基于 UIP-Kernel 开发应用程序的方法与框架

一、针对不同应用类型的处理方式

1. 周期性数据输入

采用时间（alarm）触发或者事件（中断）触发的方式，周期的将数据从外部接口导入到系统维护的“数据池”中，之后利用 **SetEvent** 或者 **ActivateTask** 的方式激活周期任务对数据进行后续处理。

2. 非周期性关键指令输入

采用事件（中断）触发的方式，获得异步产生的非周期性数据。后续处理时，若不关联到共享资源，且能较为快速地处理完，例如设置 GPIO 电平高低，或者输出短消息等，可在中断处理程序中执行；若后续操作会关联到共享资源，或者执行时间较长，或者会关联到多个逻辑，则利用任务来处理，且中断处理程序中通过 **SetEvent** 或者 **ActivateTask** 的方式激活该任务来处理。

3. 非周期性大量数据输入

有数据输入时，触发一个中断，因后续操作执行时间较长，则利用任务来处理。即，中断处理程序中通过 **SetEvent** 或者 **ActivateTask** 的方式激活任务来处理大量的输入数据。

4. 周期性数据输出

利用 alarm 获得周期执行时机，设置 alarm 控制块相关字段，alarm 到期后通过 **SetEvent** 或者 **ActivateTask** 激活处理任务输出数据；若输出操作时间较短，也可通过执行 alarm 关联的回调函数执行输出操作，从而减少任务切换次数。

5. 非周期性消息输出 1，基于任务调度随时输出

两种处理方式：

1）在一个 c 文件中统一设计实现消息输出逻辑，但对其调用则分散在整个应用软件中，即，被多个任务适时调用。

2）设计实现一个消息输出任务，等待（waiting）在事件上，或者处于终止状态，当其它任务需要输出消息时，则通过 **SendMessage** 激活该输出任务，并将输出数据传递过来；**SendMessage** 中激活任务的方式可以是设置事件（**SetEvent**），也可以是激活任务（**ActivateTask**）。

6. 非周期性指令输出 2，基于时间表、特定状态量输入、特定实时状态

设计实现一个事件处理引擎（见 framework_demo.c 源代码），每次有事件到达时利用 while(1)循环遍历一个结构体链表，若匹配上结构体链表中的某个事件，则执行对应的处理函数。该结构体中包含一个整型变量用于表征事件类型，还包含一个函数指针，指向处理函数。

该事件处理引擎被一个任务调用，该任务平时处于等待（waiting）状态，被 alarm 或者中断处理程序中的 **SendMessage** 激活，当时间到或者事件到时，触发 alarm 或者中断处理程序。

初始化时，基于需要的动作时间点或者状态量等事件，配置该结构体链表中每个结构体的成员数值。

7. 非周期性大量数据输出，某一时刻开始，进行一段时间的大量数据输出，实时性要求不高

由一个低优先级任务实现数据输出，该任务平时处于等待（waiting）状态，等待多个事件（“与”（and）模式），当其它逻辑将各数据准备好后会设置相关事件，所有事件都被设置后，该任务将进入就绪状态，被调度执行后将执行一段时间的大量数据输出；因该任务优先级较低，在执行数据输出期间可以被其它处理关键逻辑或者数据的高优先级任务抢占。

8. 数据预处理，大规模计算

一个或者多个任务实现该“大规模计算”，由数据到时触发的中断或者定时执行的 alarm 激活上述任务。根据预处理的执行时序和重要性来确定上述任务的优先级，并由该任务在大规模计算后通过调用 **SetEvent** 或者 **ActivateTask** 来同步后续逻辑的执行。因为是预处理阶段，所以可设置中等级别的优先级，能被更重要的任务抢占。

9. 数据预处理，来源于不同时间周期的数据同步

由一个任务实现该数据预处理，该任务平时处于等待（waiting）状态，等待多个事件（“与”（and）模式），多个不同时间周期的 alarm 利用回调函数将数据导入数据池，再通过 SetEvent 设置事件，当所有数据就绪后，该任务被激活继续执行，完成数据预处理。

10. 任务调度，基于周期性数据进行切换，基于控制消息执行

与（6）中执行方式类似，建立一个**任务计划表**，是一个结构体链表，在系统设计阶段离线定义控制消息的类型、个数与处理函数，并在系统软件初始化阶段注册到上述链表中。

该调度任务的核心是**计划处理引擎**，类似于前述事件处理引擎（见 framework_demo.c 源代码），每次到达计划时机时，通过 alarm 或者其它中断处理程序使该任务的 **WaitEvent** 被激活，任务执行体利用 while(1)循环遍历结构体链表，若匹配上结构体链表中的某个事件，则执行对应的处理函数，这些处理函数可以执行本地操作，也能通过调用 **SetEvent** 或者 **ActivateTask** 等系统 API 激活其它处理逻辑。

11. 参数共享和保护，大量参数的访问保护、实时接收的数据、预处理数据、状态量与消息

对上述数据池或者其它参数池，若存在读写共享的情况，即，可能有多个任务读取或者写入数据池，则需要用资源（resource）机制保护这些共享数据。

根据实际情况对上述数据池或者参数池分组，将关联若干任务的数据分为一组，从而便于设置资源的冲顶优先级。

二、软件构件、中间件与框架（framework）

1. 构件化软件开发

- 遵循统一的标准开发软件构件；
- 开发新应用不再是从头开始，而是基于构件进行组装和部署；
- 构件的重用性强，可生成自有的构件库，也可购买第三方的标准构件来组装新应用，符合社会化大分工的趋势，大幅度提高生产效率和效能；
- 解耦合，分解复杂的系统，独自验证，提高系统的可靠性和可维护性。

2. 软件构件与中间件

- 若要实现来自不同厂商软件构件的可信组装与灵活部署，必须为其提供标准化的运行环境；
- 该运行环境位于应用构件之下，底层操作系统之上，即为中间件层，属于非功能代码；
- 对于工控、汽车电子等控制类应用，该中间件一般为实时通信中间件，还包含构件运行“容器”；

- 中间件也可由第三方开发；实时通信中间件之下的实时操作系统（RTOS）也是重要“基础设施”，需要标准化，也可划入中间件范畴。

3. 构件化软件开发方法论（ Methodology）

- 整个生命周期划分成标准化的多个阶段：构件研制、研制阶段配置、研制阶段系统分析、构件与中间件组装（装配）、组装阶段配置、系统部署、组装部署阶段系统分析、系统运行时状态检查；
- 标准化：上述各阶段过程与相关工具的标准化定义与描述、构件模型的标准化、中间件的标准化、交互方式的标准化等等。（描述过程、方式、方法、工具、模板，并进行标准化定义，以方便第三发开发与使用。）

4. 基于 UIP-Kernel 实现的应用框架（Framework）

可以看出，中间件处于应用软件与地层软件/硬件之间，起到承上启下的作用，是系统的非功能代码，完成某些特定功能。可以将硬件与最上层应用软件之间的软件统称为中间件（广义中间件），也可将该广义中间件划分为若干层，每一层都是中间件（狭义中间件）。RTOS 可被视作中间件、TCP/IP 协议栈也可被视作中间件，但在软件界传统定义中，中间件一般是指实时通信中间件（例如 RT CORBA）、数据库中间件、消息传递中间件等针对大型分布式、复杂系统的中间软件，一般位于 RTOS 和底层通信协议栈之上。

应用框架（framework）是特定应用领域中被抽象出来的，具有普遍性，可被重复使用的支撑软件，也属于非功能代码，可归属中间件范畴。基于 UIP-Kernel 的多任务（多线程）编程模型，更容易开发出大型、复杂、可靠性高、满足实时性要求、可移植性强、可扩展能力强的应用框架。

如果一定要区分应用框架与中间件的细微差别，则：应用框架更接近应用层，其执行时序、执行模式是固定的，由框架开发人员设定，应用软件只能被动使用和配置参数而不能改动，框架的执行“线索”上提供了很多“钩子点”，“钩住”应用软件子模块来执行。而中间件则更接近下层，执行逻辑更具有普遍性和一般意义，“支撑”了上层执行逻辑（即，支撑上层逻辑的执行），因为更具有一般性，上层应用可利用 API 调用的方式来使用这些中间件逻辑。

我们面临的应用领域，有很多非功能代码可定义为应用框架，抽象为中间件。例如：

（1）针对被一系列的时间点（某个时刻）或者异步事件（例如外部状态）触发而执行的逻辑，可定义一个结构体，构建一个结构体链表和一个事件处理引擎（见 framework_demo.c 源代码）， 该引擎被面向计划（planning）的任务执行体调用。该任务在我们的 UIP-Kernel 实时操作系统调度下执行。

```
typedef struct _EventOrTimeHandlerEntry {
   uip_UINT                  eventOrTime;
   VOID                      (*eventOrTimeHandler)(VOID);
} EventOrTimeHandlerEntry;
/* 应用软件构造一个 EventHandlerTable 的数组，每次有事件（或者时间）到达时，遍历此数组，匹配后执行相应处理程序 */
#define       MAX_ENTRY_NUMBER       128
EventOrTimeHandlerEntry     gAppEventOrTimeHandlerTable[MAX_ENTRY_NUMBER]
/* 针对上述数组管理方式的注册函数，只是展示思想，可进一步优化 */
StatusType RegisterEventHandler( uip_UINT number, uip_UINT eventOrTime, VOID (*handler),)
{
      if(number >= MAX_ENTRY_NUMBER)
      {
         return uip_FALSE;
      }
      gAppEventOrTimeHandlerTable[number].eventOrTime = eventOrTime;
      gAppEventOrTimeHandlerTable[number].eventOrTimeHandler = handler;
      return uip_TRUE;
}
/* 事件或者时间处理机制 */
void AppEventOrTimeHandle(uip_UINT eventOrTime)
{
      uip_INT   i;
      for(i = 0; i < MAX_ENTRY_NUMBER; i++)
      {
          if(gAppEventOrTimeHandlerTable[i].eventOrTime == eventOrTime)
          {
              /* 若匹配成功，则执行对应的处理函数 */
              gAppEventOrTimeHandlerTable[number].eventOrTimeHandler();
              return uip_TRUE;
          }
      }
      return uip_FALSE;
```

```
}
/* 使用上述机制的任务执行体框架：在 waiting 状态下，等待某个事件或者时间到达，该事件或者时间是中断处理程序判断后，通过 SendMessage()发送来的，而 SendMessage 使用的消息，采用设置事件（setevent）的方式激活等待任务 AppEventOrTimeHandleTask */
TASK(AppEventOrTimeHandleTask)
{
    StatusType        appStatus;
    uip_UINT          msgBuf[1];
    while(1)
    {
        /* 被 AppDataPreProcessTask 利用 SendMessage 中的设置事件激活，周期 5ms */
        appStatus  = WaitEvent(EVENT_EVENT_OR_TIME_ARRIVE);
        if(appStatus == E_OK)
        {
            appStatus = ClearEvent(EVENT_EVENT_OR_TIME_ARRIVE);
            /* 接收发送来的具体事件或者时间数值 */
            appStatus = ReceiveMessage(MSG_INT_2_HANDLE_TASK, &msgBuf[0]);
            if(appStatus == E_OK)
            {
                AppEventOrTimeHandle(msgBuf[0]);
                /* 若传过来的是索引值，则执行如下操作 */
                //gAppEventOrTimeHandlerTable[msgBuf[0]].eventOrTimeHandler();
            }
        }
    }
}
/* 若用时间触发，则在 UIP-Kernel 内核中增加对上述框架的支撑代码，如下，每次系统 tick (毫秒级) 到时，会通过 alarm 中回调函数的方式执行如下逻辑 */
void uipSysTimeMapping(void)
{
    for(i = 0; i < MAX_ENTRY_NUMBER; i++)
    {
        if(gAppEventOrTimeHandlerTable[i].eventOrTime == uipSystemCounter.uipCtrValue)
        {
            /* 若匹配成功，则执行发送消息函数，发送数组的索引值 */
            SendMessage(MSG_INT_2_HANDLE_TASK, i);
            break;
```

```
        }
    }
}
/* 若用事件触发，则在中断处理程序中调用如下逻辑 */
void AppEventMapping(uip_UINT eventOrTime)
{
    /* 将事件直接发送到任务 */
    SendMessage(MSG_INT_2_HANDLE_TASK, eventOrTime);
    /* 或者在中断处理程序中匹配，将匹配后的索引值发送到任务 */
//  for(i = 0; i < MAX_ENTRY_NUMBER; i++)
//  {
//      if(gAppEventOrTimeHandlerTable[i].eventOrTime == eventOrTime)
//      {
//          /* 若匹配成功，则执行发送消息函数，发送数组的索引值 */
//          SendMessage(MSG_INT_2_HANDLE_TASK, i);
//          break;
//      }
//  }
}
/* 消息控制块内容如下 */
uip_UCHAR AppMsgBuf_Int2Handle[4];
{
  MSG_INT_2_HANDLE_TASK,  /* Message's id */
  0,                      /* Messages's current state(locked, unlocked,
                             overflow or not etc.) */
  uip_MSG_QUEUED,         /* Messages's type(unqueued or queued message) */
  0,                      /* Flags to be set for notifying */
  (uip_UCHAR_PTR)&AppMsgBuf_Int2Handle[0],      /* Point to buffer start address */
  (uip_UCHAR_PTR)0,       /* Point to buffer address to be writen */
  ((uip_UCHAR_PTR)&AppMsgBuf_Int2Handle[0]+sizeof(AppMsgBuf_Int2Handle)),
                          /* Point to buffer end address */
  (uip_UCHAR_PTR)0,       /* Point to buffer address to be read */
  4,                      /* Size of messages(bytes) */
  EVENT_EVENT_OR_TIME_ARRIVE,  /* Event to be set for notifying */
  (VOID *)0,              /* Callback routine called when sending message */
  TASK_EVENT_OR_TIME_HANDLE_ID,  /* The ID of task to be nofified */
  uip_COM_NOTIFICATION_SETEVENT, /* notification type */
  0,                      /* Padding byte for ARM */
```

```
    0,                              /* Padding byte for ARM */
}
```

上述机制与算法即为一个应用框架，抽象为中间件后可被多个应用软件使用。

（2）健康状态检查。基于 UIP-Kernel 构建一个检查当前运行状态，并根据状态触发相应动作的框架：提供应用逻辑“抛出”当前状态的钩子函数执行点、运行状态存储、消息传递等服务，提供启动、终止应用逻辑子模块的容器（类似软件构件执行的容器），为应用逻辑错误状态“抛出”、出现错误时的处理逻辑规划等提供标准接口。该健康状态检查框架可被各种系统软件重用。

基于上述中间件，可对研发人员分类，一部分专门负责框架开发、维护、升级，另一部分负责基于框架开发上层应用。前者具有系统架构师性质，定义整个应用软件的执行时序，每个时序的执行逻辑，指定各执行逻辑的属性；后者开发、调试针对性强的具体应用代码子模块，负责其功能与性能质量，但不必过多关心其执行时序以及与其它执行逻辑的具体交互。上述开发模式已经具备了中间件（框架）和构件化软件（具体应用代码）分离开发的特征。