2.1 QTimeEvt\_ctorX()

```
void QTimeEvt_ctorX (
    QTimeEvt *const me,
    QActive *const act,
    enum_t const sig,
    uint_fast8_t const tickRate)
```

The "extended" constructor to initialize a Time Event.

#### Details

When creating a time event, you must commit it to a specific active object `act`, tick rate `tickRate` and event signal `sig`. You cannot change these attributes later.

#### Parameters

in, out	<code>me</code>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<code>act</code>	pointer to the active object associated with this time event. The time event will post itself to this AO.
in	<code>sig</code>	signal to associate with this time event.
in	<code>tickRate</code>	system clock tick rate to associate with this time event in the range [0..15].

**Precondition** `qf_time:300`

- the signal `sig` must be valid
- the tick rate `tickRate` must be in range

#### Note

You should call [QTimeEvt\\_ctorX\(\)](#) exactly once for every Time Event object **before** arming the Time Event. The ideal place for calling [QTimeEvt\\_ctorX\(\)](#) is the constructor of the associated AO.

#### Backward Traceability

- [QTimeEvt](#): *Time Event class*

### 14.27.2.2 QTimeEvt\_armX()

```
void QTimeEvt_armX (
    QTimeEvt *const me,
    uint32_t const nTicks,
    uint32_t const interval)
```

Arm a time event (extended version for one shot or periodic time event)

#### Details

Arms a time event to fire in a specified number of clock ticks and with a specified interval. If the interval is zero, the time event is armed for one shot ('one-shot' time event). When the timeout expires, the time event gets directly posted (using the FIFO policy) into the event queue of the host active object. After posting, a one-shot time event gets automatically disarmed while a periodic time event (interval != 0) is automatically re-armed.

A time event can be disarmed at any time by calling [QTimeEvt\\_disarm\(\)](#). Also, a time event can be re-armed to fire in a different number of clock ticks by calling the [QTimeEvt\\_rearm\(\)](#).

### Parameters

in, out	<i>me</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>nTicks</i>	number of clock ticks (at the associated rate) to rearm the time event with.
in	<i>interval</i>	interval (in clock ticks) for periodic time event.

**Precondition** `qf_time > 400`

- the host AO must be valid,
- the time event must be disarmed,
- the number of clock ticks cannot be zero,
- the signal must be valid.

### Attention

Arming an already armed time event is **not** allowed and is considered a programming error. The QP/C framework will assert if it detects an attempt to arm an already armed time event.

### Backward Traceability

- [QTimeEvt: Time Event class](#)
- [DVR\\_QP\\_MC4\\_R11\\_05: Rule 11.5\(Advisory\): A conversion should not be performed from pointer to void into pointer to object](#)

### Usage

The following example shows how to arm a periodic time event as well as one-shot time event from a state machine of an active object:

```
QState Game_show_logo(Tunnel * const me, QEvt const * const e) {
    QState status_;
    switch (e->sig) {
        case Q_ENTRY_SIG: {
            // arm periodic time event
            QTimeEvt_armX(&me->blinkTimeEvt,           // <===
                           BSP_TICKS_PER_SEC/2U, // one-time delay
                           BSP_TICKS_PER_SEC/2U); // interval
            // arm a one-shot time event
            QTimeEvt_armX(&me->screenTimeEvt,
                           BSP_TICKS_PER_SEC*5U, // one-time delay
                           0U);                // interval (0 == no interval)
            . . .
            status_ = Q_HANDLED();
            break;
        }
        . . .
    }
}
```

### 14.27.2.3 QTimeEvt\_disarm()

```
bool QTimeEvt_disarm (
    QTimeEvt *const me)
```

Disarm a time event.

#### Details

Disarm the time event so it can be safely reused.

#### Remarks

Disarming an already disarmed time event is fine.

#### Parameters

in, out	me	current instance pointer (see <a href="#">SAS_QP_OO</a> )
---------	----	---

#### Returns

'true' if the time event was truly disarmed, that is, it was running. The return of 'false' means that the time event was not truly disarmed, because it was not running. The 'false' return is only possible for one-shot time events that have been automatically disarmed upon expiration. In this case the 'false' return means that the time event has already been posted or published and should be expected in the active object's state machine.

#### Backward Traceability

- [QTimeEvt](#): *Time Event class*

#### 14.27.2.4 QTimeEvt\_rearm()

```
bool QTimeEvt_rearm (
    QTimeEvt *const me,
    uint32_t const nTicks)
```

Rearm a time event.

#### Details

Rearms a time event with a new number of clock ticks. This function can be used to adjust the current period of a periodic time event or to prevent a one-shot time event from expiring (e.g., a watchdog time event). Rarming a periodic timer leaves the interval unchanged and is a convenient method to adjust the phasing of a periodic time event.

#### Parameters

in, out	me	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	nTicks	number of clock ticks (at the associated rate) to rearm the time event with.

#### Returns

'true' if the time event was running as it was re-armed. The 'false' return means that the time event was not truly rearmed because it was not running. The 'false' return is only possible for one-shot time events that have been automatically disarmed upon expiration. In this case the 'false' return means that the time event has already been posted or published and should be expected in the active object's state machine.

#### Precondition qf\_time: 600

- AO must be valid
- tick rate must be in range
- nTicks must not be zero,
- the signal of this time event must be valid

#### Backward Traceability

- [QTimeEvt](#): *Time Event class*
- [DVR\\_QP\\_MC4\\_R11\\_05](#): *Rule 11.5(Advisory): A conversion should not be performed from pointer to void into pointer to object*

**14.27.2.5 QTimeEvt\_wasDisarmed()**

```
bool QTimeEvt_wasDisarmed (
    QTimeEvt *const me)
```

Check the "was disarmed" status of a time event.

**Details**

Useful for checking whether a one-shot time event was disarmed in the [QTimeEvt\\_disarm\(\)](#) operation.

**Parameters**

<code>in, out</code>	<code>me</code>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
----------------------	-----------------	---

**Returns**

'true' if the time event was truly disarmed in the last [QTimeEvt\\_disarm\(\)](#) operation. The 'false' return means that the time event was not truly disarmed, because it was not running at that time. The 'false' return is only possible for one-shot time events that have been automatically disarmed upon expiration. In this case the 'false' return means that the time event has already been posted or published and should be expected in the active object's event queue.

**Note**

This function has a **side effect** of setting the "was disarmed" status, which means that the second and subsequent times this function is called the function will return 'true'.

**Backward Traceability**

- [QTimeEvt](#): *Time Event class*

**14.27.2.6 QTimeEvt\_currCtr()**

```
QTimeEvtCtr QTimeEvt_currCtr (
    QTimeEvt const *const me)
```

Get the current value of the down-counter of a time event.

**Details**

Useful for checking how many clock ticks (at the tick rate associated with the time event) remain until the time event expires.

**Parameters**

<code>in, out</code>	<code>me</code>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
----------------------	-----------------	---

**Returns**

For an armed time event, the function returns the current value of the down-counter of the given time event. If the time event is not armed, the function returns 0.

**Backward Traceability**

- [QTimeEvt](#): *Time Event class*

#### 14.27.2.7 QTimeEvt\_init()

```
void QTimeEvt_init (
    void ) [static], [private]
```

#### 14.27.2.8 QTimeEvt\_tick\_()

```
void QTimeEvt_tick_ (
    uint_fast8_t const tickRate,
    void const *const sender) [static], [private]
```

Processes all armed time events at every clock tick.

##### Details

This internal helper function processes all armed [QTimeEvt](#) objects associated with the tick rate `tickRate`.

This function must be called periodically from a time-tick ISR or from a task so that [QF](#) can manage the timeout events assigned to the given system clock tick rate.

##### Parameters

in	<code>tickRate</code>	clock tick rate serviced in this call [1..15].
in	<code>sender</code>	pointer to a sender object (only for <a href="#">QS</a> tracing)

##### Note

this function should be called only via the macro [QTIMEEVNT\\_TICK\\_X\(\)](#).

The calls to [QTimeEvt\\_tick\\_\(\)](#) with different `tickRate` parameter can preempt each other. For example, higher clock tick rates might be serviced from interrupts while others from tasks (active objects).

##### Backward Traceability

- [QTimeEvt](#): *Time Event class*
- [DVR\\_QP\\_MC4\\_R11\\_05](#): *Rule 11.5(Advisory): A conversion should not be performed from pointer to void into pointer to object*

#### 14.27.2.9 QTimeEvt\_expire\_()

```
QTimeEvt * QTimeEvt_expire_ (
    QTimeEvt *const me,
    QTimeEvt *const prev_link,
    QActive const *const act,
    uint_fast8_t const tickRate) [private]
```

#### 14.27.2.10 QTimeEvt\_tick1\_()

```
void QTimeEvt_tick1_ (
    uint_fast8_t const tickRate,
    void const *const sender) [static], [private]
```

Processes one clock tick for QUTest.

##### Backward Traceability

- [QTimeEvt](#): *Time Event class*

#### 14.27.2.11 QTimeEvt\_noActive()

```
bool QTimeEvt_noActive (
    uint_fast8_t const tickRate) [static]
```

Check if any time events are active at a given clock tick rate.

**Parameters**

in	<i>tickRate</i>	system clock tick rate to find out about.
----	-----------------	---

**Returns**

'true' if no time events are armed at the given tick rate and 'false' otherwise.

**Precondition** `qf_time:800`

- the tick rate must be in range

**Note**

This function should be called in critical section.

**Backward Traceability**

- [QTimeEvt: Time Event class](#)

### 14.27.3 Member Data Documentation

#### 14.27.3.1 super

`QEvt` `super` [protected]

#### 14.27.3.2 next

`struct QTimeEvt* volatile next` [private]

Link to the next time event in the list.

**Backward Traceability**

- [QTimeEvt: Time Event class](#)

#### 14.27.3.3 act

`void* act` [private]

Active object that receives the time events.

#### 14.27.3.4 ctr

`QTimeEvtCtr volatile ctr` [private]

Down-counter of the time event.

**Details**

The down-counter is decremented by 1 in every [QTimeEvt\\_tick\\_\(\)](#) call. The time event fires (gets posted or published) when the down-counter reaches zero.

**Backward Traceability**

- [QTimeEvt: Time Event class](#)

### 14.27.3.5 interval

`QTimeEvtCtr` `interval` [private]  
 Interval for periodic time event (zero for one-shot time event)

#### Details

The value of the interval is re-loaded to the internal down-counter when the time event expires, so that the time event keeps timing out periodically.

#### Backward Traceability

- `QTimeEvt`: *Time Event class*

### 14.27.3.6 tickRate

`uint8_t` `tickRate` [private]

### 14.27.3.7 flags

`uint8_t` `flags` [private]

### 14.27.3.8 QTimeEvt\_timeEvtHead\_

`QTimeEvt` `QTimeEvt_timeEvtHead_[QF_MAX_TICK_RATE]` [private]  
 Static array of heads of linked lists of time events (one for every clock tick rate)

#### Backward Traceability

- `QTimeEvt`: *Time Event class*

The documentation for this class was generated from the following files:

- `qp.h`
- `qf_time.c`
- `qp.dox`

## 14.28 QV Class Reference

`QV` non-preemptive kernel (`QV` namespace emulated as a "class" in C)

`#include "qv.h"`

#### Static Public Member Functions

- void `QV_schedDisable` (`uint_fast8_t` const `ceiling`)
- void `QV_schedEnable` (`void`)
- void `QV_onIdle` (`void`)

#### Private Attributes

- `QV_Attr` `QV_priv_`

### 14.28.1 Detailed Description

`QV` non-preemptive kernel (`QV` namespace emulated as a "class" in C)

## 14.28.2 Member Function Documentation

### 14.28.2.1 QV\_schedDisable()

```
void QV_schedDisable (
    uint_fast8_t const ceiling) [static]
```

QV selective scheduler disable

#### Details

This function disables the QV scheduler from scheduling threads below the specified ceiling. The main purpose of disabling the QV scheduler is to avoid scheduling threads that might take too long and overrun the next clock tick, for example.

#### Parameters

in	<i>ceiling</i>	preemption ceiling upto which the QV scheduler needs to be disabled
----	----------------	---

#### Note

`QV_schedDisable()` must be unlocked with `QV_schedEnable()` at some point, but this is typically done in the system clock tick ISR.

#### See also

[QV\\_schedEnable\(\)](#)

#### Usage

The following example shows how to disable the QV scheduler:

```
static QState Sporadic3_busy(Sporadic3 * const me, QEvt const * const e) {
    QState status_;
    switch (e->sig) {
        // ${AOs::Sporadic3::SM::active::busy::REMINDER}
        case REMINDER_SIG: {
            uint16_t toggles = me->total - me->done;
            if (toggles > me->per_rtc) {
                toggles = me->per_rtc;
            }
            me->done += toggles;
            for (; toggles > 0U; --toggles) {
                BSP_d4on();
                BSP_d4off();
            }
            QV_schedDisable(3U); // <== disable scheduler up to given prio.
        }
    }
}
```

### 14.28.2.2 QV\_schedEnable()

```
void QV_schedEnable (
    void) [static]
```

QV scheduler enable

#### Details

This function re-enables the QV scheduler previously disabled with `QV_schedDisable()`.

#### Note

`QV_schedEnable()` must be called (typically from the system clock tick ISR) when the application ever uses `QV_schedDisable()`.

**See also**

[QV\\_schedDisable\(\)](#)

**Usage**

The following example shows how to enable the **QV** scheduler:

```
void SysTick_Handler(void) {
    QTMEVENT_TICK_X(0U, &l_SysTick_Handler); // time events at rate 0
    QV_schedEnable(); // <== enable the scheduler to process next clock tick
    ...
}
```

**14.28.2.3 QV\_onIdle()**

```
void QV_onIdle (
    void ) [static]
```

**QV** idle callback (customized in BSPs for **QV**)

**QV\_onIdle()** is called by the non-preemptive **QV** kernel (from **QF\_run()**) when the scheduler detects that no events are available for active objects (the idle condition). This callback gives the application an opportunity to enter a power-saving CPU mode, or perform some other idle processing (such as **QS** software tracing output).

**Attention**

**QV\_onIdle()** is invoked with interrupts **DISABLED** because the idle condition can be asynchronously changed at any time by an interrupt. **QV\_onIdle()** MUST enable the interrupts internally, but not before putting the CPU into the low-power mode. (Ideally, enabling interrupts and low-power mode should happen atomically). At the very least, the function MUST enable interrupts, otherwise interrupts will remain disabled permanently.

**14.28.3 Member Data Documentation****14.28.3.1 QV\_priv\_**

**QV\_Attr** **QV\_priv\_** [private]

The documentation for this class was generated from the following files:

- [qv.h](#)
- [qv.c](#)
- [qv.dox](#)

**14.29 QV\_Attr Class Reference**

Private attributes of the **QV** kernel.

```
#include "qv.h"
```

**Private Attributes**

- **QPSet readySet**
- **uint\_fast8\_t schedCeil**

**14.29.1 Detailed Description**

Private attributes of the **QV** kernel.

**14.29.2 Member Data Documentation****14.29.2.1 readySet**

**QPSet** **readySet** [private]

Set of active-objects/threads that are ready to run in the **QV** kernel

### 14.29.2.2 schedCeil

`uint_fast8_t schedCeil [private]`

The documentation for this class was generated from the following files:

- [qv.h](#)
- [qv.dox](#)

## 14.30 QXK Class Reference

[QXK](#) dual-mode kernel ([QXK](#) namespace emulated as a "class" in C)

### 14.30.1 Detailed Description

[QXK](#) dual-mode kernel ([QXK](#) namespace emulated as a "class" in C)

#### Details

The [QXK class](#) is related to the following classes:

- [QXThread](#)
- [QXSemaphore](#)
- [QXMutex](#)

The documentation for this class was generated from the following file:

- [qk.dox](#)

## 14.31 QXK\_Attr Class Reference

Private attributes of the [QXK](#) kernel.

### 14.31.1 Detailed Description

Private attributes of the [QXK](#) kernel.

#### Note

The order and alignment of the data members in this struct might be important in [QXK](#) ports, where the members might be accessed from assembly.

The documentation for this class was generated from the following file:

- [qk.dox](#)

## 14.32 QXMutex Class Reference

Blocking Mutex of the [QXK](#) preemptive kernel.

### 14.32.1 Detailed Description

Blocking Mutex of the [QXK](#) preemptive kernel.

## Details

[QXMutex](#) is a blocking mutual exclusion mechanism that can also apply the **priority-ceiling protocol** to avoid unbounded priority inversion (if initialized with a non-zero ceiling priority, see [QXMutex\\_init\(\)](#)). In that case, [QXMutex](#) requires its own unique QP priority level, which cannot be used by any thread or any other [QXMutex](#).

If initialized with preemption-ceiling of zero, [QXMutex](#) does **not** use the priority-ceiling protocol and does not require a unique QP priority (see [QXMutex\\_init\(\)](#)).

[QXMutex](#) is **recursive** (re-entrant), which means that it can be locked multiple times (up to 255 levels) by the *same* thread without causing deadlock.

[QXMutex](#) is primarily intended for the [extended \(blocking\) threads](#), but can also be used by the [basic threads](#) through the non-blocking [QXMutex\\_tryLock\(\)](#) API.

## Note

[QXMutex](#) should be used in situations when at least one of the extended threads contending for the mutex blocks while holding the mutex (between the [QXMutex\\_lock\(\)](#) and [QXMutex\\_unlock\(\)](#) operations). If no blocking is needed while holding the mutex, the more efficient non-blocking mechanism of [selective QXK scheduler locking](#) should be used instead. [Selective scheduler locking](#) is available for both [basic threads](#) and [extended threads](#), so it is applicable to situations where resources are shared among all these threads.

## Usage

The following example illustrates how to instantiate and use the mutex to protect a shared resource (random seed of a pseudo-random number generator).

```
QXMutex l_rndMutex; // mutex to protect the random number generator
.
.
void BSP_randomSeed(uint32_t seed) {
    QXMutex_init(&l_rndMutex, N_PHILO); // <== initialize the mutex
    l_rnd = seed;
}

uint32_t BSP_random(void) { // a pseudo-random-number generator
    uint32_t rnd;

    QXMutex_lock(&l_rndMutex); // <== lock the shared random seed
    rnd = l_rnd * (3U*7U*11U*13U*23U);
    l_rnd = rnd; // set for the next time
    QXMutex_unlock(&l_rndMutex); // <== unlock the shared random seed

    return rnd;
}
```

The documentation for this class was generated from the following file:

- [qxk.dox](#)

## 14.33 QXSemaphore Class Reference

Counting Semaphore of the [QXK](#) preemptive kernel.

### 14.33.1 Detailed Description

Counting Semaphore of the [QXK](#) preemptive kernel.

## Details

[QXSemaphore](#) is a blocking mechanism intended primarily for signaling [extended threads](#). The semaphore is initialized with the maximum count (see [QXSemaphore\\_init\(\)](#)), which allows you to create a binary semaphore (when the maximum count is 1) and counting semaphore when the maximum count is > 1.

## Usage

The following example illustrates how to instantiate and use the semaphore in your application.

```
QXSemaphore BTN_sema; // semaphore to signal a button press//
int main() {
    .
    .
    // initialize the BTN_sema semaphore as binary, signaling semaphore// 
    QXSemaphore_init(&BTN_sema, // pointer to semaphore to initialize// 
                    0U, // initial semaphore count (signaling semaphore)// 
                    1U); // maximum semaphore count (binary semaphore)// 
    .
    .
}

void main_threadXYZ(QXThread * const me) {
    while (1) {
        .
        .
        QXSemaphore_wait(&BTN_sema, // <--- pointer to semaphore to wait on// 
                        QXTTHREAD_NO_TIMEOUT); // timeout for waiting// 
        .
    }
}

void GPIO_Handler(void) {
    .
    .
    QXSemaphore_signal(&BTN_sema); // <--- pointer to semaphore to signal// 
    .
}
```

The documentation for this class was generated from the following file:

- [qxk.dox](#)

## 14.34 QXThread Class Reference

eXtended (blocking) thread of the [QXK](#) preemptive kernel

### 14.34.1 Detailed Description

eXtended (blocking) thread of the [QXK](#) preemptive kernel

#### Details

[QXThread](#) represents the eXtended (blocking) thread of the [QXK](#) kernel. Each extended thread in the application must be represented by the corresponding [QXThread](#) instance

#### Note

Typically, [QXThread](#) is instantiated directly in the application code. The customization of the thread occurs in the [QXThread\\_ctor\(\)](#), where you provide the thread-handler function as the parameter.

## Usage

The following example illustrates how to instantiate and use an extended thread in your application.

```
#include "qpc.h"

QXThread blinky; // QXK extended-thread object

void main_blinky(QXThread * const me) { // thread function
    while (1) {
        BSP_ledOn();
        QXThread_delay(100U); // BLOCK
        BSP_ledOff();
        QXThread_delay(200U); // BLOCK
    }
}

int main() {
    .
}
```

```
// initialize and start blinky thread
QXThread_ctor(&blinky, &main_blinky, 0);

static uint64_t stack_blinky[40]; // stack for the thread
QXThread_start(&blinky,
    5U,           // priority
    (void *)0, 0, // event queue (not used)
    stack_blinky, sizeof(stack_blinky), // stack
    (void *)0);   // extra parameter (not used)
    .
    .
return QF_run(); // run the application
}
```

The documentation for this class was generated from the following file:

- [qxk.dox](#)

# Chapter 15

## File Documentation

### 15.1 history.dox File Reference

### 15.2 cmd\_options.dox File Reference

This file documents preprocessor macros that are usually defined on the command-line to the compiler.

#### Macros

- `#define QP_CONFIG`  
*The preprocessor switch to activate the QP configuration file [qp\\_config.h](#).*
- `#define Q_SPY`  
*The preprocessor switch to activate the QS software tracing instrumentation in QP/C Framework and QP Application code.*
- `#define Q_UTEST`  
*The preprocessor switch to activate the QUTest testing instrumentation in QP/C Framework and QP Application code.*

#### 15.2.1 Detailed Description

This file documents preprocessor macros that are usually defined on the command-line to the compiler.

#### 15.2.2 Macro Definition Documentation

##### 15.2.2.1 QP\_CONFIG

`#define QP_CONFIG`

The preprocessor switch to activate the QP configuration file [qp\\_config.h](#).

###### Details

When defined, `QP_CONFIG` causes the QP/C port to include the [qp\\_config.h](#) header file, which must be available on the include path to the compiler.

##### 15.2.2.2 Q\_SPY

`#define Q_SPY`

The preprocessor switch to activate the QS software tracing instrumentation in QP/C Framework and QP Application code.

**Details**

When defined, `Q_SPY` activates the `QS` software tracing instrumentation. When `Q_SPY` is not defined, the `QS` instrumentation in the code does not generate any code.

**Note**

The `Q_SPY` is typically defined only in the Spy build configuration and is not defined in other build configurations.

**15.2.2.3 Q\_UTEST**

```
#define Q_UTEST
```

The preprocessor switch to activate the QUTest testing instrumentation in QP/C Framework and QP Application code.

**Details**

When defined, `Q_UTEST` activates the QUTest testing instrumentation, which also requires `QS` software tracing (see `Q_SPY`).

Also, the `Q_UTEST` macro can have a value that decides whether the QP Framework "stub" is included or not. The following values are supported:

```
#define Q_UTEST // the %QP Framework testing "stub" is included
#define Q_UTEST 0 // the %QP Framework testing "stub" is NOT included
```

**Note**

The `Q_UTEST` is typically defined only in the test fixtures.

**15.3 qqueue.dox File Reference****Macros**

- `#define QF_EQQUEUE_CTR_SIZE 1U`

**TypeDefs**

- `typedef uint8_t QEQueueCtr`

**15.3.1 Macro Definition Documentation****15.3.1.1 QF\_EQQUEUE\_CTR\_SIZE**

```
#define QF_EQQUEUE_CTR_SIZE 1U
```

The size [bytes] of the ring-buffer counters used in the native `QF` event queue implementation. Valid values: 1U or 2U; default 1U.

**Details**

This macro can be defined in the `QF` port file (`qf_port.h`) to configure the `QEQueueCtr` type. Here the macro is not defined so the default of 1 byte is chosen.

**15.3.2 Typedef Documentation****15.3.2.1 QEQueueCtr**

```
typedef uint8_t QEQueueCtr
```

## 15.4 qk.dox File Reference

### 15.5 qmpool.dox File Reference

### 15.6 qp.dox File Reference

### 15.7 qp\_config.h File Reference

Sample QP/C configuration file.

#### Macros

- #define QP\_API\_VERSION 0  
*QP Framework API backwards-compatibility version.*
- #define Q\_UNSAFE  
*Disable the QP Functional Safety (FuSa) Subsystem.*
- #define Q\_SIGNAL\_SIZE 2U  
*Size of the QEvt signal [bytes].*
- #define QF\_MAX\_ACTIVE 32U  
*Maximum # Active Objects in the system (1..64)*
- #define QF\_MAX\_EPOOL 3U  
*Maximum # event pools in the system (0..15)*
- #define QF\_MAX\_TICK\_RATE 1U  
*Maximum # clock tick rates in the system (0..15)*
- #define QEVT\_PAR\_INIT  
*Event parameter initialization (RAII) for dynamic events.*
- #define QACTIVE\_CAN\_STOP  
*Enable the Active Object stop API.*
- #define QF\_EVENT\_SIZ\_SIZE 2U  
*Maximum size of dynamic events managed by QP.*
- #define QF\_TIMEEVCT\_CTR\_SIZE 4U  
*Time event counter size.*
- #define QF\_EQQUEUE\_CTR\_SIZE 1U  
*Event queue counter size.*
- #define QF\_MPOOL\_CTR\_SIZE 2U  
*Memory pool counter size (QF\_MPOOL\_CTR\_SIZE)*
- #define QF\_MPOOL\_SIZ\_SIZE 2U  
*Memory block size (QF\_MPOOL\_SIZ\_SIZE)*
- #define QS\_TIME\_SIZE 4U  
*QS timestamp size (QS\_TIME\_SIZE)*
- #define QS\_CTR\_SIZE 2U  
*QS buffer counter size (QS\_CTR\_SIZE)*
- #define QF\_ON\_CONTEXT\_SW  
*Enable context switch callback WITHOUT QS.*
- #define QF\_MEM\_ISOLATE  
*Enable MPU memory isolation.*
- #define QK\_USE\_IRQ\_NUM 31  
*Use IRQ handler for QK return-from-preemption in ARM Cortex-M.*

- `#define QK_USE_IRQ_HANDLER Reserved31_IRQHandler`  
*Use IRQ handler for QK return-from-preemption in ARM Cortex-M.*
- `#define QXK_USE_IRQ_NUM 31`  
*Use IRQ handler for QXK return-from-preemption in ARM Cortex-M.*
- `#define QXK_USE_IRQ_HANDLER Reserved31_IRQHandler`  
*Use IRQ handler for QXK return-from-preemption in ARM Cortex-M.*

### 15.7.1 Detailed Description

Sample QP/C configuration file.

#### Details

This is an example of a QP/C configuration file with the explanation of the available configuration macros. The `qp_config.h` file is optional and does not need to be provided (see the description of the `QP_CONFIG` macro below). In that case, all the listed configuration macros will be set at their **default** values.

#### Attention

The `qp_config.h` configuration file takes effect only when the command-line macro `QP_CONFIG` is defined.

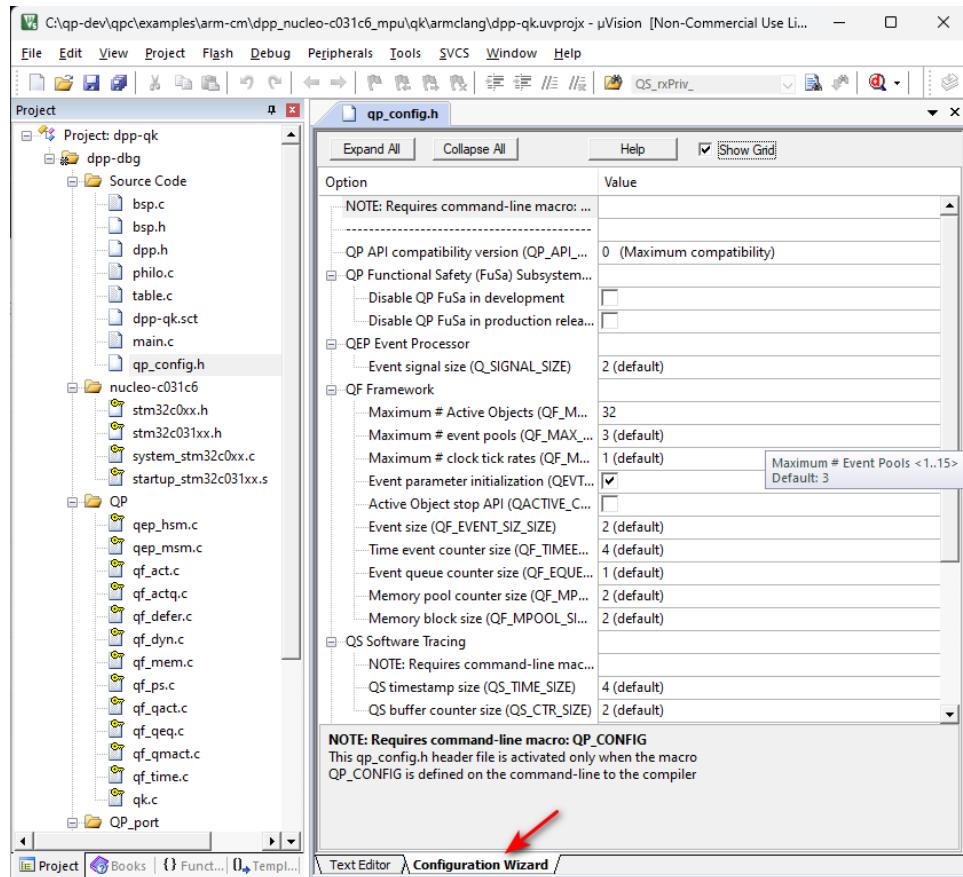
Also, if the `qp_config.h` file is provided and enabled (by defining the command-line macro `QP_CONFIG`), but any of the listed configuration macros is not defined in the provided file, the **default** value will be used.

#### Note

Some of the configuration macros listed in this file apply only to specific QP/C ports.

### Configuration Wizard support

The `qp_config.h` header files provided in the various QP/C examples are formatted to support the "Configuration Wizard" editor available in some IDEs (e.g., KEIL uVision). The screen shot below shows how to edit `qp_config.h` in that mode.



## 15.7.2 Macro Definition Documentation

### 15.7.2.1 QP\_API\_VERSION

```
#define QP_API_VERSION 0
QP Framework API backwards-compatibility version.
```

#### Details

QP API backwards compatibility with the QP/C API version. Lower `QP_API_VERSION` values enable backwards compatibility with lower (older) QP API versions.

- 0 => maximum supported compatibility
- 580 => QP 5.8.0 or newer
- 660 => QP 6.6.0 or newer
- 691 => QP 6.9.1 or newer
- 700 => QP 7.0.0 or newer
- 9999 => Latest QP API only (minimum compatibility)

For example, `QP_API_VERSION==691` will enable the compatibility layer with QP version 6.9.1 and newer, but not older than 6.9.1. `QP_API_VERSION==0` enables the maximum currently supported backwards compatibility. Conversely, `QP_API_VERSION==9999` means that no backwards compatibility layer should be enabled. Default: 0 (All supported)

### 15.7.2.2 Q\_UNSAFE

```
#define Q_UNSAFE
```

Disable the QP Functional Safety (FuSa) Subsystem.

#### Details

Defining the macro `Q_UNSAFE` disables QP FuSa Subsystem, which consists of the following facilities:

- Software assertions as a recommended technique (called Failure Assertion Programming (FAP) in IEC 61508)
- Software Self-Monitoring (SSM), which encompasses such techniques:
  - Duplicate Inverse Storage for critical variables
  - Fixed Upper Loop Bound for all loops
  - Invalid Control Flow for all unreachable code paths
  - Hardware Memory Isolation by means of Memory Protection Unit (MPU)
  - High Watermark in event queues
  - High Watermark in event pools
  - Stack Overflow detection in QP Applications
  - Stack Painting in QP Applications
  - NULL-Pointer Dereferencing protection in QP Applications

Default: undefined

#### Attention

Disabling QP FuSa Subsystem (by defining the macro `Q_UNSAFE`), especially in the final production release, **CONTRADICTS** the most fundamental principles of functional safety and is **NOT recommended**.

### 15.7.2.3 Q\_SIGNAL\_SIZE

```
#define Q_SIGNAL_SIZE 2U
```

Size of the `QEvt` signal [bytes].

#### Details

This macro controls the dynamic range of event signals.

- 1U => 1 byte (up to 255 signals)
- 2U => 2 bytes (up to 65535 signals) (default)
- 4U => 4 bytes (up to 4G signals)

### 15.7.2.4 QF\_MAX\_ACTIVE

```
#define QF_MAX_ACTIVE 32U
```

Maximum # Active Objects in the system (1..64)

#### Details

Defines the maximum # Active Objects that QP Framework can manage at any time.

- Minimum: 1
- Default: 32
- Maximum: 64 (inclusive)

### 15.7.2.5 QF\_MAX\_EPOOL

```
#define QF_MAX_EPOOL 3U
Maximum # event pools in the system (0..15)
```

#### Details

- Minimum: 0 – no event pools at all
- Default: 3
- Maximum: 15 (inclusive)

### 15.7.2.6 QF\_MAX\_TICK\_RATE

```
#define QF_MAX_TICK_RATE 1U
Maximum # clock tick rates in the system (0..15)
```

#### Details

- Minimum: 0 – no time events at all
- Default: 1
- Maximum: 15 (inclusive)

### 15.7.2.7 QEVT\_PAR\_INIT

```
#define QEVT_PAR_INIT
Event parameter initialization (RAll) for dynamic events.
```

#### Details

When defined, the macro activates initialization of event parameters while creating dynamic events. This could be used for Resource Acquisition Is Initialization (**RAll**) for dynamic events.

Default: undefined

#### See also

- [Q\\_NEW\(\)](#)
- [Q\\_NEW\\_X\(\)](#)
- [QEvt\\_init\(\)](#)

### 15.7.2.8 QACTIVE\_CAN\_STOP

```
#define QACTIVE_CAN_STOP
Enable the Active Object stop API.
```

#### Details

When defined, enable Active Object stop API (Not recommended)

Default: undefined

### 15.7.2.9 QF\_EVENT\_SIZ\_SIZE

```
#define QF_EVENT_SIZ_SIZE 2U
Maximum size of dynamic events managed by QP.
```

#### Details

This macro controls the maximum size of dynamic events managed by QP Framework.

- 1U => 1 byte dynamic range (event size up to 255 bytes)
- 2U => 2 byte dynamic range (event size up to 65535 bytes) (default)

Default: 2 byte dynamic range (64K bytes maximum event size)

### 15.7.2.10 QF\_TIMEEVT\_CTR\_SIZE

```
#define QF_TIMEEVT_CTR_SIZE 4U
Time event counter size.
```

#### Details

This macro controls the dynamic range of timeouts allowed in [QTimeEvt](#). The timeouts are counted in tick *of the associated clock tick rate*.

- 1U => 1 byte (timeouts of up to 255 ticks)
- 2U => 2 bytes (timeouts of up to 65535 ticks)
- 4U => 4 bytes (timeouts of up to  $2^{32}$  ticks) (default)

Default: 4 bytes ( $2^{32}$  dynamic range)

### 15.7.2.11 QF\_EQUEUE\_CTR\_SIZE

```
#define QF_EQUEUE_CTR_SIZE 1U
Event queue counter size.
```

#### Details

This macro controls the maximum number of events that [QEQueue](#) can hold

- 1U => 1 byte (maximum 255 events) (default)
- 2U => 2 bytes (maximum 65535 events)

Default: 1 (maximum 255 events in a queue)

### 15.7.2.12 QF\_MPOOL\_CTR\_SIZE

```
#define QF_MPOOL_CTR_SIZE 2U
Memory pool counter size (QF_MPOOL_CTR_SIZE)
```

#### Details

This macro controls the maximum number of memory blocks that [QMPool](#) can hold

- 1U => 1 byte (up to 255 memory blocks)
- 2U => 2 bytes (up to 65535 memory blocks) (default)
- 2U => 2 bytes (up to  $2^{32}$  memory blocks)

Default: 2 bytes (up to 65535 memory blocks maximum in a pool)

### 15.7.2.13 QF\_MPOOL\_SIZ\_SIZE

```
#define QF_MPOOL_SIZ_SIZE 2U
Memory block size (QF_MPOOL_SIZ_SIZE)
```

#### Details

This macro controls the maximum size of memory blocks that [QMPool](#) can hold.

- 1U => 1 byte dynamic range (event size up to 255 bytes)
- 2 U=> 2 byte dynamic range (event size up to 65535 bytes) (default)

Default: 2 byte dynamic range (64K bytes maximum block size)

### 15.7.2.14 QS\_TIME\_SIZE

```
#define QS_TIME_SIZE 4U
QS timestamp size (QS_TIME_SIZE)
```

#### Details

This macro controls the dynamic range of timestamp produced by [QS](#) software tracing.

- 1U => 1 byte (timestamp wraps around at 255 )
- 2U => 2 bytes (timestamp wraps around at 65535)
- 4U => 4 bytes (timestamp wraps around at  $2^{32}$ ) (default)

Default: 4 bytes ( $2^{32}$  dynamic range)

#### See also

- [QSTimeCtr](#)
- [QS::QS\\_onGetTime\(\)](#)

### 15.7.2.15 QS\_CTR\_SIZE

```
#define QS_CTR_SIZE 2U
QS buffer counter size (QS_CTR_SIZE)
```

#### Details

This macro controls the maximum number of bytes held in the [QS](#) TX/RX buffers.

- 1U => 1 byte (maximum 255 bytes)
- 2U => 2 bytes (maximum 65535 bytes) (default)
- 4U => 4 bytes (maximum  $2^{32}$  bytes)

Default: 2 bytes (maximum 65535 bytes in [QS](#) buffers)

### 15.7.2.16 QF\_ON\_CONTEXT\_SW

```
#define QF_ON_CONTEXT_SW
Enable context switch callback WITHOUT QS.
```

#### Details

When defined, enables context switch callback [QF\\_onContextSw\(\)](#) in the built-in kernels ([QV](#), [QK](#), [QXK](#)).

Default: undefined

### 15.7.2.17 QF\_MEM\_ISOLATE

```
#define QF_MEM_ISOLATE
Enable MPU memory isolation.
```

#### Details

When defined, enables memory isolation (requires MPU)

#### Note

Implies [QF\\_ON\\_CONTEXT\\_SW](#) (i.e., [QF\\_ON\\_CONTEXT\\_SW](#) gets defined)

### 15.7.2.18 QK\_USE\_IRQ\_NUM

```
#define QK_USE_IRQ_NUM 31
Use IRQ handler for QK return-from-preemption in ARM Cortex-M.
```

#### Details

If [QK\\_USE\\_IRQ\\_NUM](#) macro is defined, it specifies the IRQ number in ARM Cortex-M to be used as the exception for return-from-preemption in the [QK](#) kernel.

This macro should be defined only if the NMI handler is utilized in the project. The specified IRQ number must be otherwise unused.

Default: undefined

#### See also

Requires defining the macro [QK\\_USE\\_IRQ\\_HANDLER](#)

### 15.7.2.19 QK\_USE\_IRQ\_HANDLER

```
#define QK_USE_IRQ_HANDLER Reserved31_IRQHandler
Use IRQ handler for QK return-from-preemption in ARM Cortex-M.
```

#### Details

If [QK\\_USE\\_IRQ\\_HANDLER](#) macro is defined, it specifies the IRQ handler name in ARM Cortex-M to be used as the exception for return-from-preemption in the [QK](#) kernel.

This macro should be defined only if the NMI handler is utilized in the project. The specified IRQ handler must be otherwise unused.

#### See also

Requires defining the macro [QK\\_USE\\_IRQ\\_NUM](#)

### 15.7.2.20 QXK\_USE\_IRQ\_NUM

```
#define QXK_USE_IRQ_NUM 31
Use IRQ handler for QXK return-from-preemption in ARM Cortex-M.
```

#### Details

If [QXK\\_USE\\_IRQ\\_NUM](#) macro is defined, it specifies the IRQ number in ARM Cortex-M to be used as the exception for return-from-preemption in the [QXK](#) kernel.

This macro should be defined only if the NMI handler is utilized in the project. The specified IRQ number must be otherwise unused.

Default: undefined

#### See also

Requires defining the macro [QXK\\_USE\\_IRQ\\_HANDLER](#)

### 15.7.2.21 QXK\_USE\_IRQ\_HANDLER

```
#define QXK_USE_IRQ_HANDLER Reserved31_IRQHandler
```

Use IRQ handler for [QK](#) return-from-preemption in ARM Cortex-M.

#### Details

If [QXK\\_USE\\_IRQ\\_HANDLER](#) macro is defined, it specifies the IRQ handler name in ARM Cortex-M to be used as the exception for return-from-preemption in the [QK](#) kernel.

This macro should be defined only if the NMI handler is utilized in the project. The specified IRQ handler must be otherwise unused.

Default: undefined

#### See also

Requires defining the macro [QXK\\_USE\\_IRQ\\_NUM](#)

## 15.8 qp\_pkg.dox File Reference

## 15.9 qp\_port.h File Reference

Sample QP/C port.

```
#include <stdint.h>
#include <stdbool.h>
#include "qqueue.h"
#include "qmpool.h"
#include "qp.h"
#include "qk.h"
```

### Macros

- `#define Q_NORETURN _Noreturn void`  
*No-return specifier for the [Q\\_onError\(\)](#) callback function.*
- `#define QACTIVE_EQEUE_TYPE QEQueue`  
*QActive event queue type used in various QP/C ports.*
- `#define QACTIVE_OS_OBJ_TYPE void*`  
*QActive "OS-object" type used in various QP/C ports.*
- `#define QACTIVE_THREAD_TYPE void const *`  
*QActive "thread" type used in various QP/C ports.*
- `#define QF_INT_DISABLE()`  
*Disable interrupts.*
- `#define QF_INT_ENABLE()`  
*Enable interrupts.*
- `#define QF_CRIT_STAT crit_stat_t crit_stat_;`
- `#define QF_CRIT_ENTRY()`
- `#define QF_CRIT_EXIT()`
- `#define QV_CPU_SLEEP()`  
*/ def QF\_MEM\_ISOLATE*
- `#define QK_ISR_CONTEXT_()`
- `#define QK_ISR_ENTRY()`
- `#define QK_ISR_EXIT()`

- `#define QXK_ISR_CONTEXT_()`  
*Check if the code executes in the ISR context.*
- `#define QXK_CONTEXT_SWITCH_()`  
*Trigger context switch (used internally in QXK only)*
- `#define QXK_ISR_ENTRY()`  
*Define the ISR entry sequence.*
- `#define QXK_ISR_EXIT()`  
*Define the ISR exit sequence.*

## Typedefs

- `typedef unsigned int crit_stat_t`

## Functions

- `crit_stat_t critEntry (void)`
- `void critExit (crit_stat_t stat)`

### 15.9.1 Detailed Description

Sample QP/C port.

#### Details

This is just an example of a `QF` port for a generic C99 compiler. Other specific `QF` ports will define the `QF` facilities differently.

### 15.9.2 Macro Definition Documentation

#### 15.9.2.1 Q\_NORETURN

```
#define Q_NORETURN __attribute__((__noreturn))
```

No-return specifier for the `Q_onError()` callback function.

If the `Q_NORETURN` macro is undefined, the default definition uses the C99 specifier `_Noreturn`.

#### Note

The `Q_NORETURN` macro can be defined in the QP port (typically in `qep_port.h`). If such definition is provided the default won't be used.

#### 15.9.2.2 QACTIVE\_EQUEUE\_TYPE

```
#define QACTIVE_EQUEUE_TYPE QEQueue
```

`QActive` event queue type used in various QP/C ports.

#### 15.9.2.3 QACTIVE\_OS\_OBJ\_TYPE

```
#define QACTIVE_OS_OBJ_TYPE void*
```

`QActive` "OS-object" type used in various QP/C ports.

#### 15.9.2.4 QACTIVE\_THREAD\_TYPE

```
#define QACTIVE_THREAD_TYPE void const *
```

`QActive` "thread" type used in various QP/C ports.

### 15.9.2.5 QF\_INT\_DISABLE

```
#define QF_INT_DISABLE()
```

**Value:**

```
intDisable()
```

Disable interrupts.

### 15.9.2.6 QF\_INT\_ENABLE

```
#define QF_INT_ENABLE()
```

**Value:**

```
intEnable()
```

Enable interrupts.

### 15.9.2.7 QF\_CRIT\_STAT

```
#define QF_CRIT_STAT crit_stat_t crit_stat_;
```

Define the critical section status that was present before entering the critical section.

For critical sections that are allowed to nest, the critical section status must be saved and restored at the end. This macro provides the storage for saving the status.

**Note**

This macro might be empty, in which case the critical section status is not saved or restored. Such critical sections won't be able to nest. Also, note that the macro should be invoked without the closing semicolon.

### 15.9.2.8 QF\_CRIT\_ENTRY

```
#define QF_CRIT_ENTRY()
```

**Value:**

```
(crit_stat_ = critEntry())
```

Enter the critical section

If the critical section status is provided, the macro saves the critical section status from before entering the critical section. Otherwise, the macro just unconditionally enters the critical section without saving the status.

### 15.9.2.9 QF\_CRIT\_EXIT

```
#define QF_CRIT_EXIT()
```

**Value:**

```
critExit(crit_stat_)
```

Exit the critical section

If the critical section status is provided, the macro restores the critical section status saved by [QF\\_CRIT\\_ENTRY\(\)](#). Otherwise, the macro just unconditionally exits the critical section.

### 15.9.2.10 QV\_CPU\_SLEEP

```
#define QV_CPU_SLEEP()
```

**Value:**

```
do {
    __disable_interrupt(); \
    QF_INT_ENABLE(); \
    __WFI(); \
    __enable_interrupt(); \
} while (false)
```

**! def QF\_MEM\_ISOLATE**

Macro to put the CPU to sleep **safely** in the non-preemptive [QV](#) kernel (to be called from [QV\\_onIdle\(\)](#)).

### 15.9.2.11 QK\_ISR\_CONTEXT\_

```
#define QK_ISR_CONTEXT_()
```

**Value:**

```
(QK_priv_.intNest != 0U)
```

Check if the code executes in the ISR context

### 15.9.2.12 QK\_ISR\_ENTRY

```
#define QK_ISR_ENTRY()
```

**Value:**

```
do {
    QF_INT_DISABLE();
    ++QK_priv_.intNest;
    QF_QS_ISR_ENTRY(QK_priv_.intNest, QK_currPrio_);
    QF_INT_ENABLE();
} while (false)
```

Define the ISR entry sequence

### 15.9.2.13 QK\_ISR\_EXIT

```
#define QK_ISR_EXIT()
```

**Value:**

```
do {
    QF_INT_DISABLE();
    --QK_priv_.intNest;
    if (QK_priv_.intNest == 0U) {
        if (QK_sched_() != 0U) {
            QK_activate_();
        }
    }
    QF_INT_ENABLE();
} while (false)
```

Define the ISR exit sequence

### 15.9.2.14 QXK\_ISR\_CONTEXT\_

```
#define QXK_ISR_CONTEXT_()
```

**Value:**

```
(QXK_get_IPSR() != 0U)
```

Check if the code executes in the ISR context.

### 15.9.2.15 QXK\_CONTEXT\_SWITCH\_

```
#define QXK_CONTEXT_SWITCH_()
```

**Value:**

```
(trigPendSV())
```

Trigger context switch (used internally in [QXK](#) only)

### 15.9.2.16 QXK\_ISR\_ENTRY

```
#define QXK_ISR_ENTRY()
```

**Value:**

```
((void)0)
```

Define the ISR entry sequence.

### 15.9.2.17 QXK\_ISR\_EXIT

```
#define QXK_ISR_EXIT()
```

**Value:**

```

do {
    QF_INT_DISABLE();
    if (QXK_sched_() != 0U) {
        *Q_UINT2PTR_CAST(uint32_t, 0xE000ED04U) = (1U << 28U);
    }
    QF_INT_ENABLE();
    QXK_ARM_ERRATUM_838869();
} while (false)

```

Define the ISR exit sequence.

### 15.9.3 Typedef Documentation

#### 15.9.3.1 crit\_stat\_t

```
typedef unsigned int crit_stat_t
```

### 15.9.4 Function Documentation

#### 15.9.4.1 critEntry()

```
crit_stat_t critEntry (
    void )
```

#### 15.9.4.2 critExit()

```
void critExit (
    crit_stat_t stat)
```

## 15.10 qs.dox File Reference

## 15.11 qs\_pkg.dox File Reference

## 15.12 qs\_port.h File Reference

Sample QS/C port.

```
#include "qp_port.h"
#include "qs.h"
```

### Macros

- #define QS\_OBJ\_PTR\_SIZE 4U  
*object pointer size in bytes (depends on the CPU)*
- #define QS\_FUN\_PTR\_SIZE 4U  
*function pointer size in bytes (depends on the CPU)*

### 15.12.1 Detailed Description

Sample QS/C port.

#### Details

This is just an example of a [QS](#) port for a 32-bit CPU. Other specific [QS](#) ports will define the [QS](#) facilities differently.

## Remarks

[QS](#) might be used with or without the other QP Framework components, in which case the separate definitions of the macros [QS\\_CRIT\\_STAT](#), [QS\\_CRIT\\_ENTRY\(\)](#), and [QS\\_CRIT\\_EXIT\(\)](#) are needed. In this sample port [QS](#) is configured to be used with the other QP component, by simply including "qp\_port.h" before "qs.h".

## 15.12.2 Macro Definition Documentation

### 15.12.2.1 QS\_OBJ\_PTR\_SIZE

```
#define QS_OBJ_PTR_SIZE 4U  
object pointer size in bytes (depends on the CPU)
```

#### Details

- 2U => 2 bytes
- 4U => 4 bytes (default)
- 8U => 8 bytes

#### Note

[QS\\_OBJ\\_PTR\\_SIZE](#) depends on the CPU/Compiler and is defined in [qs\\_port.h](#). It cannot be overridden in [qp\\_config.h](#).

### 15.12.2.2 QS\_FUN\_PTR\_SIZE

```
#define QS_FUN_PTR_SIZE 4U  
function pointer size in bytes (depends on the CPU)
```

#### Details

- 2U => 2 bytes
- 4U => 4 bytes (default)
- 8U => 8 bytes

**Note**

`QS_FUN_PTR_SIZE` depends on the CPU/Compiler and is defined in `qs_port.h`. It cannot be overridden in `qp_config.h`.

**15.13 qsafe.dox File Reference****15.14 qv.dox File Reference****15.15 qxk.dox File Reference****15.16 sas-qp.dox File Reference****15.17 sds-qp.dox File Reference****15.18 srs-qp.dox File Reference****15.19 api.dox File Reference****15.20 dir.dox File Reference****15.21 help.dox File Reference****15.22 qp-exa.dox File Reference****15.23 qp-gs.dox File Reference****15.24 qp-main.dox File Reference****15.25 qp-ports.dox File Reference****15.26 qequeue.h File Reference**

QP native platform-independent thread-safe event queue interface.

**Classes**

- class `QEQueue`  
*Native QF Event Queue.*

**Typedefs**

- typedef `uint16_t QEQueueCtr`

**15.26.1 Detailed Description**

QP native platform-independent thread-safe event queue interface.

### Backward Traceability

- DVP\_QP\_MC4\_D04\_08

## 15.26.2 Typedef Documentation

### 15.26.2.1 QEQueueCtr

`typedef uint16_t QEQueueCtr`

The data type to store the ring-buffer counters based on the macro `QF_EQQUEUE_CTR_SIZE`.

#### Details

The dynamic range of this data type determines the maximum length of the ring buffer managed by the native `QF` event queue.

## 15.27 qk.h File Reference

QK/C (preemptive non-blocking kernel) platform-independent public interface.

### Classes

- class `QK`  
`QK` preemptive non-blocking kernel (`QK` namespace emulated as a "class" in C)
- class `QK_Attr`  
*Private attributes of the `QK` kernel.*

### Macros

- `#define QF_SCHED_STAT_ QSchedStatus lockStat_;`
- `#define QF_SCHED_LOCK_(ceil_)`
- `#define QF_SCHED_UNLOCK_()`
- `#define QACTIVE_EQQUEUE_WAIT_(me_)`
- `#define QACTIVE_EQQUEUE_SIGNAL_(me_)`
- `#define QF_EPOOL_TYPE_ QMPool`
- `#define QF_EPOOL_INIT_(p_, poolSto_, poolSize_, evtSize_)`
- `#define QF_EPOOL_EVENT_SIZE_(p_)`
- `#define QF_EPOOL_GET_(p_, e_, m_, qsld_)`
- `#define QF_EPOOL_PUT_(p_, e_, qsld_)`

### Typedefs

- `typedef uint_fast8_t QSchedStatus`

## 15.27.1 Detailed Description

QK/C (preemptive non-blocking kernel) platform-independent public interface.

### Backward Traceability

- DVP\_QP\_MC4\_D04\_08

## 15.27.2 Macro Definition Documentation

### 15.27.2.1 QF\_SCHED\_STAT\_

```
#define QF_SCHED_STAT_ QSchedStatus lockStat_;
```

### 15.27.2.2 QF\_SCHED\_LOCK\_

```
#define QF_SCHED_LOCK_(
    ceil_)
```

**Value:**

```
do { \
if (QK_ISR_CONTEXT_()) { \
    lockStat_ = 0xFFU; \
} else { \
    lockStat_ = QK_schedLock((ceil_)); \
} \
} while (false)
```

### 15.27.2.3 QF\_SCHED\_UNLOCK\_

```
#define QF_SCHED_UNLOCK_()
```

**Value:**

```
do { \
if (lockStat_ != 0xFFU) { \
    QK_schedUnlock(lockStat_); \
} \
} while (false)
```

### 15.27.2.4 QACTIVE\_EQUEUE\_WAIT\_

```
#define QACTIVE_EQUEUE_WAIT_(
    me_)
```

**Value:**

```
((void) 0)
```

### 15.27.2.5 QACTIVE\_EQUEUE\_SIGNAL\_

```
#define QACTIVE_EQUEUE_SIGNAL_(
    me_)
```

**Value:**

```
do { \
QPSet_insert(&QK_priv_.readySet, (uint_fast8_t)(me_)->prio); \
if (!QK_ISR_CONTEXT_()) { \
    if (QK_sched_() != 0U) { \
        QK_activate_(); \
    } \
} \
} while (false)
```

### 15.27.2.6 QF\_EPOOL\_TYPE\_

```
#define QF_EPOOL_TYPE_ QMPool
```

### 15.27.2.7 QF\_EPOOL\_INIT\_

```
#define QF_EPOOL_INIT_(
    p_,
    poolSto_,
    poolSize_,
    evtSize_)
```

**Value:**

```
(QMPool_init(&(p_), (poolSto_), (poolSize_), (evtSize_)))
```

### 15.27.2.8 QF\_EPOOL\_EVENT\_SIZE\_

```
#define QF_EPOOL_EVENT_SIZE_(
    p_)
```

**Value:**

```
((uint_fast16_t)(p_) .blockSize)
```

### 15.27.2.9 QF\_EPOOL\_GET\_

```
#define QF_EPOOL_GET_(
    p_,
    e_,
    m_,
    qsId_)
```

**Value:**

```
((e_) = (QEvt *)QMPool_get(&(p_), (m_), (qsId_)))
```

### 15.27.2.10 QF\_EPOOL\_PUT\_

```
#define QF_EPOOL_PUT_(
    p_,
    e_,
    qsId_)
```

**Value:**

```
(QMPool_put(&(p_), (e_), (qsId_)))
```

## 15.27.3 Typedef Documentation

### 15.27.3.1 QSchedStatus

```
typedef uint_fast8_t QSchedStatus
```

The scheduler lock status for QK\_schedLock() and QK\_schedUnlock()

The scheduler lock status for QK::schedLock() and QK::schedUnlock()

## 15.28 qmpool.h File Reference

QP native platform-independent memory pool [QMPool](#) interface.

### Classes

- class [QMPool](#)

*Native QF Memory Pool.*

### Macros

- #define [QF\\_MPOOL\\_EL](#)(evType\_)

*Memory pool element to allocate correctly aligned storage for [QMPool](#) class.*

### TypeDefs

- typedef uint16\_t [QMPoolSize](#)

*The data type to store the block-size based on the macro [QF\\_MPOOL\\_SIZ\\_SIZE](#).*

- typedef uint16\_t [QMPoolCtr](#)

*The data type to store the block-counter based on the macro [QF\\_MPOOL\\_CTR\\_SIZE](#).*

### 15.28.1 Detailed Description

QP native platform-independent memory pool [QMPool](#) interface.

### Backward Traceability

- DVP\_QP\_MC4\_D04\_08: *Directive 4.8(Advisory): If a pointer to a structure or union is never dereferenced within a translation unit then the implementation of the object should be hidden*

## 15.28.2 Macro Definition Documentation

### 15.28.2.1 QF\_MPOOL\_EL

```
#define QF_MPOOL_EL(
    evType_)
```

#### Value:

```
struct {
    void * sto_[((sizeof(evType_) - 1U) / sizeof(void *)) + \
                (sizeof(evType_) < (2U * sizeof(void *)) ? 2U : 1U)]; \
}
```

Memory pool element to allocate correctly aligned storage for [QMPool](#) class.

#### Parameters

in	<code>evType_</code>	event type (name of the subclass of <a href="#">QEvt</a> )
----	----------------------	--

### Backward Traceability

- DVP\_QP\_MC4\_D04\_09A: *Directive 4.9(Advisory): A function should be used in preference to a function-like macro where they are interchangeable (FALSE-POSITIVE diagnosis)*

#### Usage

```
static QF_MPOOL_EL(QEvt) smallPoolSto[CONFIG_QPC_SMALL_POOL_SIZE];
typedef struct {
    QEvt super;
    uint32_t data[CONFIG_QPC_MEDIUM_POOL_ENTRY_SIZE];
} mediumPool;
static QF_MPOOL_EL(mediumPool) mediumPoolSto[CONFIG_QPC_MEDIUM_POOL_SIZE];
typedef struct {
    QEvt super;
    uint32_t data[CONFIG_QPC_LARGE_POOL_ENTRY_SIZE];
} largePool;
static QF_MPOOL_EL(largePool) largePoolSto[CONFIG_QPC_LARGE_POOL_SIZE];
.

int_t main() {
    .
    // initialize Event Memory Pools...
    QF_poolInit(smallPoolSto, sizeof(smallPoolSto), sizeof(smallPoolSto[0]));
    QF_poolInit(mediumPoolSto, sizeof(mediumPoolSto), sizeof(mediumPoolSto[0]));
    QF_poolInit(largePoolSto, sizeof(largePoolSto), sizeof(largePoolSto[0]));
    .
}
```

## 15.28.3 Typedef Documentation

### 15.28.3.1 QMPoolSize

```
typedef uint16_t QMPoolSize
```

The data type to store the block-size based on the macro [QF\\_MPOOL\\_SIZ\\_SIZE](#).

#### Details

The dynamic range of this data type determines the maximum size of blocks that can be managed by the native [QF](#) event pool.

### 15.28.3.2 QMPoolCtr

`typedef uint16_t QMPoolCtr`

The data type to store the block-counter based on the macro `QF_MPOOL_CTR_SIZE`.

#### Details

The dynamic range of this data type determines the maximum number of blocks that can be stored in the pool.

## 15.29 qp.h File Reference

QP/C platform-independent public interface.

#### Classes

- class [QEvt](#)  
*Event class.*
- struct [QMState](#)  
*State object for the [QMsm](#) class (QM State Machine)*
- struct [QMTranActTable](#)  
*Transition-Action Table for the [QMsm](#) State Machine.*
- union [QAsmAttr](#)  
*Attribute of for the [QAsm](#) class (Abstract State Machine)*
- class [QAsm](#)  
*Abstract State Machine class (state machine interface)*
- struct [QAsmVtable](#)  
*Virtual table for the [QAsm](#) class.*
- class [QHsm](#)  
*Hierarchical State Machine class (QHsm-style state machine implementation strategy)*
- class [QMsm](#)  
*Hierarchical State Machine class (QMsm-style state machine implementation strategy)*
- class [QPSet](#)  
*Set of Active Objects of up to [QF\\_MAX\\_ACTIVE](#) elements.*
- struct [QSubscrList](#)  
*Subscriber List (for publish-subscribe)*
- class [QActive](#)  
*Active object class (based on the [QHsm](#) implementation strategy)*
- class [QMActive](#)  
*Active object class (based on [QMsm](#) implementation strategy)*
- class [QTimeEvt](#)  
*Time Event class.*
- class [QTicker](#)  
*"Ticker" Active Object class*

## Macros

- `#define QP_VERSION_STR "8.0.3"`  
*Version string complying with Semantic Versioning*
- `#define QP_VERSION 803U`  
*Version number for internal use (must correspond to QP\_VERSION\_STR)*
- `#define QP_RELEASE 0x6ABEE96CU`  
*Encrypted current QP release for internal use (7.3.5 on 2024-05-31)*
- `#define Q_UNUSED_PAR(par_)`  
*Helper macro to clearly mark unused parameters of functions.*
- `#define Q_DIM(array_)`
- `#define Q_UINT2PTR_CAST(type_, uint_)`  
*Perform cast from unsigned integer uint\_ to pointer of type type\_*
- `#define QEVT_INITIALIZER(sig_)`  
*Initializer for immutable (constant) QEvt instances.*
- `#define QEVT_DYNAMIC ((uint8_t)0)`  
*dummy parameter for dynamic event initialization (see QEvt::QEvt\_init())*
- `#define Q_EVT_CAST(class_)`  
*Perform downcast of an event onto a subclass of QEvt class\_*
- `#define Q_STATE_CAST(handler_)`  
*Perform cast to QStateHandler.*
- `#define Q_ACTION_CAST(action_)`  
*Perform cast to QActionHandler.*
- `#define Q_ACTION_NULL ((QActionHandler)0)`  
*Macro to provide strictly-typed zero-action to terminate action lists in the transition-action-tables.*
- `#define Q_RET_SUPER ((QState)0U)`
- `#define Q_RET_UNHANDLED ((QState)1U)`
- `#define Q_RET_HANDLED ((QState)2U)`
- `#define Q_RET_IGNORED ((QState)3U)`
- `#define Q_RET_ENTRY ((QState)4U)`
- `#define Q_RET_EXIT ((QState)5U)`
- `#define Q_RET_NULL ((QState)6U)`
- `#define Q_RET_TRAN ((QState)7U)`
- `#define Q_RET_TRAN_INIT ((QState)8U)`
- `#define Q_RET_TRAN_HIST ((QState)9U)`
- `#define Q_EMPTY_SIG ((QSignal)0U)`  
*reserved signal sent to state handler to execute the default case*
- `#define Q_ENTRY_SIG ((QSignal)1U)`  
*reserved signal sent to state handler to execute the entry case*
- `#define Q_EXIT_SIG ((QSignal)2U)`  
*reserved signal sent to state handler to execute the exit case*
- `#define Q_INIT_SIG ((QSignal)3U)`  
*reserved signal sent to state handler to execute the initial transition*
- `#define Q_USER_SIG ((enum_t)4)`  
*offset for the user signals (QP Application)*
- `#define QASM_INIT(me_, par_, qsld_)`  
*QVirtual call to the top-most initial transition in a HSM.*
- `#define QASM_DISPATCH(me_, e_, qsld_)`

- `#define QASM_IS_IN(me_, stateHndl_)`  
*Virtual call to dispatch an event to a HSM.*
- `#define Q_ASM_UPCAST(ptr_)`  
*Virtual call to check whether a SM is in a given state.*
- `#define Q_HSM_UPCAST(ptr_)`  
*Perform upcasting from a subclass of `QAsm` to the base class `QAsm`.*
- `#define Q_TRAN(target_)`  
*Take transition to the specified `target_` state.*
- `#define Q_TRAN_HIST(hist_)`  
*Take transition to the specified history of a given state. Applicable only to HSMs.*
- `#define Q_SUPER(super_)`  
*Designates the superstate of a given state. Applicable only to `QHsm` subclasses.*
- `#define Q_HANDLED()`  
*Indicate that an action has been "handled". Applies to entry/exit actions and to internal transitions.*
- `#define Q_UNHANDLED()`  
*Indicate that an internal transition has been "unhandled" due to a guard condition.*
- `#define Q_MSM_UPCAST(ptr_)`  
*Perform upcasting from a subclass of `QMsm` to the base class `QMsm`.*
- `#define QM_ENTRY(state_)`  
*Macro to call in a QM action-handler when it executes an entry action. Applicable only to `QMsm` subclasses.*
- `#define QM_EXIT(state_)`  
*Macro to call in a QM action-handler when it executes an exit action. Applicable only to `QMsm` subclasses.*
- `#define QM_TRAN(tatbl_)`  
*Macro to call in a QM state-handler when it executes a regular transition. Applicable only to `QMsm` subclasses.*
- `#define QM_TRAN_INIT(tatbl_)`  
*Macro to call in a QM state-handler when it executes an initial transition. Applicable only to `QMsm` subclasses.*
- `#define QM_TRAN_HIST(history_, tatbl_)`  
*Macro to call in a QM state-handler when it executes a transition to history. Applicable only to `QMsm` subclasses.*
- `#define QM_HANDLED()`  
*Macro to call in a QM state-handler when it handled an event. Applicable only to `QMsm` subclasses.*
- `#define QM_UNHANDLED()`  
*Indicate that an internal transition has been "unhandled" due to a guard condition. Applicable only to `QMsm` subclasses.*
- `#define QM_SUPER()`  
*Macro to call in a QM state-handler when it designates the superstate to handle an event. Applicable only to ::QMSMs.*
- `#define QM_STATE_NULL ((QMState *)0)`
- `#define Q_PRIO(prio_, pthre_)`  
*Create a `QPrioSpec` object to specify priority of an AO or a thread.*
- `#define QF_NO_MARGIN ((uint_fast16_t)0xFFFFU)`  
*Special value of margin that causes asserting failure in case event allocation or event posting fails.*
- `#define Q_NEW(evtT_, sig_, ...)`  
*Allocate a mutable (dynamic) event.*
- `#define Q_NEW_X(evtT_, margin_, sig_, ...)`  
*Non-asserting allocate a mutable (dynamic) event.*
- `#define Q_NEW_REF(evtRef_, evtT_)`  
*Create a new reference of the current event e*
- `#define Q_DELETE_REF(evtRef_)`

- `#define QACTIVE_POST(me_, e_, sender_)`  
*Delete the event reference.*
- `#define QACTIVE_POST_X(me_, e_, margin_, sender_)`  
*Invoke the direct event posting facility [QActive\\_post\\_\(\)](#).*
- `#define QACTIVE_PUBLISH(e_, sender_)`  
*Publish an event to all subscriber Active Objects.*
- `#define QTIMEEV_TICK_X(tickRate_, sender_)`  
*Invoke the system clock tick processing [QTimeEvt\\_tick\\_\(\)](#)*
- `#define QTICKER_TRIG(ticker_, sender_)`  
*Asynchronously trigger the [QTicker](#) AO to perform tick processing.*
- `#define QACTIVE_POST_LIFO(me_, e_)`  
*Post an event to an active object using the Last-In-First-Out (LIFO) policy.*
- `#define QTIMEEV_TICK(sender_)`  
*Invoke the system clock tick processing for tick rate 0.*
- `#define QF_CRIT_EXIT_NOP()`  
*No-operation for exiting a critical section.*

## Typedefs

- `typedef int int_t`  
*Alias for assertion-ID numbers in QP assertions and return from [QF\\_run\(\)](#)*
- `typedef int enum_t`
- `typedef uint16_t QSignal`  
*The signal of event [QEvt](#).*
- `typedef QEvt const * QEvtPtr`  
*Pointer to const event instances passed around in QP Framework.*
- `typedef uint_fast8_t QState`  
*Type returned from state-handler functions.*
- `typedef QState(* QStateHandler) (void *const me, QEvt const *const e)`  
*Pointer to a state-handler function.*
- `typedef QState(* QActionHandler) (void *const me)`  
*Pointer to an action-handler function.*
- `typedef void(* QXThreadHandler) (struct QXThread *const me)`  
*Pointer to an extended-thread handler function.*
- `typedef uint16_t QPrioSpec`  
*Priority specification for Active Objects in QP.*
- `typedef uint32_t QTimeEvtCtr`  
*Data type to store the block-size defined based on the macro [QF\\_TIMEEVT\\_CTR\\_SIZE](#).*
- `typedef uint32_t QPSetBits`  
*Bitmask for the internal representation of [QPSet](#) elements.*

## Functions

- `uint_fast8_t QF_LOG2 (QPSetBits const bitmask)`  
*Log-base-2 calculation when hardware acceleration is NOT provided ([QF\\_LOG2](#) not defined)*

## Variables

- char const QP\_versionStr [24]

### 15.29.1 Detailed Description

QP/C platform-independent public interface.

#### Backward Traceability

- DVP\_QP\_MC4\_D04\_08

### 15.29.2 Macro Definition Documentation

#### 15.29.2.1 QP\_VERSION\_STR

```
#define QP_VERSION_STR "8.0.3"
```

Version string complying with [Semantic Versioning](#)

##### Details

QP\_VERSION\_STR is a zero-terminated version string in the form MAJOR.MINOR.PATCH compliant with [Semantic Versioning](#).

##### Examples:

- 7.3.5-rc.1 — version 7.3.5 release-candidate-1
- 7.3.5 — version 7.3.5 (final release)

#### 15.29.2.2 QP\_VERSION

```
#define QP_VERSION 803U
```

Version number for internal use (must correspond to QP\_VERSION\_STR)

##### Details

QP\_VERSION is an unsigned integer constant of the form MAJOR|MINOR|PATCH corresponding to QP\_VERSION\_STR, with digits merged together. For example QP\_VERSION\_STR == "7.3.5-rc.1" results in QP\_VERSION == 735.

#### 15.29.2.3 QP\_RELEASE

```
#define QP_RELEASE 0x6ABEE96CU
```

Encrypted current QP release for internal use (7.3.5 on 2024-05-31)

##### Details

QP\_RELEASE contains information about QP\_VERSION (e.g., 735) and release date (e.g., 2023-06-30).

#### 15.29.2.4 Q\_UNUSED\_PAR

```
#define Q_UNUSED_PAR (
```

$$\quad \quad \quad par_{\_})$$

##### Value:

```
((void) (par_{\_}))
```

Helper macro to clearly mark unused parameters of functions.

### 15.29.2.5 Q\_DIM

```
#define Q_DIM(
    array_)

Value:
(sizeof(array_) / sizeof((array_[0U])))
```

### 15.29.2.6 Q\_UINT2PTR\_CAST

```
#define Q_UINT2PTR_CAST(
    type_,
    uint_)
```

#### **Value:**

```
((type_*)(uint_))
```

Perform cast from unsigned integer `uint_` to pointer of type `type_`

#### Details

This macro encapsulates the cast to `(type_*)`, which QP ports or application might use to access embedded hardware registers. Such uses can trigger PC-Lint "Note 923: cast from int to pointer" and this macro helps to encapsulate this deviation.

### 15.29.2.7 QEVT\_INITIALIZER

```
#define QEVT_INITIALIZER(
    sig_)
```

#### **Value:**

```
{ (QSignal)(sig_), 0x00U, 0xE0U }
```

Initializer for immutable (constant) `QEvt` instances.

#### Details

This macro encapsulates the casting of enumerated signals and the generation of the Duplicate Inverse Storage for integrity check in the `QEvt::evtTag_` attribute.

#### Usage

```
static QEvt const testEvt = QEVT_INITIALIZER(TEST_SIG);
. . .
QACTIVE_POST(AO_Xyz, &testEvt, 0U);
. . .

typedef struct {
    QEvt super; // inherit QEvt
    uint8_t id; // id of the pressed button
} ButtonPressEvt;
. . .
// immutable button-press event
static ButtonWorkEvt const pressEvt = {
    QEVT_INITIALIZER(BUTTON_PRESSED_SIG),
    .id = 123U
};
. . .
QACTIVE_POST(AO_Button, &pressEvt.super, &l_SysTick_Handler);
```

### 15.29.2.8 QEVT\_DYNAMIC

```
#define QEVT_DYNAMIC ((uint8_t)0)
dummy parameter for dynamic event initialization (see QEvt::QEvt_init())
```

### 15.29.2.9 Q\_EVT\_CAST

```
#define Q_EVT_CAST(
    class_)
```

**Value:**

```
((class_ const *) (e))
```

Perform downcast of an event onto a subclass of [QEvt](#) `class_`

**Details**

This macro encapsulates the downcast of [QEvt](#) pointers, which violates MISRA-C:2023 Rule 11.3(R). This macro helps to localize this deviation.

**Parameters**

<code>class_</code>	a subclass of <a href="#">QEvt</a>
---------------------	------------------------------------

**Backward Traceability**

- [SRS\\_QP\\_EVT\\_30](#): *QP Framework shall allow Application to create event instances with [Parameters](#) defined by the Application*
- [DVP\\_QP\\_MC4\\_D04\\_09A](#): *Directive 4.9(Advisory): A function should be used in preference to a function-like macro where they are interchangeable (FALSE-POSITIVE diagnosis)*
- [DVP\\_QP\\_MC4\\_R11\\_03B](#): *Rule 11.3(Required): A cast shall not be performed between a pointer to object type and a pointer to a different object type ([object-oriented downcast](#))*
- [DVP-QP-PCLP-826](#): *Suspicious pointer-to-pointer conversion (area too small)*

### 15.29.2.10 Q\_STATE\_CAST

```
#define Q_STATE_CAST(
    handler_)
```

**Value:**

```
((QStateHandler) (handler_))
```

Perform cast to [QStateHandler](#).

**Details**

This macro encapsulates the cast of a specific state handler function pointer to [QStateHandler](#), which violates MISRA:C-2023 Rule 11.1(R). This macro helps to localize this deviation.

**Backward Traceability**

- [DVP\\_QP\\_MC4\\_D04\\_09A](#): *Directive 4.9(Advisory): A function should be used in preference to a function-like macro where they are interchangeable (FALSE-POSITIVE diagnosis)*
- [DVP\\_QP\\_MC4\\_R11\\_01](#): *Rule 11.1(Required): Conversions shall not be performed between a pointer to a function and any other type*

**Usage**

```
void Philo_ctor(Philo * const me) {
    QActive_ctor(&me->super, Q_STATE_CAST(&Philo_initial)); // <===
    QTimeEvt_ctorX(&me->timeEvt, me, (enum_t)TIMEOUT_SIG, 0U);
}
```

### 15.29.2.11 Q\_ACTION\_CAST

```
#define Q_ACTION_CAST(  
    action_)
```

**Value:**

```
((QActionHandler) (action_))  
Perform cast to QActionHandler.
```

**Details**

This macro encapsulates the cast of a specific action handler function pointer to [QActionHandler](#), which violates MISRA:C-2023 Rule 11.1(R). This macro helps to localize this deviation.

**Backward Traceability**

- DVP\_QP\_MC4\_D04\_09A: *Directive 4.9(Advisory): A function should be used in preference to a function-like macro where they are interchangeable (FALSE-POSITIVE diagnosis)*
- DVP\_QP\_MC4\_R11\_01: *Rule 11.1(Required): Conversions shall not be performed between a pointer to a function and any other type*

### 15.29.2.12 Q\_ACTION\_NULL

```
#define Q_ACTION_NULL ((QActionHandler) 0)
```

Macro to provide strictly-typed zero-action to terminate action lists in the transition-action-tables.

### 15.29.2.13 Q\_RET\_SUPER

```
#define Q_RET_SUPER ((QState) 0U)
```

### 15.29.2.14 Q\_RET\_UNHANDLED

```
#define Q_RET_UNHANDLED ((QState) 1U)
```

### 15.29.2.15 Q\_RET\_HANDLED

```
#define Q_RET_HANDLED ((QState) 2U)
```

### 15.29.2.16 Q\_RET\_IGNORED

```
#define Q_RET_IGNORED ((QState) 3U)
```

### 15.29.2.17 Q\_RET\_ENTRY

```
#define Q_RET_ENTRY ((QState) 4U)
```

### 15.29.2.18 Q\_RET\_EXIT

```
#define Q_RET_EXIT ((QState) 5U)
```

### 15.29.2.19 Q\_RET\_NULL

```
#define Q_RET_NULL ((QState) 6U)
```

### 15.29.2.20 Q\_RET\_TRAN

```
#define Q_RET_TRAN ((QState) 7U)
```

### 15.29.2.21 Q\_RET\_TRAN\_INIT

```
#define Q_RET_TRAN_INIT ((QState)8U)
```

### 15.29.2.22 Q\_RET\_TRAN\_HIST

```
#define Q_RET_TRAN_HIST ((QState)9U)
```

### 15.29.2.23 Q\_EMPTY\_SIG

```
#define Q_EMPTY_SIG ((QSignal)0U)
reserved signal sent to state handler to execute the default case)
```

### 15.29.2.24 Q\_ENTRY\_SIG

```
#define Q_ENTRY_SIG ((QSignal)1U)
reserved signal sent to state handler to execute the entry case)
```

### 15.29.2.25 Q\_EXIT\_SIG

```
#define Q_EXIT_SIG ((QSignal)2U)
reserved signal sent to state handler to execute the exit case)
```

### 15.29.2.26 Q\_INIT\_SIG

```
#define Q_INIT_SIG ((QSignal)3U)
reserved signal sent to state handler to execute the initial transition)
```

### 15.29.2.27 Q\_USER\_SIG

```
#define Q_USER_SIG ((enum_t)4)
offset for the user signals (QP Application))
```

#### Usage

The following example illustrates the use of Q\_USER\_SIG:

```
enum AppSignals {
    SOMETHING_HAPPENED_SIG = Q_USER_SIG,
    SOMETHING_ELSE_SIG,
    TIMEOUT_SIG,
    .
    .
};
```

### 15.29.2.28 QASM\_INIT

```
#define QASM_INIT(
    me_,
    par_,
    qsId_)
```

#### Value:

```
(*((QAsm *) (me_))->vptr->init)((QAsm *) (me_), (par_), (qsId_))
QVirtual call to the top-most initial transition in a HSM.
```

#### Parameters

in,out	<i>me_</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>par_</i>	pointer the optional initialization parameter
in	<i>qsId_</i>	QS local filter ID (used only when Q_SPY is defined)

**Note**

Must be called only ONCE after the SM "constructor".

**Backward Traceability**

- **SRS\_QP\_SM\_38:** *All State Machine Implementation Strategies provided by QP Framework shall support top-most initial transition that shall be explicitly triggered independently from instantiation of the state machine object*

**Usage**

The following example illustrates how to initialize a SM, and dispatch events to it:

```
#include "qpc.h"      // QP/C public interface
#include "calc.h"     // Calc derived from QHsm

Q_DEFINE_THIS_FILE

static Calc Calc_inst; // an instance of Calc SM

int main() {
    Calc_ctor(&Calc_inst); // Calc "constructor" invokes QHsm_ctor()

    QASM_INIT(&Calc_inst, (QEvt *)0); // <== initial transition

    for (;;) { // event loop
        QEvt e = QEVT_INITIALIZER(MY_SIG);

        // wait for the next event and assign it to the event object e
        . .
        QASM_DISPATCH(&Calc_inst, &e); // dispatch e
    }
    return 0;
}
```

**15.29.2.29 QASM\_DISPATCH**

```
#define QASM_DISPATCH(
    me_,
    e_,
    qsId_)
```

**Value:**

```
(* ((QAsm *) (me_)) -> vptr -> dispatch) ((QAsm *) (me_), (e_), (qsId_))
```

Virtual call to dispatch an event to a HSM.

**Details**

Processes one event at a time in Run-to-Completion fashion.

**Parameters**

in,out	<i>me_</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>e_</i>	constant pointer the <a href="#">QEvt</a> or a class derived from <a href="#">QEvt</a> (see <a href="#">SAS_QP_OO</a> )
in	<i>qsId_</i>	<a href="#">QS</a> local filter ID (used only when <a href="#">Q_SPY</a> is defined)

**Note**

Must be called after the "constructor" and after [QASM\\_INIT\(\)](#).

## Backward Traceability

- [SRS\\_QP\\_SM\\_10](#): QP Framework shall support multiple and interchangeable State Machine Implementation Strategies

## Usage

The following example illustrates how to initialize a SM, and dispatch events to it:

```
#include "qpc.h" // QP/C public interface
#include "calc.h" // Calc derived from QHsm

Q_DEFINE_THIS_FILE

static Calc Calc_inst; // an instance of Calc SM

int main() {
    Calc_ctor(&Calc_inst); // Calc "constructor" invokes QHsm_ctor()

    QASM_INIT(&Calc_inst, (QEvt *)0); // initial transition

    for (;;) { // event loop
        QEvt e = QEVT_INITIALIZER(MY_SIG);

        // wait for the next event and assign it to the event object e
        QASM_DISPATCH(&Calc_inst, &e); // <==== dispatch e
    }
    return 0;
}
```

### 15.29.2.30 QASM\_IS\_IN

```
#define QASM_IS_IN(
    me_,
    stateHndl_)
```

#### Value:

```
(*((QAsm *) (me_))->vptr->isIn) ((QAsm *) (me_), (stateHndl_))
```

Virtual call to check whether a SM is in a given state.

#### Details

This macro applies to all QP state machines, that is subclasses of [QHsm](#) and [QMsm](#).

#### Parameters

in,out	<i>me_</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>state_</i>	state handler ( <a href="#">QStateHandler</a> type)

#### Returns

'true' if the SM is in a given state and 'false' otherwise

#### Note

Must be called after the "constructor" and after [QASM\\_INIT\(\)](#).

#### Attention

This macro must be called only on a SM that is in the "stable state configuration". Among others, this means that the state machine cannot call it in the middle of its own transition.

### Backward Traceability

- **SRS\_QP\_SM\_25:** All State Machine Implementation Strategies provided by QP Framework might supply a method for checking if a state machine is in a given state

### Usage

The following example illustrates how to check whether SM is in a given state:

```
stat = QASM_IS_IN(the_hsm, Q_STATE_CAST(&QHsmTst_s11));
```

### 15.29.2.31 Q\_ASM\_UPCAST

```
#define Q_ASM_UPCAST(
    ptr_)
```

#### Value:

```
((QAsm *) (ptr_))
```

Perform upcasting from a subclass of [QAsm](#) to the base class [QAsm](#).

#### Parameters

in, out	<i>ptr_</i>	pointer to subclass of <a href="#">QAsm</a>
---------	-------------	---

#### Returns

The upcasted pointer to the [QAsm](#) base class

#### Remarks

Upcasting from a subclass to superclass is a very frequent and **safe** operation in object-oriented programming and object-oriented languages (such as C++) perform such upcasting automatically. However, OOP is implemented in C just as a set of coding conventions (see [SAS\\_QP\\_OO](#)), and the C compiler does not "know" that certain types are related by inheritance. Therefore for C, the upcast must be performed explicitly.

Unfortunately, pointer casting violates the advisory MISRA-C:2023 Rule 11.3(R). This macro encapsulates this deviation and provides a descriptive name for the reason of this cast.

### Backward Traceability

- DVP\_QP\_MC4\_R11\_03A: *Rule 11.3(Required): A cast shall not be performed between a pointer to object type and a pointer to a different object type (object-oriented upcast) (upcast)*

### 15.29.2.32 Q\_HSM\_UPCAST

```
#define Q_HSM_UPCAST(
    ptr_)
```

#### Value:

```
((QHsm *) (ptr_))
```

Perform upcasting from a subclass of [QHsm](#) to the base class [QHsm](#).

#### Parameters

in, out	<i>ptr_</i>	pointer to subclass of <a href="#">QHsm</a>
---------	-------------	---

**Returns**

The upcasted pointer to the [QHsm](#) base class

**See also**

- [Q\\_ASM\\_UPCAST\(\)](#)

**Backward Traceability**

- DVP\_QP\_MC4\_R11\_03A: *Rule 11.3(Required): A cast shall not be performed between a pointer to object type and a pointer to a different object type (object-oriented upcast) (upcast)*

**15.29.2.33 Q\_TRAN**

```
#define Q_TRAN(
    target_)
```

**Value:**

```
((Q_ASM_UPCAST(me))->temp.fun = Q_STATE_CAST(target_), \
(QState)Q_RET_TRAN)
```

Take transition to the specified `target_` state.

**Backward Traceability**

- DVP\_QP\_MC4\_R11\_01: *Rule 11.1(Required): Conversions shall not be performed between a pointer to a function and any other type*
- DVP\_QP\_MC4\_R12\_03: *Rule 12.3(Advisory): The comma operator should not be used*
- DVP\_QP\_MC4\_R13\_04: *Rule 13.4(Advisory): The result of an assignment operator shall not be used*

**15.29.2.34 Q\_TRAN\_HIST**

```
#define Q_TRAN_HIST(
    hist_)
```

**Value:**

```
((Q_ASM_UPCAST(me))->temp.fun = (hist_), \
(QState)Q_RET_TRAN_HIST)
```

Take transition to the specified history of a given state. Applicable only to HSMs.

**Backward Traceability**

- [SRS\\_QP\\_SM\\_39](#): *All State Machine Implementation Strategies provided by QP Framework should support transitions to history. Both shallow and deep histories shall be supported*
- DVP\_QP\_MC4\_R11\_01: *Rule 11.1(Required): Conversions shall not be performed between a pointer to a function and any other type*
- DVP\_QP\_MC4\_R12\_03: *Rule 12.3(Advisory): The comma operator should not be used*

**Usage**

```
typedef struct {
    QHsm super; // inherit QHsm

    QStateHandler hist_doorClosed; // history of doorClosed
} ToastOven;

QState ToastOven_doorClosed(ToastOven * const me, QEvt const * const e) {
    QState status;
    switch (e->sig) {
        .
        .
        case Q_EXIT_SIG: {
```

```

        me->hist_doorClosed = QHsm_state(&me->super);
        status = Q_HANDLED();
        break;
    }
}
return status;
}

QState ToastOven_doorOpen(ToastOven * const me, QEvt const * const e) {
    QState status;
    switch (e->sig) {
        . .
        case CLOSE_SIG: {
            status = Q_TRAN_HIST(hist_doorClosed); // <===
            break;
        }
        default: {
            status = Q_SUPER(&QHsm_top);
            break;
        }
    }
    return status;
}

```

### 15.29.2.35 Q\_SUPER

```
#define Q_SUPER(
    super_)
```

**Value:**

```
((Q_ASM_UPCAST(me))->temp.fun = Q_STATE_CAST(super_), \
(QState)Q_RET_SUPER)
```

Designates the superstate of a given state. Applicable only to **QHsm** subclasses.

**Backward Traceability**

- **SRS\_QP\_SM\_31: All State Machine Implementation Strategies provided by QP Framework shall support states capable of holding hierarchically nested substates**
- **DVP\_QP\_MC4\_R11\_01: Rule 11.1(Required): Conversions shall not be performed between a pointer to a function and any other type**
- **DVP\_QP\_MC4\_R12\_03: Rule 12.3(Advisory): The comma operator should not be used**
- **DVP\_QP\_MC4\_R13\_04: Rule 13.4(Advisory): The result of an assignment operator shall not be used**

**Usage**

```

QState Blinky_off(Blinky * const me, QEvt const * const e) {
    QState status;
    switch (e->sig) {
        case Q_ENTRY_SIG: {
            BSP_ledOff();
            status = Q_HANDLED();
            break;
        }
        case TIMEOUT_SIG: {
            status = Q_TRAN(&Blinky_on);
            break;
        }
        default: {
            status = Q_SUPER(&QHsm_top); // <===
            break;
        }
        . .
    }
    return status;
}

```

### 15.29.2.36 Q\_HANDLED

```
#define Q_HANDLED()
```

**Value:**

```
((QState)Q_RET_HANDLED)
```

Indicate that an action has been "handled". Applies to entry/exit actions and to internal transitions.

**Backward Traceability**

- [SRS\\_QP\\_SM\\_32](#): All State Machine Implementation Strategies provided by QP Framework shall support entry actions to states
- [SRS\\_QP\\_SM\\_33](#): All State Machine Implementation Strategies provided by QP Framework shall support exit actions from states
- [SRS\\_QP\\_SM\\_36](#): All State Machine Implementation Strategies provided by QP Framework shall support internal transitions in states

### 15.29.2.37 Q\_UNHANDLED

```
#define Q_UNHANDLED()
```

**Value:**

```
((QState)Q_RET_UNHANDLED)
```

Indicate that an internal transition has been "unhandled" due to a guard condition.

**Details**

This macro must be called when a state-handler attempts to handle an event but a guard condition evaluates to 'false' and there is no other explicit way of handling the event. Applicable only to [QHsm](#) subclasses.

### 15.29.2.38 Q\_MSM\_UPCAST

```
#define Q_MSM_UPCAST(
    ptr_)
```

**Value:**

```
((QMsm *) (ptr_))
```

Perform upcasting from a subclass of [QMsm](#) to the base class [QMsm](#).

**Parameters**

in, out	<i>ptr_</i>	pointer to subclass of <a href="#">QMsm</a>
---------	-------------	---

**Returns**

The upcasted pointer to the [QMsm](#) base class

**See also**

- [Q\\_ASM\\_UPCAST\(\)](#)

**Backward Traceability**

- [DVP\\_QP\\_MC4\\_R11\\_03A](#): Rule 11.3(Required): A cast shall not be performed between a pointer to object type and a pointer to a different object type (object-oriented upcast) (upcast)

### 15.29.2.39 QM\_ENTRY

```
#define QM_ENTRY(
    state_)
```

**Value:**

```
((Q_ASM_UPCAST(me))->temp.obj = (state_), (QState)Q_RET_ENTRY)
```

Macro to call in a QM action-handler when it executes an entry action. Applicable only to **QMsm** subclasses.

**Backward Traceability**

- DVP\_QP\_MC4\_R11\_01: *Rule 11.1(Required): Conversions shall not be performed between a pointer to a function and any other type*
- DVP\_QP\_MC4\_R12\_03: *Rule 12.3(Advisory): The comma operator should not be used*
- DVP\_QP\_MC4\_R13\_04: *Rule 13.4(Advisory): The result of an assignment operator shall not be used*

### 15.29.2.40 QM\_EXIT

```
#define QM_EXIT(
    state_)
```

**Value:**

```
((Q_ASM_UPCAST(me))->temp.obj = (state_), (QState)Q_RET_EXIT)
```

Macro to call in a QM action-handler when it executes an exit action. Applicable only to **QMsm** subclasses.

**Backward Traceability**

- DVP\_QP\_MC4\_R11\_01: *Rule 11.1(Required): Conversions shall not be performed between a pointer to a function and any other type*
- DVP\_QP\_MC4\_R12\_03: *Rule 12.3(Advisory): The comma operator should not be used*
- DVP\_QP\_MC4\_R13\_04: *Rule 13.4(Advisory): The result of an assignment operator shall not be used*

### 15.29.2.41 QM\_TRAN

```
#define QM_TRAN(
    tatbl_)
```

**Value:**

```
((Q_ASM_UPCAST(me))->temp.tatbl \
= (struct QMTranActTable const *)(tatbl_), (QState)Q_RET_TRAN)
```

Macro to call in a QM state-handler when it executes a regular transition. Applicable only to **QMsm** subclasses.

**Backward Traceability**

- DVP\_QP\_MC4\_R11\_01: *Rule 11.1(Required): Conversions shall not be performed between a pointer to a function and any other type*
- DVP\_QP\_MC4\_R11\_03A: *Rule 11.3(Required): A cast shall not be performed between a pointer to object type and a pointer to a different object type (object-oriented upcast) (upcast)*
- DVP\_QP\_MC4\_R12\_03: *Rule 12.3(Advisory): The comma operator should not be used*
- DVP\_QP\_MC4\_R13\_04: *Rule 13.4(Advisory): The result of an assignment operator shall not be used*

### 15.29.2.42 QM\_TRAN\_INIT

```
#define QM_TRAN_INIT(
    tatbl_)
```

**Value:**

```
((Q_ASM_UPCAST(me))->temp.tatbl \
= (struct QMTranActTable const *)(tatbl_), (QState)Q_RET_TRAN_INIT)
```

Macro to call in a QM state-handler when it executes an initial transition. Applicable only to **QMsm** subclasses.

### Backward Traceability

- DVP\_QP\_MC4\_R11\_01: *Rule 11.1(Required): Conversions shall not be performed between a pointer to a function and any other type*
- DVP\_QP\_MC4\_R11\_03A: *Rule 11.3(Required): A cast shall not be performed between a pointer to object type and a pointer to a different object type (object-oriented upcast) (upcast)*
- DVP\_QP\_MC4\_R12\_03: *Rule 12.3(Advisory): The comma operator should not be used*
- DVP\_QP\_MC4\_R13\_04: *Rule 13.4(Advisory): The result of an assignment operator shall not be used*

### 15.29.2.43 QM\_TRAN\_HIST

```
#define QM_TRAN_HIST(
    history_,
    tatbl_)
```

#### Value:

```
((((Q_ASM_UPCAST(me))->state.obj = (history_)), \
((Q_ASM_UPCAST(me))->temp.tatbl = \
(struct QMTranActTable const *) (tatbl_))), ((QState)Q_RET_TRAN_HIST))
```

Macro to call in a QM state-handler when it executes a transition to history. Applicable only to [QMsm](#) subclasses.

### Backward Traceability

- DVP\_QP\_MC4\_R11\_01: *Rule 11.1(Required): Conversions shall not be performed between a pointer to a function and any other type*
- DVP\_QP\_MC4\_R12\_03: *Rule 12.3(Advisory): The comma operator should not be used*
- DVP\_QP\_MC4\_R13\_04: *Rule 13.4(Advisory): The result of an assignment operator shall not be used*

### 15.29.2.44 QM\_HANDLED

```
#define QM_HANDLED()
```

#### Value:

```
((QState)Q_RET_HANDLED)
```

Macro to call in a QM state-handler when it handled an event. Applicable only to [QMsm](#) subclasses.

### 15.29.2.45 QM\_UNHANDLED

```
#define QM_UNHANDLED()
```

#### Value:

```
((QState)Q_RET_UNHANDLED)
```

Indicate that an internal transition has been "unhandled" due to a guard condition. Applicable only to [QMsm](#) subclasses.

#### Details

This macro must be called when a state-handler attempts to handle an event but a guard condition evaluates to 'false' and there is no other explicit way of handling the event.

### 15.29.2.46 QM\_SUPER

```
#define QM_SUPER()
```

#### Value:

```
((QState)Q_RET_SUPER)
```

Macro to call in a QM state-handler when it designates the superstate to handle an event. Applicable only to ::QMSMs.

### 15.29.2.47 QM\_STATE\_NULL

```
#define QM_STATE_NULL ((QMState *)0)
```

### 15.29.2.48 Q\_PRIO

```
#define Q_PRIO(          \
    prio_,           \
    pthre_)
```

**Value:**

```
((QPrioSpec)((prio_) | ((pthre_) < 8U)))
```

Create a [QPrioSpec](#) object to specify priority of an AO or a thread.

**Parameters**

in	<i>prio_</i>	QF priority [1..QF_MAX_ACTIVE]
in	<i>pthre_</i>	Preemption threshold [1..QF_MAX_ACTIVE]

**Returns**

The combined 16-bit priority specification (*prio\_* in bits [0..7] and *pthre\_* in bits [8..15]).

### 15.29.2.49 QF\_NO\_MARGIN

```
#define QF_NO_MARGIN ((uint_fast16_t)0xFFFFU)
```

Special value of margin that causes asserting failure in case event allocation or event posting fails.

### 15.29.2.50 Q\_NEW

```
#define Q_NEW(          \
    evtT_,           \
    sig_,           \
    ...)           \
    (evtT_##_init((evtT_ *)QF_newX_((uint_fast16_t)sizeof(evtT_), \ \
                    QF_NO_MARGIN, (sig_)), __VA_ARGS__))
```

Allocate a mutable (dynamic) event.

**Details**

This macro allocates a mutable (dynamic) event and internally asserts that the allocation is successful. This means that the caller does not need to check the returned event pointer for validity because it is guaranteed to be not NULL.

The macro calls the internal [QF](#) function [QF\\_newX\\_\(\)](#) with argument `margin == QF_NO_MARGIN`, which causes an assertion failure when the event cannot be successfully allocated.

**Parameters**

in	<i>evtT_</i>	event type (class name) of the event to allocate
in	<i>sig_</i>	signal to assign to the newly allocated event

**Returns**

A valid event pointer cast to the type *evtT\_*.

### Attention

When the configuration macro `QEVT_PAR_INIT` is defined, the macro `Q_NEW()` becomes **variadic** and takes additional parameters, which are passed to the event initialization. In that case, *all your custom event classes* (subclasses of `QEvt`) must provide the `init()` member functions. This `init()` function needs to take at least one parameter and must return the event pointer (`me`).

### Backward Traceability

- `SDS_QP_QF`: *QF Active Object Framework*
- `DVP_QP_MC4_R10_03`: *Rule 10.3(Required): The value of an expression shall not be assigned to an object with a narrower essential type or a different essential type category*
- `DVP_QP_MC4_R11_03B`: *Rule 11.3(Required): A cast shall not be performed between a pointer to object type and a pointer to a different object type (object-oriented downcast)*
- `DVP-QP-PCLP-826`: *Suspicious pointer-to-pointer conversion (area too small)*

### Usage

The following example illustrates **non-variadic** version of `Q_NEW()` (when `QEVT_PAR_INIT` is **NOT** defined):

```
// event without parameters
QEvt *myEvt = Q_NEW(QEvt, MY_SIG); // <===
// post or publish the event...

// event with parameter(s) (event parameter(s) initialized separately)
KeypressEvt *ke = Q_NEW(KeypressEvt, KEYPRESS_SIG); // <===
ke->keyId = keyId;
// post or publish the event...
```

The following example illustrates **variadic** version of `Q_NEW()` and the **RAll** principle (Resource Acquisition Is Initialization) (when `QEVT_PAR_INIT` **IS** defined) :

```
// event without parameters (QEVT_DYNAMIC passed to QEvt_init())
QEvt *myEvt = Q_NEW(QEvt, MY_SIG, QEVT_DYNAMIC); // <===
// post or publish the event...

// event with parameters
typedef struct {
    QEvt super; // <== inherit QEvt
    // event parameters follow...
    uint8_t keyId; // ID of the key pressed
} KeypressEvt;

// when QEVT_PAR_INIT is defined, the init() member function must be provided
static inline KeypressEvt * KeypressEvt_init(KeypressEvt * const me,
                                             uint8_t keyId)
{
    if (me != (KeypressEvt *)0) { // might be NULL if allocation failed
        me->keyId = keyId;
    }
    return me;
}

// event with parameters (keyId passed to KeypressEvt_init())
KeypressEvt const *ke = Q_NEW(KeypressEvt, KEYPRESS_SIG, keyId); // <===
// . . .
```

### 15.29.2.51 Q\_NEW\_X

```
#define Q_NEW_X(
    evtt_,
    margin_,
    sig_,
    ...
)
```

#### Value:

```
(evtt_ ## _init((evtt_ *)QF_newX_(uint_fast16_t)sizeof(evtt_), \
                (margin_), (sig_), __VA_ARGS__))
```

Non-asserting allocate a mutable (dynamic) event.

## Details

This macro allocates a mutable (dynamic) event, but only if the corresponding event pool has at least of margin of free events left. If the event pool has insufficient number of events left, the macro returns NULL. The caller of this macro is responsible for checking the returned event pointer for NULL.

## Parameters

in	<i>evtT_</i>	event type (class name) of the event to allocate
in	<i>margin_</i>	number of events that must remain available in the given pool after this allocation. The special value <a href="#">QF_NO_MARGIN</a> causes asserting failure in case event allocation fails.
in	<i>sig_</i>	signal to assign to the newly allocated event

## Returns

An event pointer cast to the type *evtT\_* or NULL if the event cannot be allocated with the specified *margin* of events still left in the event pool.

## Attention

When the configuration macro [QEVT\\_PAR\\_INIT](#) is defined, the macro [Q\\_NEW\(\)](#) becomes **variadic** and takes additional parameters, which are passed to the event initialization. In that case, *all your custom event classes* (subclasses of [QEvt](#)) must to provide the [init\(\)](#) member functions. This [init\(\)](#) function needs to take at least one parameter and must return the event pointer (*me*).

## Backward Traceability

- [SDS\\_QP\\_QF: QF Active Object Framework](#)
- [DVP\\_QP\\_MC4\\_R10\\_03: Rule 10.3\(Required\): The value of an expression shall not be assigned to an object with a narrower essential type or a different essential type category](#)
- [DVP\\_QP\\_MC4\\_R11\\_03B: Rule 11.3\(Required\): A cast shall not be performed between a pointer to object type and a pointer to a different object type \(object-oriented downcast\)](#)
- [DVP-QP-PCLP-826: Suspicious pointer-to-pointer conversion \(area too small\)](#)

## Usage

The following example illustrates **non-variadic** version of [Q\\_NEW\\_X\(\)](#) (when [QEVT\\_PAR\\_INIT](#) is **NOT** defined):

```
// event without parameters
QEvt *myEvt = Q_NEW_X(QEvt, 5U, MY_SIG); // <== (margin == 5U)
if (myEvt) { // check the event pointer!
    // post or publish the event...
}

// event with parameter(s) (event parameter(s) initialized separately)
KeypressEvt *kevt = Q_NEW(KeypressEvt, 10U, KEYPRESS_SIG); // <==
if (kevt) { // check the event pointer!
    kevt->keyId = keyId;
    // post or publish the event...
}
```

The following example illustrates **variadic** version of [Q\\_NEW\\_X\(\)](#) and the **RAII** principle (Resource Acquisition Is Initialization) (when [QEVT\\_PAR\\_INIT](#) IS defined) :

```
// event without parameters (QEVT_DYNAMIC passed to QEvt_init())
QEvt *myEvt = Q_NEW_X(QEvt, MY_SIG, 5U, QEVT_DYNAMIC); // <== (margin == 5U)
if (myEvt) { // check the event pointer!
    // post or publish the event...
}

// event with parameters
typedef struct {
```

```

QEvt super; // <== inherit QEvt
// event parameters follow...
uint8_t keyId; // ID of the key pressed
} KeypressEvt;

// when QEVT_PAR_INIT is defined, the init() member function must be provided
static inline KeypressEvt * KeypressEvt_init(KeypressEvt * const me,
                                             uint8_t keyId)
{
    if (me != (KeypressEvt *)0) {
        me->keyId = keyId;
    }
    return me;
}

// event with parameters (keyId passed to KeypressEvt_init())
KeypressEvt const *kevt = Q_NEW_X(KeypressEvt, 10U, KEYPRESS_SIG, keyId); // <==
if (kevt) { // check the event pointer!
    // post or publish the event...
}

```

### 15.29.2.52 Q\_NEW\_REF

```
#define Q_NEW_REF(
    evtRef_,
    evtT_)
```

**Value:**

```
((evtRef_) = (evtT_ const *)QF_newRef_(e, (evtRef_)))
```

Create a new reference of the current event e

**Details**

The current event processed by an active object is available only for the duration of the run-to-completion (RTC) step. After that step, the current event is no longer available and the framework might recycle (garbage-collect) the event. The macro [Q\\_NEW\\_REF\(\)](#) explicitly creates a new reference to the current event that can be stored and used beyond the current RTC step, until the reference is explicitly recycled by means of the macro [Q\\_DELETE\\_REF\(\)](#).

**Parameters**

in, out	evtRef_	event reference to create
in	evtT_	event type (class name) of the event reference

**Backward Traceability**

- [SDS\\_QP\\_QF: QF Active Object Framework](#)

**Usage**

The example **defer** in the directory examples/win32/defer illustrates the use of [Q\\_NEW\\_REF\(\)](#)

### 15.29.2.53 Q\_DELETE\_REF

```
#define Q_DELETE_REF(
    evtRef_)
```

**Value:**

```
do { \
    QF_deleteRef_((evtRef_)); \
    (evtRef_) = (void *)0; \
} while (false)
```

Delete the event reference.

## Details

Every event reference created with the macro `Q_NEW_REF()` needs to be eventually deleted by means of the macro `Q_DELETE_REF()` to avoid leaking the event.

## Parameters

<code>in, out</code>	<code>evtRef_</code>	event reference to delete
----------------------	----------------------	---------------------------

## Backward Traceability

- [SDS\\_QP\\_QF: QF Active Object Framework](#)

## Usage

The example `defer` in the directory `examples/win32/defer` illustrates the use of `Q_DELETE_REF()`.

### 15.29.2.54 QACTIVE\_POST

```
#define QACTIVE_POST(
    me_,
    e_,
    sender_)
```

#### Value:

```
((void)QActive_post_((me_), (e_), QF_NO_MARGIN, (sender_)))
```

Invoke the direct event posting facility [QActive\\_post\\_\(\)](#).

## Details

This macro asserts if the queue overflows and cannot accept the event.

## Parameters

<code>in, out</code>	<code>me_</code>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
<code>in</code>	<code>e_</code>	pointer to the event to post
<code>in</code>	<code>sender_</code>	pointer to the sender object.

## Note

The `sender_` parameter is actually only used when [QS software tracing](#) is enabled (macro `Q_SPY` is defined). When [QS](#) software tracing is disabled, this macro does not use the `sender_` parameter, so it does not need to be defined.

The `sender_` parameter is actually only used when [QS software tracing](#) is enabled (macro `Q_SPY` is defined). When [QS](#) software tracing is disabled, this macro does not use the `sender_` parameter, so it does not need to be defined.

## See also

- [QActive\\_post\\_\(\)](#)

## Backward Traceability

- [QActive: Active object class \(based on the QHsm implementation strategy\)](#)
- [DVP\\_QP\\_MC4\\_R11\\_03A: Rule 11.3\(Required\): A cast shall not be performed between a pointer to object type and a pointer to a different object type \(object-oriented upcast\)](#)
- [DVP-QP-PCLP-826: Suspicious pointer-to-pointer conversion \(area too small\)](#)

### 15.29.2.55 QACTIVE\_POST\_X

```
#define QACTIVE_POST_X(
    me_,
    e_,
    margin_,
    sender_)
```

#### Value:

```
(QActive_post_((me_), (e_), (margin_), (sender_)))
```

Invoke the direct event posting facility [QActive\\_post\\_\(\)](#) without delivery guarantee.

#### Details

This macro does not assert if the queue overflows and cannot accept the event with the specified margin of free slots remaining.

#### Parameters

in,out	<i>me_</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>e_</i>	pointer to the event to post
in	<i>margin_</i>	the minimum free slots in the queue, which must still be available after posting the event. The special value <a href="#">QF_NO_MARGIN</a> causes asserting failure in case event posting fails.
in	<i>sender_</i>	pointer to the sender object.

#### Returns

'true' if the posting succeeded, and 'false' if the posting failed due to insufficient margin of free entries available in the queue.

#### Note

The *sender\_* parameter is actually only used when [QS software tracing](#) is enabled (macro [Q\\_SPY](#) is defined). When [QS](#) software tracing is disabled, this macro does not use the *sender\_* parameter, so it does not need to be defined.

The pointer to the *sender* object is not necessarily a pointer to an active object. In fact, if [QACTIVE\\_POST\\_X\(\)](#) is called from an interrupt or other context, you can create a unique object just to unambiguously identify the sender of the event.

#### Backward Traceability

- [QActive](#): Active object class (based on the [QHsm](#) implementation strategy)
- DVP\_QP\_MC4\_R11\_03A: Rule 11.3(Required): A cast shall not be performed between a pointer to object type and a pointer to a different object type ([object-oriented upcast](#))
- DVP-QP-PCLP-826: Suspicious pointer-to-pointer conversion (area too small)

#### Usage

```
extern QActive * const AO_Table;
.
.
/* typically inside a state machine action
TableEvt *pe;
Q_NEW_X(pe, TableEvt, 5U, HUNGRY_SIG); // dynamic alloc, margin==5
if (pe != (TableEvt *)0) {
    pe->philNum = me->num;
    QACTIVE_POST_X(AO_Table, &pe->super, 3U, me); // <===
}
.
.
```

### 15.29.2.56 QACTIVE\_PUBLISH

```
#define QACTIVE_PUBLISH(
    e_,
    sender_)
```

**Value:**

```
(QActive_publish_(e_), (void const *) (sender_), (sender_)->prio))
```

Publish an event to all subscriber Active Objects.

**Details**

If [Q\\_SPY](#) is defined, this macro calls `QActive_publish_()` with the `sender_` parameter to identify the publisher of the event. Otherwise, `sender_` is not used.

**Parameters**

in	<code>e_</code>	pointer to the posted event
in	<code>sender_</code>	pointer to the sender object (actually used only when <a href="#">Q_SPY</a> is defined)

**Note**

The `sender_` parameter is actually only used when [QS software tracing](#) is enabled (macro [Q\\_SPY](#) is defined). When [QS](#) software tracing is disabled, the `QACTIVE_PUBLISH()` macro does not use the `sender_` parameter, so it does not need to be defined.

**Backward Traceability**

- [QActive](#): Active object class (based on the [QHsm](#) implementation strategy)

### 15.29.2.57 QTIMEEVNT\_TICK\_X

```
#define QTMEEVNT_TICK_X(
    tickRate_,
    sender_)
```

**Value:**

```
(QTimeEvt_tick_(tickRate_, (sender_)))
```

Invoke the system clock tick processing `QTimeEvt_tick_()`

**Details**

This macro is the recommended way of invoking clock tick processing, because it provides the vital information for software tracing and avoids any overhead when the tracing is disabled.

**Parameters**

in	<code>tickRate_</code>	clock tick rate to be serviced through this call
in	<code>sender_</code>	pointer to the sender object. This parameter is actually only used when <a href="#">QS</a> software tracing is enabled (macro <a href="#">Q_SPY</a> is defined)

**Note**

The `sender_` parameter is actually only used when [QS software tracing](#) is enabled (macro `Q_SPY` is defined). When [QS](#) software tracing is disabled, this macro does not use the `sender_` parameter, so it does not need to be defined.

The `sender_` parameter is actually only used when [QS software tracing](#) is enabled (macro `Q_SPY` is defined). When [QS](#) software tracing is disabled, the [QTIMEEVN\\_TICK\\_X\(\)](#) macro does not use the `sender_` parameter, so it does not need to be defined.

**Backward Traceability**

- [QTimeEvt](#): *Time Event class*

**15.29.2.58 QTICKER\_TRIG**

```
#define QTICKER_TRIG(          \
    ticker_,           \
    sender_)
```

**Value:**

```
(QTicker_trig_((ticker_), (sender_)))
```

Asynchronously trigger the [QTicker](#) AO to perform tick processing.

**Details**

This macro is the recommended way to trigger clock tick processing, because it provides the vital information for software tracing and avoids any overhead when the tracing is disabled.

**Parameters**

in	<code>ticker_</code>	pointer to the <a href="#">QTicker</a> active object
in	<code>sender_</code>	pointer to the sender object. This parameter is actually only used when <a href="#">QS</a> software tracing is enabled (macro <code>Q_SPY</code> is defined)

**Note**

When [QS](#) software tracing is disabled, the macro not does not use the `sender_` parameter, so it does not need to be defined.

The pointer to the `sender_` object is not necessarily a pointer to an active object. In fact, when [QTICKER\\_TRIG\(\)](#) is called from an interrupt, you would create a unique object just to unambiguously identify the ISR as the sender of the time events.

**Backward Traceability**

- [QTicker](#): *"Ticker" Active Object class*

**15.29.2.59 QACTIVE\_POST\_LIFO**

```
#define QACTIVE_POST_LIFO(          \
    me_,           \
    e_)
```

**Value:**

```
(QActive_postLIFO_((me_), (e_)))
```

Post an event to an active object using the Last-In-First-Out (LIFO) policy.

**Parameters**

in, out	<i>me_</i>	current instance pointer (see <a href="#">SAS_QP_OO</a> )
in	<i>e_</i>	pointer to the event to post

**Backward Traceability**

- [QActive](#): Active object class (based on the [QHsm](#) implementation strategy)
- DVP\_QP\_MC4\_R11\_03A: Rule 11.3(Required): A cast shall not be performed between a pointer to object type and a pointer to a different object type ([object-oriented upcast](#))
- DVP-QP-PCLP-826: Suspicious pointer-to-pointer conversion (area too small)

**15.29.2.60 QTIMEEVNT\_TICK**

```
#define QTIMEEVNT_TICK(
    sender_)
```

**Value:**

```
QTIMEEVNT_TICK_X(0U, (sender_))
```

Invoke the system clock tick processing for tick rate 0.

**See also**

- [QTimeEvt::QTimeEvt\\_tick\(\)](#)

**15.29.2.61 QF\_CRIT\_EXIT\_NOP**

```
#define QF_CRIT_EXIT_NOP()
```

**Value:**

```
((void)0)
```

No-operation for exiting a critical section.

**Details**

In some [QF](#) ports the critical section exit takes effect only on the next machine instruction. If this next instruction is another entry to a critical section, the critical section won't be really exited, but rather the two adjacent critical sections would be merged. The [QF\\_CRIT\\_EXIT\\_NOP\(\)](#) macro contains minimal code required to prevent such merging of critical sections in such merging of critical sections in [QF](#) ports, in which it can occur.

**15.29.3 Typedef Documentation****15.29.3.1 int\_t**

```
typedef int int_t
```

Alias for assertion-ID numbers in QP assertions and return from QF\_run()

**15.29.3.2 enum\_t**

```
typedef int enum_t
```

**15.29.3.3 QSignal**

```
typedef uint16_t QSignal
```

The signal of event [QEvt](#).

## Details

The relationship between an event and a signal is as follows. A signal in UML is the specification of an asynchronous stimulus that triggers reactions, and as such is an essential part of an event. (The signal conveys the type of the occurrence—what happened?) However, an event can also contain additional quantitative information about the occurrence in form of event parameters.

### 15.29.3.4 QEvtPtr

```
typedef QEvt const* QEvtPtr
```

Pointer to const event instances passed around in QP Framework.

### 15.29.3.5 QState

```
typedef uint_fast8_t QState
```

Type returned from state-handler functions.

### 15.29.3.6 QStateHandler

```
typedef QState(* QStateHandler) (void *const me, QEvt const *const e)
```

Pointer to a state-handler function.

### 15.29.3.7 QActionHandler

```
typedef QState(* QActionHandler) (void *const me)
```

Pointer to an action-handler function.

### 15.29.3.8 QXThreadHandler

```
typedef void(* QXThreadHandler) (struct QXThread *const me)
```

Pointer to an extended-thread handler function.

### 15.29.3.9 QPrioSpec

```
typedef uint16_t QPrioSpec
```

Priority specification for Active Objects in QP.

## Details

Active Object priorities in QP are integer numbers in the range [1..QF\_MAX\_ACTIVE], whereas the special priority number 0 is reserved for the lowest-priority idle thread. The QP Framework uses the *direct* priority numbering, in which higher numerical values denote higher urgency. For example, an AO with priority 32 has higher urgency than an AO with priority 23.

QPrioSpec allows an application developer to assign **two** priorities to a given AO (see also [Q\\_PRIO\(\)](#)):

1. The "QF-priority", which resides in the least-significant byte of the QPrioSpec data type. The "QF-priority" must be **unique** for each thread in the system and higher numerical values represent higher urgency (direct priority numbering).
2. The "preemption-threshold" priority, which resides in the most-significant byte of the QPrioSpec data type. The second priority cannot be lower than the "QF-priority", but does NOT need to be unique.

In the QP native preemptive kernels, like [QK](#), the "preemption-threshold" priority is used as to implement the "preemption-threshold scheduling" (PTS). It determines the conditions under which a given thread can be *preempted* by other threads. Specifically, a given thread can be preempted only by another thread with a *higher* priority than the "preemption-threshold" of the original thread.

**Note**

For backwards-compatibility, `QPrioSpec` data type might contain only the "QF-priority" component (and the "preemption-threshold" component left at zero). In that case, the "preemption-threshold" will be assumed to be the same as the "QF-priority". This corresponds exactly to the previous semantics of AO priority.

**Remarks**

When QP runs on top of 3rd-party kernels/RTOSes or general-purpose operating systems, the second priority can have different meaning, depending on the specific RTOS/GPOS used. Priority threshold is supported in QP ports to ThreadX.

**15.29.3.10 QTimeEvtCtr**

```
typedef uint32_t QTimeEvtCtr
```

Data type to store the block-size defined based on the macro `QF_TIMEEVT_CTR_SIZE`.

**Details**

The dynamic range of this data type determines the maximum block size that can be managed by the pool.

**15.29.3.11 QPSetBits**

```
typedef uint32_t QPSetBits
```

Bitmask for the internal representation of `QPSet` elements.

**15.29.4 Function Documentation****15.29.4.1 QF\_LOG2()**

```
uint_fast8_t QF_LOG2 (
    QPSetBits const bitmask)
```

Log-base-2 calculation when hardware acceleration is NOT provided (`QF_LOG2` not defined)

**15.29.5 Variable Documentation****15.29.5.1 QP\_versionStr**

```
char const QP_versionStr[24] [extern]
```

**15.30 qp\_pkg.h File Reference**

Internal (package scope) QP/C interface.

**Classes**

- class `QF_Attr`

*Private attributes of the QF framework.*

**Macros**

- `#define QACTIVE_CAST_(ptr_)`
- `#define Q_PTR2UINT_CAST_(ptr_)`
- `#define QTE_FLAG_IS_LINKED (1U << 7U)`
- `#define QTE_FLAG_WAS_DISARMED (1U << 6U)`

### 15.30.1 Detailed Description

Internal (package scope) QP/C interface.

### 15.30.2 Macro Definition Documentation

#### 15.30.2.1 QACTIVE\_CAST\_

```
#define QACTIVE_CAST_(
    ptr_)
```

**Value:**

```
((QActive *) (ptr_))
```

Internal helper macro to encapsulate a MISRA deviation

**Parameters**

in	ptr_	active object pointer
----	------	-----------------------

This internal macro encapsulates the violation of MISRA-C:2023 Rule 11.5(A).

**Backward Traceability**

- DVR\_QP\_MC4\_R11\_05
- DVP\_QP\_MC4\_R11\_03B
- DVP-QP-PCLP-826

#### 15.30.2.2 Q\_PTR2UINT\_CAST\_

```
#define Q_PTR2UINT_CAST_(
    ptr_)
```

**Value:**

```
((uintptr_t) (ptr_))
```

Internal helper macro to cast pointers to integers

**Parameters**

in	ptr_	object pointer
----	------	----------------

**Returns**

The integer representation of the pointer

**Backward Traceability**

- DVP\_QP\_MC4\_D04\_09A (false-positive)
- DVR\_QP\_MC4\_R11\_04

#### 15.30.2.3 QTE\_FLAG\_IS\_LINKED

```
#define QTE_FLAG_IS_LINKED (1U << 7U)
```

#### 15.30.2.4 QTE\_FLAG\_WAS\_DISARMED

```
#define QTE_FLAG_WAS_DISARMED (1U << 6U)
```

## 15.31 qpc.h File Reference

QP/C interface including the backwards-compatibility layer.

```
#include "qp_port.h"
#include "qsafe.h"
#include "qs_port.h"
```

### Macros

- #define QP\_API\_VERSION 0
  - #define QM\_SUPER\_SUB(host\_)
  - #define QM\_TRAN\_EP(tatbl\_)
  - #define QM\_TRAN\_XP(xp\_, tatbl\_)
  - #define QACTIVE\_START(me\_, prioSpec\_, qSto\_, qLen\_, stkSto\_, stkSize\_, par\_)
  - #define QXTHREAD\_START(me\_, prioSpec\_, qSto\_, qLen\_, stkSto\_, stkSize\_, par\_)
  - #define Q\_onAssert(module\_, id\_)
  - #define Q\_ALLEGGE\_ID(id\_, expr\_)
  - #define Q\_ALLEGGE(expr\_)
  - #define Q\_ASSERT\_COMPILE(expr\_)
  - #define QHSM\_INIT(me\_, par\_, qsld\_)
  - #define QHSM\_DISPATCH(me\_, e\_, qsld\_)
  - #define QHsm\_isIn(me\_, state\_)
  - #define QF\_PUBLISH(e\_, sender\_)
  - #define QF\_TICK\_X(tickRate\_, sender\_)
- Invoke the system clock tick processing.*
- #define QF\_TICK(sender\_)
- Invoke the system clock tick processing.*
- #define QF\_getQueueMin(prio\_)

### Typedefs

- typedef char `char_t`

### 15.31.1 Detailed Description

QP/C interface including the backwards-compatibility layer.

### 15.31.2 Macro Definition Documentation

#### 15.31.2.1 QP\_API\_VERSION

```
#define QP_API_VERSION 0
```

#### 15.31.2.2 QM\_SUPER\_SUB

```
#define QM_SUPER_SUB (
    host_)
```

##### **Value:**

```
error "submachines no longer supported"
```

### 15.31.2.3 QM\_TRAN\_EP

```
#define QM_TRAN_EP (
    tatbl_)

Value:
error "submachines no longer supported"
```

### 15.31.2.4 QM\_TRAN\_XP

```
#define QM_TRAN_XP (
    xp_,
    tatbl_)

Value:
error "submachines no longer supported"
```

### 15.31.2.5 QACTIVE\_START

```
#define QACTIVE_START (
    me_,
    prioSpec_,
    qSto_,
    qLen_,
    stkSto_,
    stkSize_,
    par_)

Value:
(QActive_start((QActive *) (me_), (prioSpec_), \
(qSto_), (qLen_), (stkSto_), (stkSize_), (par_)))
```

**Deprecated** Macro for starting an Active Object. Use [QActive::QActive\\_start\(\)](#) instead.

### 15.31.2.6 QXTHREAD\_START

```
#define QXTHREAD_START (
    me_,
    prioSpec_,
    qSto_,
    qLen_,
    stkSto_,
    stkSize_,
    par_)

Value:
(QXThread_start((QXThread *) (me_), (prioSpec_), \
(qSto_), (qLen_), (stkSto_), (stkSize_), (par_)))
```

**Deprecated** Macro for starting an eXtended Thread. Use [QXThread::QXThread\\_start\(\)](#) instead.

### 15.31.2.7 Q\_onAssert

```
#define Q_onAssert (
    module_,
    id_)

Value:
Q_onError(module_, id_)
```

**Deprecated** Assertion failure handler. Use [Q\\_onError\(\)](#) instead.

### 15.31.2.8 Q\_ALLEGGE\_ID

```
#define Q_ALLEGGE_ID(
    id_,
    expr_)
```

**Value:**

```
if (!(expr_)) { \
    QF_CRIT_STAT \
    QF_CRIT_ENTRY(); \
    Q_onError(&Q_this_module_[0], (id_)); \
    QF_CRIT_EXIT(); \
} else ((void)0)
```

**Deprecated** #Q\_NASSERT preprocessor switch to disable QP assertions

**Deprecated** general purpose assertion with user-specified ID number that **always** evaluates the `expr_` expression.

**Note**

The use of this macro is no longer recommended.

### 15.31.2.9 Q\_ALLEGGE

```
#define Q_ALLEGGE(
    expr_)
```

**Value:**

```
Q_ALLEGGE_ID(__LINE__, (expr_))
```

**Deprecated** general purpose assertion without ID number that **always** evaluates the `expr_` expression. Instead of ID number, this macro is based on the standard `__LINE__` macro.

**Note**

The use of this macro is no longer recommended.

### 15.31.2.10 Q\_ASSERT\_COMPILE

```
#define Q_ASSERT_COMPILE(
    expr_)
```

**Value:**

```
Q_ASSERT_STATIC(expr_)
```

Static (compile-time) assertion.

**Deprecated** Use `Q_ASSERT_STATIC()` or better yet `_Static_assert()` instead.

### 15.31.2.11 QHSM\_INIT

```
#define QHSM_INIT(
    me_,
    par_,
    qsId_)
```

**Value:**

```
QASM_INIT((me_), (par_), (qsId_))
```

**Deprecated** instead use: `QASM_INIT()`

### 15.31.2.12 QHSM\_DISPATCH

```
#define QHSM_DISPATCH(
    me_,
    e_,
    qsId_)
```

**Value:**

```
QASM_DISPATCH((me_), (e_), (qsId_))
```

**Deprecated** instead use: [QASM\\_DISPATCH\(\)](#)

### 15.31.2.13 QHsm\_isIn

```
#define QHsm_isIn(
    me_,
    state_)
```

**Value:**

```
QASM_IS_IN((QAsm *) (me_), (state_))
```

**Deprecated** instead use: [QASM\\_IS\\_IN\(\)](#)

### 15.31.2.14 QF\_PUBLISH

```
#define QF_PUBLISH(
    e_,
    sender_)
```

**Value:**

```
QACTIVE_PUBLISH((e_), (sender_))
```

### 15.31.2.15 QF\_TICK\_X

```
#define QF_TICK_X(
    tickRate_,
    sender_)
```

**Value:**

```
QTIMEEVNT_TICK_X((tickRate_), (sender_))
```

Invoke the system clock tick processing.

**Deprecated** superseded by [QTIMEEVNT\\_TICK\\_X\(\)](#)

### 15.31.2.16 QF\_TICK

```
#define QF_TICK(
    sender_)
```

**Value:**

```
QTIMEEVNT_TICK(sender_)
```

Invoke the system clock tick processing.

**Deprecated** superseded by [QTIMEEVNT\\_TICK\(\)](#)

### 15.31.2.17 QF\_getQueueMin

```
#define QF_getQueueMin(
    prio_)
```

**Value:**

```
(QActive_getQueueMin((prio_)))
```

**Deprecated** instead use: [QActive::QActive\\_getQueueMin\(\)](#)

### 15.31.3 Typedef Documentation

#### 15.31.3.1 char\_t

```
typedef char char\_t
```

**Deprecated** plain 'char' is no longer forbidden in MISRA-C:2023

## 15.32 qs\_dummy.h File Reference

QS/C dummy public interface.

### Classes

- struct [QS\\_TProbe](#)  
*QUTest Test-Probe attributes.*

### Macros

- #define [QS\\_INIT\(arg\\_\)](#)
- #define [QS\\_EXIT\(\)](#)
- #define [QS\\_DUMP\(\)](#)
- #define [QS\\_GLB\\_FILTER\(rec\\_\)](#)
- #define [QS\\_LOC\\_FILTER\(qsId\\_\)](#)
- #define [QS\\_BEGIN\\_ID\(rec\\_, qsId\\_\)](#)
- #define [QS\\_END\(\)](#)
- #define [QS\\_BEGIN\\_INCRIT\(rec\\_, qsId\\_\)](#)
- #define [QS\\_END\\_INCRIT\(\)](#)
- #define [QS\\_I8\(width\\_, data\\_\)](#)
- #define [QS\\_U8\(width\\_, data\\_\)](#)
- #define [QS\\_I16\(width\\_, data\\_\)](#)
- #define [QS\\_U16\(width\\_, data\\_\)](#)
- #define [QS\\_I32\(width\\_, data\\_\)](#)
- #define [QS\\_U32\(width\\_, data\\_\)](#)
- #define [QS\\_F32\(width\\_, data\\_\)](#)
- #define [QS\\_F64\(width\\_, data\\_\)](#)
- #define [QS\\_I64\(width\\_, data\\_\)](#)
- #define [QS\\_U64\(width\\_, data\\_\)](#)
- #define [QS\\_ENUM\(group\\_, value\\_\)](#)
- #define [QS\\_STR\(str\\_\)](#)
- #define [QS\\_MEM\(mem\\_, size\\_\)](#)
- #define [QS\\_SIG\(sig\\_, obj\\_\)](#)
- #define [QS\\_OBJ\(obj\\_\)](#)
- #define [QS\\_FUN\(fun\\_\)](#)
- #define [QS\\_SIG\\_DICTIONARY\(sig\\_, obj\\_\)](#)
- #define [QS\\_OBJ\\_DICTIONARY\(obj\\_\)](#)
- #define [QS\\_OBJ\\_ARR\\_DICTIONARY\(obj\\_, idx\\_\)](#)
- #define [QS\\_FUN\\_DICTIONARY\(fun\\_\)](#)
- #define [QS\\_USR\\_DICTIONARY\(rec\\_\)](#)
- #define [QS\\_ENUM\\_DICTIONARY\(value\\_, group\\_\)](#)
- #define [QS\\_ASSERTION\(module\\_, loc\\_, delay\\_\)](#)
- #define [QS\\_FLUSH\(\)](#)

- #define QS\_TEST\_PROBE\_DEF(fun\_)
- #define QS\_TEST\_PROBE(code\_)
- #define QS\_TEST\_PROBE\_ID(id\_, code\_)
- #define QS\_TEST\_PAUSE()
- #define QS\_OUTPUT()
- #define QS\_RX\_INPUT()
- #define QS\_RX\_PUT(b\_)
- #define QS\_ONLY(code\_)
- #define QS\_BEGIN\_PRE(rec\_, qsld\_)
- #define QS\_END\_PRE()
- #define QS\_U8\_PRE(data\_)
- #define QS\_2U8\_PRE(data1\_, data2\_)
- #define QS\_U16\_PRE(data\_)
- #define QS\_U32\_PRE(data\_)
- #define QS\_TIME\_PRE()
- #define QS\_SIG\_PRE(sig\_)
- #define QS\_EVS\_PRE(size\_)
- #define QS\_OBJ\_PRE(obj\_)
- #define QS\_FUN\_PRE(fun\_)
- #define QS\_EQC\_PRE(ctr\_)
- #define QS\_MPC\_PRE(ctr\_)
- #define QS MPS\_PRE(size\_)
- #define QS\_TEC\_PRE(ctr\_)
- #define QS\_CRIT\_STAT
- #define QS\_CRIT\_ENTRY()
- #define QS\_CRIT\_EXIT()
- #define QS\_TR\_CRIT\_ENTRY()
- #define QS\_TR\_CRIT\_EXIT()
- #define QS\_TR\_ISR\_ENTRY(isrnest\_, prio\_)
- #define QS\_TR\_ISR\_EXIT(isrnest\_, prio\_)

## Typedefs

- typedef uint32\_t QSTimeCtr  
*Unsigned integer type for QS timestamps.*

## Functions

- void QS\_initBuf (uint8\_t \*const sto, uint\_fast32\_t const stoSize)
- uint16\_t QS\_getByte (void)
- uint8\_t const \* QS\_getBlock (uint16\_t \*const pNbytes)
- void QS\_doOutput (void)
- uint8\_t QS\_onStartup (void const \*arg)
- void QS\_onCleanup (void)
- void QS\_onFlush (void)  
*Flush the QS output buffer.*
- QSTimeCtr QS\_onGetTime (void)  
*Return the current timestamp for QS trace records.*
- void QS\_rxInitBuf (uint8\_t \*const sto, uint16\_t const stoSize)
- void QS\_rxParse (void)
- void QS\_onTestSetup (void)

- void `QS_onTestTeardown` (void)
- void `QS_onTestEvt` (`QEvt` \*e)
- void `QS_onTestPost` (void const \*sender, `QActive` \*recipient, `QEvt` const \*e, bool status)
- void `QS_onTestLoop` (void)

### 15.32.1 Detailed Description

QS/C dummy public interface.

### 15.32.2 Macro Definition Documentation

#### 15.32.2.1 QS\_INIT

```
#define QS_INIT(
    arg_)

Value:
((uint8_t)1U)
```

#### 15.32.2.2 QS\_EXIT

```
#define QS_EXIT()
Value:
(void)0
```

#### 15.32.2.3 QS\_DUMP

```
#define QS_DUMP()
Value:
(void)0
```

#### 15.32.2.4 QS\_GLB\_FILTER

```
#define QS_GLB_FILTER(
    rec_)

Value:
(void)0
```

#### 15.32.2.5 QS\_LOC\_FILTER

```
#define QS_LOC_FILTER(
    qsId_)

Value:
(void)0
```

#### 15.32.2.6 QS\_BEGIN\_ID

```
#define QS_BEGIN_ID(
    rec_,
    qsId_)

Value:
if (false) {
```

#### 15.32.2.7 QS\_END

```
#define QS_END()
Value:
}
```

### 15.32.2.8 QS\_BEGIN\_INCRIT

```
#define QS_BEGIN_INCRIT(
    rec_,
    qsId_)
```

**Value:**

```
if (false) {
```

### 15.32.2.9 QS\_END\_INCRIT

```
#define QS_END_INCRIT()
```

**Value:**

```
}
```

### 15.32.2.10 QS\_I8

```
#define QS_I8(
    width_,
    data_)
```

**Value:**

```
((void) 0)
```

### 15.32.2.11 QS\_U8

```
#define QS_U8(
    width_,
    data_)
```

**Value:**

```
((void) 0)
```

### 15.32.2.12 QS\_I16

```
#define QS_I16(
    width_,
    data_)
```

**Value:**

```
((void) 0)
```

### 15.32.2.13 QS\_U16

```
#define QS_U16(
    width_,
    data_)
```

**Value:**

```
((void) 0)
```

### 15.32.2.14 QS\_I32

```
#define QS_I32(
    width_,
    data_)
```

**Value:**

```
((void) 0)
```

### 15.32.2.15 QS\_U32

```
#define QS_U32(
    width_,
    data_)
```

**Value:**

```
(void) 0)
```

**15.32.2.16 QS\_F32**

```
#define QS_F32(  
    width_,  
    data_)
```

**Value:**

```
(void) 0)
```

**15.32.2.17 QS\_F64**

```
#define QS_F64(  
    width_,  
    data_)
```

**Value:**

```
(void) 0)
```

**15.32.2.18 QS\_I64**

```
#define QS_I64(  
    width_,  
    data_)
```

**Value:**

```
(void) 0)
```

**15.32.2.19 QS\_U64**

```
#define QS_U64(  
    width_,  
    data_)
```

**Value:**

```
(void) 0)
```

**15.32.2.20 QS\_ENUM**

```
#define QS_ENUM(  
    group_,  
    value_)
```

**Value:**

```
(void) 0)
```

**15.32.2.21 QS\_STR**

```
#define QS_STR(  
    str_)
```

**Value:**

```
(void) 0)
```

**15.32.2.22 QS\_MEM**

```
#define QS_MEM(  
    mem_,  
    size_)
```

**Value:**

```
(void) 0)
```

**15.32.2.23 QS\_SIG**

```
#define QS_SIG(
    sig_,
    obj_)
```

**Value:**

```
((void) 0)
```

**15.32.2.24 QS\_OBJ**

```
#define QS_OBJ(
    obj_)
```

**Value:**

```
((void) 0)
```

**15.32.2.25 QS\_FUN**

```
#define QS_FUN(
    fun_)
```

**Value:**

```
((void) 0)
```

**15.32.2.26 QS\_SIG\_DICTIONARY**

```
#define QS_SIG_DICTIONARY(
    sig_,
    obj_)
```

**Value:**

```
((void) 0)
```

**15.32.2.27 QS\_OBJ\_DICTIONARY**

```
#define QS_OBJ_DICTIONARY(
    obj_)
```

**Value:**

```
((void) 0)
```

**15.32.2.28 QS\_OBJ\_ARR\_DICTIONARY**

```
#define QS_OBJ_ARR_DICTIONARY(
    obj_,
    idx_)
```

**Value:**

```
((void) 0)
```

**15.32.2.29 QS\_FUN\_DICTIONARY**

```
#define QS_FUN_DICTIONARY(
    fun_)
```

**Value:**

```
((void) 0)
```

**15.32.2.30 QS\_USR\_DICTIONARY**

```
#define QS_USR_DICTIONARY(
    rec_)
```

**Value:**

```
((void) 0)
```

**15.32.2.31 QS\_ENUM\_DICTIONARY**

```
#define QS_ENUM_DICTIONARY(
    value_,
    group_)
```

**Value:**

```
((void) 0)
```

**15.32.2.32 QS\_ASSERTION**

```
#define QS_ASSERTION(
    module_,
    loc_,
    delay_)
```

**Value:**

```
((void) 0)
```

**15.32.2.33 QS\_FLUSH**

```
#define QS_FLUSH()
```

**Value:**

```
((void) 0)
```

**15.32.2.34 QS\_TEST\_PROBE\_DEF**

```
#define QS_TEST_PROBE_DEF(
    fun_)
```

QS macro to define the Test-Probe for a given fun\_

**Backward Traceability**

- DVP\_QS\_MC4\_R11\_01
- DVP\_QS\_MC4\_R15\_05
- DVP\_QS\_PCLP\_823

**15.32.2.35 QS\_TEST\_PROBE**

```
#define QS_TEST_PROBE(
    code_)
```

**15.32.2.36 QS\_TEST\_PROBE\_ID**

```
#define QS_TEST_PROBE_ID(
    id_,
    code_)
```

QS\_TEST\_PROBE QS macro to apply a Test-Probe

**Backward Traceability**

- DVR\_QS\_MC4\_R11\_05

QS macro to apply a Test-Probe

**15.32.2.37 QS\_TEST\_PAUSE**

```
#define QS_TEST_PAUSE()
```

**Value:**

```
((void) 0)
```

QS macro to pause test execution and enter the test event-loop

**15.32.2.38 QS\_OUTPUT**

```
#define QS_OUTPUT()
```

**Value:**

```
((void) 0)
```

**15.32.2.39 QS\_RX\_INPUT**

```
#define QS_RX_INPUT()
```

**Value:**

```
((void) 0)
```

**15.32.2.40 QS\_RX\_PUT**

```
#define QS_RX_PUT(
    b_)
```

**Value:**

```
((void) 0)
```

Macro to call QS\_rxPut() when **Q\_SPY** is defined (resolves to nothing when **Q\_SPY** is not defined)

**15.32.2.41 QS\_ONLY**

```
#define QS_ONLY(
    code_)
```

**Value:**

```
((void) 0)
```

**15.32.2.42 QS\_BEGIN\_PRE**

```
#define QS_BEGIN_PRE(
    rec_,
    qsId_)
```

**Value:**

```
if (false) {
```

**15.32.2.43 QS\_END\_PRE**

```
#define QS_END_PRE()
```

**Value:**

```
}
```

**15.32.2.44 QS\_U8\_PRE**

```
#define QS_U8_PRE(
    data_)
```

**Value:**

```
((void) 0)
```

**15.32.2.45 QS\_2U8\_PRE**

```
#define QS_2U8_PRE(
    data1_,
    data2_)
```

**Value:**

```
((void) 0)
```

**15.32.2.46 QS\_U16\_PRE**

```
#define QS_U16_PRE(  
    data_)
```

**Value:**

```
(void) 0
```

**15.32.2.47 QS\_U32\_PRE**

```
#define QS_U32_PRE(  
    data_)
```

**Value:**

```
(void) 0
```

**15.32.2.48 QS\_TIME\_PRE**

```
#define QS_TIME_PRE()
```

**Value:**

```
(void) 0
```

**15.32.2.49 QS\_SIG\_PRE**

```
#define QS_SIG_PRE(  
    sig_)
```

**Value:**

```
(void) 0
```

**15.32.2.50 QS\_EVS\_PRE**

```
#define QS_EVS_PRE(  
    size_)
```

**Value:**

```
(void) 0
```

**15.32.2.51 QS\_OBJ\_PRE**

```
#define QS_OBJ_PRE(  
    obj_)
```

**Value:**

```
(void) 0
```

**15.32.2.52 QS\_FUN\_PRE**

```
#define QS_FUN_PRE(  
    fun_)
```

**Value:**

```
(void) 0
```

**15.32.2.53 QS\_EQC\_PRE**

```
#define QS_EQC_PRE(  
    ctr_)
```

**Value:**

```
(void) 0
```

**15.32.2.54 QS\_MPC\_PRE**

```
#define QS_MPC_PRE(
    ctr_)
```

**Value:**

```
((void) 0)
```

**15.32.2.55 QS\_MPS\_PRE**

```
#define QS_MPS_PRE(
    size_)
```

**Value:**

```
((void) 0)
```

**15.32.2.56 QS\_TEC\_PRE**

```
#define QS_TEC_PRE(
    ctr_)
```

**Value:**

```
((void) 0)
```

**15.32.2.57 QS\_CRIT\_STAT**

```
#define QS_CRIT_STAT
```

Internal **QS** macro for defining the critical section status

**15.32.2.58 QS\_CRIT\_ENTRY**

```
#define QS_CRIT_ENTRY()
```

**Value:**

```
((void) 0)
```

Internal macro for entering a critical section

**15.32.2.59 QS\_CRIT\_EXIT**

```
#define QS_CRIT_EXIT()
```

**Value:**

```
((void) 0)
```

Internal macro for exiting a critical section

**15.32.2.60 QS\_TR\_CRIT\_ENTRY**

```
#define QS_TR_CRIT_ENTRY()
```

**Value:**

```
((void) 0)
```

**15.32.2.61 QS\_TR\_CRIT\_EXIT**

```
#define QS_TR_CRIT_EXIT()
```

**Value:**

```
((void) 0)
```

**15.32.2.62 QS\_TR\_ISR\_ENTRY**

```
#define QS_TR_ISR_ENTRY(
    isrnest_,
    prio_)
```

**Value:**

```
(void) 0
```

**15.32.2.63 QS\_TR\_ISR\_EXIT**

```
#define QS_TR_ISR_EXIT(\
    isrnest_,\
    prio_)
```

**Value:**

```
(void) 0
```

**15.32.3 Typedef Documentation****15.32.3.1 QSTimeCtr**

`typedef uint32_t QSTimeCtr`

Unsigned integer type for **QS** timestamps.

**Details**

The dynamic range of **QSTimeCtr** is configurable by the macro **QS\_TIME\_SIZE**, which can take values 1-byte (256 counts), 2-bytes (64K counts) and 4-bytes (4G counts).

**See also**

- [QS\\_TIME\\_SIZE](#)
- [QS::QS\\_onGetTime\(\)](#)

**15.32.4 Function Documentation****15.32.4.1 QS\_initBuf()**

```
void QS_initBuf (\
    uint8_t *const sto, \
    uint_fast32_t const stoSize)
```

Initialize the QS-TX data buffer

**Details**

This function should be called from [QS\\_onStartup\(\)](#) to provide QS-TX (transmit channel) with the data buffer.

**Parameters**

in	<i>sto</i>	pointer to the storage for the transmit buffer
in	<i>stoSize</i>	size in [bytes] of the storage buffer. Currently the size of the <b>QS</b> buffer cannot exceed 64KB.

**Remarks**

**QS** can work with quite small data buffers (e.g., 1024 bytes), but you will start losing data (tracing information) if the buffer is too small for the bursts of tracing activity. The right size of the buffer depends on the data production rate and the data output rate. **QS** offers flexible filtering to reduce the data production rate.

**Note**

If the data output rate cannot keep up with the production rate, **QS** will start overwriting the older data with newer data. This is consistent with the "last-is-best" **QS** policy. The **QS** data protocol sequence counters and check sums on each QS-record allow the QSPY host utility to easily detect any data loss. (See also functions [QS\\_getByte\(\)](#) and [QS\\_getBlock\(\)](#) for performing QS-TX output.)

#### 15.32.4.2 QS\_getByte()

```
uint16_t QS_getByte (
    void )
```

Byte-oriented interface to the QS-TX data buffer

##### Details

This function delivers one byte at a time from the [QS](#) data buffer.

##### Returns

the byte in the least-significant 8-bits of the 16-bit return value if the byte is available. If no more data is available at the time, the function returns ::QS\_EOD (End-Of-Data).

##### Attention

[QS\\_getByte\(\)](#) should be called from within a critical section (interrupts disabled).

#### 15.32.4.3 QS\_getBlock()

```
uint8_t const * QS_getBlock (
    uint16_t *const pNbytes)
```

Block-oriented interface to the QS-TX data buffer

##### Details

This function delivers a contiguous block of data from the [QS](#) data buffer. The function returns the pointer to the beginning of the block, and writes the number of bytes in the block to the location pointed to by `pNbytes`. The argument `pNbytes` is also used as input to provide the maximum size of the data block that the caller can accept.

##### Parameters

in, out	<code>pNbytes</code>	pointer to the number of bytes to send. On input, <code>pNbytes</code> specifies the maximum number of bytes that the function can provide. On output, <code>pNbytes</code> contains the actual number of bytes available.
---------	----------------------	--

##### Returns

if data is available, the function returns pointer to the contiguous block of data and sets the value pointed to by `pNbytes` to the # available bytes. If data is available at the time the function is called, the function returns NULL pointer and sets the value pointed to by `pNbytes` to zero.

##### Note

Only the NULL return from [QS\\_getBlock\(\)](#) indicates that the [QS](#) buffer is empty at the time of the call. The non-NULL return often means that the block is at the end of the buffer and you need to call [QS\\_getBlock\(\)](#) again to obtain the rest of the data that "wrapped around" to the beginning of the [QS](#) data buffer.

##### Attention

[QS\\_getBlock\(\)](#) should be called from within a critical section (interrupts disabled).

#### 15.32.4.4 QS\_doOutput()

```
void QS_doOutput (
    void )
```

#### 15.32.4.5 QS\_onStartup()

```
uint8_t QS_onStartup (
    void const * arg)
```

#### 15.32.4.6 QS\_onCleanup()

```
void QS_onCleanup (
    void )
```

#### 15.32.4.7 QS\_onFlush()

```
void QS_onFlush (
    void )
```

Flush the [QS](#) output buffer.

##### Details

This function is used at the end of [QS](#) dictionaries and other [QS](#) records that need to be transmitted immediately. [QS\\_onFlush\(\)](#) is typically used only during the initial transient and should NOT be used thereafter because it is busy-waiting for the transmission of the data.

##### Attention

Typically, no critical section (or interrupt disabling) should be used in [QS\\_onFlush\(\)](#) to avoid nesting of critical sections in case [QS\\_onFlush\(\)](#) is called from [Q\\_onError\(\)](#).

#### 15.32.4.8 QS\_onGetTime()

```
QSTimeCtr QS_onGetTime (
    void )
```

Return the current timestamp for [QS](#) trace records.

##### Details

This function is defined in the application and is called for every [QS](#) trace record that requires a timestamp. The function shall return the current timestamp as an unsigned integer of the type [QSTimeCtr](#). The dynamic range of [QSTimeCtr](#) is configurable by the macro [QS\\_TIME\\_SIZE](#), which can take values 1-byte (256 counts), 2-bytes (64K counts) and 4-bytes (4G counts).

As this is a callback from [QS](#) to the application, the properties of the timestamp depend entirely on the application. Typically, the function requires a hardware timer with the following properties:

- sufficiently fine granularity (typically sub-microsecond)
- natural warp-around by the end of the dynamic range (e.g., 0xFFFFFFFF -> 0 for a 4-byte timestamp)

##### Attention

The [QS\\_onGetTime\(\)](#) function is called within a critical section.

## Example

The following code implements [QS\\_onGetTime\(\)](#) with a free-running, 32-bit timer in STM32U545 MCU:

```
QSTimeCtr QS_onGetTime(void) {
    return TIM5->CNT; // 32-bit Timer5 count
}

// where Timer5 is configured inside QS_onStartup() as follows:
uint8_t QS_onStartup(void const *arg) {
    .
    .
    // configure Timer5 for delivering QS time-stamp.....
    SET_BIT(RCC->APB1ENR1, RCC_APB1ENR1_TIM5EN);

    uint32_t tmp = TIM5->CR1;
    MODIFY_REG(tmp, TIM_CR1_DIR | TIM_CR1_CMS, 0U); // counter-mode=Up
    MODIFY_REG(tmp, TIM_CR1_CKD, 0U); // clock-division=1
    WRITE_REG(TIM5->CR1, tmp);

    WRITE_REG(TIM5->ARR, 0xFFFFFFFFU); // auto-reload
    WRITE_REG(TIM5->PSC, 0U); // prescaler=1
    SET_BIT(TIM5->EGR, TIM_EGR_UG); // update event to reload prescaler

    CLEAR_BIT(TIM5->CR1, TIM_CR1_ARPE); // disable ARR preload
    MODIFY_REG(TIM5->SMCR, TIM_SMCR_SMS | TIM_SMCR_ECE, 0U); // internal clock
    MODIFY_REG(TIM5->CR2, TIM_CR2_MMS, 0U); // reset timer synchronization
    CLEAR_BIT(TIM5->SMCR, TIM_SMCR_MSM); // disable master-slave

    // enable Timer5 to start time stamp
    SET_BIT(TIM5->CR1, TIM_CR1_CEN);
    .
}
```

## See also

- [QS\\_TIME\\_SIZE](#)
- [QSTimeCtr](#)

### 15.32.4.9 QS\_rxInitBuf()

```
void QS_rxInitBuf (
    uint8_t *const sto,
    uint16_t const stoSize)
```

Initialize the QS-RX data buffer

#### Details

This function should be called from [QS\\_onStartup\(\)](#) to provide QS-RX (receive channel) with the data buffer. The QS-RX channel requires the initialization of the QS-TX channel (see [QS\\_initBuf\(\)](#)). The QS-RX channel is optional and does not need to be initialized when not needed.

#### Parameters

in	<i>sto</i>	pointer to the storage for the QS-RX receive buffer
in	<i>stoSize</i>	size in [bytes] of the storage buffer

#### Remarks

[QS](#) can work with quite small data buffers (e.g., 128 bytes), but you will start losing data (commands to the target) if the buffer is too small for the commands that are being sent. The right size of the buffer depends on the commands and the data input rate.

#### Note

If the QS-RX processing cannot keep up with the input rate, [QS](#) will start overwriting the older data with newer data. The [QS](#) data protocol sequence counters and check sums on each QS-record allow the QS-RX parser (see [QS\\_rxParse\(\)](#)) to detect any data loss, but the corrupted commands won't be executed.

#### 15.32.4.10 QS\_rxParse()

```
void QS_rxParse (
    void )
```

Parse and process QS-RX (receive channel) data bytes

##### Details

The application must call this function repeatedly to process the incoming commands from the QS-RX input channel. The frequency of the calls must be high enough to avoid overflowing the QS-RX input buffer (see [QS\\_rxInitBuf\(\)](#), [QS\\_rxPut\(\)](#), and [QS\\_RX\\_PUT\(\)](#)).

##### Note

This function should be called from a single thread of execution. An ideal place to call this function is the idle-thread. Also, this function must be called outside a critical section (with interrupts enabled).

#### 15.32.4.11 QS\_onTestSetup()

```
void QS_onTestSetup (
    void )
```

#### 15.32.4.12 QS\_onTestTeardown()

```
void QS_onTestTeardown (
    void )
```

#### 15.32.4.13 QS\_onTestEvt()

```
void QS_onTestEvt (
    QEvt * e)
```

#### 15.32.4.14 QS\_onTestPost()

```
void QS_onTestPost (
    void const * sender,
    QActive * recipient,
    QEvt const * e,
    bool status)
```

#### 15.32.4.15 QS\_onTestLoop()

```
void QS_onTestLoop (
    void )
```

## 15.33 qsafe.h File Reference

QP Functional Safety (FuSa) Subsystem.

### Macros

- #define QF\_CRIT\_STAT
- #define QF\_CRIT\_ENTRY()
- #define QF\_CRIT\_EXIT()
- #define Q\_ASSERT\_INCRIT(id\_, expr\_)

*General-purpose assertion with user-specified ID number (in critical section)*

- `#define Q_ERROR_INCRIT(id_)`  
*Assertion with user-specified ID for a wrong path through the code (in critical section)*
- `#define Q_ASSERT_ID(id_, expr_)`  
*General-purpose assertion (with user-specified ID number)*
- `#define Q_ERROR_ID(id_)`  
*Assertion for a wrong path through the code (with user-specified ID)*
- `#define Q_ASSERT(expr_)`  
*General-purpose assertion (with ID provided in LINE)*
- `#define Q_ERROR()`  
*Assertion for a wrong path through the code (with ID provided in LINE)*
- `#define Q_REQUIRE_ID(id_, expr_)`  
*Assertion for checking a precondition (with user-specified ID number)*
- `#define Q_REQUIRE(expr_)`  
*Assertion for checking a precondition (with ID provided in LINE)*
- `#define Q_REQUIRE_INCRIT(id_, expr_)`  
*Assertion for checking a precondition (in critical section)*
- `#define Q_ENSURE_ID(id_, expr_)`  
*Assertion for checking a postcondition (with user-specified ID number)*
- `#define Q_ENSURE(expr_)`  
*Assertion for checking a postcondition*
- `#define Q_ENSURE_INCRIT(id_, expr_)`  
*Assertion for checking a postcondition (in critical section)*
- `#define Q_INVARIANT_ID(id_, expr_)`  
*Assertion for checking an invariant (with user-specified ID number)*
- `#define Q_INVARIANT(expr_)`  
*Assertion for checking an invariant.*
- `#define Q_INVARIANT_INCRIT(id_, expr_)`
- `#define Q_ASSERT_STATIC(expr_)`
- `#define Q_NORETURN _Noreturn void`
- `#define Q_DIM(array_)`

## Functions

- `Q_NORETURN Q_onError (char const *const module, int_t const id)`  
*Custom error handler Callback function invoked after detecting an error (part of QP Functional Safety (FuSa) Subsystem).*

### 15.33.1 Detailed Description

QP Functional Safety (FuSa) Subsystem.

This header file is part of the QP Functional Safety (FuSa) Subsystem and contains the following facilities:

- Software assertions (Failure Assertion Programming (FAP) in IEC 61508)
- Software Self-Monitoring (SSM) techniques:
  - Duplicate Inverse Storage for critical variables
  - Fixed Upper Loop Bound for all loops
  - Invalid Control Flow for all unreachable code paths
  - Hardware Memory Isolation by means of Memory Protection Unit (MPU)
  - High Watermark in event queues

- High Watermark in event pools
- Stack Overflow detection in QP Applications
- Stack Painting in QP Applications
- NULL-Pointer Dereferencing protection in QP Applications

**Note**

This header file can be used in C, C++, and mixed C/C++ programs.

**Attention**

The preprocessor switch [Q\\_UNSAFE](#) disables the QP Functional Safety System. However, it is generally **NOT RECOMMENDED**, especially in the production code. Instead, the failure callback [Q\\_onError\(\)](#) should be very carefully designed, implemented, and tested in various failure modes.

## 15.33.2 Macro Definition Documentation

### 15.33.2.1 QF\_CRIT\_STAT

```
#define QF_CRIT_STAT
```

### 15.33.2.2 QF\_CRIT\_ENTRY

```
#define QF_CRIT_ENTRY()
```

**Value:**

```
(void) 0
```

### 15.33.2.3 QF\_CRIT\_EXIT

```
#define QF_CRIT_EXIT()
```

**Value:**

```
(void) 0
```

### 15.33.2.4 Q\_ASSERT\_INCRIT

```
#define Q_ASSERT_INCRIT(  
    id_,  
    expr_)
```

**Value:**

```
((expr_) ? ((void) 0) : Q_onError(&Q_this_module_[0], (id_)))
```

General-purpose assertion with user-specified ID number (in critical section)

**Details****Parameters**

in	<i>id_</i>	ID number (unique within the module) of the assertion
in	<i>expr_</i>	Boolean expression to check

### Attention

This macro must be called inside already established critical section. The evaluation of the expression `expr_` as well as calling of `Q_onError()` happens inside that critical section.

The assertion expression (`expr_`) must be possibly simple, have **no side effects**, and quick to evaluate because the evaluation happens inside a critical section. Also, the expression must **NOT** call any functions that might use critical sections inside (because this would cause nesting of critical sections, which might not be supported).

### Backward Traceability

- DVP\_QP\_MC4\_D04\_09A: *Directive 4.9(Advisory): A function should be used in preference to a function-like macro where they are interchangeable (FALSE-POSITIVE diagnosis)* (false-positive)

### Forward Traceability

#### 15.33.2.5 Q\_ERROR\_INCRIT

```
#define Q_ERROR_INCRIT(  
    id_)
```

##### Value:

```
(Q_onError(&Q_this_module_[0], (id_)))
```

Assertion with user-specified ID for a wrong path through the code (in critical section)

##### Details

##### Parameters

in	<code>id_</code>	ID number (unique within the module) of the assertion
----	------------------	---

### Attention

This macro must be called inside already established critical section. The call to `Q_onError()` happens inside that critical section.

### Backward Traceability

- DVP\_QP\_MC4\_D04\_09A: *Directive 4.9(Advisory): A function should be used in preference to a function-like macro where they are interchangeable (FALSE-POSITIVE diagnosis)* (false-positive)

### Forward Traceability

#### 15.33.2.6 Q\_ASSERT\_ID

```
#define Q_ASSERT_ID(  
    id_,  
    expr_)
```

##### Value:

```
do { \  
    QF_CRIT_STAT \  
    QF_CRIT_ENTRY(); \  
    (expr_) ? ((void)0) : Q_onError(&Q_this_module_[0], (id_)); \  
    QF_CRIT_EXIT(); \  
} while (false)
```

General-purpose assertion (with user-specified ID number)

## Details

Evaluates the Boolean expression `expr_` and does nothing else when it evaluates to 'true'. However, when `expr_` evaluates to 'false', the `Q_ASSERT_ID()` macro calls the no-return function `Q_onError()`.

## Parameters

in	<code>id_</code>	ID number (unique within the module) of the assertion
in	<code>expr_</code>	Boolean expression to check

## Attention

This macro uses critical section and the evaluation of the expression `expr_` as well as calling of `Q_onError()` happens inside the critical section.

The assertion expression (`expr_`) must be possibly simple, have **no side effects**, and quick to evaluate because the evaluation happens inside a critical section. Also, the expression must **NOT** call any functions that might use critical sections inside (because this would cause nesting of critical sections, which might not be supported).

## Backward Traceability

- DVP\_QP\_MC4\_D04\_09A: *Directive 4.9(Advisory): A function should be used in preference to a function-like macro where they are interchangeable (FALSE-POSITIVE diagnosis)* (false-positive)

## Forward Traceability

### 15.33.2.7 Q\_ERROR\_ID

```
#define Q_ERROR_ID(
    id_)
```

#### Value:

```
do { \
    QF_CRIT_STAT \
    QF_CRIT_ENTRY(); \
    Q_onError(&Q_this_module_[0], (id_)); \
    QF_CRIT_EXIT(); \
} while (false)
```

Assertion for a wrong path through the code (with user-specified ID)

## Details

Calls the `Q_onError()` callback if ever executed. This assertion takes the user-supplied parameter `id_` to identify the location of this assertion within the file. This avoids the volatility of using line numbers, which change whenever a line of code is added or removed upstream from the assertion.

## Parameters

in	<code>id_</code>	ID number (unique within the module) of the assertion
----	------------------	---

## Backward Traceability

- DVP\_QP\_MC4\_D04\_09A: *Directive 4.9(Advisory): A function should be used in preference to a function-like macro where they are interchangeable (FALSE-POSITIVE diagnosis)* (false-positive)

## Forward Traceability

### 15.33.2.8 Q\_ASSERT

```
#define Q_ASSERT(
    expr_)
```

**Value:**

`Q_ASSERT_ID(__LINE__, (expr_))`

General-purpose assertion (with ID provided in **LINE**)

**Details**

Equivalent to [Q\\_ASSERT\\_ID\(\)](#), except it uses **LINE** to identify the assertion within a file.

**Parameters**

in	<code>expr_</code>	Boolean expression to check
----	--------------------	-----------------------------

**Backward Traceability**

- DVP\_QP\_MC4\_D04\_09A: *Directive 4.9(Advisory): A function should be used in preference to a function-like macro where they are interchangeable (FALSE-POSITIVE diagnosis) (false-positive)*

### 15.33.2.9 Q\_ERROR

```
#define Q_ERROR()
```

**Value:**

`Q_ERROR_ID(__LINE__)`

Assertion for a wrong path through the code (with ID provided in **LINE**)

**Details**

Calls the [Q\\_onError\(\)](#) callback if ever executed.

**Note**

This macro identifies the problem location with the line number, which might change as the code is modified.

**Backward Traceability**

- DVP\_QP\_MC4\_D04\_09A: *Directive 4.9(Advisory): A function should be used in preference to a function-like macro where they are interchangeable (FALSE-POSITIVE diagnosis) (false-positive)*

### 15.33.2.10 Q\_REQUIRE\_ID

```
#define Q_REQUIRE_ID(
    id_,
    expr_)
```

**Value:**

`Q_ASSERT_ID((id_), (expr_))`

Assertion for checking a precondition (with user-specified ID number)

**Details**

Equivalent to [Q\\_ASSERT\\_ID\(\)](#), except the name provides a better documentation of the intention of this assertion.

**Parameters**

in	<i>id_</i>	ID number (unique within the module) of the assertion
in	<i>expr_</i>	Boolean expression

**Backward Traceability**

- DVP\_QP\_MC4\_D04\_09A: *Directive 4.9(Advisory): A function should be used in preference to a function-like macro where they are interchangeable (FALSE-POSITIVE diagnosis)* (false-positive)

**Forward Traceability****15.33.2.11 Q\_REQUIRE**

```
#define Q_REQUIRE(
    expr_)
```

**Value:**

[Q\\_ASSERT\(expr\\_\)](#)

Assertion for checking a precondition (with ID provided in **LINE**)

**Details**

Equivalent to [Q\\_ASSERT\(\)](#), except the name provides a better documentation of the intention of this assertion.

**Parameters**

in	<i>expr_</i>	Boolean expression
----	--------------	--------------------

**Backward Traceability**

- DVP\_QP\_MC4\_D04\_09A: *Directive 4.9(Advisory): A function should be used in preference to a function-like macro where they are interchangeable (FALSE-POSITIVE diagnosis)* (false-positive)

**Forward Traceability****15.33.2.12 Q\_REQUIRE\_INCRIT**

```
#define Q_REQUIRE_INCRIT(
    id_,
    expr_)
```

**Value:**

[Q\\_ASSERT\\_INCRIT\(\(id\\_\), \(expr\\_\)\)](#)

Assertion for checking a precondition (in critical section)

**Details**

Equivalent to [Q\\_ASSERT\\_INCRIT\(\)](#), except the name provides a better documentation of the intention of this assertion.

**Parameters**

in	<i>id_</i>	ID number (unique within the module) of the assertion
in	<i>expr_</i>	Boolean expression

**Backward Traceability**

- DVP\_QP\_MC4\_D04\_09A: *Directive 4.9(Advisory): A function should be used in preference to a function-like macro where they are interchangeable (FALSE-POSITIVE diagnosis)* (false-positive)

**Forward Traceability****15.33.2.13 Q\_ENSURE\_ID**

```
#define Q_ENSURE_ID(
    id_,
    expr_)
```

**Value:**

`Q_ASSERT_ID((id_), (expr_))`

Assertion for checking a postcondition (with user-specified ID number)

**Details**

Equivalent to `Q_ASSERT_ID()`, except the name provides a better documentation of the intention of this assertion.

**Parameters**

in	<i>id_</i>	ID number (unique within the module) of the assertion
in	<i>expr_</i>	Boolean expression

**Forward Traceability****15.33.2.14 Q\_ENSURE**

```
#define Q_ENSURE(
    expr_)
```

**Value:**

`Q_ASSERT(expr_)`

Assertion for checking a postcondition

**Details**

Equivalent to `Q_ASSERT()`, except the name provides a better documentation of the intention of this assertion.

**Parameters**

in	<i>expr_</i>	Boolean expression
----	--------------	--------------------

### Backward Traceability

- DVP\_QP\_MC4\_D04\_09A: Directive 4.9(Advisory): A function should be used in preference to a function-like macro where they are interchangeable (FALSE-POSITIVE diagnosis) (false-positive)

### Forward Traceability

#### 15.33.2.15 Q\_ENSURE\_INCRIT

```
#define Q_ENSURE_INCRIT(
    id_,
    expr_)
```

##### Value:

[Q\\_ASSERT\\_INCRIT\(\)](#)

Assertion for checking a postcondition (in critical section)

##### Details

Equivalent to [Q\\_ASSERT\\_INCRIT\(\)](#), except the name provides a better documentation of the intention of this assertion.

##### Parameters

in	<i>id_</i>	ID number (unique within the module) of the assertion
in	<i>expr_</i>	Boolean expression

### Backward Traceability

- DVP\_QP\_MC4\_D04\_09A: Directive 4.9(Advisory): A function should be used in preference to a function-like macro where they are interchangeable (FALSE-POSITIVE diagnosis) (false-positive)

### Forward Traceability

#### 15.33.2.16 Q\_INVARIANT\_ID

```
#define Q_INVARIANT_ID(
    id_,
    expr_)
```

##### Value:

[Q\\_ASSERT\\_ID\(\)](#)

Assertion for checking an invariant (with user-specified ID number)

##### Details

Equivalent to [Q\\_ASSERT\\_ID\(\)](#), except the name provides a better documentation of the intention of this assertion.

##### Parameters

in	<i>id_</i>	ID number (unique within the module) of the assertion
in	<i>expr_</i>	Boolean expression

### Backward Traceability

- DVP\_QP\_MC4\_D04\_09A: Directive 4.9(Advisory): A function should be used in preference to a function-like macro where they are interchangeable (FALSE-POSITIVE diagnosis) (false-positive)

#### 15.33.2.17 Q\_INVARIANT

```
#define Q_INVARIANT(
    expr_)
```

##### Value:

`Q_ASSERT(expr_)`

Assertion for checking an invariant.

##### Details

Equivalent to `Q_ASSERT()`, except the name provides a better documentation of the intention of this assertion.

##### Parameters

in	expr_	Boolean expression
----	-------	--------------------

### Backward Traceability

- DVP\_QP\_MC4\_D04\_09A: Directive 4.9(Advisory): A function should be used in preference to a function-like macro where they are interchangeable (FALSE-POSITIVE diagnosis) (false-positive)

### Forward Traceability

#### 15.33.2.18 Q\_INVARIANT\_INCRIT

```
#define Q_INVARIANT_INCRIT(
    id_,
    expr_)
```

##### Value:

`Q_ASSERT_INCRIT((id_), (expr_))`

#### 15.33.2.19 Q\_ASSERT\_STATIC

```
#define Q_ASSERT_STATIC(
    expr_)
```

##### Value:

`extern char Q_static_assert_{(expr_) ? 1 : -1}`

Static (compile-time) assertion.

This type of assertion deliberately causes a compile-time error when the `expr_` Boolean expression evaluates to FALSE. The macro exploits the fact that in C/C++ a dimension of an array cannot be negative. The compile-time assertion has no runtime side effects.

##### Parameters

in	expr_	Compile-time Boolean expression
----	-------	---------------------------------

**Note**

The static assertion macro is provided for backwards compatibility with older C standards. Newer C11 supports `_Static_assert()`, which should be used instead of [Q\\_ASSERT\\_STATIC\(\)](#).

**Backward Traceability**

- DVP\_QP\_MC4\_D04\_09A (false-positive)

**15.33.2.20 Q\_NORETURN**

```
#define Q_NORETURN __attribute__((__noreturn))
```

**15.33.2.21 Q\_DIM**

```
#define Q_DIM( array_) \
    (sizeof(array_) / sizeof((array_)[0U]))
```

**15.33.3 Function Documentation****15.33.3.1 Q\_onError()**

```
Q_NORETURN Q_onError( \
    char const *const module, \
    int_t const id)
```

Custom error handler Callback function invoked after detecting an error (part of QP Functional Safety (FuSa) Subsystem).

**Details**

This callback function needs to be defined in the application to perform any corrective action after an **unrecoverable error** has been detected. The [Q\\_onError\(\)](#) function is the last line of defense after the system failure and its implementation should be very **carefully** designed and **tested** under various fault conditions, including but not limited to: stack overflow/corruption, calling [Q\\_onError\(\)](#) from an ISR or other hardware exception, etc.

**Parameters**

in	<i>module</i>	name of the file/module in which the assertion failed (constant, zero-terminated C string)
in	<i>id</i>	ID of the assertion within the module. This could be a line number or a user-specified ID-number.

**Returns**

This callback function should **not return** (see [Q\\_NORETURN](#)), as continuation after an unrecoverable error makes no sense.

**Attention**

[Q\\_onError\(\)](#) must be called within a critical section (typically with interrupts **disabled**).

**Note**

During debugging, [Q\\_onError\(\)](#) is an ideal place to put a breakpoint. For deployment, it is **NOT RECOMMENDED** to implement [Q\\_onError\(\)](#) as an endless loop that ties up the CPU (denial of service).

Called by the following: [Q\\_ASSERT\\_ID\(\)](#), [Q\\_ERROR\\_ID\(\)](#), [Q\\_REQUIRE\\_ID\(\)](#), [Q\\_ENSURE\\_ID\(\)](#), [Q\\_INVARIANT\\_ID\(\)](#) as well as: [Q\\_ASSERT\(\)](#), [Q\\_ERROR\(\)](#), [Q\\_REQUIRE\(\)](#), [Q\\_ENSURE\(\)](#), and [Q\\_INVARIANT\(\)](#).

**Backward Traceability**

- FMEDA\_QA\_00: *Failure Mode: Fault detection and self-monitoring are inactive.*
- FMEDA\_QA\_01: *Failure Mode: Software resumes normal operation after detecting a fault.*
- FMEDA\_QA\_02: *Failure Mode: Custom Error Handler fails to reach Safe State due to already compromised system.*

**Forward Traceability**

## 15.34 qstamp.h File Reference

Date/time for time-stamping the QP builds (used in [QS](#) software tracing)

**Variables**

- char const [Q\\_BUILD\\_DATE](#) [12]
- char const [Q\\_BUILD\\_TIME](#) [9]

### 15.34.1 Detailed Description

Date/time for time-stamping the QP builds (used in [QS](#) software tracing)

### 15.34.2 Variable Documentation

#### 15.34.2.1 [Q\\_BUILD\\_DATE](#)

```
char const Q_BUILD_DATE[12]  [extern]
```

#### 15.34.2.2 [Q\\_BUILD\\_TIME](#)

```
char const Q_BUILD_TIME[9]  [extern]
```

## 15.35 qv.h File Reference

QV/C (non-preemptive kernel) platform-independent public interface.

**Classes**

- class [QV](#)  
*[QV](#) non-preemptive kernel ([QV](#) namespace emulated as a "class" in C)*
- class [QV\\_Attr](#)  
*Private attributes of the [QV](#) kernel.*

## Macros

- #define QF\_SCHED\_STAT\_
- #define QF\_SCHED\_LOCK\_(dummy)
- #define QF\_SCHED\_UNLOCK\_()
- #define QACTIVE\_EQUEUE\_WAIT\_(me\_)
- #define QACTIVE\_EQUEUE\_SIGNAL\_(me\_)
- #define QF\_EPOOL\_TYPE\_QMPool
- #define QF\_EPOOL\_INIT\_(p\_, poolSto\_, poolSize\_, evtSize\_)
- #define QF\_EPOOL\_EVENT\_SIZE\_(p\_)
- #define QF\_EPOOL\_GET\_(p\_, e\_, m\_, qsld\_)
- #define QF\_EPOOL\_PUT\_(p\_, e\_, qsld\_)

### 15.35.1 Detailed Description

QV/C (non-preemptive kernel) platform-independent public interface.

#### Backward Traceability

- DVP\_QP\_MO4\_D04\_08

### 15.35.2 Macro Definition Documentation

#### 15.35.2.1 QF\_SCHED\_STAT\_

```
#define QF_SCHED_STAT_
```

#### 15.35.2.2 QF\_SCHED\_LOCK\_

```
#define QF_SCHED_LOCK_(
    dummy)
```

##### Value:

```
((void) 0)
```

#### 15.35.2.3 QF\_SCHED\_UNLOCK\_

```
#define QF_SCHED_UNLOCK_()
```

##### Value:

```
((void) 0)
```

#### 15.35.2.4 QACTIVE\_EQUEUE\_WAIT\_

```
#define QACTIVE_EQUEUE_WAIT_(
    me_)
```

##### Value:

```
((void) 0)
```

#### 15.35.2.5 QACTIVE\_EQUEUE\_SIGNAL\_

```
#define QACTIVE_EQUEUE_SIGNAL_(
    me_)
```

##### Value:

```
QPSet_insert(&QV_priv_.readySet, (uint_fast8_t)(me_)->prio)
```

#### 15.35.2.6 QF\_EPOOL\_TYPE\_

```
#define QF_EPOOL_TYPE_QMPool
```

### 15.35.2.7 QF\_EPOOL\_INIT\_

```
#define QF_EPOOL_INIT_(
    p_,
    poolSto_,
    poolSize_,
    evtSize_)
```

**Value:**

```
(QMPool_init(&(p_), (poolSto_), (poolSize_), (evtSize_)))
```

### 15.35.2.8 QF\_EPOOL\_EVENT\_SIZE\_

```
#define QF_EPOOL_EVENT_SIZE_(
    p_)
```

**Value:**

```
((uint_fast16_t)(p_).blockSize)
```

### 15.35.2.9 QF\_EPOOL\_GET\_

```
#define QF_EPOOL_GET_(
    p_,
    e_,
    m_,
    qsId_)
```

**Value:**

```
((e_) = (QEvt *)QMPool_get(&(p_), (m_), (qsId_)))
```

### 15.35.2.10 QF\_EPOOL\_PUT\_

```
#define QF_EPOOL_PUT_(
    p_,
    e_,
    qsId_)
```

**Value:**

```
(QMPool_put(&(p_), (e_), (qsId_)))
```

## 15.36 qep\_hsm.c File Reference

[QHsm](#) class implementation

```
#include "qp_port.h"
#include "qsafe.h"
#include "qs_port.h"
#include "qs_pkg.h"
```

### Macros

- #define QHSM\_MAX\_NEST\_DEPTH\_ 6

### 15.36.1 Detailed Description

[QHsm](#) class implementation

### 15.36.2 Macro Definition Documentation

#### 15.36.2.1 QHSM\_MAX\_NEST\_DEPTH\_

```
#define QHSM_MAX_NEST_DEPTH_ 6
```

## 15.37 qep\_msm.c File Reference

[QMsm](#) class implementation

```
#include "qp_port.h"
#include "qsafe.h"
#include "qs_port.h"
#include "qs_pkg.h"
```

### Macros

- `#define QMSM_MAX_ENTRY_DEPTH_ ((int_fast8_t)4)`  
*maximum depth of entry levels in a MSM for tran. to history*

### 15.37.1 Detailed Description

[QMsm](#) class implementation

### 15.37.2 Macro Definition Documentation

#### 15.37.2.1 QMSM\_MAX\_ENTRY\_DEPTH\_

```
#define QMSM_MAX_ENTRY_DEPTH_ ((int_fast8_t)4)
maximum depth of entry levels in a MSM for tran. to history
```

## 15.38 qf\_act.c File Reference

defines Active Object management internal data

```
#include "qp_port.h"
#include "qp_pkg.h"
#include "qsafe.h"
#include "qs_port.h"
#include "qs_pkg.h"
```

### Functions

- `uint_fast8_t QF_LOG2 (QPSetBits const bitmask)`  
*Log-base-2 calculation when hardware acceleration is NOT provided ([QF\\_LOG2](#) not defined)*

### Variables

- `char const QP_versionStr [24] = "QP/C " QP_VERSION_STR`
- `QF_Attr QF_priv_`

### 15.38.1 Detailed Description

defines Active Object management internal data

- `QActive::QActive_registry_`
- `QF::QF_priv_`
- `QF_LOG2()`

## 15.38.2 Function Documentation

### 15.38.2.1 QF\_LOG2()

```
uint_fast8_t QF_LOG2 (
    QPSetBits const bitmask)
```

Log-base-2 calculation when hardware acceleration is NOT provided (QF\_LOG2 not defined)

## 15.38.3 Variable Documentation

### 15.38.3.1 QP\_versionStr

```
char const QP_versionStr[24] = "QP/C " QP_VERSION_STR
```

### 15.38.3.2 QF\_priv\_

```
QF_Attr QF_priv_
```

## 15.39 qf\_actq.c File Reference

Active Object queue operations based on the QEQueue class.

```
#include "qp_port.h"
#include "qp_pkg.h"
#include "qsafe.h"
#include "qs_port.h"
#include "qs_pkg.h"
```

### 15.39.1 Detailed Description

Active Object queue operations based on the QEQueue class.

#### Details

- [QActive::QActive\\_post\\_\(\)](#)
- [QActive::QActive\\_postLIFO\\_\(\)](#)

#### Note

This source file is needed only when the QActive base class is configured to use QEQueue for the event queue. In QP Framework ports to 3-rd party RTOSes that use a different event queue, this file should **NOT** be included in the build.

## 15.40 qf\_defer.c File Reference

event deferral facilities

```
#include "qp_port.h"
#include "qp_pkg.h"
#include "qsafe.h"
#include "qs_port.h"
#include "qs_pkg.h"
```

### 15.40.1 Detailed Description

event deferral facilities

Details

- [QActive::QActive\\_defer\(\)](#)
- [QActive::QActive\\_recall\(\)](#)
- [QActive::QActive\\_flushDeferred\(\)](#)

## 15.41 qf\_dyn.c File Reference

dynamic event management facilities

```
#include "qp_port.h"
#include "qp_pkg.h"
#include "qsafe.h"
#include "qs_port.h"
#include "qs_pkg.h"
```

### 15.41.1 Detailed Description

dynamic event management facilities

Details

- [QF::QF\\_poolInit\(\)](#)
- [QF::QF\\_newX\\_\(\)](#)
- [QF::QF\\_gc\(\)](#)
- [QF::QF\\_poolGetMaxBlockSize\(\)](#)

## 15.42 qf\_mem.c File Reference

[QMPool](#) implementation

```
#include "qp_port.h"
#include "qp_pkg.h"
#include "qsafe.h"
#include "qs_port.h"
#include "qs_pkg.h"
```

### 15.42.1 Detailed Description

[QMPool](#) implementation

Details

- [QMPool::QMPool\\_init\(\)](#)
- [QMPool::QMPool\\_get\(\)](#)
- [QMPool::QMPool\\_put\(\)](#)

## 15.43 qf\_ps.c File Reference

publish-subscribe facilities

```
#include "qp_port.h"
#include "qp_pkg.h"
#include "qsafe.h"
#include "qs_port.h"
#include "qs_pkg.h"
```

### 15.43.1 Detailed Description

publish-subscribe facilities

Details

- [QActive::QActive\\_psInit\(\)](#)
- [QActive::QActive\\_publish\\_\(\)](#)
- [QActive::QActive\\_subscribe\(\)](#)
- [QActive::QActive\\_unsubscribe\(\)](#)
- [QActive::QActive\\_unsubscribeAll\(\)](#)

## 15.44 qf\_qact.c File Reference

Active Object management facilities.

```
#include "qp_port.h"
#include "qp_pkg.h"
#include "qsafe.h"
#include "qs_port.h"
#include "qs_pkg.h"
```

### 15.44.1 Detailed Description

Active Object management facilities.

Details

- [QActive::QActive\\_ctor\(\)](#)
- [QActive::QActive\\_register\\_\(\)](#)
- [QActive::QActive\\_unregister\\_\(\)](#)

## 15.45 qf\_qeq.c File Reference

[QEQueue](#) implementation

```
#include "qp_port.h"
#include "qp_pkg.h"
#include "qsafe.h"
#include "qs_port.h"
#include "qs_pkg.h"
```

### 15.45.1 Detailed Description

QEQueue implementation

## 15.46 qf\_qmact.c File Reference

QMActive implementation

```
#include "qp_port.h"
#include "qp_pkg.h"
#include "qsafe.h"
#include "qs_port.h"
#include "qs_pkg.h"
```

### 15.46.1 Detailed Description

QMActive implementation

## 15.47 qf\_time.c File Reference

QTimeEvt implementation

```
#include "qp_port.h"
#include "qp_pkg.h"
#include "qsafe.h"
#include "qs_port.h"
#include "qs_pkg.h"
```

### 15.47.1 Detailed Description

QTimeEvt implementation

## 15.48 qk.c File Reference

preemptive non-blocking QK kernel implementation

```
#include "qp_port.h"
#include "qp_pkg.h"
#include "qsafe.h"
#include "qs_port.h"
#include "qs_pkg.h"
```

### 15.48.1 Detailed Description

preemptive non-blocking QK kernel implementation

## 15.49 qstamp.c File Reference

QS time stamp.

```
#include "qstamp.h"
```

## Variables

- char const Q\_BUILD\_DATE [12] = \_\_DATE\_\_
- char const Q\_BUILD\_TIME [9] = \_\_TIME\_\_

### 15.49.1 Detailed Description

QS time stamp.

#### Details

This file shall be recompiled in every software build to time-stamp it in order to distinguish that build from any other.

### 15.49.2 Variable Documentation

#### 15.49.2.1 Q\_BUILD\_DATE

```
char const Q_BUILD_DATE[12] = __DATE__
```

#### 15.49.2.2 Q\_BUILD\_TIME

```
char const Q_BUILD_TIME[9] = __TIME__
```

## 15.50 qv.c File Reference

non-preemptive non-blocking QV kernel implementation

```
#include "qp_port.h"
#include "qp_pkg.h"
#include "qsafe.h"
#include "qs_port.h"
#include "qs_pkg.h"
```

### 15.50.1 Detailed Description

non-preemptive non-blocking QV kernel implementation

# Index

act  
    QAsmAttr, 329  
    QMTranActTable, 378  
    QTimeEvt, 393  
Active Objects, 87  
actPrio  
    QK\_Attr, 360  
actThre  
    QK\_Attr, 360  
addr  
    QS\_TProbe, 381  
Algorithm Viewpoint, 226  
API Reference, 231  
api.dox, 417  
ARM Cortex-M, 31  
ARM Cortex-R, 60

bits  
    QPSet, 380  
blockSize  
    QMPool, 368

char\_t  
    qpc.h, 455  
cmd\_options.dox, 401  
    Q\_SPY, 401  
    Q\_UTEST, 402  
    QP\_CONFIG, 401  
Context Viewpoint, 170  
crit\_stat\_t  
    qp\_port.h, 415  
critEntry  
    qp\_port.h, 415  
critExit  
    qp\_port.h, 415  
ctr  
    QTimeEvt, 393

data  
    QS\_TProbe, 381  
Deprecated APIs, 237  
Dining Philosophers Problem (DPP), 15  
dir.dox, 417  
dispatch  
    QAsmVtable, 330  
DOC\_SAS\_QP, 165

DOC\_SDS\_QP, 179  
DOC\_SRS\_QP, 79  
Downloading & Installing QP/C, 8

embOS, 62  
end  
    QEQueue, 336  
    QMPool, 368  
entryAction  
    QMState, 377  
enum\_t  
    qp.h, 447  
ePool\_  
    QF\_Attr, 349  
eQueue  
    QActive, 326  
Event Delivery Mechanisms, 117  
Event Memory Management, 126  
Events, 98  
Examples, 73  
exitAction  
    QMState, 377

flags  
    QTimeEvt, 394  
freeHead  
    QMPool, 368  
FreeRTOS, 63  
frontEvt  
    QEQueue, 336  
fun  
    QAsmAttr, 329  
getStateHandler  
    QAsmVtable, 330  
Getting Started, 7

head  
    QEQueue, 336  
help.dox, 417  
history.dox, 401

idx  
    QS\_TProbe, 381  
init  
    QAsmVtable, 330

initAction  
     QMState, 377  
 int\_t  
     qp.h, 447  
 Interaction Viewpoint, 193  
 Interface Viewpoint, 229  
 interval  
     QTimeEvt, 393  
 intNest  
     QK\_Attr, 360  
 isIn  
     QAsmVtable, 330  
 lockCeil  
     QK\_Attr, 360  
 Low-Power Example, 20  
 maxPool\_  
     QF\_Attr, 349  
 MSP430, 61  
 Native Ports (Built-in Kernels), 31  
 next  
     QTimeEvt, 393  
 nextPrio  
     QK\_Attr, 360  
 nFree  
     QEQueue, 336  
     QMPool, 368  
 nMin  
     QEQueue, 336  
     QMPool, 368  
 Non-Functional Requirements, 160  
 Non-Preemptive Kernel, 144  
 Non-preemptive QV Kernel, 36  
 nTot  
     QMPool, 368  
 obj  
     QAsmAttr, 330  
 osObject  
     QActive, 326  
 Overview, 1, 83  
 PC-Lint-Plus, 28  
 poolNum\_  
     QEvt, 340  
 Ports, 27  
 Ports to General-Purpose OSes, 70  
 Ports to Third-Party RTOS, 61  
 POSIX (Multithreaded), 71  
 POSIX-QV (Single Threaded), 71  
 Preemptive Dual-Mode Kernel, 155  
 Preemptive Non-Blocking Kernel, 149  
 Preemptive Non-Blocking QK Kernel, 43  
 prio  
     QActive, 325  
 pthre  
     QActive, 325  
 Q\_ACTION\_CAST  
     qp.h, 428  
 Q\_ACTION\_NULL  
     qp.h, 429  
 Q\_ALLEGGE  
     qpc.h, 453  
 Q\_ALLEGGE\_ID  
     qpc.h, 452  
 Q\_ASM\_UPCAST  
     qp.h, 433  
 Q\_ASSERT  
     qsafe.h, 473  
 Q\_ASSERT\_COMPILE  
     qpc.h, 453  
 Q\_ASSERT\_ID  
     qsafe.h, 472  
 Q\_ASSERT\_INCRIT  
     qsafe.h, 471  
 Q\_ASSERT\_STATIC  
     qsafe.h, 478  
 Q\_BUILD\_DATE  
     qstamp.c, 488  
     qstamp.h, 480  
 Q\_BUILD\_TIME  
     qstamp.c, 488  
     qstamp.h, 480  
 Q\_DELETE\_REF  
     qp.h, 442  
 Q\_DIM  
     qp.h, 426  
     qsafe.h, 479  
 Q\_EMPTY\_SIG  
     qp.h, 430  
 Q\_ENSURE  
     qsafe.h, 476  
 Q\_ENSURE\_ID  
     qsafe.h, 476  
 Q\_ENSURE\_INCRIT  
     qsafe.h, 477  
 Q\_ENTRY\_SIG  
     qp.h, 430  
 Q\_ERROR  
     qsafe.h, 474  
 Q\_ERROR\_ID  
     qsafe.h, 473  
 Q\_ERROR\_INCRIT  
     qsafe.h, 472  
 Q\_EVT\_CAST  
     qp.h, 427

Q\_EXIT\_SIG  
    qp.h, 430  
Q\_HANDLED  
    qp.h, 435  
Q\_HSM\_UPCAST  
    qp.h, 433  
Q\_INIT\_SIG  
    qp.h, 430  
Q\_INVARIANT  
    qsafe.h, 478  
Q\_INVARIANT\_ID  
    qsafe.h, 477  
Q\_INVARIANT\_INCRIT  
    qsafe.h, 478  
Q\_MSM\_UPCAST  
    qp.h, 436  
Q\_NEW  
    qp.h, 439  
Q\_NEW\_REF  
    qp.h, 442  
Q\_NEW\_X  
    qp.h, 440  
Q\_NORETURN  
    qp\_port.h, 412  
    qsafe.h, 479  
Q\_onAssert  
    qpc.h, 452  
Q\_onError  
    qsafe.h, 479  
Q\_PRIO  
    qp.h, 439  
Q\_PTR2UINT\_CAST\_  
    qp\_pkg.h, 450  
Q\_REQUIRE  
    qsafe.h, 475  
Q\_REQUIRE\_ID  
    qsafe.h, 474  
Q\_REQUIRE\_INCRIT  
    qsafe.h, 475  
Q\_RET\_ENTRY  
    qp.h, 429  
Q\_RET\_EXIT  
    qp.h, 429  
Q\_RET\_HANDLED  
    qp.h, 429  
Q\_RET\_IGNORED  
    qp.h, 429  
Q\_RET\_NULL  
    qp.h, 429  
Q\_RET\_SUPER  
    qp.h, 429  
Q\_RET\_TRAN  
    qp.h, 429  
Q\_RET\_TRAN\_HIST  
    qp.h, 430  
Q\_RET\_TRAN\_INIT  
    qp.h, 429  
Q\_RET\_UNHANDLED  
    qp.h, 429  
Q\_SIGNAL\_SIZE  
    qp\_config.h, 406  
Q\_SPY  
    cmd\_options.dox, 401  
Q\_STATE\_CAST  
    qp.h, 428  
Q\_SUPER  
    qp.h, 435  
Q\_TRAN  
    qp.h, 434  
Q\_TRAN\_HIST  
    qp.h, 434  
Q\_UINT2PTR\_CAST  
    qp.h, 427  
Q\_UNHANDLED  
    qp.h, 436  
Q\_UNSAFE  
    qp\_config.h, 406  
Q\_UNUSED\_PAR  
    qp.h, 426  
Q\_USER\_SIG  
    qp.h, 430  
Q\_UTEST  
    cmd\_options.dox, 402  
QActionHandler  
    qp.h, 448  
QActive, 311  
    eQueue, 326  
    osObject, 326  
    prio, 325  
    pthre, 325  
    QActive\_ctor, 314  
    QActive\_defer, 323  
    QActive\_evtLoop\_, 325  
    QActive\_flushDeferred, 324  
    QActive\_get\_, 318  
    QActive\_getQueueMin, 320  
    QActive\_maxPubSignal\_, 326  
    QActive\_post\_, 316  
    QActive\_postFIFO\_, 325  
    QActive\_postLIFO\_, 318  
    QActive\_psInit, 319  
    QActive\_publish\_, 319  
    QActive\_recall, 324  
    QActive\_register\_, 315  
    QActive\_registry\_, 326  
    QActive\_setAttr, 314  
    QActive\_start, 314  
    QActive\_stop, 315

QActive\_subscribe, 321  
 QActive\_subscrList\_, 326  
 QActive\_unregister\_, 316  
 QActive\_unsubscribe, 322  
 QActive\_unsubscribeAll, 322  
 super, 325  
 thread, 325  
**QACTIVE\_CAN\_STOP**  
 qp\_config.h, 407  
**QACTIVE\_CAST\_**  
 qp\_pkg.h, 450  
**QActive\_ctor**  
 QActive, 314  
**QActive\_defer**  
 QActive, 323  
**QACTIVE\_EQUEUE\_SIGNAL\_**  
 qk.h, 419  
 qv.h, 481  
**QACTIVE\_EQUEUE\_TYPE**  
 qp\_port.h, 412  
**QACTIVE\_EQUEUE\_WAIT\_**  
 qk.h, 419  
 qv.h, 481  
**QActive\_evtLoop\_**  
 QActive, 325  
**QActive\_flushDeferred**  
 QActive, 324  
**QActive\_get\_**  
 QActive, 318  
**QActive\_getQueueMin**  
 QActive, 320  
**QActive\_maxPubSignal\_**  
 QActive, 326  
**QACTIVE\_OS\_OBJ\_TYPE**  
 qp\_port.h, 412  
**QACTIVE\_POST**  
 qp.h, 443  
**QActive\_post\_**  
 QActive, 316  
**QACTIVE\_POST\_LIFO**  
 qp.h, 446  
**QACTIVE\_POST\_X**  
 qp.h, 443  
**QActive\_postFIFO\_**  
 QActive, 325  
**QActive\_postLIFO\_**  
 QActive, 318  
**QActive\_psInit**  
 QActive, 319  
**QACTIVE\_PUBLISH**  
 qp.h, 444  
**QActive\_publish\_**  
 QActive, 319  
**QActive\_recall**  
 QActive, 324  
**QActive\_register\_**  
 QActive, 315  
**QActive\_registry\_**  
 QActive, 326  
**QActive\_setAttr**  
 QActive, 314  
**QACTIVE\_START**  
 qpc.h, 452  
**QActive\_start**  
 QActive, 314  
**QActive\_stop**  
 QActive, 315  
**QActive\_subscribe**  
 QActive, 321  
**QActive\_subscrList\_**  
 QActive, 326  
**QACTIVE\_THREAD\_TYPE**  
 qp\_port.h, 412  
**QActive\_unregister\_**  
 QActive, 316  
**QActive\_unsubscribe**  
 QActive, 322  
**QActive\_unsubscribeAll**  
 QActive, 322  
**QActiveDummy**, 327  
**QAsm**, 327  
 QAsm\_ctor, 328  
 state, 328  
 temp, 328  
 vptr, 328  
**QAsm\_ctor**  
 QAsm, 328  
**QASM\_DISPATCH**  
 qp.h, 431  
**QASM\_INIT**  
 qp.h, 430  
**QASM\_IS\_IN**  
 qp.h, 432  
**QAsmAttr**, 329  
 act, 329  
 fun, 329  
 obj, 330  
 tabl, 329  
 thr, 329  
**QAsmVtable**, 330  
 dispatch, 330  
 getStateHandler, 330  
 init, 330  
 isIn, 330  
 qep\_hsm.c, 482  
 QHSM\_MAX\_NEST\_DEPTH\_, 482  
 qep msm.c, 483  
 QMSM\_MAX\_ENTRY\_DEPTH\_, 483

QEQueue, 331  
end, 336  
frontEvt, 336  
head, 336  
nFree, 336  
nMin, 336  
QEQueue\_get, 334  
QEQueue\_getNFree, 334  
QEQueue\_getNMin, 335  
QEQueue\_init, 332  
QEQueue\_isEmpty, 335  
QEQueue\_post, 332  
QEQueue\_postLIFO, 333  
ring, 336  
tail, 336  
qqueue.dox, 402  
QEQueueCtr, 402  
QF\_EQUEUE\_CTR\_SIZE, 402  
qqueue.h, 417  
QEQueueCtr, 418  
QEQueue\_get  
QEQueue, 334  
QEQueue\_getNFree  
QEQueue, 334  
QEQueue\_getNMin  
QEQueue, 335  
QEQueue\_init  
QEQueue, 332  
QEQueue\_isEmpty  
QEQueue, 335  
QEQueue\_post  
QEQueue, 332  
QEQueue\_postLIFO  
QEQueue, 333  
QEQueueCtr  
qqueue.dox, 402  
qqueue.h, 418  
QEvt, 337  
poolNum\_, 340  
QEvt\_ctor, 338  
QEvt\_init, 338  
QEvt\_refCtr\_dec\_, 340  
QEvt\_refCtr\_inc\_, 339  
refCtr\_, 340  
sig, 340  
QEvt\_ctor  
QEvt, 338  
QEVT\_DYNAMIC  
qp.h, 427  
QEvt\_init  
QEvt, 338  
QEVT\_INITIALIZER  
qp.h, 427  
QEVT\_PAR\_INIT  
qp\_config.h, 407  
QEvt\_refCtr\_dec\_  
QEvt, 340  
QEvt\_refCtr\_inc\_  
QEvt, 339  
QEvtPtr  
qp.h, 448  
QF, 341  
QF\_bzero\_, 348  
QF\_deleteRef\_, 348  
QF\_gc, 347  
QF\_gcFromISR, 348  
QF\_getPoolMin, 345  
QF\_init, 342  
QF\_newRef\_, 347  
QF\_newX\_, 346  
QF\_onCleanup, 343  
QF\_onContextSw, 344  
QF\_onStartup, 343  
QF\_poolGetMaxBlockSize, 345  
QF\_poolInit, 344  
QF\_priv\_, 349  
QF\_psInit, 349  
QF\_run, 343  
QF\_stop, 342  
qf\_act.c, 483  
QF\_LOG2, 484  
QF\_priv\_, 484  
QP\_versionStr, 484  
qf\_actq.c, 484  
QF\_Attr, 349  
ePool\_, 349  
maxPool\_, 349  
QF\_bzero\_  
QF, 348  
QF\_CRIT\_ENTRY  
qp\_port.h, 413  
qsafe.h, 471  
QF\_CRIT\_EXIT  
qp\_port.h, 413  
qsafe.h, 471  
QF\_CRIT\_EXIT\_NOP  
qp.h, 447  
QF\_CRIT\_STAT  
qp\_port.h, 413  
qsafe.h, 471  
qf\_defer.c, 484  
QF\_deleteRef\_  
QF, 348  
qf\_dyn.c, 485  
QF\_EPOOL\_EVENT\_SIZE\_  
qk.h, 419  
qv.h, 482  
QF\_EPOOL\_GET\_

qk.h, 420  
 qv.h, 482  
**QF\_EPOOL\_INIT\_**  
 qk.h, 419  
 qv.h, 481  
**QF\_EPOOL\_PUT\_**  
 qk.h, 420  
 qv.h, 482  
**QF\_EPOOL\_TYPE\_**  
 qk.h, 419  
 qv.h, 481  
**QF\_EQUEUE\_CTR\_SIZE**  
 qenqueue.dox, 402  
 qp\_config.h, 408  
**QF\_EVENT\_SIZ\_SIZE**  
 qp\_config.h, 407  
**QF\_gc**  
 QF, 347  
**QF\_gcFromISR**  
 QF, 348  
**QF\_getPoolMin**  
 QF, 345  
**QF\_getQueueMin**  
 qpc.h, 454  
**QF\_init**  
 QF, 342  
**QF\_INT\_DISABLE**  
 qp\_port.h, 412  
**QF\_INT\_ENABLE**  
 qp\_port.h, 413  
**QF\_LOG2**  
 qf\_act.c, 484  
 qp.h, 449  
**QF\_MAX\_ACTIVE**  
 qp\_config.h, 406  
**QF\_MAX\_EPOOL**  
 qp\_config.h, 406  
**QF\_MAX\_TICK\_RATE**  
 qp\_config.h, 407  
 qf\_mem.c, 485  
**QF\_MEM\_ISOLATE**  
 qp\_config.h, 409  
**QF\_MPOOL\_CTR\_SIZE**  
 qp\_config.h, 408  
**QF\_MPOOL\_EL**  
 qmpool.h, 421  
**QF\_MPOOL\_SIZ\_SIZE**  
 qp\_config.h, 408  
**QF\_newRef\_**  
 QF, 347  
**QF\_newX\_**  
 QF, 346  
**QF\_NO\_MARGIN**  
 qp.h, 439

QF\_ON\_CONTEXT\_SW  
 qp\_config.h, 409  
**QF\_onCleanup**  
 QF, 343  
**QF\_onContextSw**  
 QF, 344  
**QF\_onStartup**  
 QF, 343  
**QF\_poolGetMaxBlockSize**  
 QF, 345  
**QF\_poolInit**  
 QF, 344  
**QF\_priv\_**  
 QF, 349  
 qf\_act.c, 484  
**qf\_ps.c**, 486  
**QF\_psInit**  
 QF, 349  
**QF\_PUBLISH**  
 qpc.h, 454  
 qf\_qact.c, 486  
 qf\_qeq.c, 486  
 qf\_qmact.c, 487  
**QF\_run**  
 QF, 343  
**QF\_SCHED\_LOCK\_**  
 qk.h, 418  
 qv.h, 481  
**QF\_SCHED\_STAT\_**  
 qk.h, 418  
 qv.h, 481  
**QF\_SCHED\_UNLOCK\_**  
 qk.h, 419  
 qv.h, 481  
**QF\_stop**  
 QF, 342  
**QF\_TICK**  
 qpc.h, 454  
**QF\_TICK\_X**  
 qpc.h, 454  
 qf\_time.c, 487  
**QF\_TIMEEVT\_CTR\_SIZE**  
 qp\_config.h, 408  
**QFreeBlock**, 350  
**QHsm**, 350  
 QHsm\_childState, 355  
 QHsm\_ctor, 352  
 QHsm\_dispatch\_, 353  
 QHsm\_enter\_target\_, 356  
 QHsm\_getStateHandler\_, 354  
 QHsm\_init\_, 352  
 QHsm\_isIn\_, 353  
 QHsm\_state, 355  
 QHsm\_top, 354

QHsm\_tran\_complex\_, 356  
QHsm\_tran\_simple\_, 356  
super, 356  
QHsm\_childState  
  QHsm, 355  
QHsm\_ctor  
  QHsm, 352  
QHSM\_DISPATCH  
  qpc.h, 453  
QHsm\_dispatch\_  
  QHsm, 353  
QHsm\_enter\_target\_  
  QHsm, 356  
QHsm\_getStateHandler\_  
  QHsm, 354  
QHSM\_INIT  
  qpc.h, 453  
QHsm\_init\_  
  QHsm, 352  
QHsm\_isIn  
  qpc.h, 454  
QHsm\_isIn\_  
  QHsm, 353  
QHSM\_MAX\_NEST\_DEPTH\_  
  qep\_hsm.c, 482  
QHsm\_state  
  QHsm, 355  
QHsm\_top  
  QHsm, 354  
QHsm\_tran\_complex\_  
  QHsm, 356  
QHsm\_tran\_simple\_  
  QHsm, 356  
QHsmDummy, 356  
QK, 357  
  QK\_activate\_, 357  
  QK\_onIdle, 359  
  QK\_priv\_, 359  
  QK\_sched\_, 357  
  QK\_sched\_act\_, 357  
  QK\_schedLock, 358  
  QK\_schedUnlock, 358  
qk.c, 487  
qk.dox, 403  
qk.h, 418  
  QACTIVE\_EQUEUE\_SIGNAL\_, 419  
  QACTIVE\_EQUEUE\_WAIT\_, 419  
  QF\_EPOOL\_EVENT\_SIZE\_, 419  
  QF\_EPOOL\_GET\_, 420  
  QF\_EPOOL\_INIT\_, 419  
  QF\_EPOOL\_PUT\_, 420  
  QF\_EPOOL\_TYPE\_, 419  
  QF\_SCHED\_LOCK\_, 418  
  QF\_SCHED\_STAT\_, 418  
QF\_SCHED\_UNLOCK\_, 419  
QSchedStatus, 420  
QK\_activate\_  
  QK, 357  
QK\_Attr, 359  
  actPrio, 360  
  actThre, 360  
  intNest, 360  
  lockCeil, 360  
  nextPrio, 360  
  readySet, 360  
QK\_ISR\_CONTEXT\_  
  qp\_port.h, 413  
QK\_ISR\_ENTRY  
  qp\_port.h, 414  
QK\_ISR\_EXIT  
  qp\_port.h, 414  
QK\_onIdle  
  QK, 359  
QK\_priv\_  
  QK, 359  
QK\_sched\_  
  QK, 357  
QK\_sched\_act\_  
  QK, 357  
QK\_schedLock  
  QK, 358  
QK\_schedUnlock  
  QK, 358  
QK\_USE\_IRQ\_HANDLER  
  qp\_config.h, 410  
QK\_USE\_IRQ\_NUM  
  qp\_config.h, 410  
QM\_ENTRY  
  qp.h, 436  
QM\_EXIT  
  qp.h, 437  
QM\_HANDLED  
  qp.h, 438  
QM\_STATE\_NULL  
  qp.h, 438  
QM\_SUPER  
  qp.h, 438  
QM\_SUPER\_SUB  
  qpc.h, 451  
QM\_TRAN  
  qp.h, 437  
QM\_TRAN\_EP  
  qpc.h, 451  
QM\_TRAN\_HIST  
  qp.h, 438  
QM\_TRAN\_INIT  
  qp.h, 437  
QM\_TRAN\_XP

qpc.h, 452  
 QM\_UNHANDLED  
 qp.h, 438  
 QMAActive, 360  
     QMAActive\_ctor, 363  
     super, 363  
 QMAActive\_ctor  
     QMAActive, 363  
 QMPool, 363  
     blockSize, 368  
     end, 368  
     freeHead, 368  
     nFree, 368  
     nMin, 368  
     nTot, 368  
     QMPool\_get, 366  
     QMPool\_init, 364  
     QMPool\_put, 367  
     start, 368  
 qmpool.dox, 403  
 qmpool.h, 420  
     QF\_MPOOL\_EL, 421  
     QMPoolCtr, 421  
     QMPoolSize, 421  
 QMPool\_get  
     QMPool, 366  
 QMPool\_init  
     QMPool, 364  
 QMPool\_put  
     QMPool, 367  
 QMPoolCtr  
     qmpool.h, 421  
 QMPoolSize  
     qmpool.h, 421  
 QMsm, 368  
     QMsm\_childStateObj, 374  
     QMsm\_ctor, 370  
     QMsm\_dispatch\_, 372  
     QMsm\_enterHistory\_, 375  
     QMsm\_execTatbl\_, 374  
     QMsm\_exitToTranSource\_, 375  
     QMsm\_getStateHandler\_, 373  
     QMsm\_init\_, 371  
     QMsm\_isIn\_, 372  
     QMsm\_stateObj, 373  
     super, 376  
 QMsm\_childStateObj  
     QMsm, 374  
 QMsm\_ctor  
     QMsm, 370  
 QMsm\_dispatch\_  
     QMsm, 372  
 QMsm\_enterHistory\_  
     QMsm, 375  
     QMsm\_execTatbl\_,  
     QMsm\_exitToTranSource\_,  
     QMsm\_getStateHandler\_,  
     QMsm\_init\_,  
     QMsm\_isIn\_,  
     QMsm\_stateObj,  
     super,  
 QMsm\_childStateObj  
     QMsm, 374  
 QMsm\_ctor  
     QMsm, 370  
 QMsm\_dispatch\_  
     QMsm, 372  
 QMsm\_enterHistory\_  
     QMsm, 375  
     QMsm\_execTatbl\_,  
     QMsm\_exitToTranSource\_,  
     QMsm\_getStateHandler\_,  
     QMsm\_init\_,  
     QMsm\_isIn\_,  
     QMsm\_stateObj,  
     super,  
 QMsm\_execTatbl\_  
     QMsm, 374  
 QMsm\_exitToTranSource\_  
     QMsm, 375  
 QMsm\_getStateHandler\_  
     QMsm, 373  
 QMsm\_init\_  
     QMsm, 371  
 QMsm\_isIn\_  
     QMsm, 372  
 QMSM\_MAX\_ENTRY\_DEPTH\_  
     qep\_msm.c, 483  
 QMsm\_stateObj  
     QMsm, 373  
 QMState, 376  
     entryAction, 377  
     exitAction, 377  
     initAction, 377  
     stateHandler, 377  
     superstate, 377  
 QMTranActTable, 377  
     act, 378  
     target, 378  
 qp-exa.dox, 417  
 qp-gs.dox, 417  
 qp-main.dox, 417  
 qp-ports.dox, 417  
 qp.dox, 403  
 qp.h, 422  
     enum\_t, 447  
     int\_t, 447  
     Q\_ACTION\_CAST, 428  
     Q\_ACTION\_NULL, 429  
     Q\_ASM\_UPCAST, 433  
     Q\_DELETE\_REF, 442  
     Q\_DIM, 426  
     Q\_EMPTY\_SIG, 430  
     Q\_ENTRY\_SIG, 430  
     Q\_EVT\_CAST, 427  
     Q\_EXIT\_SIG, 430  
     Q\_HANDLED, 435  
     Q\_HSM\_UPCAST, 433  
     Q\_INIT\_SIG, 430  
     Q\_MSM\_UPCAST, 436  
     Q\_NEW, 439  
     Q\_NEW\_REF, 442  
     Q\_NEW\_X, 440  
     Q\_PRIO, 439  
     Q\_RET\_ENTRY, 429  
     Q\_RET\_EXIT, 429  
     Q\_RET\_HANDLED, 429  
     Q\_RET\_IGNORED, 429  
     Q\_RET\_NULL, 429  
     Q\_RET\_SUPER, 429

Q\_RET\_TRAN, 429  
Q\_RET\_TRAN\_HIST, 430  
Q\_RET\_TRAN\_INIT, 429  
Q\_RET\_UNHANDLED, 429  
Q\_STATE\_CAST, 428  
Q\_SUPER, 435  
Q\_TRAN, 434  
Q\_TRAN\_HIST, 434  
Q\_UINT2PTR\_CAST, 427  
Q\_UNHANDLED, 436  
Q\_UNUSED\_PAR, 426  
Q\_USER\_SIG, 430  
QActionHandler, 448  
QACTIVE\_POST, 443  
QACTIVE\_POST\_LIFO, 446  
QACTIVE\_POST\_X, 443  
QACTIVE\_PUBLISH, 444  
QASM\_DISPATCH, 431  
QASM\_INIT, 430  
QASM\_IS\_IN, 432  
QEVT\_DYNAMIC, 427  
QEVT\_INITIALIZER, 427  
QEvtPtr, 448  
QF\_CRIT\_EXIT\_NOP, 447  
QF\_LOG2, 449  
QF\_NO\_MARGIN, 439  
QM\_ENTRY, 436  
QM\_EXIT, 437  
QM\_HANDLED, 438  
QM\_STATE\_NULL, 438  
QM\_SUPER, 438  
QM\_TRAN, 437  
QM\_TRAN\_HIST, 438  
QM\_TRAN\_INIT, 437  
QM\_UNHANDLED, 438  
QP\_RELEASE, 426  
QP\_VERSION, 426  
QP\_VERSION\_STR, 426  
QP\_versionStr, 449  
QPrioSpec, 448  
QPSetBits, 449  
QSignal, 447  
QState, 448  
QStateHandler, 448  
QTICKER\_TRIG, 446  
QTIMEEV\_TICK, 447  
QTIMEEV\_TICK\_X, 445  
QTimeEvtCtr, 449  
QXThreadHandler, 448  
QP/C Tutorial, 12  
QP\_API\_VERSION  
    qp\_config.h, 405  
    qpc.h, 451  
QP\_CONFIG  
    cmd\_options.dox, 401  
    qp\_config.h, 403  
        Q\_SIGNAL\_SIZE, 406  
        Q\_UNSAFE, 406  
        QACTIVE\_CAN\_STOP, 407  
        QEVT\_PAR\_INIT, 407  
        QEVT\_QUEUE\_CTR\_SIZE, 408  
        QEVT\_EVENT\_SIZ\_SIZE, 407  
        QF\_MAX\_ACTIVE, 406  
        QF\_MAX\_EPOOL, 406  
        QF\_MAX\_TICK\_RATE, 407  
        QF\_MEM\_ISOLATE, 409  
        QF\_MPOOL\_CTR\_SIZE, 408  
        QF\_MPOOL\_SIZ\_SIZE, 408  
        QF\_ON\_CONTEXT\_SW, 409  
        QF\_TIMEEV\_CTR\_SIZE, 408  
        QK\_USE\_IRQ\_HANDLER, 410  
        QK\_USE\_IRQ\_NUM, 410  
        QP\_API\_VERSION, 405  
        QS\_CTR\_SIZE, 409  
        QS\_TIME\_SIZE, 409  
        QXK\_USE\_IRQ\_HANDLER, 410  
        QXK\_USE\_IRQ\_NUM, 410  
    qp\_pkg.dox, 411  
    qp\_pkg.h, 449  
        Q\_PTR2UINT\_CAST\_, 450  
        QACTIVE\_CAST\_, 450  
        QTE\_FLAG\_IS\_LINKED, 450  
        QTE\_FLAG\_WAS\_DISARMED, 450  
    qp\_port.h, 411  
        crit\_stat\_t, 415  
        critEntry, 415  
        critExit, 415  
        Q\_NORETURN, 412  
        QACTIVE\_EQEUE\_TYPE, 412  
        QACTIVE\_OS\_OBJ\_TYPE, 412  
        QACTIVE\_THREAD\_TYPE, 412  
        QF\_CRIT\_ENTRY, 413  
        QF\_CRIT\_EXIT, 413  
        QF\_CRIT\_STAT, 413  
        QF\_INT\_DISABLE, 412  
        QF\_INT\_ENABLE, 413  
        QK\_ISR\_CONTEXT\_, 413  
        QK\_ISR\_ENTRY, 414  
        QK\_ISR\_EXIT, 414  
        QV\_CPU\_SLEEP, 413  
        QXK\_CONTEXT\_SWITCH\_, 414  
        QXK\_ISR\_CONTEXT\_, 414  
        QXK\_ISR\_ENTRY, 414  
        QXK\_ISR\_EXIT, 414  
    QP\_RELEASE  
        qp.h, 426  
    QP\_VERSION  
        qp.h, 426

QP\_VERSION\_STR  
     qp.h, 426  
 QP\_versionStr  
     qf\_act.c, 484  
     qp.h, 449  
 qpc.h, 451  
     char\_t, 455  
     Q\_ALLEGGE, 453  
     Q\_ALLEGGE\_ID, 452  
     Q\_ASSERT\_COMPILE, 453  
     Q\_onAssert, 452  
     QACTIVE\_START, 452  
     QF\_getQueueMin, 454  
     QF\_PUBLISH, 454  
     QF\_TICK, 454  
     QF\_TICK\_X, 454  
     QHSM\_DISPATCH, 453  
     QHSM\_INIT, 453  
     QHsm\_isIn, 454  
     QM\_SUPER\_SUB, 451  
     QM\_TRAN\_EP, 451  
     QM\_TRAN\_XP, 452  
     QP\_API\_VERSION, 451  
     QXTHREAD\_START, 452  
 QPrioSpec  
     qp.h, 448  
 QPSet, 378  
     bits, 380  
     QPSet\_findMax, 379  
     QPSet\_hasElement, 379  
     QPSet\_insert, 379  
     QPSet\_isEmpty, 379  
     QPSet\_notEmpty, 379  
     QPSet\_remove, 379  
     QPSet\_setEmpty, 379  
 QPSet\_findMax  
     QPSet, 379  
 QPSet\_hasElement  
     QPSet, 379  
 QPSet\_insert  
     QPSet, 379  
 QPSet\_isEmpty  
     QPSet, 379  
 QPSet\_notEmpty  
     QPSet, 379  
 QPSet\_remove  
     QPSet, 379  
 QPSet\_setEmpty  
     QPSet, 379  
 QPSetBits  
     qp.h, 449  
 QS, 380  
 qs.dox, 415  
 QS\_2U8\_PRE

qs\_dummy.h, 462  
 QS\_ASSERTION  
     qs\_dummy.h, 461  
 QS\_BEGIN\_ID  
     qs\_dummy.h, 457  
 QS\_BEGIN\_INCRIT  
     qs\_dummy.h, 457  
 QS\_BEGIN\_PRE  
     qs\_dummy.h, 462  
 QS\_CRIT\_ENTRY  
     qs\_dummy.h, 464  
 QS\_CRIT\_EXIT  
     qs\_dummy.h, 464  
 QS\_CRIT\_STAT  
     qs\_dummy.h, 464  
 QS\_CTR\_SIZE  
     qp\_config.h, 409  
 QS\_doOutput  
     qs\_dummy.h, 466  
 qs\_dummy.h, 455  
     QS\_2U8\_PRE, 462  
     QS\_ASSERTION, 461  
     QS\_BEGIN\_ID, 457  
     QS\_BEGIN\_INCRIT, 457  
     QS\_BEGIN\_PRE, 462  
     QS\_CRIT\_ENTRY, 464  
     QS\_CRIT\_EXIT, 464  
     QS\_CRIT\_STAT, 464  
     QS\_doOutput, 466  
     QS\_DUMP, 457  
     QS\_END, 457  
     QS\_END\_INCRIT, 458  
     QS\_END\_PRE, 462  
     QS\_ENUM, 459  
     QS\_ENUM\_DICTIONARY, 460  
     QS\_EQC\_PRE, 463  
     QS\_EVS\_PRE, 463  
     QS\_EXIT, 457  
     QS\_F32, 459  
     QS\_F64, 459  
     QS\_FLUSH, 461  
     QS\_FUN, 460  
     QS\_FUN\_DICTIONARY, 460  
     QS\_FUN\_PRE, 463  
     QS\_getBlock, 466  
     QS\_getByte, 465  
     QS\_GLB\_FILTER, 457  
     QS\_I16, 458  
     QS\_I32, 458  
     QS\_I64, 459  
     QS\_I8, 458  
     QS\_INIT, 457  
     QS\_initBuf, 465  
     QS\_LOC\_FILTER, 457

QS\_MEM, 459  
QS\_MPC\_PRE, 463  
QS MPS\_PRE, 464  
QS\_OBJ, 460  
QS\_OBJ\_ARR\_DICTIONARY, 460  
QS\_OBJ\_DICTIONARY, 460  
QS\_OBJ\_PRE, 463  
QS\_onCleanup, 467  
QS\_onFlush, 467  
QS\_onGetTime, 467  
QS\_ONLY, 462  
QS\_onStartup, 467  
QS\_onTestEvt, 469  
QS\_onTestLoop, 469  
QS\_onTestPost, 469  
QS\_onTestSetup, 469  
QS\_onTestTeardown, 469  
QS\_OUTPUT, 461  
QS\_RX\_INPUT, 462  
QS\_RX\_PUT, 462  
QS\_rxInitBuf, 468  
QS\_rxParse, 468  
QS\_SIG, 459  
QS\_SIG\_DICTIONARY, 460  
QS\_SIG\_PRE, 463  
QS\_STR, 459  
QS\_TEC\_PRE, 464  
QS\_TEST\_PAUSE, 461  
QS\_TEST\_PROBE, 461  
QS\_TEST\_PROBE\_DEF, 461  
QS\_TEST\_PROBE\_ID, 461  
QS\_TIME\_PRE, 463  
QS\_TR\_CRIT\_ENTRY, 464  
QS\_TR\_CRIT\_EXIT, 464  
QS\_TR\_ISR\_ENTRY, 464  
QS\_TR\_ISR\_EXIT, 465  
QS\_U16, 458  
QS\_U16\_PRE, 462  
QS\_U32, 458  
QS\_U32\_PRE, 463  
QS\_U64, 459  
QS\_U8, 458  
QS\_U8\_PRE, 462  
QS\_USR\_DICTIONARY, 460  
QSTimeCtr, 465  
QS\_DUMP  
    qs\_dummy.h, 457  
QS\_END  
    qs\_dummy.h, 457  
QS\_END\_INCRIT  
    qs\_dummy.h, 458  
QS\_END\_PRE  
    qs\_dummy.h, 462  
QS\_ENUM  
    qs\_dummy.h, 459  
QS\_ENUM\_DICTIONARY  
    qs\_dummy.h, 460  
QS\_EQC\_PRE  
    qs\_dummy.h, 463  
QS\_EVS\_PRE  
    qs\_dummy.h, 463  
QS\_EXIT  
    qs\_dummy.h, 457  
QS\_F32  
    qs\_dummy.h, 459  
QS\_F64  
    qs\_dummy.h, 459  
QS\_Filter, 380  
QS\_FLUSH  
    qs\_dummy.h, 461  
QS\_FUN  
    qs\_dummy.h, 460  
QS\_FUN\_DICTIONARY  
    qs\_dummy.h, 460  
QS\_FUN\_PRE  
    qs\_dummy.h, 463  
QS\_FUN\_PTR\_SIZE  
    qs\_port.h, 416  
QS\_getBlock  
    qs\_dummy.h, 466  
QS\_getByte  
    qs\_dummy.h, 465  
QS\_GLB\_FILTER  
    qs\_dummy.h, 457  
QS\_I16  
    qs\_dummy.h, 458  
QS\_I32  
    qs\_dummy.h, 458  
QS\_I64  
    qs\_dummy.h, 459  
QS\_I8  
    qs\_dummy.h, 458  
QS\_INIT  
    qs\_dummy.h, 457  
QS\_initBuf  
    qs\_dummy.h, 465  
QS\_LOC\_FILTER  
    qs\_dummy.h, 457  
QS\_MEM  
    qs\_dummy.h, 459  
QS\_MPC\_PRE  
    qs\_dummy.h, 463  
QS MPS\_PRE  
    qs\_dummy.h, 464  
QS\_OBJ  
    qs\_dummy.h, 460  
QS\_OBJ\_ARR\_DICTIONARY  
    qs\_dummy.h, 460

QS\_OBJ\_DICTIONARY  
     qs\_dummy.h, 460  
 QS\_OBJ\_PRE  
     qs\_dummy.h, 463  
 QS\_OBJ\_PTR\_SIZE  
     qs\_port.h, 416  
 QS\_onCleanup  
     qs\_dummy.h, 467  
 QS\_onFlush  
     qs\_dummy.h, 467  
 QS\_onGetTime  
     qs\_dummy.h, 467  
 QS\_ONLY  
     qs\_dummy.h, 462  
 QS\_onStartup  
     qs\_dummy.h, 467  
 QS\_onTestEvt  
     qs\_dummy.h, 469  
 QS\_onTestLoop  
     qs\_dummy.h, 469  
 QS\_onTestPost  
     qs\_dummy.h, 469  
 QS\_onTestSetup  
     qs\_dummy.h, 469  
 QS\_onTestTeardown  
     qs\_dummy.h, 469  
 QS\_OUTPUT  
     qs\_dummy.h, 461  
 qs\_pkg.dox, 415  
 qs\_port.h, 415  
     QS\_FUN\_PTR\_SIZE, 416  
     QS\_OBJ\_PTR\_SIZE, 416  
 QS\_RX\_INPUT  
     qs\_dummy.h, 462  
 QS\_RX\_PUT  
     qs\_dummy.h, 462  
 QS\_rxInitBuf  
     qs\_dummy.h, 468  
 QS\_rxParse  
     qs\_dummy.h, 468  
 QS\_SIG  
     qs\_dummy.h, 459  
 QS\_SIG\_DICTIONARY  
     qs\_dummy.h, 460  
 QS\_SIG\_PRE  
     qs\_dummy.h, 463  
 QS\_STR  
     qs\_dummy.h, 459  
 QS\_TEC\_PRE  
     qs\_dummy.h, 464  
 QS\_TEST\_PAUSE  
     qs\_dummy.h, 461  
 QS\_TEST\_PROBE  
     qs\_dummy.h, 461  
 QS\_TEST\_PROBE\_DEF  
     qs\_dummy.h, 461  
 QS\_TEST\_PROBE\_ID  
     qs\_dummy.h, 461  
 QS\_TIME\_PRE  
     qs\_dummy.h, 463  
 QS\_TIME\_SIZE  
     qp\_config.h, 409  
 QS\_TProbe, 380  
     addr, 381  
     data, 381  
     idx, 381  
 QS\_TR\_CRIT\_ENTRY  
     qs\_dummy.h, 464  
 QS\_TR\_CRIT\_EXIT  
     qs\_dummy.h, 464  
 QS\_TR\_ISR\_ENTRY  
     qs\_dummy.h, 464  
 QS\_TR\_ISR\_EXIT  
     qs\_dummy.h, 465  
 QS\_U16  
     qs\_dummy.h, 458  
 QS\_U16\_PRE  
     qs\_dummy.h, 462  
 QS\_U32  
     qs\_dummy.h, 458  
 QS\_U32\_PRE  
     qs\_dummy.h, 463  
 QS\_U64  
     qs\_dummy.h, 459  
 QS\_U8  
     qs\_dummy.h, 458  
 QS\_U8\_PRE  
     qs\_dummy.h, 462  
 QS\_USR\_DICTIONARY  
     qs\_dummy.h, 460  
 qsafe.dox, 417  
 qsafe.h, 469  
     Q\_ASSERT, 473  
     Q\_ASSERT\_ID, 472  
     Q\_ASSERT\_INCRIT, 471  
     Q\_ASSERT\_STATIC, 478  
     Q\_DIM, 479  
     Q\_ENSURE, 476  
     Q\_ENSURE\_ID, 476  
     Q\_ENSURE\_INCRIT, 477  
     Q\_ERROR, 474  
     Q\_ERROR\_ID, 473  
     Q\_ERROR\_INCRIT, 472  
     Q\_INVARIANT, 478  
     Q\_INVARIANT\_ID, 477  
     Q\_INVARIANT\_INCRIT, 478  
     Q\_NORETURN, 479  
     Q\_onError, 479

Q\_REQUIRE, 475  
Q\_REQUIRE\_ID, 474  
Q\_REQUIRE\_INCRIT, 475  
QF\_CRIT\_ENTRY, 471  
QF\_CRIT\_EXIT, 471  
QF\_CRIT\_STAT, 471  
QSchedStatus  
  qk.h, 420  
QSignal  
  qp.h, 447  
QSpyId, 381  
qstamp.c, 487  
  Q\_BUILD\_DATE, 488  
  Q\_BUILD\_TIME, 488  
qstamp.h, 480  
  Q\_BUILD\_DATE, 480  
  Q\_BUILD\_TIME, 480  
QState  
  qp.h, 448  
QStateHandler  
  qp.h, 448  
QSTimeCtr  
  qs\_dummy.h, 465  
QSubscrList, 381  
  set, 382  
QTE\_FLAG\_IS\_LINKED  
  qp\_pkg.h, 450  
QTE\_FLAG\_WAS\_DISARMED  
  qp\_pkg.h, 450  
QTicker, 382  
  QTicker\_ctor, 385  
  QTicker\_dispatch\_, 385  
  QTicker\_init\_, 385  
  QTicker\_trig\_, 385  
  super, 385  
QTicker\_ctor  
  QTicker, 385  
QTicker\_dispatch\_  
  QTicker, 385  
QTicker\_init\_  
  QTicker, 385  
QTICKER\_TRIG  
  qp.h, 446  
QTicker\_trig\_  
  QTicker, 385  
QTimeEvt, 385  
  act, 393  
  ctr, 393  
  flags, 394  
  interval, 393  
  next, 393  
  QTimeEvt\_armX, 388  
  QTimeEvt\_ctorX, 388  
  QTimeEvt\_currCtr, 391  
  QTimeEvt\_disarm, 389  
  QTimeEvt\_expire\_, 392  
  QTimeEvt\_init, 391  
  QTimeEvt\_noActive, 392  
  QTimeEvt\_rearm, 390  
  QTimeEvt\_tick1\_, 392  
  QTimeEvt\_tick\_, 392  
  QTimeEvt\_timeEvtHead\_, 394  
  QTimeEvt\_wasDisarmed, 390  
  super, 393  
  tickRate, 394  
  QTimeEvt\_armX  
    QTimeEvt, 388  
  QTimeEvt\_ctorX  
    QTimeEvt, 388  
  QTimeEvt\_currCtr  
    QTimeEvt, 391  
  QTimeEvt\_disarm  
    QTimeEvt, 389  
  QTimeEvt\_expire\_  
    QTimeEvt, 392  
  QTimeEvt\_init  
    QTimeEvt, 391  
  QTimeEvt\_noActive  
    QTimeEvt, 392  
  QTimeEvt\_rearm  
    QTimeEvt, 390  
  QTIMEEVNT\_TICK  
    qp.h, 447  
  QTimeEvt\_tick1\_  
    QTimeEvt, 392  
  QTimeEvt\_tick\_  
    QTimeEvt, 392  
  QTIMEEVNT\_TICK\_X  
    qp.h, 445  
  QTimeEvt\_timeEvtHead\_  
    QTimeEvt, 394  
  QTimeEvt\_wasDisarmed  
    QTimeEvt, 390  
  QTimeEvtCtr  
    qp.h, 449  
  QV, 394  
    QV\_onIdle, 396  
    QV\_priv\_, 396  
    QV\_schedDisable, 395  
    QV\_schedEnable, 395  
  qv.c, 488  
  qv.dox, 417  
  qv.h, 480  
    QACTIVE\_EQUEUE\_SIGNAL\_, 481  
    QACTIVE\_EQUEUE\_WAIT\_, 481  
    QF\_EPOOL\_EVENT\_SIZE\_, 482  
    QF\_EPOOL\_GET\_, 482  
    QF\_EPOOL\_INIT\_, 481

QF\_EPOOL\_PUT\_, 482  
 QF\_EPOOL\_TYPE\_, 481  
 QF\_SCHED\_LOCK\_, 481  
 QF\_SCHED\_STAT\_, 481  
 QF\_SCHED\_UNLOCK\_, 481  
**QV\_Attr**, 396  
 readySet, 396  
 schedCeil, 396  
**QV\_CPU\_SLEEP**  
 qp\_port.h, 413  
**QV\_onIdle**  
 QV, 396  
**QV\_priv\_**  
 QV, 396  
**QV\_schedDisable**  
 QV, 395  
**QV\_schedEnable**  
 QV, 395  
**QXK**, 397  
 qxk.dox, 417  
**QXK\_Attr**, 397  
**QXK\_CONTEXT\_SWITCH\_**  
 qp\_port.h, 414  
**QXK\_ISR\_CONTEXT\_**  
 qp\_port.h, 414  
**QXK\_ISR\_ENTRY**  
 qp\_port.h, 414  
**QXK\_ISR\_EXIT**  
 qp\_port.h, 414  
**QXK\_USE\_IRQ\_HANDLER**  
 qp\_config.h, 410  
**QXK\_USE\_IRQ\_NUM**  
 qp\_config.h, 410  
**QXMutex**, 397  
**QXSemaphore**, 398  
**QXThread**, 399  
**QXTHREAD\_START**  
 qpc.h, 452  
**QXThreadHandler**  
 qp.h, 448  
  
**readySet**  
 QK\_Attr, 360  
 QV\_Attr, 396  
**refCtr\_**  
 QEvt, 340  
**Resource Viewpoint**, 175  
**Revision History**, 239  
**ring**  
 QEQueue, 336  
  
 sas-qp.dox, 417  
**SAS\_OS\_API**, 175  
**SAS\_OSAL\_API**, 174  
**SAS\_QP\_AF**, 169  
  
 SAS\_QP\_API, 174  
**SAS\_QP\_APP**, 171  
**SAS\_QP\_CLS**, 173  
**SAS\_QP\_EDA**, 168  
**SAS\_QP\_EMM**, 177  
**SAS\_QP\_FRM**, 171  
**SAS\_QP\_MEM**, 175  
**SAS\_QP\_OO**, 168  
**SAS\_QP\_OS**, 172  
**SAS\_QP\_OSAL**, 172  
 schedCeil  
 QV\_Attr, 396  
 sds-qp.dox, 417  
**SDS\_QA\_QHsm\_choice**, 210  
**SDS\_QA\_QHsm\_decl**, 203  
**SDS\_QA\_QHsm\_entry**, 206  
**SDS\_QA\_QHsm\_exit**, 207  
**SDS\_QA\_QHsm\_hist**, 211  
**SDS\_QA\_QHsm\_hist\_tran**, 211  
**SDS\_QA\_QHsm\_intern**, 209  
**SDS\_QA\_QHsm\_nest\_init**, 208  
**SDS\_QA\_QHsm\_state**, 205  
**SDS\_QA\_QHsm\_top\_init**, 204  
**SDS\_QA\_QHsm\_tran**, 208  
**SDS\_QA\_QMsm\_choice**, 220  
**SDS\_QA\_QMsm\_decl**, 212  
**SDS\_QA\_QMsm\_entry**, 217  
**SDS\_QA\_QMsm\_exit**, 217  
**SDS\_QA\_QMsm\_hist**, 221  
**SDS\_QA\_QMsm\_hist\_tran**, 222  
**SDS\_QA\_QMsm\_intern**, 220  
**SDS\_QA\_QMsm\_nest\_init**, 218  
**SDS\_QA\_QMsm\_state**, 215  
**SDS\_QA\_QMsm\_top\_init**, 214  
**SDS\_QA\_QMsm\_tran**, 219  
**SDS\_QA\_START**, 193  
**SDS\_QP\_CRIT**, 229  
**SDS\_QP\_MELOC**, 199  
**SDS\_QP\_POST**, 195  
**SDS\_QP\_PUB**, 196  
**SDS\_QP\_QActive**, 189  
**SDS\_QP\_QAsm**, 185  
**SDS\_QP\_QEP**, 183  
**SDS\_QP\_QEvt**, 184  
**SDS\_QP\_QF**, 183  
**SDS\_QP\_QHsm**, 186  
**SDS\_QP\_QHsm\_ctor**, 226  
**SDS\_QP\_QHsm\_dispatch**, 227  
**SDS\_QP\_QHsm\_init**, 226  
**SDS\_QP\_QHsm\_tran-complex**, 227  
**SDS\_QP\_QHsm\_tran-simple**, 227  
**SDS\_QP\_QMctive**, 191  
**SDS\_QP\_QMsm**, 188  
**SDS\_QP\_QMsm\_ctor**, 227

- SDS\_QP\_QMsm\_disp, 228  
SDS\_QP\_QMsm\_init, 227  
SDS\_QP\_QMsm\_tat, 228  
SDS\_QP\_QMsm\_tran, 228  
SDS\_QP\_QTimeEvt, 191  
SDS\_QP\_TELC, 224  
set  
    QSubscrList, 382  
sig  
    QEvt, 340  
Simple Blinky Application, 13  
Software Architecture Specification, 165  
Software Design Specification, 179  
Software Requirements Specification, 79  
Software Tracing, 136  
srs-qp.dox, 417  
SRS\_QA\_EDM\_60, 122  
SRS\_QP\_AO\_00, 91  
SRS\_QP\_AO\_01, 91  
SRS\_QP\_AO\_10, 92  
SRS\_QP\_AO\_11, 92  
SRS\_QP\_AO\_20, 92  
SRS\_QP\_AO\_21, 93  
SRS\_QP\_AO\_22, 93  
SRS\_QP\_AO\_23, 94  
SRS\_QP\_AO\_30, 94  
SRS\_QP\_AO\_31, 94  
SRS\_QP\_AO\_32, 95  
SRS\_QP\_AO\_40, 95  
SRS\_QP\_AO\_50, 96  
SRS\_QP\_AO\_51, 96  
SRS\_QP\_AO\_60, 96  
SRS\_QP\_AO\_70, 97  
SRS\_QP\_EDG\_10, 119  
SRS\_QP\_EDM\_00, 119  
SRS\_QP\_EDM\_01, 119  
SRS\_QP\_EDM\_50, 120  
SRS\_QP\_EDM\_51, 120  
SRS\_QP\_EDM\_52, 121  
SRS\_QP\_EDM\_53, 121  
SRS\_QP\_EDM\_54, 121  
SRS\_QP\_EDM\_55, 122  
SRS\_QP\_EDM\_61, 123  
SRS\_QP\_EDM\_62, 123  
SRS\_QP\_EDM\_64, 123  
SRS\_QP\_EDM\_65, 124  
SRS\_QP\_EDM\_66, 124  
SRS\_QP\_EDM\_80, 124  
SRS\_QP\_EDM\_81, 125  
SRS\_QP\_EMM\_00, 127  
SRS\_QP\_EMM\_10, 127  
SRS\_QP\_EMM\_11, 128  
SRS\_QP\_EMM\_20, 128  
SRS\_QP\_EMM\_30, 128  
SRS\_QP\_EMM\_40, 129  
SRS\_QP\_EVT\_00, 98  
SRS\_QP\_EVT\_20, 99  
SRS\_QP\_EVT\_21, 99  
SRS\_QP\_EVT\_22, 99  
SRS\_QP\_EVT\_23, 99  
SRS\_QP\_EVT\_30, 100  
SRS\_QP\_EVT\_31, 100  
SRS\_QP\_EVT\_40, 100  
SRS\_QP\_EVT\_41, 101  
SRS\_QP\_NF\_01, 160  
SRS\_QP\_NF\_02, 160  
SRS\_QP\_NF\_03, 160  
SRS\_QP\_NF\_04, 161  
SRS\_QP\_NF\_10, 163  
SRS\_QP\_NF\_11, 163  
SRS\_QP\_NF\_12, 163  
SRS\_QP\_NF\_13, 164  
SRS\_QP\_NF\_14, 164  
SRS\_QP\_NF\_20, 161  
SRS\_QP\_NF\_21, 161  
SRS\_QP\_NF\_40, 162  
SRS\_QP\_NF\_41, 162  
SRS\_QP\_NF\_50, 163  
SRS\_QP\_QK\_00, 152  
SRS\_QP\_QK\_10, 153  
SRS\_QP\_QK\_20, 153  
SRS\_QP\_QK\_21, 153  
SRS\_QP\_QK\_30, 154  
SRS\_QP\_QK\_31, 154  
SRS\_QP\_QS\_00, 139  
SRS\_QP\_QS\_01, 139  
SRS\_QP\_QS\_10, 140  
SRS\_QP\_QS\_11, 140  
SRS\_QP\_QS\_20, 141  
SRS\_QP\_QS\_21, 141  
SRS\_QP\_QS\_30, 141  
SRS\_QP\_QS\_31, 142  
SRS\_QP\_QS\_40, 142  
SRS\_QP\_QS\_50, 143  
SRS\_QP\_QV\_00, 146  
SRS\_QP\_QV\_10, 146  
SRS\_QP\_QV\_11, 146  
SRS\_QP\_QV\_12, 147  
SRS\_QP\_QV\_20, 147  
SRS\_QP\_QV\_21, 147  
SRS\_QP\_QXK\_00, 156  
SRS\_QP\_QXK\_10, 157  
SRS\_QP\_QXK\_11, 157  
SRS\_QP\_QXK\_12, 157  
SRS\_QP\_QXK\_13, 157  
SRS\_QP\_QXK\_20, 158  
SRS\_QP\_QXK\_21, 158  
SRS\_QP\_QXK\_22, 158

SRS\_QP\_SM\_00, 104  
 SRS\_QP\_SM\_01, 105  
 SRS\_QP\_SM\_10, 105  
 SRS\_QP\_SM\_20, 105  
 SRS\_QP\_SM\_21, 106  
 SRS\_QP\_SM\_22, 107  
 SRS\_QP\_SM\_23, 107  
 SRS\_QP\_SM\_24, 108  
 SRS\_QP\_SM\_25, 108  
 SRS\_QP\_SM\_30, 109  
 SRS\_QP\_SM\_31, 110  
 SRS\_QP\_SM\_32, 110  
 SRS\_QP\_SM\_33, 110  
 SRS\_QP\_SM\_34, 111  
 SRS\_QP\_SM\_35, 112  
 SRS\_QP\_SM\_36, 113  
 SRS\_QP\_SM\_37, 114  
 SRS\_QP\_SM\_38, 115  
 SRS\_QP\_SM\_39, 115  
 SRS\_QP\_SM\_40, 116  
 SRS\_QP\_TM\_00, 132  
 SRS\_QP\_TM\_10, 132  
 SRS\_QP\_TM\_11, 132  
 SRS\_QP\_TM\_20, 133  
 SRS\_QP\_TM\_21, 133  
 SRS\_QP\_TM\_22, 133  
 SRS\_QP\_TM\_23, 134  
 SRS\_QP\_TM\_30, 134  
 SRS\_QP\_TM\_40, 135  
 start  
     QMPool, 368  
 state  
     QAsm, 328  
 State Dynamics Viewpoint, 202  
 State Machines, 102  
 stateHandler  
     QMState, 377  
 Structure Viewpoint, 182  
 super  
     QActive, 325  
     QHsm, 356  
     QMActive, 363  
     QMsm, 376  
     QTicker, 385  
     QTimeEvt, 393  
 superstate  
     QMState, 377  
 tail  
     QEQueue, 336  
 target  
     QMTranActTable, 378  
 tatbl  
     QAsmAttr, 329