

FrameWork多线程框架使用手册

- 第一章、 多线程框架介绍
- 第二章、 入门指南
- 第三章、 调度原理
- 第四章、 消息队列
- 第五章、 如何写一个线程
- 第六章、 延迟消息队列
- 第七章 、 框架下的编程规则

第一章：多线程框架介绍

FrameWork:

FrameWork 多线程框架是用于嵌入式应用的软件框架。它是一个轻量级 RTOS 的替代品，实现基于优先级的非抢占式调度，通过消息队列在线程间通信。可以适用不同的 MCU，代码 100% 是 ANSI 的 C 语言编写的。

本文档的目的:

本文档旨在说明在嵌入式应用中如何配置、使用 **FrameWork** 多线程框架。同时，也对框架的内部结构进行了说明。

1.1 需求:

需要有一个目标板运行本软件。

目标系统（硬件）:

你的目标系统必须有:

- 1、 一个 MCU
- 2、 最少的 ROM 和 RAM
- 3、 一个 UART（方便测试）

最小系统:

- 1、 ROM 1.2K BYTE
- 2、 RAM 150 BYTE

大的系统:

- 1、 ROM 2.5K BYTE
- 2、 RAM >150byte (取决于消息队列配置的大小)

编译系统:

和 ANSI C 兼容的 C 编译器。

1.2 FrameWork 多线程框架的特点:

- 1、 移植容易，只修改开关中断指令。
- 2、 支持多线程（目前是 8 个，可以很容易更改为 16、32 个或更多）切换。
- 3、 支持状态机。
- 4、 不用压栈，RAM 占用最少。
- 5、 线程间用消息队列通信，只有一种通信方式，简单易用。
- 6、 不改变原有编程方式。

1.3 数据类型 :

```
typedef unsigned char  BOOLEAN;           /* 布尔变量 */
typedef unsigned char  INT8U;             /* 无符号 8 位整型变量 */
typedef signed   char  INT8S;             /* 有符号 8 位整型变量 */
typedef unsigned short INT16U;           /* 无符号 16 位整型变量 */
typedef signed   short INT16S;           /* 有符号 16 位整型变量 */
typedef unsigned int   INT32U;           /* 无符号 32 位整型变量 */
typedef signed   int   INT32S;           /* 有符号 32 位整型变量 */
typedef float        FP32;               /* 单精度浮点数（32 位长度） */
typedef double       FP64;               /* 双精度浮点数（64 位长度） */
```

1.4 开关中断

在不同的 MCU 中要修改以下三条语句（下列是 ARM 下的代码）

```
__swi(0x00) void SwiHandle(int Handle);

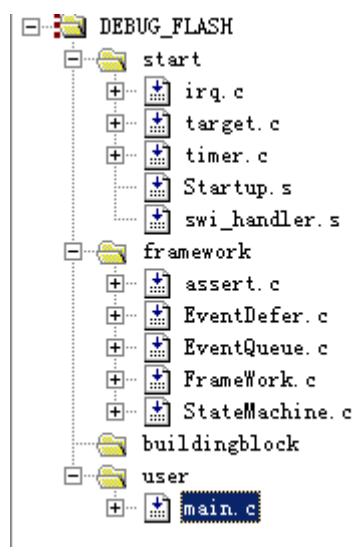
#define OS_ENTER_CRITICAL()  SwiHandle(0)

#define OS_EXIT_CRITICAL()   SwiHandle(1)
```

第二章 入门指南

2.1 把源文件加到工程中

把 FrameWork 和应用分离，这是我们推荐的，下图中 framework 中的文件和目录中的文件类似。



框架文件结构：

FrameWork.c：调度函数和线程初始化

FrameWork.h：头文件

MsgQ.c：消息队列

MsgQ.h：头文件

StateMachine.c：消息分发(状态机)

StateMachine.h：头文件

MsgDefer.c：消息延迟（可选是否编译）

MsgQDefer.h：头文件

FW_CFG.h：框架配置文件

C 源文件中要包含的文件:

DATA_CPU.H (包括数据类型和开关中断的宏替换)

2.2 如何配置:

配置文件: **FW_CFG.h**

```

#define EVENTDEFER_EN      0      //是否编译消息延迟函数
#define HFSM_EN            1      //使用层次状态机, 如果有效会增加 1K 代
                                码

typedef   INT8U   EVENT_MSG;      // 定义消息数据类型
INT8U,INT16U,INT32U

#define MSG_PARAM_SIZE    1      //消息参数大小:0,1,2,4,如果=0, 则消息发
送函数不带参数
//为每一个线程配置消息队列的大小
#define PRIO1_MSG_BUF_SIZE 20    //线程 1 消息队列大小,1-N
#define PRIO2_MSG_BUF_SIZE 30
#define PRIO3_MSG_BUF_SIZE 40
#define PRIO4_MSG_BUF_SIZE 50
#define PRIO5_MSG_BUF_SIZE 60
#define PRIO6_MSG_BUF_SIZE 20
#define PRIO7_MSG_BUF_SIZE 10
#define PRIO8_MSG_BUF_SIZE 10    //线程 8 消息队列大小,1-N

//为每一个线程配置延迟消息队列的大小
#if EVENTDEFER_EN
#define PRIO1_DEFER_BUF_SIZE 1    //线程 1 延迟消息队列大小,1-N
#define PRIO2_DEFER_BUF_SIZE 1
#define PRIO3_DEFER_BUF_SIZE 1
#define PRIO4_DEFER_BUF_SIZE 1
#define PRIO5_DEFER_BUF_SIZE 10
#define PRIO6_DEFER_BUF_SIZE 10
#define PRIO7_DEFER_BUF_SIZE 1
#define PRIO8_DEFER_BUF_SIZE 1    //线程 8 延迟消息队列大小,1-N
#endif

#include   "..\FRAMEWORK\assert.h"
#include   "..\FRAMEWORK\StateMachine.h"
#if HFSM_EN
#define SM_STRU HFSM

```

```
#else
#define SM_STRU FSM
#endif
#include "..\FRAMEWORK\FrameWork.h"
#include "..\FRAMEWORK\MsgQ.h"
#if EVENTDEFER_EN
#include "..\FRAMEWORK\MsgDefer.h"
#endif
```

- 1、 用户需要修改消息队列大小
- 2、 有限状态机只支持一层状态机，代码较少，应用容易，初学者（或不熟悉状态机）请选择有限状态机。
- 3、 对层次状态机熟悉（复杂程序）可选择层次状态机
- 4、 消息延迟可以选择是否编译。

2.3 线程初始化：

调用函数 `void ThreadInitTran (INT8U prio,STAT_PTR firststat)` 完成一个线程的初始化。输入参数：线程优先级号（1---8），第一个状态函数名

功能：线程初始化，并执行进入动作。

如果不熟悉状态机，第一个状态函数名可以认为是一个任务，但这个任务必须收到触发它的消息消息才能执行。

比如下面是一条初始化代码：

```
ThreadInitTran (1,(STAT_PTR)& ActObj7TmrTick);
```

表示 1 号线程初始化为 ActObj7TmrTick 状态，ActObj7TmrTick 本身是一函数，也是状态。也就是一个线程可以有不同的状态，在不同状态下可以执行不同的操作（也就是运行不同的函数）。

ActObj7TmrTick 也可以是一个普通函数：

```
STAT_PTR ActObj7TmrTick(SM_STRU *psm)
{
    switch (MSG(psm)) {
        case TICK_TIMEOUT_EVT://10ms
            Tmr_Tick();
            return (STAT_PTR)0;
    }
    return (STAT_PTR)&HfsmTop;
}
```

上面函数中，当有线程或中断发消息给这个线程时，本线程程序被执行，这个程序只处理：`TICK_TIMEOUT_EVT`，当收到这个消息，则运行 `Tmr_Tick()`。

2.4 调度开始：

把要运行的线程都用 `ThreadInitTran()` 初始化后，在主循环中调用 `ThreadScheduler()` 就可以了。

```
while(1){
    ThreadScheduler ();
}
```

因为是消息驱动的，所以任何一个线程收到消息时，就会被调度。

2.5 例程（在 LPC23XX/LPC24XX 开发板上运行）

例程中表四个线程：

```
void InitialObj(void)
{
    ThreadInitTran(1,(STAT_PTR)&COMM1_S0);
    ThreadInitTran(2,(STAT_PTR)&COMM2RxTx);
    ThreadInitTran(5,(STAT_PTR)&CanRxTxData);
    ThreadInitTran(7,(STAT_PTR)&ActObj7TmrTick);
}
```

线程 1 演示一个层次式状态机，通过 UART0 输入做消息驱动状态机运转，同时串口会打印出状态机的转换过程。

线程 2 是 串口 1 收发演示，收到一个字节后，再把这个字节发出去。

线程 5 是 CAN 总线的一个收发实验。

线程 7 是个定时构件，被 10MS 执行一次。

第三章、调度原理

3.1 调度原理：

- 1、 用一个字节变量的每一位代表一个任务是否就绪，1为就绪，0为休眠。
- 2、 这个字节从高位到低位代表的任务，优先级也从高到低。
- 3、 通过查表从就绪的任务中找出最高优先级的任务并执行，同时清就绪标志。

就绪表 ActObjReadySet

1	0	1	0	0	0	0	0
---	---	---	---	---	---	---	---

位：	7	6	5	4	3	2	1	0
任务号：	8	7	6	5	4	3	2	1

上表表示有两任务：任务8和任务6 就绪。

因为8位优先级高，我们来查表：

```
PRIORITY_TABLE[] = {0, 1, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4};
```

```
ready = ThreadReadyList; // 10100000
```

```
if (ready != 0) {
    if ((ready & 0xF0) != 0U) {
        prionum = PRIORITY_TABLE[ready >> 4] + 4;
    } else {
        prionum = PRIORITY_TABLE[ready];
    }
}
```

查表结果为4，4+4=8

计算结果为8，所以优先级为8的任务先执行，并清就绪位，完成后再次计算结果为6，优先级为6的任务再执行。

通过从不同的线程结构中取得函数指针来执行不同的线程函数

3.2 使用任务调度的优势

- 1、多个线程同时就绪时，高优先级先执行。
- 2、高优先级线程，最长等待时间是上一个正执行线程的完成时间
- 3、因为主循环时间最长时是最长线程的执行时间，所以有些中断中执行的代码可以移到任务中。
- 4、可以减少条件语句。
- 5、使软件结构更合理，清晰。

3.3 被调度函数指针的获取：

1、初始化代码：

```
ThreadInitTran(1,(STAT_PTR)&COMM1_S0);  
COMM1_S0 的函数指针赋给了 1 号线程的状态指针变量：
```

```
struct fsmtag {  
  
    STAT_PTR  StatFunPtr;  
  
    EVENT_MSG EventMsg;  
  
    #if (MSG_PARAM_SIZE != 0)  
  
    EVENT_PARAM EventParam;  
  
    #endif
```

};当前状态指针变量 StatFunPtr 中保存了： COMM1_S0 的函数指针。
当调度时，运行这个函数指针指向的函数。

2、状态转换：

TRAN(target_) 是一个宏替换，代码如下：

```
#define TRAN(target_) \  
    (((FSM *)psm)->StatFunPtr = (STAT_PTR)(target_), \  
    \
```

((FSM *)psm)->EventMsg = (EVENT_MSG)0 //为执行进入和退出动作。

比如：

TRAN(&COMM1_S211);

把状态函数指针变量：StatFunPtr 变为 COMM1_S211

下次这个线程被调度时，就执行：COMM1_S211（）。

注：对不熟悉状态机或程序较简单，就可不用这部分（指状态转换）。

第四章 消息队列

4.1 消息队列说明

消息队列是线程间通信的方式，每一个线程有一个队列与一对应，队列长度可设定，一个线程向另一个线程发消息时，调用函数：

```
MSGQUEUE_EXT void MsgQPost(INT8U prio,EVENT_MSG msg,EVENT_PARAM par);
MSGQUEUE_EXT void MsgQPostISR(INT8U prio,EVENT_MSG msg,EVENT_PARAM par);
```

（注： 第二个函数为中断中调用。）

消息发送到消息队列后，对应的线程马上就绪，当没有更高优先级的线程就绪时，收到该消息的线程被调度，线程中的程序会运行对应该消息的处理程序。

从队列中取出消息由调度函数（ThreadScheduler（））来完成，用户不用干预，只需发送消息就可以了。

4.2 消息队列函数：

向队列发送消息有两个函数：

```
MSGQUEUE_EXT void MsgQPost(INT8U prio,EVENT_MSG msg,EVENT_PARAM par);
MSGQUEUE_EXT void MsgQPostISR(INT8U prio,EVENT_MSG msg,EVENT_PARAM par);
```

第一个参数就是消息的目的地线程号；

第二个参数就是发送什么消息；

第三个参数可选，可同时传递一个参数。

比如：

```
c = U0RBR;
MsgQPostISR (3,COMRX_EVT,c);
```

串口中断中调用上面两条代码，把收到数据的消息（COMRX_EVT）发送到 3 号线程，3 号线程就会就绪，同时知道串口收到数据，可以去处理，后面的参数 C 可用可不用，如用则串口不再用缓冲区，用消息队列代替就可以了。

如果在线程中调用则用另外一个函数：

```
MsgQPost (3,COMRX_EVT,c);
```

第五章 如何写一个线程

5.1 线程介绍:

线程(thread)是依序执行的一组可执行的动作, 线程的总体也被称作任务。由调度器 ThreadScheduler()来调度执行, 每一个线程都是消息驱动的, 当某线程的队列中有消息时, 它才会就绪并被调度执行。所以每一个线程都必须处理消息。

5.2 不用状态机时的线程程序编写

如果还不熟悉状态机, 从前后台转向本框架, 则只需遵守一个简单的格式, 编程方式不用改变, 或者你可以拷贝原来的代码。

比如: 你原来的按键扫描程序: KeyScan(); 在主循环中每 20MS 调用一次。现在改在 FramwWork 框架下, 用 1 号线程: objStat1

1、线程初始化: ThreadInitTran(1,(STAT_PTR)&ObjStat1);

2、线程程序:

```
STAT_PTR  ObiStat1(SM_STRU *me)
{
    switch (MSG(me)) {
        case KEY_SCAN_EVT://20ms
            KeyScan ();
            return (STAT_PTR)0;
    }
    return (STAT_PTR)&HfsmTop;
}
```

你的 KeyScan () 程序和前后台没有区别, 只需要另外的线程或中断每 20MS 发一个消息 KEY_SCAN_EVT, EvtPostQue(1, KEY_SCAN_EVT, 0);这个线程现在只处理一个消息, 就是 20MS 的按键扫描消息。

如果你的 LCD 刷新程序 LCDDISP () 也想放在这个线程, 50MS

刷新一次。

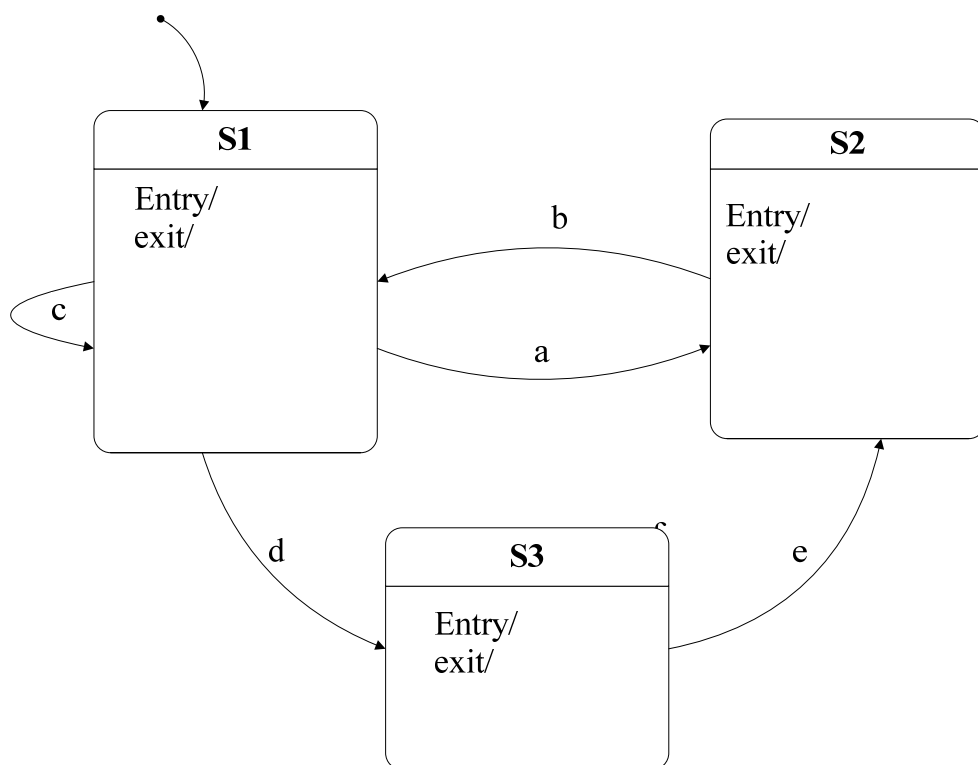
则程序修改为：

```
STAT_PTR  ObjStat1(SM_STRU *psm)
{
    switch (MSG(psm)) {
        case KEY_SCAN_EVT://20ms
            KeyScan ();
            return (STAT_PTR)0;
        case LCD_DISP_EVT:
            LCDDISP();
            return (STAT_PTR)0;
    }
    return (STAT_PTR)&HfsmTop;
}
```

5.3 用有限状态机方式的线程编写

每一个线程就是一个状态机，每一个状态就是一个状态函数，改变状态就是改变函数指针。

看下面的状态图：



状态的进入和退出动作就是向串口发送进入或退出某状态，消息

由串口输入，程序如下：

1、初始化时调用如下函数：

```
ThreadInitTran(1,(STAT_PTR)&COMM1_FSM_S1); //有限状态机
```

2：状态图代码：

```
STAT_PTR COMM1_FSM_S1(SM_STRU *psm)
{
    switch (MSG(psm)) {
        case ENTRY_EVT:
            PrintCOMM(COMM1,"FSM TEST:please input 0A,0B,0C,0D,0E ");
            PrintCOMM(COMM1,"S1-ENTRY;");
            TmrStart(0,5000);
            return (STAT_PTR)0;
        case EXIT_EVT:
            PrintCOMM(COMM1,"S1-EXIT;");
            return (STAT_PTR)0;
        case 0x0A:
            PrintCOMM(COMM1,"S1--0A;");
            TRAN(&COMM1_FSM_S2);
            return (STAT_PTR)0;
        case 0x0d:
            PrintCOMM(COMM1,"S1--0D;");
            TRAN(&COMM1_FSM_S3);
            return (STAT_PTR)0;
        case TIPSMOUT_EVT:
            TmrStartISR(0,5000);
            PrintCOMM(COMM1,"HFSM TEST:please input 0A,0B,0C,0D,0E,0F;");
            return (STAT_PTR)0;
    }
    return (STAT_PTR)0;
}
```

```
STAT_PTR COMM1_FSM_S2(SM_STRU *psm)
{
    switch (MSG(psm)) {
        case ENTRY_EVT:
            PrintCOMM(COMM1,"S2-ENTRY;");
            return (STAT_PTR)0;
        case EXIT_EVT:
            PrintCOMM(COMM1,"S2-EXIT;");
```

```
        return (STAT_PTR)0;
    case 0x0B:
        PrintCOMM(COMM1,"S2--0B;");
        TRAN(&COMM1_FSM_S1);
        return (STAT_PTR)0;
    }
    return (STAT_PTR)0;
}

STAT_PTR COMM1_FSM_S3(SM_STRU *psm)
{
    switch (MSG(psm)) {
        case ENTRY_EVT:
            PrintCOMM(COMM1,"S3-ENTRY;");
            return (STAT_PTR)0;
        case EXIT_EVT:
            PrintCOMM(COMM1,"S3-EXIT;");
            return (STAT_PTR)0;
        case 0x0E:
            PrintCOMM(COMM1,"S3-0E;");
            TRAN(&COMM1_FSM_S2);
            return (STAT_PTR)0;
    }
    return (STAT_PTR)0;
}
```

3、代码说明:

(1)、代码和图完全一致，每一个函数就是一个状态，每一个函数中都有一个处理消息的 SWITCH 语句。

(2)、ENTRY_EVT 是进入动作执行的消息，由框架代码自动执行。

如果有进入动作，就填写相关代码，如没有就可以没有这个 CASE 语句。

(3)、EXIT_EVT 是退出动作执行的消息，由框架代码自动执行。

如果有退出动作，就填写相关代码，如没有就可以没有这个 CASE 语句。

(4)、每一个 CASE 就是一个消息。

(5)、TRAN () 是状态转换，参数是状态名（也就是状态函数名）。

(6) 在 ENTRY_EVT 和 EXIT_EVT 下面的代码中：不能调用 TRAN ()。（切记）

第六章：延迟消息队列(可选)

延迟消息对简化状态模型是一种重要的技术。当一个消息在不适当的或难以处理的时候到来时，可以延迟该消息，当状态机能处理它时，再唤醒它。

延迟消息有两个函数：

1、 `BOOLEAN MsgDefer(INT8U prio,EVENT_MSG msg,EVENT_PARAM par);`

向延迟队列发送一个消息。（延迟一个消息）

2、 `void MsgRecall(INT8U prio);`

如果延迟队列中有消息，则取出放到消息队列中去处理。

（唤醒一个消息）

这两个函数不复杂，容易理解，但使用这两个函数，就需要技巧。

当某状态不能处理这个消息，就把消息放入延迟消息队列

比如：

```
if(GetState(6) == &Com2Rx){
    MsgQPost (6,msg); //如果当前状态是 Com2Rx，则发送消息
}else{
    MsgDefer (6,msg); //如果不是 Com2Rx 状态，则延迟这个消息。
}
```

在能处理延迟消息的状态函数的进入动作中调用：

上面的例子，当状态回到 Com2Rx 时，在它的进入动作中调用

`MsgRecall ()`：调用后这个消息就进入正常的消息队列中被分发。

```
STAT_PTR Com2Rx (HFSM *psm)
{
    switch (MSG(psm)) {
        case ENTRY_EVT:
            MsgRecall (6);
    }
}
```

```
return (STAT_PTR)0;
```

```
... ..
```

第七章、框架下的编程规则

框架的编程规则

- 1、各线程不能共享资源。可以由一个线程管理共享资源。

其它线程通过这个线程来访问共享资源。

- 2、线程间通过消息队列相互作用。
- 3、每一个线程都不能阻塞，以互相等待。

状态处理器的编程规则

- 1、进入、退出和初始转换用保留消息。

```
#define    EMPTY_EVT        0    // 用于查找状态层次
#define    ENTRY_EVT        1    // 进入动作消息
#define    EXIT_EVT         2    // 退出动作消息
#define    INIT_EVT         3    // 初始转换消息
```

- 2、不能在进入或退出动作中进行状态转换。
- 3、对于层次式状态机，不能处理消息时返回超状态。