QPC

1. 模块

QEP 分层事件处理器

QP/C 中，对象行为由**层次化状态机 (UML 状态图)** 定义，其关键在于**状态嵌套**。这种机制的价值在于，它通过允许**子状态仅特化（定义差异）于其超状态**的行为，有效避免了传统 FSM 中因重复定义共性行为而导致的**状态-转换爆炸**，从而实现了行为的**高度共享与复用**。

- QHsm class
- QHsm_ctor()
- QHSM_INIT()
- QHSM_DISPATCH()
- QHsm_isIn()
- QHsm_state()
- QHsm_top()
- QMsm class
- QMsm_ctor()
- QMsm_isInState()
- QMsm_stateObj()

QF 活动对象框架

QF 是一个可移植的、事件驱动的、实时框架，用于执行活动对象（并发状态机），专为实时嵌入式 (RTE) 系统而设计。

Active Objects

- QActive class
- QActive_ctor()
- QACTIVE_START()
- QACTIVE_POST()
- QACTIVE_POST_X()
- QACTIVE_POST_LIFO()
- QActive_defer()
- QActive_recall()
- QActive_flushDeferred()
- QActive_stop()
- QMActive class
- QMActive_ctor()

Publish-Subscribe

- QSubscrList (Subscriber List struct)
- QF_psInit()
- QF_PUBLISH()
- QActive_subscribe()
- QActive_unsubscribe()

- QActive_unsubscribeAll()

Dynamic Events

- QEvt class
- QF_poolInit()
- Q_NEW()
- Q_NEW_X()
- Q_NEW_REF()
- Q_DELETE_REF()
- QF_gc()

Time Events

- QTimeEvt class
- QF_TICK_X()
- QTimeEvt_ctorX()
- QTimeEvt_armX()
- QTimeEvt_disarm()
- QTimeEvt_rearm()
- QTimeEvt_ctr()
- QTicker active object

Event Queues (raw thread-safe)

- QEQueue class
- QEQueue_init()
- QEQueue_post()
- QEQueue_postLIFO()
- QEQueue_get()
- QEQueue_getNFree()
- QEQueue_getNMin()
- QEQueue_isEmpty()
- QEQueueCtr()

Memory Pools

- QMPool class
- QMPool_init()
- QMPool_get()
- QMPool_put()

QV 协作内核

QV 是一个简单的协作内核（以前称为“Vanilla”内核）。该内核一次执行一个活动对象，并在处理每个事件之前执行基于优先级的调度。

Arm Cortex M

QV 内核是一种协作式（Cooperative）内核，其工作原理本质上类似于传统的前台-后台系统（即"超级循环"）：

1. 执行模式与上下文

- **活跃对象执行：** 所有的**活跃对象 (Active Objects)** 都在主循环（背景）中执行。
- **中断返回：** 中断（前台）处理完成后，总是**返回到被抢占的地方**继续执行主循环。
- **处理器模式：**
  - **主循环 (Main Loop) / 应用程序代码** 在**特权线程模式 (Privileged Thread mode)** 下执行。
  - **异常（包括所有中断）** 总是由 **特权处理模式 (Privileged Handler mode)** 处理。
- **堆栈使用：** QV 内核**只使用主堆栈指针 (Main Stack Pointer)**，不使用也不初始化进程堆栈指针 (Process Stack Pointer)。

2. 中断管理与临界区

- **避免竞态条件：** 为了避免主循环和中断之间发生**竞态条件 (race conditions)**，QV 内核会**短暂地禁用中断。**
- **进入中断：** ARM Cortex-M 在进入中断上下文时，**不会自动禁用中断**（即不会设置 `PRIMASK` 或 `BASEPRI`）。
- **ISR 建议：**
  - 一般情况下，**不应该在中断服务程序 (ISR) 内部禁用中断。**
  - 特别是，调用 **QP 服务**（如 `QF_PUBLISH()`、`QF_TICK_X()`、`QACTIVE_POST()` 等）时，**必须保持中断开启状态**，以避免临界区嵌套问题。
- **中断优先级：** 如果不希望某个中断被其他中断抢占，可以通过配置 **NVIC**，为其设置一个**更高的优先级**（即**更小的数值**）。

3. 初始化

- **初始化功能：** `QF_init()` 函数会调用 `QV_init()`，将 MCU 中所有可用的 IRQ 的中断优先级设置为一个安全值 `QF_BASEPRI`（主要针对 ARMv7 架构）。

qep_port.h

包含了 stdint.h、stdbool.h

qf_port.h

该文件指定了中断禁用策略（QF 临界区）以及 QF 的配置常量

qv_port.h

- 该文件提供了宏 QV_CPU_SLEEP()，该宏指定如何在协作式 QV 内核中安全地进入 CPU 睡眠模式
- 为了避免中断唤醒活动对象和进入睡眠状态之间出现竞争条件，协作式 QV 内核在禁用中断的情况下调用 QV_CPU_SLEEP() 回调。

qv_port.c

该文件定义了函数 QV_init()，该函数对于 ARMv7-M 架构，将所有 IRQ 的中断优先级设置为安全值 QF_BASEPRI。

ISR

为活跃对象生成事件（即调用 `QACTIVE_POST()` 或 `QF_PUBLISH()` 等 QP 服务）

ARM EABI（嵌入式应用程序二进制接口）要求堆栈 8 字节对齐，而某些编译器仅保证 4 字节对齐。因此，一些编译器（例如 GNU-ARM）提供了一种将 ISR 函数指定为中断的方法。例如，GNU-ARM 编译器提供了 attribute((interrupt)) 指定，可以保证 8 字节堆栈对齐。

```
// QF 的时间事件管理
void SysTick_Handler(void) __attribute__((__interrupt__));
void SysTick_Handler(void) {
     // ~ ~ ~
     QF_TICK_X(0U, &l_SysTick_Handler); /* process all armed time events */
}
```

FPU

QV idle

当没有事件可用时，非抢占式 QV 内核会调用平台特定的回调函数 QV_onIdle()，您可以使用该函数来节省 CPU 资源，或执行任何其他“空闲”处理（例如 Quantum Spy 软件跟踪输出）。

必须在中断被禁用时被调用(避免与可能投递事件的中断发生**竞态条件**)

必须在内部重新启用中断(CPU 进入低功耗模式后，需要中断机制来唤醒)

```
void QV_onIdle(void)
{
#if defined NDEBUG
    /* Put the CPU and peripherals to the low-power mode */
    QV_CPU_SLEEP(); /* atomically go to sleep and enable interrupts */
#else
    QF_INT_ENABLE(); /* just enable interrupts */
#endif
}
```

API

- QV_INIT()
- QF_run()
- QV_onIdle()
- QV_CPU_SLEEP()

QK 抢占式非阻塞内核

QK 是一个小型抢占式、基于优先级、非阻塞内核，专为执行活动对象而设计。QK 运行活动对象的方式与优先级中断控制器（例如 ARM Cortex-M 中的 NVIC）使用单个堆栈运行中断的方式相同。活动对象以运行至完成 (RTC) 的方式处理其事件，并从调用堆栈中移除自身，这与嵌套中断在完成后从堆栈中移除自身的方式相同。同时，高优先级活动对象可以抢占低优先级活动对象，就像优先级中断控制器下中断可以相互抢占一样。QK 满足速率单调调度（也称为速率单调分析 RMA）的所有要求，可用于硬实时系统。

QXK 抢占式阻塞内核

QS 软件追踪组件

2. 移植

QP/C 发行版包含许多 QP/C 移植版本，这些版本分为三类：

1. 原生移植版本：使 QP/C 能够“原生”运行在裸机处理器上，使用内置内核（QV、QK 或 QXK）。
2. 第三方 RTOS 移植版本：使 QP/C 能够在第三方实时操作系统 (RTOS) 上运行。
3. 第三方操作系统移植版本：使 QP/C 能够在第三方操作系统 (OS)（例如 Windows 或 Linux）上运行。

Arm Cortex-M Port

与任何实时内核一样，QP 实时框架需要禁用中断才能访问代码的关键部分，并在访问完成后重新启用中断。

中断

QP 框架在 ARM Cortex-M 处理器上采用了**选择性禁用中断**的策略，将中断分为两大类：

- **“内核感知”中断 (Kernel-Aware)： 被允许调用 QP 服务**（例如，发布或投递事件）。
- **“内核无感知”中断 (Kernel-Unaware)： 不被允许调用任何 QP 服务。**它们只能通过**触发一个“内核感知”中断**来进行间接通信（由该“内核感知”中断来投递或发布事件）。

1. 针对 Cortex-M3/M4/M7 (ARMv7-M) 架构
    - QP 不会完全禁用所有中断，即使在临界区 (Critical Sections) 内。它使用 `BASEPRI` 寄存器来选择性地禁用中断。
    - “内核无感知”中断 (Kernel-Unaware)： 永不被禁用
    - “内核感知”中断 (Kernel-Aware)： 在 QP 临界区内会被禁用
2. 针对 Cortex-M0/M0+ (ARMv6-M) 架构
    - 实现方式： 由于这些架构没有实现 `BASEPRI` 寄存器，QP 必须使用 `PRIMASK` 寄存器来全局禁用中断
    - 结果： 在此架构下，所有中断都是“内核感知”的

注意事项和建议

- **QP 5.9.x 及以上：** `QF_init()` 会将所有 IRQ 的优先级设置为“内核感知”值 `QF_BASEPRI`。
- **最佳实践： 强烈建议**应用程序在 `QF_onStartup()` 中**显式设置**所有使用中断的优先级。
- **第三方库风险：** 需警惕 STM32Cube 等第三方库可能会**意外更改**中断优先级和分组，建议在运行 QP 应用前将优先级改回适当的值。
- **设置函数：** 应使用 CMSIS 提供的 `NVIC_SetPriority()` 函数来设置每个中断的优先级。请注意，`NVIC_SetPriority()` **传入的值**与**最终存储在 NVIC 寄存器中的值（CMSIS priorities vs.**

NVIC values）是不同的。

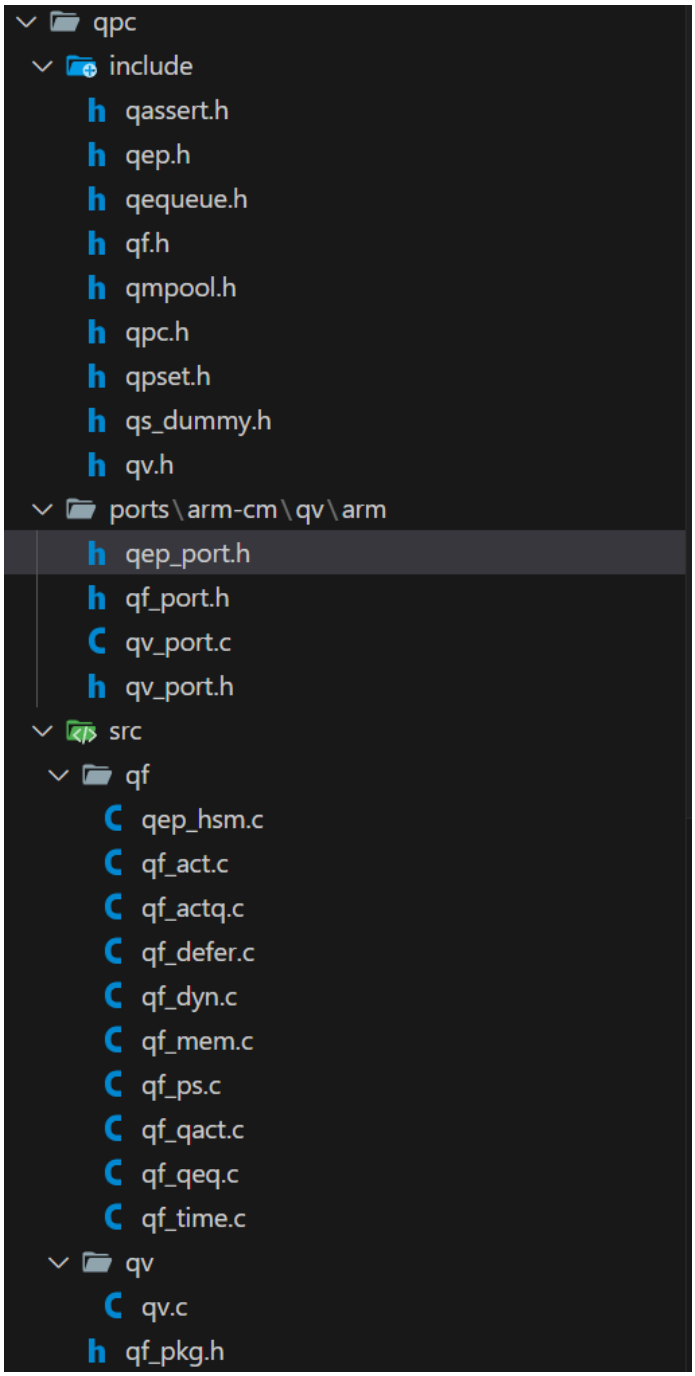
3. 集成

Arm Cortex M 裸机集成qpc（qv）所需文件(无qs软件跟踪)

1. 只包含`qpc.h`头文件即可
2. **不要修改qpc源代码文件**
3. 使用QPC提供的宏来操作活动对象，如QACTIVE_START、QACTIVE_POST、QF_TICK_X

`ports/arm-cm/qv/`：移植

- `arm`：armcc编译器
- `armclang`：armclang编译器
- `gnu`：gcc_arm编译器



4.状态转换

状态配置稳定：me->temp.fun == me->state.fun

转换规则

LCA(最近公共祖先)不执行 ENTRY 和 EXIT 事件

自转换：当状态机执行转换（Q_TRAN）到当前状态自身时，会发生以下顺序的动作：

1. 执行当前状态的 Q_EXIT_SIG。
2. 执行目标状态（即自身）的 Q_ENTRY_SIG。
3. 执行目标状态的 Q_INIT_SIG（如果

Example:

回调

```
void Q_onAssert(char const *const module, int loc)
{
    for (;;) {
    }
}

void QF_onCleanup(void)
{
}

void QF_onStartup(void)
{
    // 为所有ISR设置优先级
    NVIC_SetPriority(SysTick_IRQn, QF_AWARE_ISR_CMSIS_PRI);

    // enable IRQs
}

void QV_onIdle(void)
{
#if defined NDEBUG
    /* Put the CPU and peripherals to the low-power mode */
    QV_CPU_SLEEP(); /* atomically go to sleep and enable interrupts */
#else
    QF_INT_ENABLE(); /* just enable interrupts */
#endif
}
```

QActive

`QACTIVE_START` 内部使用了 `Q_ASSERT`，所以必须调用 `Q_DEFINE_THIS_FILE` 或 `Q_DEFINE_THIS_MODULE`

QTimeEvt

QTicker

```c
// 设置最大Tick Rate 'QPC\ports\arm-cm\qv\arm\qf_port.h'
#define QF_MAX_TICK_RATE        4U
// QScreen.c
void QScreen_Ctor(void)
{
    QScreen *const me = &l_screen;
    QActive_ctor(&me->super, Q_STATE_CAST(&QScreen_initial));
    QTimeEvt_ctorX(&me->timer, &me->super, SIG_SCREEN_TIMEOUT, 0);
}
// QMain.c
static QTicker s_ticker0;
QActive *AO_Ticker0 = &s_ticker0.super;
void SysTick_Handler(void) {
    // post a don't-care event to Ticker0
    QACTIVE_POST(AO_Ticker0, 0, 0); // 等价于 QF_TICK_X(0U, 0);
}
void StartActiveObjects(void)
{
    uint8_t priority = 1;
    QTicker_ctor(&s_ticker0, 3); // ticker AO for tick rate 0
    QACTIVE_START(AO_Ticker0, priority++, 0, 0, 0, 0, 0);
}
```

NOTE

1. 状态处理函数中初始化 `QState status = Q_HANDLED();`，以防信号处理后未初始化status
2. 订阅（`QActive_subscribe()`）的信号 **只能通过** `QF_PUBLISH()` **投递（发布）。**
   而 `QACTIVE_POST()` **是直接发送给某一个活动对象的，不走订阅系统。**
3. 静态事件和动态事件(事件池)
   - 静态事件

```c
static QEvt evt = {SIG_LED_TOGGLE, 0, 0};
QACTIVE_POST(AO_LED, &evt, 0);
```

   - 动态事件

```c
QF_PoolInit(poolSto, poolSize, sizeof(MyEvt));
MyEvt *pe = QF_NEW(MyEvt, SIG_MY_EVENT);
QACTIVE_POST(AO, &pe->super, 0);// 或 QF_PUBLISH
```

     1. QTimeEvt_armX 400错误：定时器**重复启动**导致断言失败 `Q_REQUIRE_ID(400, t->ctr == 0U);`
     2. `QEvt::poolId_` 为0表示静态事件，此时`QEvt::refCtr_`不用于引用计数
4. 使用QF_NO_MARGIN的函数会有断言失败机制
5. 使用 QACTIVE_START 同文件必须定义 Q_DEFINE_THIS_FILE
6. `QACTIVE_POST_X`会断言失败, `QACTIVE_POST`不会断言失败
7. 宏 **QF_AWARE_ISR_CMSIS_PRI** 在应用程序中用于作为枚举"QF-aware"中断优先级的偏移量.
   - "QF aware"的中断:
     - 优先级大于等于 **QF_AWARE_ISR_CMSIS_PRI**

        - 可以调用 QF 服务
    - "QF unaware"的中断：
        - 优先级小于 **QF_AWARE_ISR_CMSIS_PRI**
        - 不能调用任何 QF 服务