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

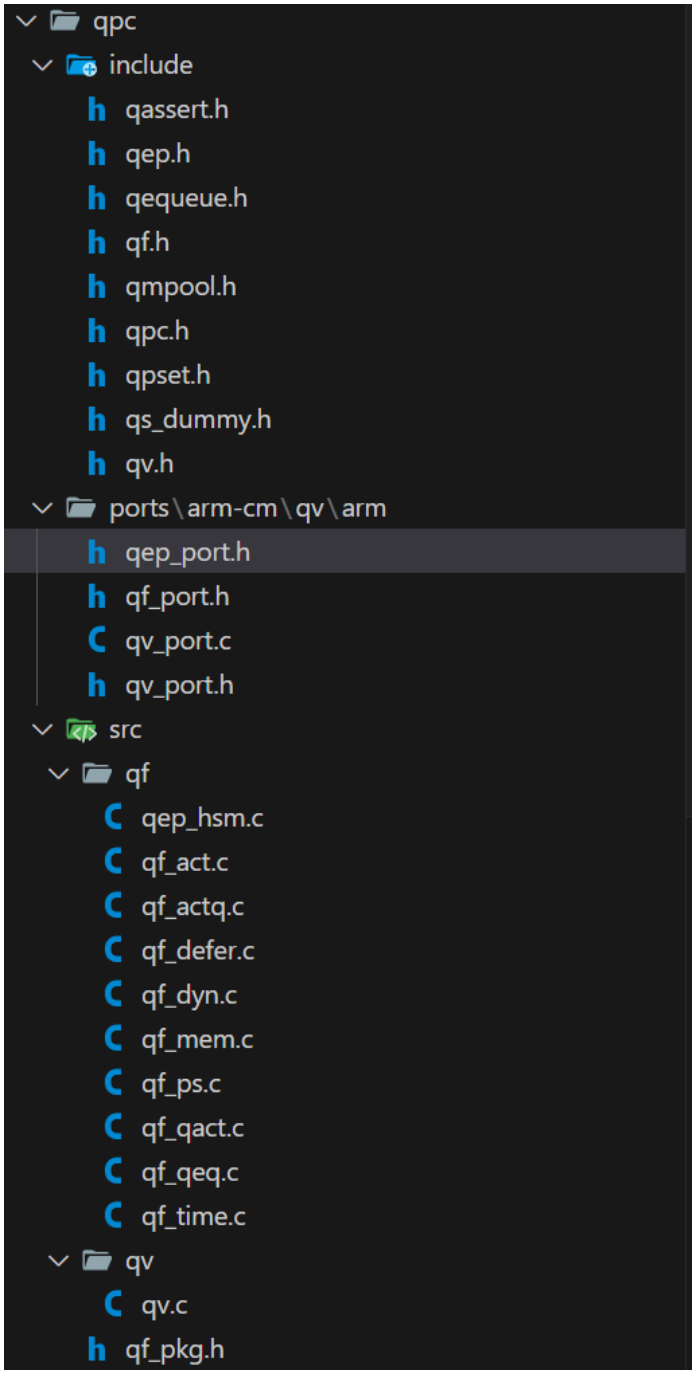
注意事项和建议

- **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



注意

1. 任何地方用到QP的仅需包含qpc.h
2. **不需要修改qpc源代码文件**

include

qassert.h

- 断言失败进入 Q_onAssert
- `QACTIVE_START` 内部使用了 `Q_ASSERT`，所以必须调用 `Q_DEFINE_THIS_FILE` 或 `Q_DEFINE_THIS_MODULE`

```
#define Q_ASSERT(test_) ((test_) ? (void)0 : Q_onAssert(&Q_this_module_[0], __LINE__))
```

qep.h

```
typedef struct {
    QSignal sig;                /*!< 事件实例的信号 */
    uint8_t poolId_;            /*!< 所属事件池 ID（静态事件为 0）*/
    uint8_t volatile refCtr_; /*!< 引用计数器 */
} QEvt;

// 修改目标状态为target
#define Q_TRAN(target_)  \
    ((Q_HSM_UPCAST(me))->temp.fun = Q_STATE_CAST(target_), (QState)Q_RET_TRAN)

// 修改目标状态为super
#define Q_SUPER(super_)  \
    ((Q_HSM_UPCAST(me))->temp.fun = Q_STATE_CAST(super_), (QState)Q_RET_SUPER)
```

- `poolId_` 为0表示静态事件，此时 `refCtr_` 不用于引用计数

qequeue.h

```

```

qf.h

```
#include "qpset.h"
```

- QActive

```
// 活动对象基类 (基于 ::QHsm 实现)
struct QActive {
    QHsm super;
    QEQueue eQueue;
    uint8_t prio;
};

// QActive 类虚表
struct QActiveVtable {
    // [virtual] 启动活动对象
    void (*start)(QActive *const me, uint_fast8_t prio,
                  QEvt const **const qSto, uint_fast16_t const qLen,
```

```
                    void *const stkSto, uint_fast16_t const stkSize,
                    void const *const par);
    // [virtual] FIFO 异步发送事件给活动对象
    bool (*post)(QActive *const me, QEvt const *const e,
                 uint_fast16_t const margin);
    // [virtual] LIFO 异步发送事件给活动对象
    void (*postLIFO)(QActive *const me, QEvt const *const e);
};


// QM 建模工具使用
typedef struct {
    QActive super;
} QMActive;
typedef QActiveVtable QMActiveVtable;
void QMActive_ctor(QMActive *const me, QStateHandler initial);


// QTicker 是一个高效的活动对象, 专门用于以指定 tick 频率
typedef struct {
    QActive super; /*!< inherits ::QActive */
} QTicker;
void QTicker_ctor(QTicker *const me, uint_fast8_t tickRate);
```

public 函数

```
// QActive public 函数
#define QACTIVE_START(...)
#define QACTIVE_POST(...)       // 不会断言失败(FIFO)
#define QACTIVE_POST_X(...)     // 会断言失败(FIFO)
#define QACTIVE_POST_LIFO(...)
```

protected 函数

```
// QActive protected 函数
void QActive_ctor(QActive *const me, QStateHandler initial);
void QActive_stop(QActive *const me);

// 订阅信号 @p sig, 以便传递给活动对象 @p me
void QActive_subscribe(QActive const *const me, enum_t const sig);

// 取消订阅信号 @p sig, 使其不再传递给活动对象 @p me
void QActive_unsubscribe(QActive const *const me, enum_t const sig);

// 取消订阅所有信号, 使其不再传递给活动对象 @p me
void QActive_unsubscribeAll(QActive const *const me);

// 将事件 @p e 延迟存储到指定的事件队列 @p eq 中
bool QActive_defer(QActive const *const me, QEQueue *const eq, QEvt const
*const e);

// 从指定的事件队列 @p eq 中取回一个之前延迟的事件
bool QActive_recall(QActive *const me, QEQueue *const eq);

```

```
// 清空指定的延迟队列 @p eq
uint_fast16_t QActive_flushDeferred(QActive const *const me, QEQueue *const
eq);

// 通用的附加属性设置 (useful in QP ports)
void QActive_setAttr(QActive *const me, uint32_t attr1, void const *attr2);
```

- QTimeEvt 事件事件类

```
typedef struct QTimeEvt {
    QEvt super;                          // inherits ::QEvt
    struct QTimeEvt *volatile next;      // 指向链表中下一个时间事件
    void *volatile act;                  // 接收时间事件的活动对象
    QTimeEvtCtr volatile ctr;            // 计数器
    QTimeEvtCtr interval;                // 重载值
} QTimeEvt;
```

public 函数

```
// 构造函数, 初始化时间事件
void QTimeEvt_ctorX(QTimeEvt *const me, QActive *const act,
                    enum_t const sig, uint_fast8_t tickRate);

// 启动一个时间事件(单次或周期性),并直接投递事件
void QTimeEvt_armX(QTimeEvt *const me,
                   QTimeEvtCtr const nTicks, QTimeEvtCtr const interval);

// 重新启动一个时间事件
bool QTimeEvt_rearm(QTimeEvt *const me, QTimeEvtCtr const nTicks);

// 取消启动一个时间事件
bool QTimeEvt_disarm(QTimeEvt *const me);

// 检查时间事件是否"被取消"
bool QTimeEvt_wasDisarmed(QTimeEvt *const me);

// 获取时间事件当前的递减计数器值
QTimeEvtCtr QTimeEvt_currCtr(QTimeEvt const *const me);
```

- QF facilities

```
// 订阅列表
typedef QPSet QSubscrList;

/* public functions */
void QF_init(void);

/*! 发布-订阅机制初始化 */
void QF_psInit(QSubscrList *const subscrSto, enum_t const maxSignal);

/*! 事件池初始化, 用于事件的动态分配 */
void QF_poolInit(void *const poolSto, uint_fast32_t const poolSize,
uint_fast16_t const evtSize);

```

```
/*! 获取任意已注册事件池的块大小 */
uint_fast16_t QF_poolGetMaxBlockSize(void);

/*! 将控制权交给 QF 以运行应用程序 */
int_t QF_run(void);

/*! 应用层调用该函数以停止 QF 应用程序，并将控制权交还给 OS/内核 */
void QF_stop(void);


// QF 回调
void QF_onStartup(void);
void QF_onCleanup(void);


// 事件发布
void QF_publish_(QEvt const *const e);
#define QF_PUBLISH(e_, dummy_) (QF_publish_(e_))


// 在时钟节拍中处理事件事件
void QF_tickX_(uint_fast8_t const tickRate);
#define QF_TICK_X(tickRate_, dummy) (QF_tickX_(tickRate_))
// 如果在指定的时钟速率下没有已启动的时间事件，则返回 'true'
bool QF_noTimeEvtsActiveX(uint_fast8_t const tickRate);


/*! 注册一个活动对象，使其由框架管理 */
void QF_add_(QActive *const a);

/*! 将活动对象从框架中移除 */
void QF_remove_(QActive *const a);

/*! 获取指定事件池的最小剩余空闲条目数 */
uint_fast16_t QF_getPoolMin(uint_fast8_t const poolId);

/*! 获取指定事件队列的最小剩余空闲条目数 */
uint_fast16_t QF_getQueueMin(uint_fast8_t const prio);


/*! 内部 QF 实现：创建新的动态事件 */
QEvt *QF_newX_(uint_fast16_t const evtSize, uint_fast16_t const margin,
enum_t const sig);
// 分配一个动态事件 (断言版本)
#define Q_NEW(evtT_, sig_)
// 分配一个动态事件 (非断言版本)
#define Q_NEW_X(e_, evtT_, margin_, sig_)

/*! 内部 QF 实现：创建新的事件引用 */
QEvt const *QF_newRef_(QEvt const *const e, void const *const evtRef);
/*! 内部 QF 实现：删除事件引用 */
void QF_deleteRef_(void const *const evtRef);
// 创建当前事件 `e` 的新引用
#define Q_NEW_REF(evtRef_, evtT_)
// 删除事件引用
```

```
#define Q_DELETE_REF(evtRef_)
```

qmpool.h

```

```

qpc.h

```
#include "qf_port.h"      /* QF/C port from the port directory */
#include "qassert.h"      /* QP embedded systems-friendly assertions */
```

qpset.h

```

```

qv.h

协作式内核

```
#include "qequeue.h"  /* QV kernel uses the native QP event queue  */
#include "qmpool.h"   /* QV kernel uses the native QP memory pool  */
#include "qpset.h"    /* QV kernel uses the native QP priority set */

#define QF_EQUEUE_TYPE      QEQueue
// QF_run() 中调用
void QV_onIdle(void);

/* QV 内核特有的调度器加锁机制 (但在 QV 中不需要) */
#define QF_SCHED_STAT_
#define QF_SCHED_LOCK_(dummy)   ((void)0)
#define QF_SCHED_UNLOCK_(dummy) ((void)0)

/* QF 原生事件队列操作 */
#define QACTIVE_EQUEUE_WAIT_(me_) \
    Q_ASSERT_ID(0, (me_)->eQueue.frontEvt != (QEvt *)0)

#define QACTIVE_EQUEUE_SIGNAL_(me_) \
    QPSet_insert(&QV_readySet_, (uint_fast8_t)(me_)->prio)



/* QF 原生事件池操作 */
#define QF_EPOOL_TYPE_ QMPool

// QF_EPOOL_INIT_ 事件池初始化
#define QF_EPOOL_INIT_(p_, poolSto_, poolSize_, evtSize_) \
    (QMPool_init(&(p_), (poolSto_), (poolSize_), (evtSize_)))

// QF_EPOOL_EVENT_SIZE_ 事件池大小
#define QF_EPOOL_EVENT_SIZE_(p_) ((uint_fast16_t)(p_).blockSize)
```

```
32
33 // QF_EPOOL_GET_
34 #define QF_EPOOL_GET_(p_, e_, m_, qs_id_) \
35     ((e_) = (QEvt *)QMPool_get(&(p_), (m_), (qs_id_)))
36
37 // QF_EPOOL_PUT_
38 #define QF_EPOOL_PUT_(p_, e_, qs_id_) \
39     (QMPool_put(&(p_), (e_), (qs_id_)))
```

qpc.h

```
1 #include "qf_port.h"      /* QF/C port from the port directory */
2 #include "qassert.h"      /* QP embedded systems-friendly assertions */
```

ports

移植目录

qep_port.h

```
1 #include <stdint.h>  /* Exact-width types. WG14/N843 C99 Standard */
2 #include <stdbool.h> /* Boolean type.      WG14/N843 C99 Standard */
3 #include "qep.h"     /* QEP platform-independent public interface */
```

qf_port.h

```
1 #include "qep_port.h" /* QEP port */
2 #include "qv_port.h"  /* QV cooperative kernel port */
3 #include "qf.h"       /* QF platform-independent public interface */
```

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

qv_port.h

```
1 #include "qv.h" /* QV platform-independent public interface */
```

src

qep_hsm.c

```
// 保留事件
static QEvt const QEP_reservedEvt_[] = {
    { (QSignal)QEP_EMPTY_SIG_, 0U, 0U },
    { (QSignal)Q_ENTRY_SIG,    0U, 0U },
    { (QSignal)Q_EXIT_SIG,     0U, 0U },
    { (QSignal)Q_INIT_SIG,     0U, 0U }
};

/** 在一个状态转换函数中执行 保留事件动作
 *  QEP_TRIG_(me->temp.fun, QEP_EMPTY_SIG_)
 *  QEP_TRIG_(t, Q_INIT_SIG)
 */
#define QEP_TRIG_(state_, sig_) ((*(state_))(me, &QEP_reservedEvt_[(sig_)]))

// 状态处理函数执行退出动作Q_EXIT_SIG
#define QEP_EXIT_(state_, qs_id_)   QEP_TRIG_(state_, Q_EXIT_SIG)

// 状态处理函数执行进入动作Q_ENTRY_SIG
#define QEP_ENTER_(state_, qs_id_)  QEP_TRIG_(state_, Q_ENTRY_SIG)
```

```
void QHsm_ctor(QHsm * const me, QStateHandler initial) {
     /* QHsm virtual table */
    static struct QHsmVtable const vtable = {
        &QHsm_init_,
        &QHsm_dispatch_
    };
    me->vptr      = &vtable;
    me->state.fun = Q_STATE_CAST(&QHsm_top);
    me->temp.fun  = initial;
}

// 最顶层初始转换 QHsm_top → MyState_Initial → MyState_Active
void QHsm_init_(QHsm * const me, void const * const e)
{
    QStateHandler t = me->state.fun;
    QState r;

    /** @pre the virtual pointer must be initialized, the top-most initial
    * transition must be initialized, and the initial transition must not
    * be taken yet.
    */
    Q_REQUIRE_ID(200, (me->vptr != (struct QHsmVtable *)0)
                      && (me->temp.fun != Q_STATE_CAST(0))
                      && (t == Q_STATE_CAST(&QHsm_top)));

    /* execute the top-most initial tran. */
    r = (*me->temp.fun)(me, Q_EVT_CAST(QEvt));

    /* the top-most initial transition must be taken */
    Q_ASSERT_ID(210, r == (QState)Q_RET_TRAN);

    /* drill down into the state hierarchy with initial transitions... */
```

```
    do {
        QStateHandler path[QHSM_MAX_NEST_DEPTH_]; /* tran entry path array
*/
        int_fast8_t ip = 0; /* tran entry path index */

        path[0] = me->temp.fun;
        (void)QEP_TRIG_(me->temp.fun, QEP_EMPTY_SIG_);
        while (me->temp.fun != t) {
            ++ip;
            Q_ASSERT_ID(220, ip < (int_fast8_t)Q_DIM(path));
            path[ip] = me->temp.fun;
            (void)QEP_TRIG_(me->temp.fun, QEP_EMPTY_SIG_);
        }
        me->temp.fun = path[0];

        /* retrace the entry path in reverse (desired) order... */
        do {
            QEP_ENTER_(path[ip], qs_id); /* enter path[ip] */
            --ip;
        } while (ip >= 0);

        t = path[0]; /* current state becomes the new source */

        r = QEP_TRIG_(t, Q_INIT_SIG); /* execute initial transition */

    } while (r == (QState)Q_RET_TRAN);

    me->state.fun = t; /* change the current active state */
    me->temp.fun  = t; /* mark the configuration as stable */
}

QState QHsm_top(void const * const me, QEvt const * const e) {
    (void)me;
    (void)e;
    return (QState)Q_RET_IGNORED; /* the top state ignores all events */
}

void QHsm_dispatch_(QHsm * const me, QEvt const * const e) {
    QStateHandler t = me->state.fun;
    QStateHandler s;
    QState r;

    /** @pre the current state must be initialized and
    * the state configuration must be stable
    */
    Q_REQUIRE_ID(400, (t != Q_STATE_CAST(0))
                      && (t == me->temp.fun));

    /* process the event hierarchically... */
    do {
        s = me->temp.fun;
        r = (*s)(me, e); /* invoke state handler s */

        if (r == (QState)Q_RET_UNHANDLED) { /* unhandled due to a guard? */
            r = QEP_TRIG_(s, QEP_EMPTY_SIG_); /* find superstate of s */
```

```
87          }
88      } while (r == (QState)Q_RET_SUPER);
89
90      /* transition taken? */
91      if (r >= (QState)Q_RET_TRAN) {
92          QStateHandler path[QHSM_MAX_NEST_DEPTH_];
93          int_fast8_t ip;
94
95          path[0] = me->temp.fun; /* save the target of the transition */
96          path[1] = t;
97          path[2] = s;
98
99          /* exit current state to transition source s... */
100         for (; t != s; t = me->temp.fun) {
101             if (QEP_TRIG_(t, Q_EXIT_SIG) == (QState)Q_RET_HANDLED) {
102                 (void)QEP_TRIG_(t, QEP_EMPTY_SIG_); /* find superstate of t
    */
103             }
104         }
105
106         ip = QHsm_tran_(me, path);
107
108         /* retrace the entry path in reverse (desired) order... */
109         for (; ip >= 0; --ip) {
110             QEP_ENTER_(path[ip], qs_id);  /* enter path[ip] */
111         }
112
113         t = path[0];      /* stick the target into register */
114         me->temp.fun = t; /* update the next state */
115
116         /* drill into the target hierarchy... */
117         while (QEP_TRIG_(t, Q_INIT_SIG) == (QState)Q_RET_TRAN) {
118             ip = 0;
119             path[0] = me->temp.fun;
120
121             (void)QEP_TRIG_(me->temp.fun, QEP_EMPTY_SIG_);/*find superstate
    */
122
123             while (me->temp.fun != t) {
124                 ++ip;
125                 path[ip] = me->temp.fun;
126                 (void)QEP_TRIG_(me->temp.fun, QEP_EMPTY_SIG_);/* find super
    */
127             }
128             me->temp.fun = path[0];
129
130             /* entry path must not overflow */
131             Q_ASSERT_ID(410, ip < QHSM_MAX_NEST_DEPTH_);
132
133             /* retrace the entry path in reverse (correct) order... */
134             do {
135                 QEP_ENTER_(path[ip], qs_id); /* enter path[ip] */
136                 --ip;
137             } while (ip >= 0);
138
```

```
            t = path[0]; /* current state becomes the new source */
        }
    }

    me->state.fun = t; /* change the current active state */
    me->temp.fun  = t; /* mark the configuration as stable */
}

// LCA 最近公共祖先
static int_fast8_t QHsm_tran_(QHsm * const me, QStateHandler
path[QHSM_MAX_NEST_DEPTH_]) {
    int_fast8_t ip = -1; /* transition entry path index */
    int_fast8_t iq; /* helper transition entry path index */
    QStateHandler t = path[0];
    QStateHandler const s = path[2];
    QState r;

    /* (a) check source==target (transition to self)... */
    if (s == t) {
        QEP_EXIT_(s, qs_id); /* exit the source */
        ip = 0; /* enter the target */
    }
    else {
        (void)QEP_TRIG_(t, QEP_EMPTY_SIG_); /* find superstate of target */

        t = me->temp.fun;

        /* (b) check source==target->super... */
        if (s == t) {
            ip = 0; /* enter the target */
        }
        else {
            (void)QEP_TRIG_(s, QEP_EMPTY_SIG_); /* find superstate of src
*/

            /* (c) check source->super==target->super... */
            if (me->temp.fun == t) {
                QEP_EXIT_(s, qs_id); /* exit the source */
                ip = 0; /* enter the target */
            }
            else {
                /* (d) check source->super==target... */
                if (me->temp.fun == path[0]) {
                    QEP_EXIT_(s, qs_id); /* exit the source */
                }
                else {
                    /* (e) check rest of source==target->super->super..
                    * and store the entry path along the way
                    */
                    iq = 0; /* indicate that LCA not found */
                    ip = 1; /* enter target and its superstate */
                    path[1] = t;      /* save the superstate of target */
                    t = me->temp.fun; /* save source->super */

                    /* find target->super->super... */
```

```
192                     r = QEP_TRIG_(path[1], QEP_EMPTY_SIG_);
193                     while (r == (QState)Q_RET_SUPER) {
194                         ++ip;
195                         path[ip] = me->temp.fun; /* store the entry path */
196                         if (me->temp.fun == s) { /* is it the source? */
197                             iq = 1; /* indicate that LCA found */
198 
199                             /* entry path must not overflow */
200                             Q_ASSERT_ID(510,
201                                 ip < QHSM_MAX_NEST_DEPTH_);
202                             --ip; /* do not enter the source */
203                             r = (QState)Q_RET_HANDLED; /* terminate loop */
204                         }
205                          /* it is not the source, keep going up */
206                         else {
207                             r = QEP_TRIG_(me->temp.fun, QEP_EMPTY_SIG_);
208                         }
209                     }
210 
211                     /* the LCA not found yet? */
212                     if (iq == 0) {
213 
214                         /* entry path must not overflow */
215                         Q_ASSERT_ID(520, ip < QHSM_MAX_NEST_DEPTH_);
216 
217                         QEP_EXIT_(s, qs_id); /* exit the source */
218 
219                         /* (f) check the rest of source->super
220                         *                   == target->super->super...
221                         */
222                         iq = ip;
223                         r = (QState)Q_RET_IGNORED; /* LCA NOT found */
224                         do {
225                             if (t == path[iq]) { /* is this the LCA? */
226                                 r = (QState)Q_RET_HANDLED; /* LCA found */
227                                 ip = iq - 1; /* do not enter LCA */
228                                 iq = -1; /* cause termintion of the loop */
229                             }
230                             else {
231                                 --iq; /* try lower superstate of target */
232                             }
233                         } while (iq >= 0);
234 
235                         /* LCA not found? */
236                         if (r != (QState)Q_RET_HANDLED) {
237                             /* (g) check each source->super->...
238                             * for each target->super...
239                             */
240                             r = (QState)Q_RET_IGNORED; /* keep looping */
241                             do {
242                                 /* exit t unhandled? */
243                                 if (QEP_TRIG_(t, Q_EXIT_SIG)
244                                     == (QState)Q_RET_HANDLED)
245                                 {
246 
```

```
247                                     (void)QEP_TRIG_(t, QEP_EMPTY_SIG_);
248                                 }
249                                 t = me->temp.fun; /* set to super of t */
250                                 iq = ip;
251                                 do {
252                                     /* is this LCA? */
253                                     if (t == path[iq]) {
254                                         /* do not enter LCA */
255                                         ip = (int_fast8_t)(iq - 1);
256                                         iq = -1; /* break out of inner loop
    */
257                                         /* break out of outer loop */
258                                         r = (QState)Q_RET_HANDLED;
259                                     }
260                                     else {
261                                         --iq;
262                                     }
263                                 } while (iq >= 0);
264                             } while (r != (QState)Q_RET_HANDLED);
265                         }
266                     }
267                 }
268             }
269         }
270     }
271     return ip;
272 }
```

> 状态配置稳定：me->temp.fun == me->state.fun

转换规则

qv.c

QF_Init 内部调用了 QV_INIT

qf_actq.c

活动对象队列

qf_qact.c

活动对象

qf_qep.c

事件队列

qf_time.c

```
// 启动定时器
void QTimeEvt_armX(QTimeEvt * const me,
                   QTimeEvtCtr const nTicks, QTimeEvtCtr const interval);
```

时间事件QTimEvt 是静态事件

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

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

   - 动态事件

   ```
   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);`
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