# TOPPERS中级实装讲座

## （H8/3069F版：基本）

## 基础篇：第1天

TOPPERS工程

培训工作组



# 本教材的使用条件

NEXCESS中级课程02
应用实时OS的软件设计技术

Copyright (C) 2006 by 名古屋大学嵌入式软件技术人材培养计划
Copyright (C) 2006 by 本田晋也、高田广章、 山本雅基

只有满足下述(1)～(4)的条件，上述著作权者才允许其无偿使用、复制、更改、翻译、翻印（以下称为使用）本教材（包括由本教材改编・翻译的材料，以下同）。

(1) 本教材中的著作权标志等原样不变地包含在资料中。
(2) 翻印本资料时，需通过下述网站报告翻印形式等内容。
http://www.nces.is.nagoya-u.ac.jp/NEXCESS/REPORT/
(3) 更改・翻译本资料时，资料中应包含更改・翻译本资料的原因。
而且，更改・翻译者的著作权问题要与本框内的著作权问题分开记述。
(4) 因使用本资料而造成的任何直接或间接的损害，均与上述享有著作权的作者无关。

※ 本资料的一部分内容是利用了文部科学省科学技术振兴调整费，作为名古屋大学嵌入式软件技术人材培养程序（NEXCESS）的一环编制而成的。
※ 本资料中提到的商品名、服务名等是各公司的商标或者注册商标。



# 关于本教材的注意事项

1．关于著作权的表述

---

<TOPPERS中级实装讲座基础篇(H8-3069F版：基本）第1天>

Copyright (C) 2005-2006 by 高田广章 名古屋大学
Copyright (C) 2005-2006 by 山本雅基 名古屋大学
Copyright (C) 2005-2006 by 本田晋也 名古屋大学

只有满足下述(1)～(3)的条件，上述享有著作权的作者才允许无偿使用、复制、更改、翻印（以下称为使用）本资料（包括由本资料改编的文件，以下同）。

(1) 使用本资料时，上述著作权记述以及使用条件应保持原样包含在资料中。
(2) 更改本资料时，资料中应包含更改本资料的原因。但是，作为TOPPERS工程活动的一个环节更改本资料时，则无需记述更改本资料的原因。
(3) 因使用本资料而造成的任何直接或间接的损害，均与上述享有著作权的作者以及TOPPERS工作组无关。

---

2．关于本资料如果有任何意见、建议、想法或疑问，请通过电子邮件与TOPPERS办事处进行联系。

3．关于本资料的内容，出于调整和改善的目的，可能会对内容进行修订，且不予通告。

本资料中使用了Microsoft公司的Clip Art Gallery中的内容。
TRON是“The Real-time Operating system Nucleus”的简称，ITRON是“Industrial TRON”的简称，μITRON是“Micro Industrial TRON” 的简称，TOPPERS/JSP是“Toyohashi Open Platform for EmbeddedReal-Time System/Just Standard Profile Kernel”的简称。
本资料中提到的商品名以及商标名是各公司的商标或注册商标。



# 日 程 表

- 第1天

| | |
|---|---|
| 1．开发环境的确认 | 0.5小时 |
| 2．基本程序设计 | 1.5小时 |
| 3．ITRON规范的概要 | 0.5小时 |
| 4．ITRON API解说 | 2.5小时 |
| 5．碗面定时器的开发 | 1.0小时 |

- 第2天

| | |
|---|---|
| 1．TCP/IP的概要 | 1.0小时 |
| 2．TRON TCP/IP API 规范 | 1.0小时 |
| 3．TINET（TCP/IP栈） | 0.5小时 |
| 4．TCP服务器程序设计 | 2.0小时 |
| 5．TCP客户端程序设计 | 1.0小时 |
| 6．总结 | 0.5小时 |



## 基础篇的目标

- 学习AKI-H8/3069F电路板的设置、TOPPERS/JSP Kernel的构建方法、样例程序的运行方法
- 学习操作微处理器板上设备的程序的编程方法
- 学习应用ITRON规范API的多任务程序设计
- 理解TCP/IP的基本概念
- 学习使用ITRON TCP/IP API规范API的网络程序设计



# 开发环境的确认

1. 配备物品的确认
2. 开发对象的说明
3. 开发环境的说明



TOPPERS

## 配备物品的确认：教材、电路板、电缆

- PC机（Windows-PC）
- 教材
- PizzaFactory2
  - AKI-H8/3069F电路板+ I/O扩展板
  - 电路板配套手册・CD-ROM
  - 液晶基板
  - RS232C电缆
  - 10base-T 交叉电缆
  - AKI-H8/3069F电路板用电源



TOPPERS

| | |
|---|---|
| ・PC | 以下工具是必需的<br>PizzaFactory2,(Cygwin也可以）<br>（在Cygwin下进行开发的时候，可以用AKIH8 3069F电路板配套手册和CD-ROM进行安装)<br>编辑器<br>通信软件 |
| ・文本 | 本文本，μITRON4.0规范书摘要 |
| ・微处理器板 | 请确认AKI-H8/3069F与I/O扩展板、液晶基板的安装显示 |
| ・RS232C电缆 | 进行PC和电路板之间的通信<br>用于下载简易监控程序的显示、用户程序等 |
| ・10base-T交叉电缆 | 在从第2天开始的实习中，用于与PC之间的数据通信 |
| ・AKI-H8/3069F电路板用电源 | 给板提供电源 |

## 配备物品的确认：程序目录

- 开发程序在**environment**（程序目录）下
- 构造器（**configurator**）中已含在**Windows**下运行的**exe**
- 在除**Windows**之外的环境下使用的时候，需要重新构建构造器（**configurator**）
- 目录构成（**environment**以下）
  - jsp-1.4.2
    - TOPPERS/JSP Kernel 1.4.2+ TINET 1.3 源代码
    - 实习用程序(从第1天到第4天通用)
  - tcp_server
    - Windows用服务器程序
  - h8mon
    - H8用简易监控程序的源代码




开发程序在“environment”目录下。下面说明一下子目录的内容。

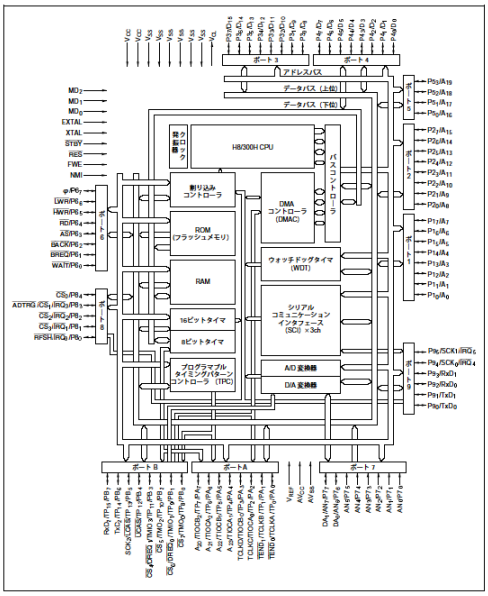
- jsp-1.4.2　在TOPPERS/JSP Kernel 1.4.2＋TINET1.3源代码中添加了实习用程序的开发环境
- h8mon　写入到AKIH8/3069F中的ROM监视器的源代码、写入格式数据、文档
- tcp_server　第2天的讲座中使用的PC上的TCP服务器程序
- netDevice　应用程序实习篇的讲座中使用的网络设备用服务器程序
- ExtBoard　在应用程序实习篇（第1天）的讲座中使用
  碗面模拟器：在扩展板上追加3个按钮
- potsim3069　在应用程序实习篇的讲座中使用
  话题烧水壶模拟器AKIH8/3069F电路版
- DeviceManager　Windows版设备管理器的运行方式
  用于netDevice的使用

# 开发环境的确认

1. 配备物品的确认
2. 开发对象的说明
3. 开发环境的说明

## 开发对象：处理器（H8/3069F）

- 瑞萨产16bit微处理器
- 最大运行频率: 25MHz
- 自带FLASH: 512KB
- 自带RAM: 16KB
- 外围电路
  - DMA控制器4通道
  - 16bit定时器3通道
  - SCI 3通道
  - 10bit A/D 8通道
  - 8bit D/A 2通道





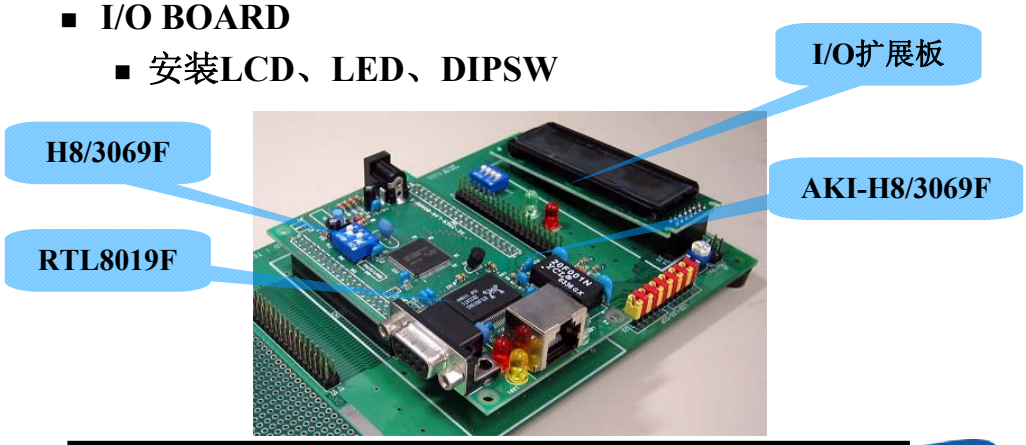

在中级实装讲座中，使用的是瑞萨16位微处理器H8/3069F。

它的最大运行频率是25MHz，而在AKIH8/3069F板上使用的是20MHz。在单片上内置有512KB的FLASH内存和16KB的静态RAM。在AKIH8/3069F板上安装有2MB的SDRAM外设。在本实习中，将H8用简易监控程序写入到512KB的FLASH内存上，然后使用该监控程序将TOPPERS/JSP和开发的应用程序下载到SDRAM中来验证程序。

外围电路基本上只使用并行I/O端口。

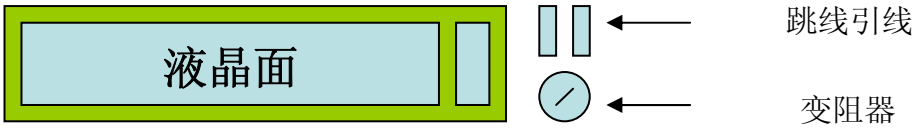
## 开发对象：电路板（AKI-H8/3069F）

- 搭载H8/3069F
- 搭载NE2000兼容网络控制器、RTL8019AS
- 内置有H8/3069F内置FLASH ROM用写入器电路
- I/O BOARD
  - 安装LCD、LED、DIPSW






下图是AKIH8/3069F电路板、I/O板的概略图。在I/O板上安装液晶基板。为了让液晶正确显示，需要安装跳线引线，并将变阻器设置为合适的值。这将会在后边的实习中进行确认。

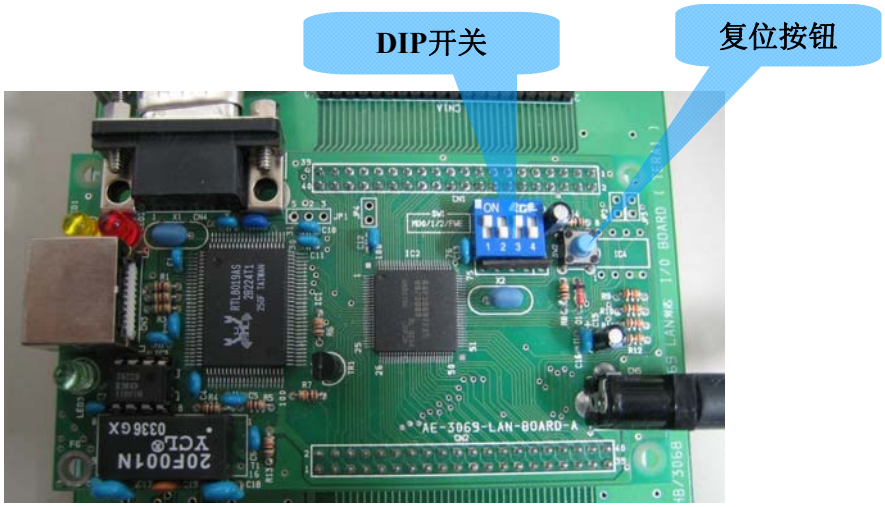



AKIH8/3069F板上安装了NE2000兼容的RTL8019AS网络控制器和LAN socket、串行驱动器和9针的D-SUB连接器。这些控制是由H8用TOPPERS-JSP和TINET来执行。

I/O板上安装了2个LED、4位的DIPSW、液晶基板。

## 开发对象：复位按钮与DIP开关

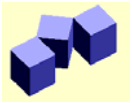
- 板上安装了复位按钮与用于模式变更的DIP开关



# 开发环境的确认

1. 配备物品的确认
2. 开发对象的说明
3. 开发环境的说明

## 开发环境：PizzaFactory2



http://support.toppers-open.org/

- 支持TOPPERS Kernel的开发环境的二进制发布包
- 收录了创建TOPPERS Kernel时所需的工具类以及Kernel的源代码
- 提供Windows运行方式(exe?)的安装程序，比Cygwin环境容易安装
- 编译器采用的是基于MinGW的GCC，而不是Cygwin。与基于Cygwin的编译器相比，它的速度要快两倍以上
- 运行桌面上的图标“PizzaFactory2CommandLine”，控制台就会被启动



TOPPERS

在本实习中，在PizzaFactory2的开发环境下进行程序开发。它比Cygwin更容易安装、卸载。因为可以连同开发环境一起安装，所以要特别推荐给对GNU的安装没有把握的人。而且与Cygwin相比还具有上述特征。

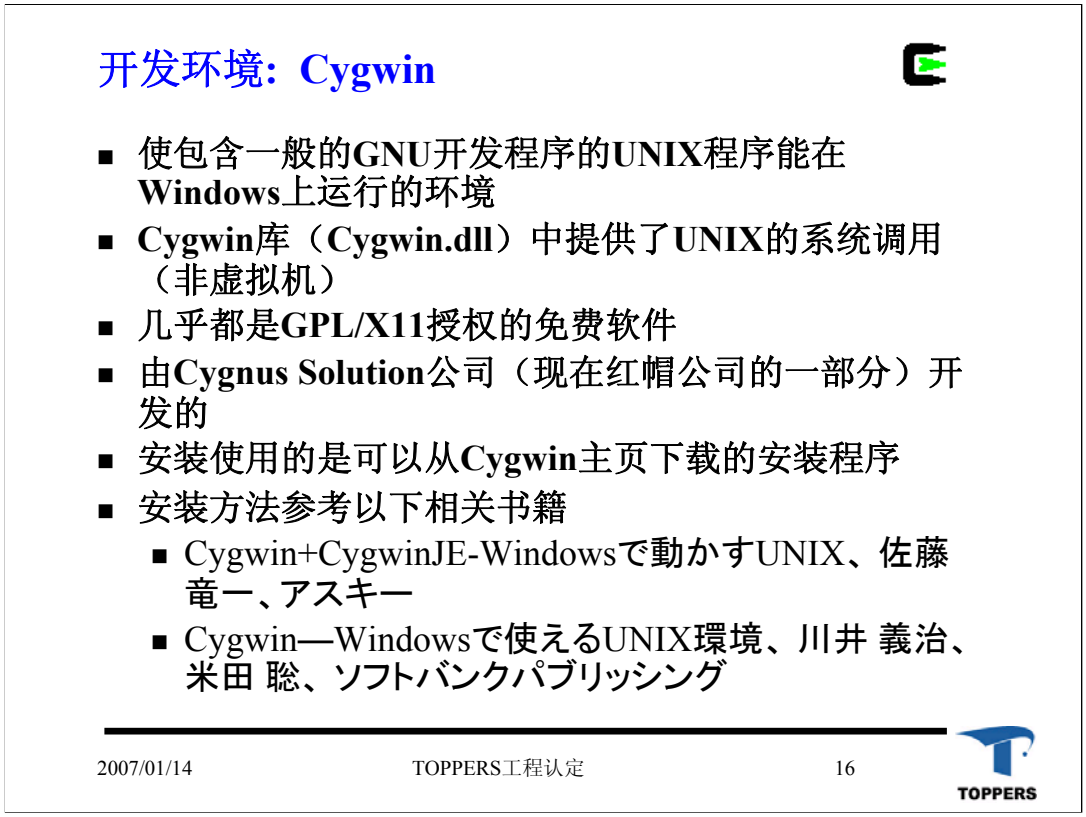
## 开发环境：GCC

- 由GNU工程开发的开源编译器
- GCC是“GNU Compiler Collection”的简称。正如它的名字一样，它支持多种语言（C、C++、Objective-C、FORTRAN、Java、Ada）
- 支持多种类型的处理器
  - Alpha、ARM、AVR、H8、IA64、M32R、M68K、MIPS、SH、SPARC、V850
- GCC作为汇编器和连接器调用binutils。
  - 汇编器(gas)、连接器(ld)、二进制输出（objdump）
- 作为调试器，GNU工程同样提供开源软件gdb



TOPPERS

在PizzaFactory2中，GNU开发环境是以二进制数据的形式进行安装的。自己在Cygwin上进行安装时需要一定的技巧。在TOPPERS/JSP的文档“doc目录”中有记述了GNU安装方法的文档“gnu_install.txt”，请参考。

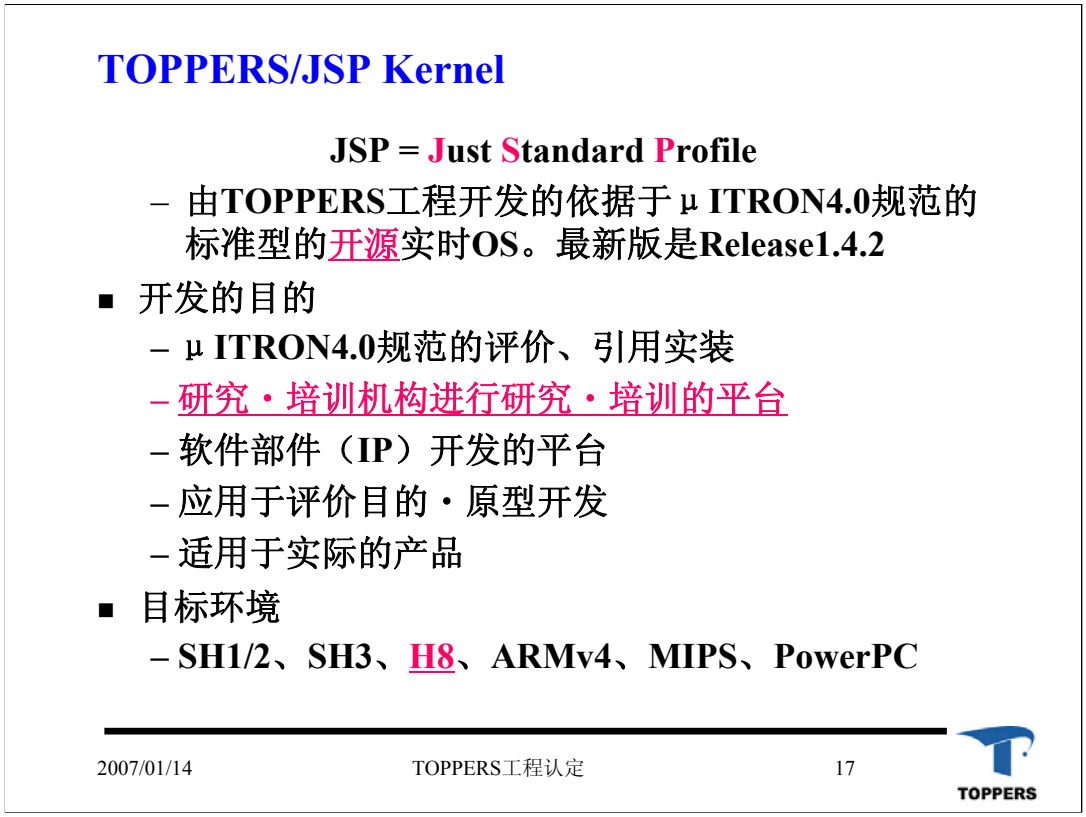



Cygwin等同于TOPPERS/JSP的H8版的开发环境。推荐给想使用开源软件廉价构建开发环境的人。基本可以从Cygwin的主页上下载来进行安装。不过在环境设置以及日文化方面需要一定的技巧。如果购买CD-ROM配套的书籍进行安装的话，会比较容易安装。

*相关书籍参考中文名：

**Cygwin+CygwinJE-Windows下运行的UNIX，佐藤龙一，ASCII**

**Cygwin－Windows下可使用的UNIX环境，川井义治、米田聪，软银出版社**




在本实习中，AKIH8/3069F电路板上运行的RTOS使用TOPPERS/JSP。

TOPPERS/JSP是基于*μITRON4.0规范*的标准流程文档的开源实时OS。TOPPERS/JSP是以下述目的进行开发的。

1）开发的目的是进行*μITRON4.0规范*的评价、引用实装。

2）目标是搭建研究・培训机构进行嵌入式技术研究・培训的平台。

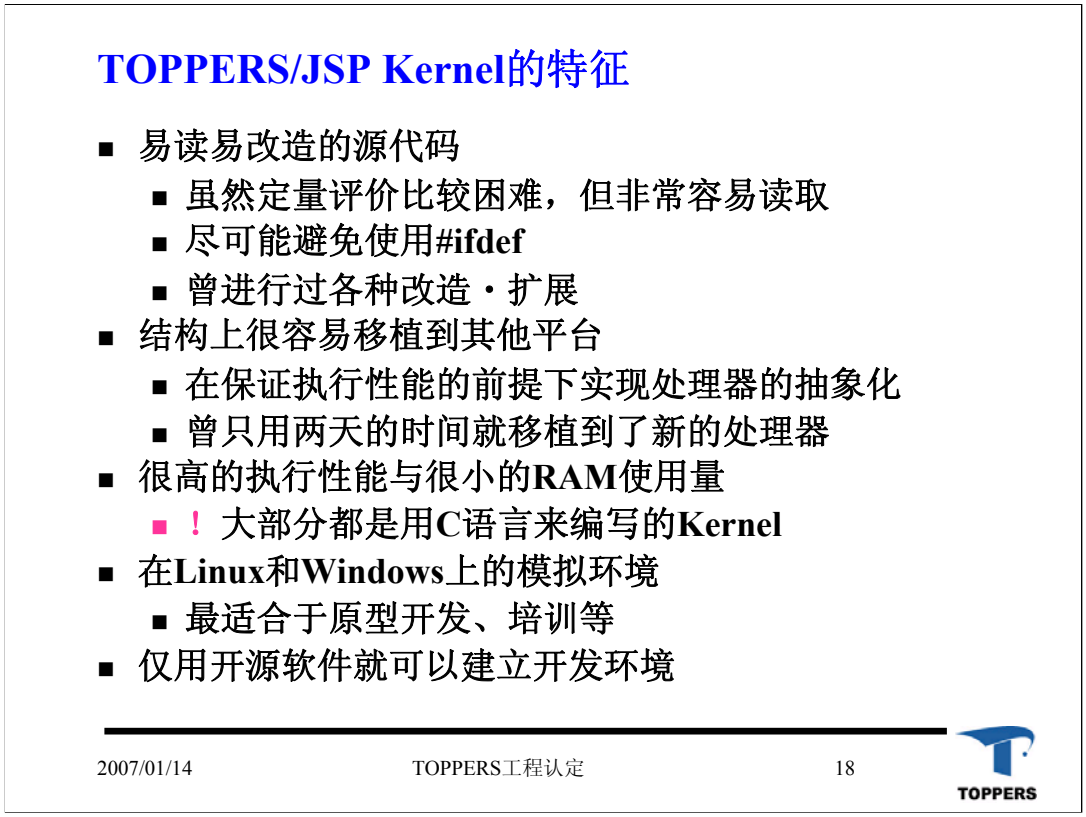
3）目标是搭建开发嵌入式软件的软件部件（IP）的平台。

4）应用于评价目的・原型开发

5）适用于实际的产品

TOPPERS/JSP适用于各种各样的目标环境。本次是以H8的AKIH8/3069F电路板为目标。该目标依赖部分记述在以下源代码目录下。

jsp-1.4.2/config/h8和jsp-1.4.2/config/h8/akih8_3069f




TOPPERS/JSP的特征：

1）易读易改的源代码

虽然定量评价比较困难，但非常容易读取。

尽可能避免使用条件编译语句(#if，#ifdef等)。

曾有过多次改造、扩展。在μITRON4.0规范的功能扩展・确认上也使用了JSP。

2）很容易移植到其他平台

在不降低执行性能的前提下实现了处理器的抽象化。

曾经用两天的时间就移植到了新的处理器上。

3）很高的执行性能与很小的RAM使用量

作为大部分都是用C语言来记述的实时Kernel，它具有惊人的执行性能，而且实现了在运行时占用很少的内存。

4）Linux和Windows上的模拟环境

拥有Linux和Windows上的模拟环境。最适合于原型开发、教育等。

5）仅用开源软件就可以使用开发环境

可以直接使用Linux、Cygwin等开发环境。

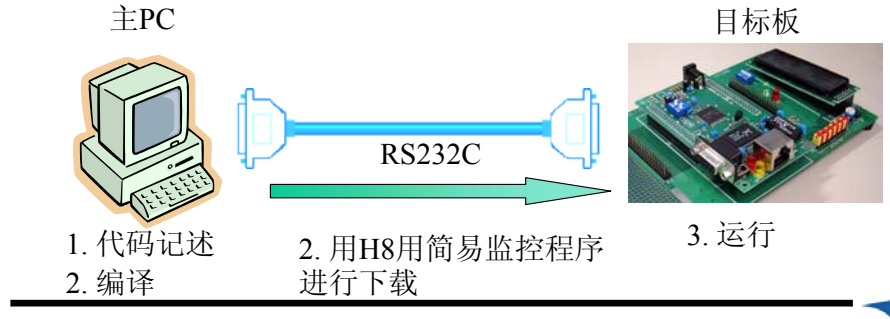
# 基本程序设计

1. 开发环境的构建
2. 样例程序的运行
3. 外围设备程序设计



## 开发环境的构建：开发环境的概要

- 目标板先将H8用简易监控程序写入到内置Flash ROM中，然后用此程序进行启动
- 程序在主PC上使用交叉编译器H8用GCC进行编译
- 使用H8用简易监控程序经由RS232C将编译过的程序拷贝到目标板的RAM上运行






TOPPERS

下面简要说明一下开发环境。

下面简单介绍一下开发环境。

开发完的程序并不是直接写入到AKIH8/3069F电路板上的FLASH ROM中运行，而是先将H8用简易监控程序"H8用简易监控程序"写入到FLASH ROM中。

开发完的程序并不是直接写入到AKIH8/3069F电路板上的FLASH ROM中运行，而是先将H8用简易监控程序"H8用简易监控程序"写入到FLASH ROM中。电源启动后，通过RS232C在PC上的通信软件中显示提示符。

了。此监控程序在插上电源后就会通过RS232C在PC中的通信软件上显示提示符。 创建的程序使用H8用简易监控程序与通信软件的传送功能，下载到AKIH8/3069F板上的SDRAM空间中。下载结束后监视器会恢复到提示符要求状态，所以要用命令使程序在板上运行。

创建的程序要使用H8用简易监控程序和通信软件的传送功能下载到AKIH8/3069F电路板上的SDRAM空间中。

开发人员在PC的开发环境下开发在TOPPERS/JSP下运行的程序，并转换成编译→链接→下载的格式，然后在实机上运行。另外，编译、链接、格式转换是通过make命令实现的。

开发人员在PC的开发环境下开发在TOPPERS/JSP下运行的程序，并转换成编译→链接→下载的格式，然后在实机上运行。另外，编译、链接、格式转换是通过make命令实现的。

开发人员在PC上的开发环境下开发运行在TOPPERS/JSP下的程序，流程为编译→链接→转换为下载格式，然后在实机上运行。另外，编译、链接、格式转换使用的是make命令。

## 开发环境的构建：H8用简易监控程序

- 是用于经由RS232C将用户程序下载到RAM上并进行运行的程序
- 可以从TOPPERS工程的主页上下载
- 对H8/3069F内置Flash ROM的写入次数是有限制的，所以每次都将用户程序写入到内置FlashROM 进行调试并不是太好的方法
- 先将H8简易监控程序写入到内置Flash ROM中，然后将用户程序下载到H8/3069F的内置RAM或外部扩展的RAM上并进行调试
- 也具备显示・变更内存内容的功能
- 详细内容请参照程序目录下的./h8mon/readme.txt




下面介绍一下H8简易监控程序。

该程序可以从TOPPERS工程的主页上下载。AKIH8/3069F板上有用于运行的2MB的内存。可以在此空间运行程序和数据。而且，对CPU内置FLASH ROM的写入次数是有限制的。如果对FLASH ROM进行多次写入，有时可能会发生写入错误。因此，本实习的运行环境是使用H8简易监控程序进行下载运行的环境。关于监视器的详细内容请参照./h8mon/readme.txt。

## 开发环境的构建：监控程序的写入


- 将AKI-H8/3069F上的DIP开关设置为内置Flash ROM写入模式（DIP1:ON、DIP2:ON、DIP3:OFF、DIP4:ON）
- 通过RS232C连接目标板与主机PC，打开目标板的电源(插上电源的一瞬间LED3会亮一下)
- 打开控制台，移动到程序目录的./h8mon/mon3069f下
- 使用h8write.exe将mon3069.mot写入到内置Flash ROM中

```
$ cd  ./h8mon/mon3069f
$ ls
3069.S*  3069.h*  3069.ld*  Makefile*  load3069.mot*  mon3069.mot*
$h8write.exe -3069 -f20 mon3069.mot COM1
H8/3069F is ready!  2002/5/20 Yukio Mituiwa.
writing
ingore record
ingore record
.....
EEPROM Writing is successed.
```

指定连接的COM端口

2007/01/14 TOPPERS工程认定 22

TOPPERS

下面说明一下H8简易监控程序的写入步骤。（本实习中不需要写入）

对FLASH ROM进行写入使用的是AKIH8/3069F电路板上添加的CD-ROM中的h8write.exe程序。

1）在断开电源的状态下，将AKIH8/3069F板上的DIPSW设置为FLASH ROM写入模式。

将DIP-SW设置为1-ON、2-ON、3-OFF、4-ON。

2）连接RS232C电缆，插上目标板的电源。插上电源时LED3会亮一下。

3）通过DOS窗口（或PizzaFactory2、或Cygwin）的命令行移动到./h8mon/mon3069f，并执行以下命令。

> h8write.exe –3069 –f20 mon3069.mot COM1(返回)

该命令的COM1是假定已将RS232C电缆连接到COM1上的输入。已连接到其他端口时请指定此端口。

4）写入结束后显示“EEPROM Writing is successed.”。

## 开发环境的构建：监控程序的运行确认


- 写入结束后关掉目标板的电源
- 将AKI-H8/3069F上的DIP开关设置为普通运行模式（模式5）（DIP1:ON、DIP2:OFF、DIP3:ON、DIP4:OFF）
- 将通信软件(TeraTerm等)的端口设定为速度38400bps、数据8bit、无奇偶校验位、停止位1bit的方式，通过RS232C进行通讯连接
- 打开目标板的电源，确认终端软件上是否输出H8用简易监控程序的启动LOG

输入?

```
1:H8/3069F Monitor v1.12 Copyright (C) 1999-2004 CSE Tomakomai NCT
1:?
H8/3069F Monitor v1.12 Copyright (C) 1999-2004 CSE Tomakomai NCT
-------------------------------------------------------------------------------
ld                  load program        | go[<ad>]      go program
dw[<ad>[<ln>]] dump word                | db[<ad>[<ln>]] dump byte
ew[<ad>]        enter word
fb[<ad>[<ln>[<in>[<ic>[<rp>]]]]]  fill byte
?           help
1:
```

2007/01/14 TOPPERS工程认定 23

TOPPERS

确认监视器的运行：

1）写入结束后请切断目标板的电源。

2）将AKIH8/3069F板的DIP-SW设置为普通运行模式（模式5）。将DIP-SW设置为1-ON、2-OFF、3-ON、4-OFF。

3）在PC上启动通信软件（TeraTerm等），并如下进行串口的设置。

Baud rate:38400

Data:8bit

Parity:none

Stop:1bit

Flow control:XON/XOFF

4）插上目标板的电源，并确认灯头（burner）显示以及是否显示提示符。

5）用？命令显示帮助菜单。

# 基本程序设计

1. 开发环境的构建
2. 样例程序的运行
3. 外围设备程序设计



## 样例程序的运行：sample1

- TOPPERS/JSP的发布包中包含有用于确认Kernel基本动作的样例程序Sample1
- sample1由接收控制台输入的主任务和3个并行执行的任务构成
- 每次并行执行的任务执行一定次数的空循环时，都会显示表示任务处于执行中的消息
- 主任务等待控制台的输入（等待时执行并行执行的任务），并对输入进行相应处理
  - 通过sample1的编译、下载、执行，来学习TOPPERS/JSP Kernel的创建・执行方法




TOPPERS/JSP的发布包中包含有用于确认Kernel的基本运行的样例程序sample1。sample1启动后会启动4个任务。主任务解析控制台的输入，并通过使用服务调用给予其他3个任务运行的指示。这3个任务则从控制台显示运行状态。

为了确认TOPPERS/JSP的运行，请尝试创建此程序并在AKIH8/3069F上运行。

## 样例程序的运行：Kernel库

- 有两种TOPPERS/JSP Kernel的创建方法
  - 同时创建应用程序与Kernel
  - 将Kernel作为库进行创建，并与应用程序进行链接
- 如果同时创建应用程序与Kernel，则需要一定的编译时间和磁盘容量等。所以本次实习采用将Kernel作为库进行创建并与应用程序进行链接的方法
- 在程序目录下创建目录./jsp-1.4.2/OBJ/AKIH8_3069F/libkerne，并按照以下的顺序创建Kernel库（libkernel.a）

```
$ cd jsp-1.4.2/OBJ/AKIH8_3069F
$ mkdir libkernel
$ cd libkernel
$ ../../../configure -C h8 -S akih8_3069f
$ make depend
$ make libkernel.a
```




有两种TOPPERS/JSP的Kernel的创建方法。一种方法是在每次创建应用程序（如sample1）时都重新创建Kernel。另一种方法是先将Kernel进行库化，在创建应用程序时再作为库进行链接。前者在重新创建后Kernel本身也要进行编译、链接，这样在创建上就要浪费一定的时间。而后者则缩短了创建的时间。但后者在变更了Kernel相关的定义时必须重新创建Kernel本身，而在创建应用程序时不会自动进行创建，这样就需要自己来进行管理。

在本次讲座中必须多次重新创建应用程序。所以，还是将Kernel进行库化来进行创建吧！在本次讲座的第1天、第2天、第3、4天中，Kernel的设置条件是不一样的。必须根据需要来重新创建Kernel库（libkernel.a）。

Kernel库在jsp-1.4.2/OBJ/AKIH8_3069F/libkernel中进行创建。创建libkernel目录并移动到该目录下，通过下述命令进行创建。

../../../configure –C h8 –S akih8_3069f

make depend

make libkernel.a

## 样例程序的运行：sample1的创建

- 在程序目录下创建目录./jsp-1.4.2/OBJ/AKIH8_3069F/sample1并执行configure
- 执行configure生成用于创建样例程序的文件
  - Makefile : make文件
  - sample1.c/h : 样例程序的源代码
  - sample1.cfg : 样例程序的配置文件

```
$ cd jsp-1.4.2/OBJ/AKIH8_3069F
$ mkdir sample1
$ cd sample1
$ ../../../configure -C h8 -S akih8_3069f
configure: Generating Makefile from ../sample/Makefile.
configure: Generating sample1.c from ../sample/sample1.c.
configure: Generating sample1.h from ../sample/sample1.h.
configure: Generating sample1.cfg from ../sample/sample1.cfg.
$ls
Makefile sample1.c sample1.cfg sample1.h
```



TOPPERS

在jsp-1.4.2/OBJ/AKIH8_3069F/sample1目录下创建sample1。创建sample1目录并移动到该目录下。执行下面的命令，创建Make文件与源文件。

../../../configure –C h8 –S akih8_3069f

如果执行configure，将会生成用于创建样例程序的文件。

Makefile : Make文件

sample1.c/h : 样例程序的源代码

sample1.cfg : 样例程序的配置文件

## 样例程序的运行：sample1的创建

- **在程序目录下创建目录./jsp-1.4.2/OBJ/AKIH8_3069F/sample1 并执行configure**

```
$ cd jsp-1.4.2/OBJ/AKIH8_3069F
$ mkdir sample1
$ cd sample1
$ ../../../configure -C h8 -S akih8_3069f
```

- **用编辑器打开./jsp-1.4.2/OBJ/AKIH8_3069F/sample1/中创建的Makefile，在第81行的KERNEL_LIB处指定刚刚创建Kernel库的目录**

```
#
#Kernel库（libkernel.a）的目录名
#（Kernel库也为make对象时定义为空）
#
KERNEL_LIB = ../libkernel
```

- **执行make depend和make，进行编译并和Kernel库进行链接，确认生成jsp.srec**




在jsp-1.4.2/OBJ/AKIH8_3069F/sample1目录下创建sample1。创建sample1目录并移动到该目录下。执行下面的命令，创建Make文件和源文件。

../../../configure –C h8 –S akih8_3069f

为了查看Kernel库，改写为在Makefile的第81行的KERNEL_LIB处设置Kernel库的基地址（../libkernel）。通过下述命令进行创建。

make depend

make

## 样例程序的运行：电路板的设置

- 将AKI-H8/3069F上的DIP开关设置为通常运行模式（模式5）（DIP1:ON、DIP2:OFF、DIP3:ON、DIP4:OFF）
- 以速度8400bps、数据8bit、无奇偶校验位、停止位1bit的方式将终端软件(TeraTerm等)连接到连接了RS232C的端口上
- 打开目标板的电源，确认终端软件中是否输出H8用简易监控程序的启动LOG

输入?

```
1:H8/3069F Monitor v1.12 Copyright (C) 1999-2004 CSE Tomakomai NCT
1:?
H8/3069F Monitor v1.12 Copyright (C) 1999-2004 CSE Tomakomai NCT
-------------------------------------------------------------------------------
ld                  load program        | go[<ad>]      go program
dw[<ad>[<ln>]] dump word                | db[<ad>[<ln>]] dump byte
ew[<ad>]        enter word
fb[<ad>[<ln>[<in>[<ic>[<rp>]]]]]  fill byte
?           help
1:
```

2007/01/14 TOPPERS工程认定 29

TOPPERS

ROM监视器的命令：

ld 将摩托罗拉S记录形式的调试模块下载到RAM中。

go [<addr>] 从地址<addr> 开始执行调试模块。如果省略<addr>，则会从指定为复位异常处理向量地址的地址开始执行。

dw [<addr> [<len>]]

从地址<addr>开始输出长度为<len>Word（2个字节）的数据。如果省略<addr>，则从上次输出的数据的下一个数据开始输出256字节。但如果是初次执行，则从内置RAM的开始地址开始输出256字节。

db [<addr> [<len>]]

从地址<addr>开始输出长度为<len>Byte的数据。如果省略<addr>，则从上次输出的数据的下一个数据开始输出256字节。但如果是初次执行，则从内置RAM的开始地址开始输出256字节。

ew [<addr>] 从地址<addr>开始变为Word数据的写入模式。如果省略<addr>，则从上次写入的最后一个地址的下一个地址开始变成写入模式。在写入模式下，不进行任何输入而是直接换行时，不对显示的内存进行写入。而且，只要输入“.”就可以关闭写入模式。

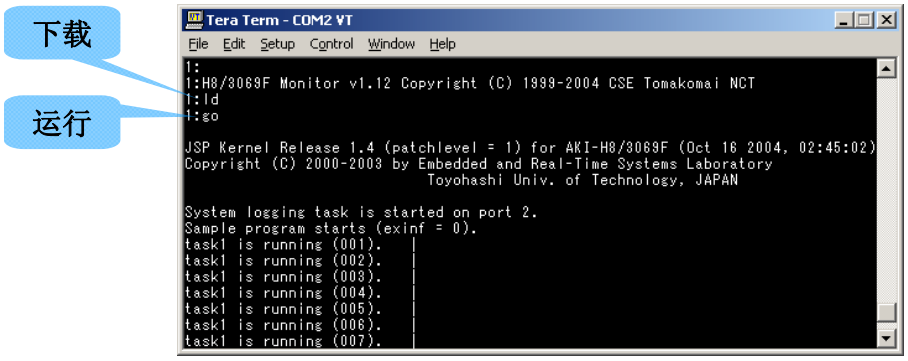
fb [<addr> [<len> [<init> [<inc> [<repeat>]]]]]

从地址<addr> 开始写入长度为<len>Byte的数据。<addr> 的省略值是内置RAM的开始地址，<len> 的省略值是256字节。<init> 是开始写入的数据类型的初始值，<inc>是开始写入的数据类型的增加值，<repeat> 是写入的重复次数。

? 输出简单命令的格式。

## 样例程序的运行：sample1的下载

- 打开终端软件，和H8用简易监控程序连接
- 执行H8用简易监控程序的ld命令，通过终端软件的文件传送功能下载生成的jsp.srec（如果用TeraTerm则可以通过拖放来下载）
- 下载后，通过go命令启动Kernel






TOPPERS

sample1的操作是通过从终端软件输入命令来实现的。

命令表如下所示：

'1' : 以后的命令对TASK1执行。
'2' : 以后的命令对TASK2执行。
'3' : 以后的命令对TASK3执行。
'a' : 用act_tsk启动任务。
'A' : 用can_act取消对任务的启动要求。
'e' : 使任务调用ext_tsk并结束。
't' : 用ter_tsk强制结束任务。
'>' : 将任务的优先级设为HIGH_PRIORITY 。
'=' : 将任务的优先级设为MID_PRIORITY 。
'<' : 将任务的优先级设为LOW_PRIORITY 。
'G' : 用get_pri读取任务的优先级。
's' : 使任务调用slp_tsk并设为唤醒等待。
'S' : 使任务调用tslp_tsk(10秒)并设为唤醒等待。
'w' : 用wup_tsk唤醒任务。
'W' : 用can_wup取消对任务的唤醒要求。
'l' : 用rel_wai强制解除任务的等待。
'u' : 用sus_tsk将任务设为强制等待状态。
'm' : 用rsm_tsk解除任务的强制等待状态。
'M' : 用frsm_tsk强制解除任务的强制等待状态。

## 样例程序的运行：sample1的命令

- 从终端软件输入命令来控制sample1
- 如果用‘s’命令停止task1，显示将会变为task2的执行
- 如果用‘2’命令将对象变为task2并用‘s’命令停止，显示将会变为task3的执行
- 如果用‘3’命令将对象变为task3并用‘s’命令停止，那么3个任务将会停止，显示也停止
- 如果用‘1’命令将对象变为task2并用‘w’命令再次执行，显示将会变为task1的执行
- 也请用其他命令试试




'd' : 使任务调用dly_tsk(10秒)，并等待一定的时间。
'x' : 对任务要求类型0 x 0001的异常处理。
'X' : 对任务要求类型0 x 0002的异常处理。
'y' : 使任务调用dis_tex，并禁止任务异常。
'Y' : 使任务调用ena_tex，并允许任务异常。
'r' : 循环3个优先级的就绪队列。
'c' : 运行周期句柄。
'C' : 停止周期句柄。
'V' : 用vxget_tim读取性能评价用系统时间两次。
'v' : 显示已发行的系统调用（默认）。

# 基本程序设计

1. 开发环境的构建
2. 程序的编译与运行
3. 外围设备程序设计



## 外围设备程序设计

- 基于sample1，创建操作I/O扩展板上的LED和DIP开关的程序
- 学习以下几点
  - 手册的阅读方法
  - TOPPERS/JSP Kernel的系统接口层（SIL）的使用方法
- 完成程序
  - 程序目录的
    ./jsp-1.4.3/OBJ/AKIH8_3069F/device




修改sample1，创建操作I/O扩展板上的LED和DIP-SW的程序。外围设备程序（device.h和device.c）在./jsp-1.4.2/OBJ/AKIH8_3069F/device目录下。使用它们来修改程序。

在TOPPERS/JSP中是使用Tron协会制定的“设备驱动设计方针”中的系统接口层（SIL）来进行外部端口的访问。后面会对LED、DIP-SW等硬件接口以及其访问方法进行说明。

## 外围设备程序设计：LED与DIP

- I/O板上安装了两个LED（LED1与LED2）以及一个4列DIP开关（照片上从右向左依次为DIP1、DIP2、DIP3、DIP4）
- LED与DIP分别是怎样连接到AKI-H8/3069F上的？请参照I/O扩展板的手册和AKI-H8/3069F的手册进行确认




TOPPERS

在I/O扩展板上，安装了两个LED(LED1与LED2)以及一个4列DIP开关（DIP-SW）。LED与DIP-SW通过I/O板侧面的10个引脚的跳线（LCD-LED-DIP）连接到CPU板的CN2A连接器上。

## 外围设备程序：LED与DIP的连接

- LED与DIP分别连接在LCD-LED-DIP上
- LCD-LED-DIP可以通过跳线连接临近的引脚
- 连接了LED与DIP的引脚旁边的引脚连接在CN2A上
- CN2A连接在AKI-H8/3069F的引脚上


| | LED-LCD-DIP | | | CN2A | H8/3069f |
|---|---|---|---|---|---|
| LED1 | 21 | - | 22 | 11 | D6/P46 |
| LED2 | 23 | - | 24 | 12 | D7/P47 |
| DIP1 | 13 | - | 14 | 37 | P50/A16 |
| DIP2 | 15 | - | 16 | 38 | P51/A17 |
| DIP3 | 17 | - | 18 | 39 | P52/A18 |
| DIP4 | 19 | - | 20 | 40 | P53/A19 |



TOPPERS

LED与DIP-SW通过跳线（LCD-LED-DIP）以及连接器CN2A，连接在CPU板上的3069F-CPU的并口上。如果跳线没有连接，则无法实现LED的控制、DIP-SW状态的获取等。

LED、DIP-SW、CPU之间的连接如下所示：

LED1　端口4的6位
LED2　端口4的7位
DIP1　端口5的0位
DIP2　端口5的1位
DIP3　端口5的2位
DIP4　端口5的3位

## 外围设备程序：LED的连接端口

- LED连接在I/O端口4的BIT6与7上
- 此端口在标记上（D6/P46）与其他功能是互斥的
- 参照H8/3069F的手册中I/O端口4的相关说明（P328）
  - 兼用于数据总线的8bit输入输出端口
  - 在模式1～5（扩展模式）下，如果通过总线宽度控制寄存器（ABWCR）将空间0～7都置为8位的访问空间，则会变成8位总线模式，而且端口4变成输入输出端口
  - 在模式7下，端口4是输入输出端口
- 确认处理器的运行模式
  - AKI-H8/3069F中在模式5下使用
- 由于是模式5，所以需要设置总线宽度控制寄存器（ABWCR），并设为8位总线模式
- 根据手册确认ABWCR（P162）
  - 地址0xee020，8位寄存器。如果写入“1”，则设置为8位访问空间



TOPPERS

连接到LED的I/O端口4在各执行模式下的设定是不同的。单片模式：在模式7下固定为输入输出端口。扩展模式：在模式1～5以及8位总线模式下是输入输出端口，在16位总线模式下是数据输入输出端子。总线宽度由总线宽度控制寄存器（ABWCR）进行设置。该寄存器可以设置从空间0到7的总线宽度。此寄存器可以设定空间0到7的存总线宽度线模式或8位总线模式。而由于默认模式是8位总线模式，所以不需要进行设置。端口4可以作为输入输出端口来使用。

连接到LED的I/O端口4的设置是根据运行模式的不同而不同。单片模式：在模式7下固定为输入输出端口。扩展模式：在模式1～5下，8位总线模式时是输入输出端口，16位总线模式时是数据输入输出端子。总线宽度由总线宽度控制寄存器（ABWCR）进行设置。该寄存器可以设置从空间0到7的总线宽度。此寄存器可以设定空间0到7的存总线宽度线模式或8位总线模式。而由于默认模式是8位总线模式，所以不需要进行设置。端口4可以作为输入输出端口来使用。

在模式5时是在8位总线模式下使用外部内存，而默认的就是8位总线模式，所以不需要设定，端口4可以作为输入输出端口进行使用。

本实习中是通过device.c中的initial_led函数对ABWCR寄存器进行初始化。

本实习中是通过device.c中的initial_led函数对ABWCR寄存器进行初始化。

## 外围设备程序：LED的连接端口

- 确认I/O端口4的构成寄存器（P329）
  - P4DDR（0xee003）端口4数据方向寄存器(1为输出端口)
  - P4DR ：0xfffd3端口4数据寄存器
  - P4PCR : 0xee03e端口4输入拉高（pullup）MOS控制寄存器
  - (P4DDR为0，如果设为1，MOS就会变为ON)
- 根据上述内容，如下LED端口的初始化和操作如下
  - 初始化
    - 在ABWCR（0xee020）中写入0xff并设置为8位总线模式
    - 在P4DDR(0xee003)的BIT6和7中写入“1”并设置为输出
  - LED操作
    - 在P4DR的BIT6和7中写入数据




I/O端口4由3个寄存器构成。

P4DDR（端口4数据方向寄存器）

是8位写入专用寄存器，可以对各BIT设置端口4的各端子的输入输出。
关于各BIT，1为输出端口，0为输入端口。

P4DR（端口4数据寄存器）

是8位可读写的寄存器，保存端口4的输出数据。
当端口4作为输出端口时，输出本寄存器的值。
而且，如果读该寄存器，P4DDR为0的BIT将会读取端子的逻辑电平，P4DDR为1的BIT将会读取P4DR的值。

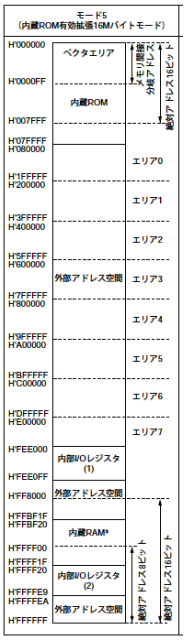
P4PCR（端口4输入上拉MOS控制寄存器）

P4PCR是8位可读写的寄存器，根据各BIT控制内置在端口4中的输入上拉MOS。在模式1～5（扩展模式）的8位总线模式和模式7（单片模式）下，如果在P4DDR清零（输入端口的）的状态下将P4PCR设为1，那么输入上拉MOS将会变为ON。

用初始化函数（initial_led）将ABWCR和P4DDR进行初始化。在LED操作函数(set_led)中，通过P4DR的6、7位的设置实现LED的点亮与熄灭。

# 外围设备程序：模式与地址

- **H8/3069F为模式5时，地址宽度为24位**
- **而外围设备的地址为20位**
  - ✓ **ex. ABWCR : 0xee020**
- **因此，在访问外围设备时，高位4位必须赋值为0xf**
  - ✓ **手册上写的不够周全……**






TOPPERS

在H8/3069F的模式5（H8/3069F版TOPPERS/JSP对应的模式）下，地址宽度为24位。I/O端口等寄存器、内置RAM等配置在0xFEE000地址之后的空间中，所以ABWCR寄存器配置在0xFEE020地址中。

## 外围设备程序：系统接口层

- **TOPPERS/JSP Kernel与通用OS不同，它不存在用户模式与Kernel模式之分。因此可以直接从用户程序操作设备**
- **TOPPERS/JSP Kernel支持一部分系统接口层（SIL）**
- **SIL是根据为提高设备驱动的便携性而制定的ITRON设备驱动设计指南而被提出的**
- **使用SIL的程序要引用s_services.h**
- **本次使用的是访问设备寄存器的函数群**

| | |
|---|---|
| VB sil_reb_mem(VP mem) | **从mem的地址以8位的单位读取数据** |
| void sil_wrb_mem(VP mem、VB data) | **以8位的单位将数据写入到mem地址中** |
| VH sil_reh_mem(VP mem) | **从mem的地址以16位的单位读取数据** |
| void sil_wrh_mem(VP mem、VH data) | **以16位的单位将数据写入到mem地址中** |
| VW sil_rew_mem(VP mem) | **从mem的地址以32位的单位读取数据** |
| void sil_wrw_mem(VP mem、VW data) | **以32位的单位将数据写入到mem地址中** |



TOPPERS

在“设备驱动设计方针”中，将设备驱动分为3个组件。

GDIC（General Device Interface Component）

PDIC（Primitive Device Interface Component）

SIL（System Interface Layer）

在TOPPERS/JSP中，按照与设备的访问接口SIL的规范对设备进行访问。具体来讲，有以下访问函数。

| | |
|---|---|
| VB sil_reb_mem(VP mem); | 从mem的地址以8位的单位读入数据 |
| void sil_wrb_mem(VP mem, VB data); | 以8位的单位将数据写入到mem地址中 |
| VH sil_reh_mem(VP mem); | 从mem的地址以16位的单位读入数据 |
| void sil_wrh_mem(VP mem, VH data); | 以16位的单位将数据写入到mem地址中 |
| VW sil_rew_mem(VP mem); | 从mem的地址以32位的单位读入数据 |
| void sil_wrw_mem(VP mem, VW data); | 以32位的单位将数据写入到mem地址中 |

## 外围设备程序：工程的文件追加

- 在sample1中追加device.h device.c，编写LED操作函数
- 在工程中追加了文件时，为了编译此文件，要在Makefile的UTASK_COBJ中追加.o文件

```
ifdef USE_CXX
    UTASK_CXXOBJS = $(UNAME).o
    UTASK_COBJS =
else
    UTASK_COBJS = $(UNAME).o  device.o
endif
```

支持C++时追加到此处

- 在工程中追加了文件时，为了重新构建依赖关系，必须执行下面的命令

```
$ make realclean
$ make depend
```




首先是在sampe1目录下，使用编译器在sample1目录下创建device.h和device.c的两个源文件。可以是未编辑的文件。在Makefile中的UTASK_COBJS中追加device.o，请确认上述make命令是否能够正确创建。

## 外围设备程序：头文件

- 程序创建以下两个外部函数

```
extern void initial_led(void);
extern void set_led(LED led, CONDITION req);
```

- 将LED与CONDITION作为列举型进行定义

```
typedef enum{ LED1,  LED2 }LED;
typedef enum{ON, OFF }CONDITION;
```




为了从sample1.c读取LED的初始化以及LED操作函数，要定义作为函数的external声明及参数进行使用的enum。这一点在device.h中有记述，所以只要改写成从sample1.c引用device.h就可以了。

## 外围设备程序：sample1.c的变更

- 在sample1.c的主任务程序追加main_task启动时调用函数initial_led() 初始化LED的代码
- 输入“4”时LED1灯亮
- 输入“5”时LED1灯灭
- 输入“6”时LED2灯亮
- 输入“7”时LED2灯灭

```
case '3':
        tskno = 3;
        tskid = TASK3;
        break;
case '4':
        set_led(LED1, ON);
        break;
case '5':
        set_led(LED1, OFF);
        break;
case '6':
        set_led(LED2, ON);
        break;
case '7':
        set_led(LED2, OFF);
        break;
case 'a':
```




修改sample1.c的正文。修改成：在启动main_task时进行设备的初始化，在main_task函数的开头调用initial_led函数。然后，为了对LED进行设置，要在主循环的switch语句中追加case“4”到“7”，并追加对应各处理的LED操作函数。

另外，请在sample1.c的引用文件中追加device.h。

#include “device.h”

## 外围设备程序：程序例（1/2）

- 引用文件、宏定义、全局变量

```
#include <s_services.h>
#include “device.h”
#define ABWCR        0xfee020  /* 总线宽度控制寄存器 */
#define P4DDR        0xfee003  /* 数据方向寄存器 */
#define P4DR         0xffffd3  /* 数据寄存器*/

#define P4DDR_B6     0x40      /* LED1连接端口的BIT */
#define P4DDR_B7     0x80      /* LED2连接端口的BIT  */

unsigned char cled;  /* 保存LED的设置状态 */
```

- 初始化函数

```
void
initial_led(){
        cled = 0x00;
        /* 8位总线模式 */
        sil_wrb_mem((VP)ABWCR、0xff);
        /* P4的BIT6与BIT7为输出端口 */
        sil_wrb_mem((VP)P4DDR, sil_reb_mem((VP)P4DDR) | (P4DDR_B6 | P4DDR_B7));
        /* P4的输出数据初始化(初始状态为熄灯状态) */
        sil_wrb_mem((VP)P4DR, sil_reb_mem((VP)P4DR) & ~(P4DDR_B6|P4DDR_B7));
}
```

2007/01/14 TOPPERS工程认定 43

TOPPERS

请追加以下两个device.c的引用文件。如果追加s_services.h，就可以记述SIL记述的函数。

请追加以下两个device.c的引用文件。如果追加s_services.h，就可以记述SIL记述的函数。

```
#include <s_services.h>
#include “device.h”
```

关于P4DDR端口的初始化步骤，按照上面的叙述应该是在读取P4DDR端口后指定输出BIT并进行重新写入。但P4DDR端口是write only端口，根据端口公司的手册在读取时会读取为1。所以，P4端口都被设定为输出端口。在I/O端口上，P4端口的其他位被设定为LCD用输出。所以在运行上没有问题。关于P4DDR端口的初始值（包括LCD的设定）以下记述方法比较恰当。

关于P4DDR端口的初始化步骤，按照上面的叙述应该是在读取P4DDR端口后指定输出BIT并进行重新写入。但P4DDR端口是write only端口，根据端口公司的手册在读取时会读取为1。所以，P4端口都被设定为输出端口。在I/O端口上，P4端口的其他位被设定为LCD用输出。所以在运行上没有问题。关于P4DDR端口的初始值（包括LCD的设定）以下记述方法比较恰当。

```
#define P4DDR_LCD 0xfc
sil_wrb_mem((VP)P4DDR, (P4DDR_B6|P4DDR_B7|P4DDR_LCD));
```

#define P4DDR_LCD 0xfc

sil_wrb_mem((VP)P4DDR, (P4DDR_B6|P4DDR_B7|P4DDR_LCD));

# 外围设备程序：程序例（2/2）

- **LED连接端口的写入**

```
void
led_out(unsigned char led_data){
      sil_wrb_mem((VP)P4DR,
                        sil_reb_mem((VP)P4DR) & ~(P4DDR_B6 | P4DDR_B7)
                        |(led_data & (P4DDR_B6 | P4DDR_B7)));
}
```

- **输出函数**

```
void
set_led(LED led、CONDITION req){
    unsigned char rled = cled;
    switch(led){
      case LED1:
            if (req == OFF)
              rled &= ~P4DDR_B6;
            else
              rled |= P4DDR_B6;
            break;
      case LED2:
            if (req == OFF)
              rled &= ~P4DDR_B7;
            else
              rled |= P4DDR_B7;
            break;
     default:
            break;
    }
    if (cled != rled) {
            cled = rled;
            led_out(cled);
    }
}
```

## 外围设备程序：DIP开关

- 变更为：按下“8”使DIP开关的状态输出到控制台
- DIP开关连接在端口5上
  - DIP1 : P50/ DIP2 : P51/ DIP3 : P52/ DIP4 : P53
- 端口5构成寄存器
  - P5DDR : 0xfee004 : 数据方向寄存器 (1为输出端口)
  - P5DR : 0xffffd4 : 端口5数据寄存器
  - P5PCR : 0xfee03f : 端口5输入Pull up
- 在模式5下，端口5通过P5DDR的设置可成为地址总线或输入端口
  - 如果将P5DDR设为0x00置为输入模式，端口就是有效的
  - Pull up置为ON
    DIP开关ON时为“0”，OFF时为“1”




DIP-SW的1到4引脚连接在端口5的0位到3位上。端口5为模式1到4时，是4位的地址输出端子。为模式7时，是4位的输入端口。为模式5（本设置）时，是4位的输出端子与4位的输入端口两用，可以通过P5DDR寄存器的设置来进行切换。

P5DDR（端口5数据方向寄存器）

是8位的只写寄存器，可以根据各BIT设置端口5的各端子的输入输出。并于各BIT，1为输出端口或地址输出端子，0为输入端口。BIT7～4是保留位，固定为1。

P5DR（端口5数据寄存器）

是8位的可读写寄存器，保存端口5的输出数据。

端口5作为输出端口时，输出本寄存器的值。

而且，如果读取此寄存器，P5DDR为0的BIT将会被读取端子的逻辑电平，P5DDR为1的BIT将会被读取P5DR的值。

BIT7～4是保留位，固定为1。

P5PCR（端口5输入上拉MOS控制寄存器）

P5PCR是8位的可读写寄存器，根据各BIT控制内置在端口5中的输入上拉MOS。

在模式5（扩展模式）及模式7（单片模式）下，如果在P5DDR清零的（输入端口的）状态下将P5PCR设为1，那么输入上拉MOS则为ON。

BIT7～4是保留位，固定为1。

在本硬件中需进行上拉（pullup）。

## 外围设备程序：接口（1/2）

- 程序创建以下两个外部函数

```
extern void initial_switch(void);
extern CONDITION get_switch(SWITCH sw);
```

- 将SWITCH与CONDITION作为列举型进行定义

```
typedef enum{
    ON,
    OFF
}CONDITION;
```

```
typedef enum{
    SW1,
    SW2,
    SW3,
    SW4
}SWITCH;
```




在device.h引用文件中进行开关用的定义。

initial_switch函数进行开关设备的初始化。

get_switch函数获取DIP-SW的状态。如果指定4种SWITC类型中的一种作为参数，那么作为返回值会返回CONDIITION类型的状态。

## 外围设备程序：接口（2/2）

- 在sample1.c的主任务函数main_task中追加调用initial_switch()函数的代码，初始化DIP
- 如果将vmsk_log函数的第一参数变更为LOG_UPTO(LOG_NOTICE)，那么DIP的各BIT的状态将会输出到控制台

```
if (get_switch(SW1) == ON) {
  syslog(LOG_NOTICE, "SW1 is ON");
}
else {
  syslog(LOG_NOTICE, "SW1 is OFF");
}

if (get_switch(SW2) == ON) {
  syslog(LOG_NOTICE, "SW2 is ON");
}
else {
  syslog(LOG_NOTICE, "SW2 is OFF");
}
……..
```




在sample1.c中修改main_task函数，并追加initial_switch函数进行开关的初始化。修改为使用get_switch函数来LOG显示开关的状态。

## 外围设备程序：程序例（1/2）

- 宏定义

```
#define P5DDR       0xfee004   /* 端口5方向寄存器 */
#define P5DR        0xffffd4   /* 端口5数据寄存器 */
#define P5PCR       0xfee03f   /* 端口5Pull Up寄存器 */
#define P5DR_B0     0x01       /* DIP1连接端口的BIT */
#define P5DR_B1     0x02       /* DIP2连接端口的BIT */
#define P5DR_B2     0x04       /* DIP3连接端口的BIT */
#define P5DR_B3     0x08       /* DIP4连接端口的BIT */

#define P5PCR_PULLUP 0x0f
```

- 初始化函数

```
void
initial_switch(){
        sil_wrb_mem((VP)P5DDR, 0x00);  /* 输入端口 */
        sil_wrb_mem((VP)P5PCR, P5PCR_PULLUP); /* PULL UP */
}
```




DIP-SW对应端口的宏定义记载在device.c文件中。为了作为输入来获取DIP-SW，必须进行上拉设置。其中的P5PCR_PULLUP定义是用于定义上拉对象的4个BIT的。

initial_switch函数实现用于DIP-SW查看的设置。通过以下函数对端口5的4位进行输入设置。

```
sil_wrb_mem((VP)P5DDR, 0x00);
```

通过以下函数对DIP-SW的对象BIT进行上拉。

```
sil_wrb_mem((VP)P5PCR,P5PCR_PULLUP);
```

# 外围设备程序：开关状态的获取

- 端口状态的获取

```
unsigned char
sense_switch(void){
  return(sil_reb_mem((VP)(P5DR))&(P5DR_B0|P5DR_B1|P5DR_B2|P5DR_B3));
}
```

- 开关状态的提取

```
CONDITION
get_switch(SWITCH sw){
    unsigned char    key;
    CONDITION result = OFF;

    key = sense_switch();
    switch(sw){
    case SW1:
        if((key & P5DR_B0) == 0)
            result = ON;
        break;
    case SW2:
        if((key & P5DR_B1) == 0)
            result = ON;
        break;
    case SW3:
        if((key & P5DR_B2) == 0)
            result = ON;
        break;
    case SW4:
        if((key & P5DR_B3) == 0)
            result = ON;
        break;
    default:
        break;
    }
    return result;
}
```




sense_switch函数读入DIP-SW对应的端口5的内容。关于B3到B0对应的BIT，0时为ON，1时为OFF。

get_switch函数使用sense_switch函数来判定参数指定的DIP-SW的当前状态。作为返回值，在ON状态下返回ON，在OFF状态下返回OFF。

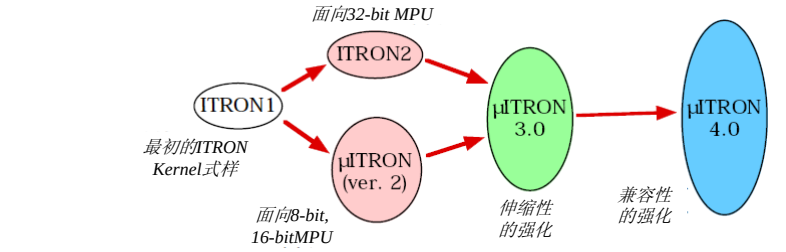
# ITRON规范的概要



# ITRON规范

- ITRON工程
  - TRON工程的一个子工程
- ITRON规范
  - ITRON工程中制定的实时性Kernel规范
  - 制定并公布迄今为止4代ITRON的规范
  - 代表了当今实时性Kernel技术高度的规范

➡ 嵌入式系统开发的开放平台






TOPPERS

TRON（The Real-time Operating system Nucleus）是1984年由东京大学的坂村健博士以构建理想的计算机架构为目的而开创的工程。在产业界及各大学的共同努力下，以实现全新的计算机体系为目标。TRON工程的推进一直都是以几个实际的领域为对象，以各子工程为单位来实行具体规范的标准化。其中包括ITRON（嵌入式系统用适时OS规范及与其相关的规范）、BTRON（PC、工作站用OS规范及与其相关的规范）、CTRON（以通信控制、信息处理为目的的OS界面规范）、TRON电子机器HMI（各种电子机器的人机界面的标准方针）等子工程。

最初的ITRON规范是在1987年诞生的“ITRON1规范”。基于此ITRON1规范又开发了多个实时性Kernel并进行应用，这在规范的适应性的检证上起到了非常重要的作用。此后，又出现了具有适用于8～16位的MCU的功能的μITRON规范(Ver.2.0)以及适用于大规模的32位处理器的ITRON2.0规范，它们都是在1989年进行公开的。由于μITRON规范在受限的源MCU中也能发挥它实用的性能，所以它被安装在很多的MCU中，同时也应用在很多的嵌入式机器中。随着μITRON规范被广泛应用于各领域，需求也越来越明确，而且μITRON规范开始应用于32位的MCU，这就要求它要具备适用于8位到32位的各种规模的MCU的可伸缩性。于是，在1993年重新制定了μITRON3.0规范并进行了公开。

此后，经历了ITRON TCP/IP API规范、JTRON规范的制定，μITRON3.0规范就有必要重新考虑。于是，在1999年又公开了μITRON4.0规范。μITRON4.0规范是嵌入式系统开发OPEN化的基础。

## ITRON规范的特征


- 实现了OS的小型轻便性
  - 也适用于单片机
- 规范容易理解
  - 重视技术人员培训的标准化
- 完全开放的标准规范
  - 安装不发生使用版权费
- 可以/已经安装于各种处理器
  - 从8位单片机到32位RISC微处理器
  - 可以容易移植到不同的处理器上
- 已使用于多种设备中
  - 是嵌入式系统领域内使用范围最广的OS规范
- 得到很多制造商/开发商的支持




ITRON规范的特征：

1）实现了OS的小型轻便性，即使是单片微处理器也能够发挥它本身的性能。

2）规范很容易理解。这一点是重视技术人员培训的标准化来制定规范的结果。

3）是完全开放的标准规范，不需要支付使用费就可以进行安装。

4）从8位单片微处理器到32位RISC处理器，对各种处理器都可以进行安装，而且已经进行了安装。所以，可以很容易移植到不同的处理器上。

5）曾在多个嵌入式设备上使用。

6）得到多个制造商/厂家的支持。

## ITRON规范的Kernel的功能(μITRON4.0规范)

- **Kernel的功能**
  - 任务管理功能
  - 任务附属同步功能
  - 任务异常处理功能
  - 同步・通信功能
  - 扩展同步・通信功能
  - 内存池功能
  - 时间管理功能
  - 系统状态管理功能
  - 中断管理功能
  - 服务调用管理功能

- **服务调用数**
  - 全功能版
    - **服务调用: 166**
    - **静态API : 21**
  - 标准版
    - **服务调用: 70**
    - **静态API : 11**
  - 汽车控制用版
    - **服务调用: 43**
    - **静态API : 8**
  - 最小版

! 没有规定用于I/O操作的功能

2007/01/14 TOPPERS工程认定 53

下面介绍一下*μITRON4.0规范*。

Kernel的功能：

1）任务管理功能：管理多任务功能的基本功能

2）任务附属同步功能：为任务功能准备的基本同步功能

3）任务异常处理功能：任务中发生异常事件时在任务语境下进行处理的功能

4）同步・通信功能：通过与任务相独立的对象来实现任务间的同步・通信的功能

5）扩展同步・通信功能：通过与任务相独立的对象来实现高度任务间的同步・通信的功能

6）内存池功能：通过软件进行动态内存管理的功能

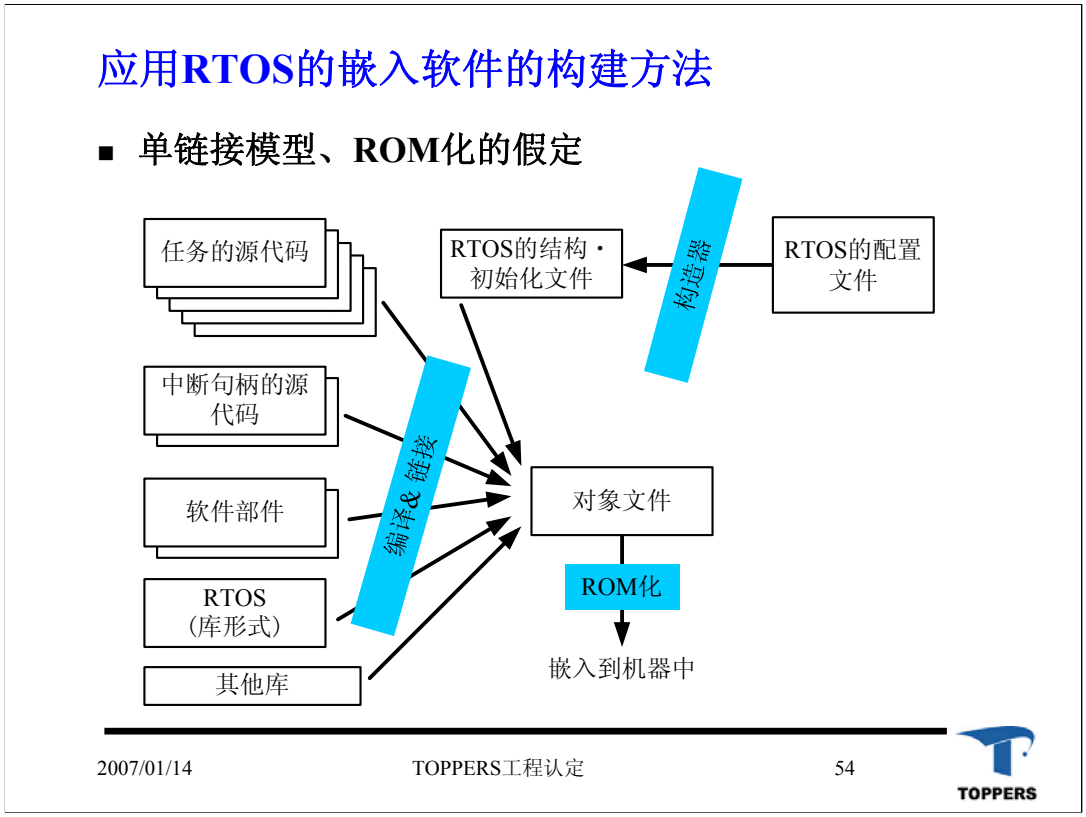
7）时间管理功能：进行依赖于时间的处理的功能

8）系统状态管理功能：变更・查看系统状态的功能

9）中断管理功能：对由外部中断启动的中断句柄及中断服务例程进行管理的功能

10）服务管理功能：对扩展服务调用进行定义及调用的功能

在*μITRON4.0规范*中，为了更好地实现可伸缩性，在全功能版与最小版之间又制定了标准版及汽车控制用版。




下面介绍一下使用了RTOS的嵌入式软件的构建方法。基本上是作为1个链接模型对假设ROM化的方法进行介绍。

用户程序是作为任务的源代码进行创建的。将中断相关的处理作为中断句柄的源代码进行记述。在这些程序中添加软件部件、库化的RTOS以及其他的库，将从RTOS的配置文件生成的RTOS的构成・初始化文件进行编译・链接，并在进行对象文件化、ROM化后嵌入到机器中。

## RTOS的配置

- 决定RTOS的配置、初始状态等的步骤
- 什么样的构成能够变更取决于各RTOS。如：
  - 使用的功能/不使用的功能
  - 各对象（如任务、信号量）等的最大值
  - 任务优先级的等级数
  - 各对象的生成・定义信息
    （静态生成对象时）
- 配置机制
  - 根据配置改变RTOS的代码的方法（使用条件编译等）
  - 将配置结果生成一个（或多个）源代码的方法




将决定RTOS的构成、初始状态的步骤称为RTOS的结构。是什么样的构成以及是否为初始状态取决于各RTOS。比如说，在变更作为RTOS使用/不使用的功能时，或设置对象的最大数时，或设置优先级的等级数时，或静态生成对象时，都可以设置生成定义信息。

按照结构在RTOS的代码中添加变更并将构成结果生成一个或多个源代码的机制叫做结构机制。

## μITRON4.0规范的配置

- 静态生成Kernel对象
  - 将对象生成信息、配置文件中的记述方法进行标准化（静态API）
- 静态API的例子

```
CRE_TSK(tskid, {tskatr, exinf, task, itskpri, stksz, stk});
ATT_ISR({intatr, exinf, intno, isr});
```

- 构造器解析静态API，生成Kernel的内部信息




在μITRON4.0规范中，可以静态生成Kernel对象。创建配置文件，通过用静态API在此文件中记述对象的生成信息，构造器就可以自动生成源代码。

静态API的一个例子是CRE_TSK，它指定静态生成的任务对象的属性。

## 多任务机制

- 调度方式
  - 在μITRON4.0中，支持基于任务优先级的多任务调度机制
  - 优先级相同的任务以FCFS（First Come First Served）方式进行调度。此种调度方式具备改变优先级相同的任务的执行顺序的功能。利用这一点也可以实现优先级相同的任务的循环调度
  - 优先级的分配基本是静态的，但也具备动态变更优先级的功能
- 任务状态
  - 支持以下7种状态：

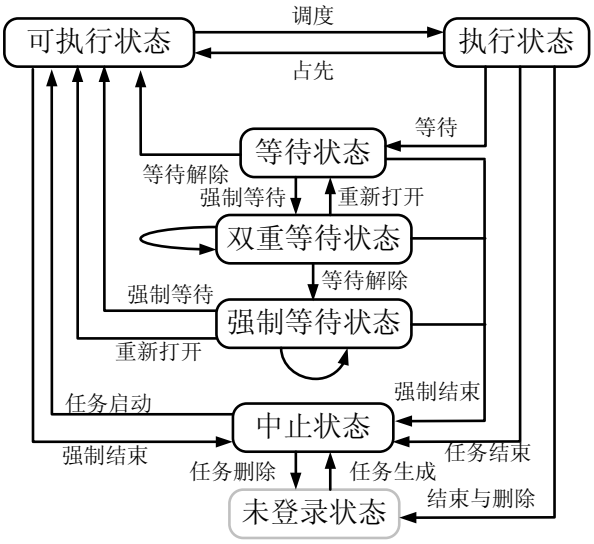
    执行、可执行、中止、等待、强制等待、双重等待、未登录




下面介绍一下*μITRON4.0规范*的多任务机制。任务的调度方式是基于优先级的调度。优先级相同的任务排队等待，以FCFS（FirstCome First Served）的方式来执行。即使是优先级相同，也可以通过循环排队顺序的功能（rot_rdq服务调用）来改变执行的顺序。而且，通过优先级变更的功能（chg_pri服务调用）还可以实现优先级的动态变更。

任务状态支持执行、可执行、中止、等待、强制等待、双重等待、未登录7种状态。

## 任务的状态迁移




- **执行状态（RUNNING）**
- **可执行状态（READY）**
- **等待状态（WAITING）**
  - 任务自行进入等待状态
- **强制等待状态（SUSPENDED）**
  - 由其他任务使之进入等待状态
- **双重等待状态（WAITTING-SUSPENDED）**
  - 上面两种状态重叠的状态
- **中止状态（DORMANT）**
- **未登录状态 (NON-EXISTENT)**



TOPPERS

任务的7种状态：

1）执行状态（RUNNING）

当前任务正在执行的状态。不过，在执行非任务语境时，基本是将开始执行非任务语境前执行的任务作为执行状态的任务。

2）可执行状态（READY）

虽然此任务可以执行，但由于有一个比此任务优先级更高的任务正在执行，所以此任务现在还不能执行的状态。

3）等待状态（WAITING）

在条件齐备前，通过调用任务自己中断执行的服务调用来中断执行的状态。

4）强制等待状态（SUSPENDED）

被其他任务强制中断执行的状态。但在μITRON4.0规范中，也可以实现任务自己成为强制等待状态。

5）双重等待状态（WAITING-SUSPENDED）

等待状态与强制等待状态重叠的状态。如果将处于等待状态的任务转移到强制等待状态，就会出现双重等待状态。

6）中止状态（DORMANT）

任务还没有启动，或已执行完的状态。

7）未登录状态（NON-EXISTENT）

任务还没有生成，或还没有登录到删除后的系统的虚拟状态。

# ITRON API解说

1. 任务相关的API
2. 通过程序进行确认
3. 同步附属功能
4. 通过程序确认同步附属功能



## ITRON规范的服务・调用（API）

- 服务・调用名称的方针

  例）act_tsk

  表示操作对象,这里是任务（task）

  表示操作方法,这里是启动（activate）

  –如果服务・调用成功了，就会作为返回值返回E_OK

- 可以从中断句柄（非任务语境）进行调用服务・调用
  使用以“i”开头的名称来进行区分

  例）iact_tsk

- 存在不能从中断句柄进行调用的API

  –进入等待状态的API




服务・调用是用于从任务等应用・程序向适时・Kernel发送要求的接口，一般被称为API。一直到μITRON3.0规范它都被称为“系统・调用”。ITRON规范的服务・调用名称有命名方针，前半部分表示操作方法，后半部分表示操作对象。在服务调用“act_tsk”中，act是表示启动的“activate”的缩写，tsk是任务的缩写。如果服务・调用正常结束，作为返回值将会返回E_OK。

从中断句柄（非任务语境）进行调用的服务・调用使用以“i”开头的名称来进行区分。由于非任务语境无等待状态，所以无法调用要求等待状态的服务・调用。

## 静态API的详细说明

| | | |
|---|---|---|
| CRE_TSK | 任务生成（静态API） | 【标准】 |
| cre_tsk | 任务生成 | 【标准外】 |

【静态API】

CRE_TSK（ID tskid、{ATR tskatr, VP_INT exinf, FP task,
PRI itskpri, SIZE stksz, VP stk}）

【C语言API】

ER ercd = cre_tsk(ID tskid, T_CTSK *pk_ctsk)

【参数】

| | | |
|---|---|---|
| ID | tskid | 生成对象的任务的ID号 |
| T_CTSK | *pk_ctsk | 包含任务生成信息的包 |
| ATR | tskatr | 任务属性（编写语言、初始状态） |
| VP_INT | exinf | 任务的扩展信息（给任务的参数） |
| FP | task | 任务的启动地址 |
| PRI | itskpri | 任务的启动优先级 |
| SIZE | stksz | 任务栈空间的大小（字节数） |
| VP | stk | 任务栈空间的首地址 |




在*μITRON4.0规范*中，对静态生成构成系统的对象的方法（构造器）进行了标准化，规定了静态API。静态API记述在配置・文件中，是表示对象的生成内容的API。C语言API的服务・调用的名称是以小写字母进行记述的，但静态API的名称则是以大写字母进行记述。参数与C语言API是基本相同的，在C语言API中作为结构体进行指定的参数在静态API中也是以同样的形式进行记述。

# 任务管理功能

## 直接操作任务状态的功能

- 任务管理功能的服务・调用

| | |
|---|---|
| cre_tsk | 任务的生成 |
| del_tsk | 任务的删除 |
| act_tsk、iact_tsk | 任务的启动 |
| can_act | 任务的启动要求的无效化 |
| ext_tsk | 本任务的结束 |
| ter_tsk | 任务的强制结束 |
| chg_pri | 任务优先级的变更 |
| get_pri | 任务优先级的参照 |
| ref_tsk | 任务的状态参照 |

- 任务的启动要求（act_tsk）的排队等待及其无效化（can_act）




任务管理功能是用于直接操作・查看任务状态的功能。其中包括生成・删除任务的功能、启动・结束任务的功能、取消启动任务要求的功能、改变任务优先级的功能、查看任务状态的功能。任务是以ID号来进行识别的对象，其ID号叫做任务ID。

为了控制执行顺序，任务具有基本优先级与当前优先级两种优先级。如果只说任务的优先级，则是指任务的当前优先级。任务的基本优先级在任务启动时被初始化为任务启动时的优先级。对任务的启动要求排队等待。也就是说，如果想再一次启动已经启动过的任务，就会留下此任务的启动要求，在此任务结束后会自动进行再启动。不过，通过指定启动代码来启动任务的服务调用（sta_tsk）的启动要求是不进行排队等待的。

在任务结束时，除了互斥体的锁定解除以外，适时・Kernel不会释放任务已获得的资源（信号量资源、内存块等）。这些资源必须由应用程序来释放。

## 任务附属同步功能

### 直接操作任务实现同步的功能

- 任务附属同步功能的服务・调用

| | |
|---|---|
| slp_tsk、tslp_tsk | 使本任务进入唤醒等待状态 |
| wup_tsk、iwup_tsk | 别的任务的唤醒 |
| can_wup | 唤醒要求的无效化 |
| rel_wai、irel_wai | 任务等待状态的强制解除 |
| sus_tsk | 使任务进入强制等待 |
| rsm_tsk、frsm_tsk | 从强制等待中释放任务 |
| dly_tsk | 本任务的延迟 |

- 任务唤醒要求（wup_tsk）的排队等待及其无效化（can_wup）
- 带有超时的唤醒等待（tslp_tsk）与本任务的延迟（dly_tsk）




任务附属同步功能是通过直接操作任务的状态来取得同步的功能。其中包括使任务处于唤醒等待的功能以及从此处唤醒的功能、取消任务唤醒要求的功能、强制解除任务等待状态的功能、使任务进入强制等待状态的功能以及从此处重启的功能、任务自己延迟执行的功能。

对任务的唤醒功能排队等待。也就是说，如果想唤醒非唤醒等待状态的任务，就会留下此任务唤醒要求的记录。过后要使此任务进入唤醒等待状态时，此任务不会进入唤醒等待状态。

对任务的强制等待要求是嵌套的。也就是说，如果想使已处于强制等待状态（包括双重等待状态）的任务再次进入强制等待状态，就会留下使此任务进入强制等待状态的记录。过后想解除此任务的强制等待状态（包括双重等待状态）进行重启时，强制等待状态不会解除，此任务也不会重启。

## 时间管理功能：概要

- 系统时间管理
  - 管理系统时间（Kernel管理的当前时间）的功能
- 时间事件句柄
  - 经过一定的时间进行调用的例程
  - 具备以下Timer句柄
    - 周期句柄（周期性调用）
    - Alarm句柄（在指定的时间调用）
    - Overrun句柄（Overrun时调用）
- 用于实现时间同步的其他功能
  - 任务的延迟（使任务等待一定的时间）
  - 进入等待状态的服务・调用的超时功能




时间管理功能是为了实现依赖于时间的处理的功能。其中包括系统时间管理、周期句柄、Alarm句柄、Overrun句柄的各种功能。将周期句柄、Alarm句柄、Overrun・句柄统称为时间事件句柄。

・系统时间管理

系统时间管理功能是用于操作系统时间的功能。其中包括设置・查看系统时间的功能、提供时间控制装置（stick）来更新系统时间的功能。

系统时间在系统初始化时初始化为0。此后，在每次应用程序调用提供时间控制装置（stick）的服务调用（isig_tim）时都会更新一次。调用isig_tim时更新的系统时间的程序取决于实装的情况。但在Kernel内部也可能具备更新系统时间的功能，所以此时不需要支持isig_tim。

・时间事件句柄

周期句柄是在一定周期内启动的时间事件句柄。在周期句柄功能中包括生成・删除周期句柄的功能、开始・停止周期句柄运作的功能、查看周期句柄状态的功能。周期句柄是以ID号进行识别的对象，其ID号叫做周期句柄ID。周期句柄的启动周期与启动相位可以在生成周期句柄时对各周期句柄进行设置。在操作周期句柄时，Kernel根据设置的启动周期与启动相位来确定下一次启动周期句柄的时间。在生成周期句柄时的时间加上启动相位的时间就是下一次启动的时间。一旦确定了启动周期句柄的时间，就会将此周期句柄的扩展信息(exinf)作为参数来启动周期句柄。在这种情况下，有时会用启动周期句柄的时间加上启动周期的时间来重新确定下一次启动的时间。周期句柄的启动相位原则上不能长于启动周期。如果指定了时间长于启动周期的启动相位，这种情况下的运行取决于实装的情况。获取周期句柄是否在运行的状态。如果周期句柄没在运行，即使到了启动周期句柄的时间也不会启动周期句柄，而只是确定下一次启动的时间。如果调用开始运行周期句柄的服务调用(sta_cyc)，就会使周期句柄进入运行状态，必要的话会重新确定下一次启动周期句柄的时间。如果调用停止周期句柄运行的服务调用(stp_cyc)，就会使周期句柄进入不运行状态。在生成周期句柄后处于哪种状态可以通过周期句柄的属性来确定。

## 时间管理功能：系统调用

- 用于系统时间管理的服务・调用

| | |
|---|---|
| set_tim | 系统时钟的设置 |
| get_tim | 系统时钟的读取 |
| isig_tim | time tic供给 |

- 用于周期句柄操作的服务・调用

| | |
|---|---|
| cre_cyc | 周期句柄生成 |
| del_cyc | 周期句柄的删除 |
| sta_cyc | 周期句柄的启动 |
| stp_cyc | 周期句柄停止 |




Alarm句柄是在指定的时间启动的时间事件句柄。在Alarm句柄功能中包括生成・删除Alarm句柄的功能、开始・停止Alarm句柄运行的功能、查看Alarm句柄状态的功能。Alarm句柄是以ID号进行识别的对象，其ID号叫做Alarm句柄ID。

启动Alarm句柄的时间（叫做Alarm句柄的启动时间）可以对每个Alarm句柄进行设置。一旦到了Alarm句柄的启动时间，就会将此Alarm句柄的扩展信息(exinf)作为参数来启动Alarm句柄。并不是在Alarm句柄生成后就设置Alarm句柄的启动时间，此时Alarm句柄是处于停止状态的。如果调用开始运行Alarm句柄的服务调用(sta_arm)，就会将Alarm句柄的启动时间设置为根据调用服务调用的时间开始指定的相对时间后。如果调用停止Alarm句柄运行的服务调用(stp_arm)，就会解除Alarm句柄的启动时间并停止Alarm句柄的运行。

Overrun句柄是在任务超过了设置的时间使用处理器时启动的时间事件句柄。在Overrun句柄功能中包括定义Overrun句柄的功能、开始・停止Overrun句柄运行的功能、查看Overrun句柄状态的功能。

作为Overrun句柄的启动条件而进行设置的时间（叫做任务的上限处理器时间）可以对每个任务进行设置。Kernel对设置了上限处理器时间的任务管理其上限处理器时间设置后任务使用的累积处理器时间（叫做任务的使用处理器时间），如果任务的使用处理器时间超过上限处理器时间就启动Overrun句柄。由于系统内可以定义的Overrun句柄只有一个，所以会将引起启动的任务ID号(tsikid)与扩展信息(exinf)作为参数传递给此Overrun句柄。任务生成后不是马上设置任务的上限处理器时间。如果调用开始运行Overrun句柄的服务调用(sta_ovr)，就会解除指定任务的上限处理器时间的设置。在任务引起Overrun句柄的启动或任务结束时也会解除此任务上限处理器时间的设置。在任务使用的处理器时间中至少要包含此任务（包括任务异常处理例程）以及由此任务调用的服务调用的执行时间。相反地，不包含其他任务（包括任务异常处理例程）以及由其他任务调用的服务调用的执行时间。

## 系统状态管理功能

### 参照/变更系统状态的功能

- 系统状态管理功能的服务・调用

| | |
|---|---|
| rot_rdq, irot_rdq | 任务执行顺序的循环 |
| get_tid, iget_tid | 执行状态的任务ID的获取 |
| loc_cpu, iloc_cpu | 进入CPU锁定状态 |
| unl_cpu_iunl_cpu | CPU锁定状态的解除 |
| dis_dsp | 调度禁止 |
| ena_dsp | 调度许可 |
| sns_ctx | 是否是非任务语境执行中? |
| sns_loc | 是否是CPU锁定状态? |
| snd_dsp | 是否是调度禁止状态? |
| sns_dpn | 是否是调度保留状态? |




系统状态管理功能是变更・查看系统状态的功能。其中包括循环任务优先级的功能、查看处于执行状态的任务ID的功能、进入・解除CPU锁定状态的功能、禁止・解除任务调度的功能、查看语境以及系统状态等功能。

## 中断处理与中断管理功能

- 中断处理（外部中断）
  - 中断句柄可以写在应用程序中
  - 如果发生中断，就会经由Kernel内的出入口处理来调用中断句柄
  - 可以在中断句柄中调用服务・调用来进行任务的启动/唤醒等
  - 中断句柄处理优先于任务
    - 调度延迟
- 中断管理功能的服务・调用

| | |
|---|---|
| def_int | 中断句柄的定义 |

  - 此外，还有禁止/许可中断等功能




中断管理功能是用于管理由外部中断启动的中断句柄以及中断服务例程的功能。其中包括定义中断句柄的功能、生成・删除中断服务例程的功能、查看中断服务例程状态的功能、禁止・许可中断的功能、变更・查看中断屏蔽的功能。中断服务例程是以ID号进行识别的对象，其ID号叫做中断服务例程ID。

中断管理功能中具有下列数据类型。

INHNO　中断句柄号

INTNO　中断号

IXXXX　中断屏蔽

中断屏蔽的数据类型中的XXXX部分在实装定义上是对应于目标处理器的架构的适当的字符串。中断句柄的记述形式取决于实装的情况。在启动中断服务例程时，将此中断服务例程的扩展信息(exinf)作为参数进行传递。中断服务例程的C语言记述形式如下所示：

```
void isr (VP_INT exinf)
{
        中断服务例程本体
}
```

# ITRON API解说

1. 任务相关的API
2. 通过程序进行确认
3. 同步附属功能
4. 同步附属功能程序设计



## 通过Sample进行确认

**以sample1为题材来学习JSP Kernel的**
**配置方法、调度以及任务相关的API的使用方法**

- **移到程序目录的./jsp-1.4.2/OBJ/AKIH8_3069F/sample1下**
- **运行make realclean并返回到初始状态，然后确认文件**
- **运行make depend，确认运行了什么，生成了什么**

```
$ cd ./jsp-1.4.2/OBJ/AKIH8_3069F/sample1
$ make realclean
$ ls
Makefile*  sample1.c  sample1.cfg  sample1.h
$ make depend
h8300-hms-gcc -E –I. -I../../../include  -I../../../config/h8/akih8_3069f -I../../../conf
ig/h8  -DCPU_CLOCK=20000000 -DLABEL_ASM -DAKI_MONITOR
-x c-header sample1.cfg > tmpfile1
../../../cfg/cfg -s tmpfile1 -c -obj -cpu h8 -system akih8_3069f
rm -f tmpfile1
Generating Makefile.depend.
$ls
Makefile*       kernel_cfg.c  kernel_id.h     sample1.c    sample1.h
Makefile.depend  kernel_chk.c  kernel_obj.dat  sample1.cfg vector.S
```



TOPPERS

以sample1为题材来确认使用JSP Kernel进行创建的步骤。在sample1目录下执行以下命令。

| | |
|---|---|
| make realclean | 清除创建文件 |
| ls | 确认初始状态的目录 |
| make depend | 构建依赖关系、运行Kernel关系的构造器 |
| ls | 确认文件 |

新生成以下文件：

Makefile.depend、kernel_cfg.c、kernel_id.h、kernel_chk.c、kernel_obj.dat vector.S

Makefile.depend是记述Makefile生成的源码与引用文件的依赖关系的文件。其他文件是由构造器生成的文件。

在H8的JSP1.4.1的实装中，需要手动将中断向量的指定追加到vector table中，但从JSP1.4.2开始已变更为使用PERL脚本自动生成了。vector.s是保存自动生成的vector table的汇编程序・源码。

## 配置：生成文件

- 用-E选项运行H8用的GCC。-E选项只运行预处理器
- 通过预处理器将sample1.cfg转换成tmpfile1

```
h8300-hms-gcc -E –I. –I../../../include  -I../../../config/h8/akih8_3069f –I../../../conf
ig/h8  -DCPU_CLOCK=20000000 -DLABEL_ASM -DAKI_MONITOR
-x c-header sample1.cfg > tmpfile1
```

- 对生成的tmpfile1运行构造器（cfg）

```
../../../cfg/cfg -s tmpfile1 -c -obj -cpu h8 -system akih8_3069f
```

- 构造器生成下述文件。其中与配置相关的是kernel_cfg.c与kenrel_id.h两个文件

```
kernel_cfg.c  kernel_id.h kernel_chk.c  kernel_obj.dat
```

2007/01/14 TOPPERS工程认定 70

TOPPERS

在TOPPERS/JSP的结构下进行哪些处理呢？首先，通过预处理器将配置文件（sample1.cfg）转换成tmpfile1文件。预处理器将引用文件中的定义转换成实数，转换成容易构成的数据形式。以tmpfile1文件为对象，构造器（jsp-1.4.2/cfg/cfg）生成用C语言可以进行编译的引用文件（kernel_id.h）及源文件（kerel_cfg.c）。kernel_id.h是Kernel对象的定义用引用文件，kernel_cfg.c是Kernel对象的数据空间定义、表、初始化函数的源文件。

## 配置：sample1的配置文件

- 由于构造器无法读入C语言的记述，所以要定义_MACRO_ONLY。C语言的记述写在此宏中
- 在sample1的配置文件（sample1.cfg）中，用静态API生成了四个任务及一个周期句柄
- 在最后三行引用的是系统时钟驱动、串行接口驱动、系统LOG任务的配置文件

```
#define _MACRO_ONLY
#include "sample1.h"
INCLUDE("¥"sample1.h¥"");
CRE_TSK(TASK1, { TA_HLNG, (VP_INT) 1, task, MID_PRIORITY, STACK_SIZE, NULL });
CRE_TSK(TASK2, { TA_HLNG, (VP_INT) 2, task, MID_PRIORITY, STACK_SIZE, NULL });
CRE_TSK(TASK3, { TA_HLNG, (VP_INT) 3, task, MID_PRIORITY, STACK_SIZE, NULL });
CRE_TSK(MAIN_TASK, { TA_HLNG|TA_ACT, 0, main_task, MAIN_PRIORITY,
                                                STACK_SIZE, NULL });
….
CRE_CYC(CYCHDR1, { TA_HLNG, 0, cyclic_handler, 2000, 0 });

#include "../../../systask/timer.cfg"
#include "../../../systask/serial.cfg"
#include "../../../systask/logtask.cfg"
```

2007/01/14 TOPPERS工程认定 71

TOPPERS

下面以sample1.cfg为对象来说明一下配置文件。由于构造器不以C语言中的Profile这样的函数记述为解析对象，所以当配置文件的引用文件中存在这样的记述时，删除掉就可以正确转换。在配置文件的开头定义_MACRO_ONLY，通过预处理器删除#ifndef定义的C语言记述。在sample1.cfg中，定义了四个任务及一个周期句柄。

sample1.cfg中最后的引用文件是系统中使用的驱动、Kernel的扩展功能用配置文件。

| | |
|---|---|
| timer.cfg | 系统定时器的驱动的配置文件 |
| serial.cfg | TOPPERS/JSP中使用的串行驱动的配置文件 |
| logtask.cfg | syslog用配置文件 |

## 配置：kernel_id.h

- **kernel_id.h中定义了对象的ID自动分配结果**
- **在用户程序中，先用宏记述API使用的ID，然后引用此文件**

- kernel_id.h

```
#define CYCHDR1          1
#define LOGTASK          5
#define MAIN_TASK        4
#define SERIAL_RCV_SEM1  1
#define SERIAL_RCV_SEM2  3
#define SERIAL_SND_SEM1  2
#define SERIAL_SND_SEM2  4
#define TASK1            1
#define TASK2            2
#define TASK3            3
```

- 配置文件

```
CRE_TSK(TASK1, { TA_HLNG, …
CRE_TSK(TASK2, { TA_HLNG, …
CRE_TSK(TASK3, { TA_HLNG, …
CRE_TSK(MAIN_TASK, { TA_HLNG …
```

- 用户程序

```
/*
 * 任务的启动
 */
act_tsk(TASK1);
act_tsk(TASK2);
act_tsk(TASK3);
```




kernel_id.h定义了Kernel对象的ID自动分配结果。在上述例子中，对配置文件中的CRE_TSK静态API声明分配数值。

在TASK1中是1，在TASK2中是2，在TASK3中是3，在MAIN_TASK中是4。LOGTASK的5是用logtask.cfg定义的记录任务的任务ID。也对周期句柄的ID“CYCHDR1”分配了数值。关于这些分配，通过引用kernel_id.h，也可以在用户程序中实现作为实数进行编译。

## 配置：kernel_cfg.c

- Kernel构成･初始化文件
- 由静态API生成的信息创建的Kernel内部数据、初始化例程
- 和用户程序同样进行编译･链接

```
#define TNUM_TSKID 5
const ID _kernel_tmax_tskid = (TMIN_TSKID + TNUM_TSKID - 1);
static __STK_UNIT __stack_TASK1[__TCOUNT_STK_UNIT(1024)];
static __STK_UNIT __stack_TASK2[__TCOUNT_STK_UNIT(1024)];
static __STK_UNIT __stack_TASK3[__TCOUNT_STK_UNIT(1024)];
static __STK_UNIT __stack_MAIN_TASK[__TCOUNT_STK_UNIT(1024)];
static __STK_UNIT __stack_LOGTASK[__TCOUNT_STK_UNIT(LOGTASK_STACK_SIZE)];
const TINIB _kernel_tinib_table[TNUM_TSKID] = {
        {0, (VP_INT)(( VP_INT ) 1), (FP)(task), INT_PRIORITY(10), …},
        {0, (VP_INT)(( VP_INT ) 2), (FP)(task), INT_PRIORITY(10), …},
        {0, (VP_INT)(( VP_INT ) 3), (FP)(task), INT_PRIORITY(10), …},
        {0x00u | 0x02u, (VP_INT)(0)、(FP)(main_task), INT_PRIORITY(5), …},
        {0x00u | 0x02u, (VP_INT)(( VP_INT ) 2), (FP)(logtask), ..}
};
const ID _kernel_torder_table[TNUM_TSKID] = {1,2,3,4,5};
TCB _kernel_tcb_table[TNUM_TSKID];
```



TOPPERS

kernel_cfg.c是Kernel构成・初始化文件。生成了从配置文件生成的Kernel内部数据、Kernel对象的数据定义、初始化例程等记述。此文件与用户程序同时进行编译・链接。

在上述例子中，_kernel_tmax_tskid是任务ID最大数的常量数据，_stack_TASK1是并行执行的任务(task)的任务ID1（TASK1）用栈空间，_kernel_tinib_table是TCB（任务・控制・块）的初始化表，_kernel_tcb_table是TCB的数据空间。

## 静态API：执行状态下的任务生成

- 作成目录./jsp-1.4.2/OBJ/AKIH8_3069F/task_api/，执行configure，作成sample1的工程
- 在中止状态下生成并行执行的TASK1、TASK2、TASK3，通过在主任务中执行act_tsk()进入可执行状态
- 如下变更sample1
  - 不在主任务中调用act_tsk()
  - 变更静态API，在可执行状态下生成三个任务
- 尝试增加一个并行执行的任务




尝试改造sample1。创建task_api目录，生成sample1工程。如下修改此工程的程序。

1）在当前程序中，在中止状态下生成TASK1、TASK2、TASK3三个并行执行的任务，并通过主任务中的act_tsk()服务调用进入执行状态。修改主任务与配置文件，修改为：在主任务中也可以不执行act_tsk()，在执行状态下生成三个任务。

2）将并行执行的任务数从三增加到四。

## 静态API：执行状态下的任务生成

- 对CRE_TSK的任务属性指定TA_ACT，使对象任务进入可执行状态，在任务生成时进行的处理的基础上，进行任务启动时的处理，任务状态为可执行状态

```
CRE_TSK(TASK1, { TA_HLNG|TA_ACT, (VP_INT) 1,
                 task, MID_PRIORITY, STACK_SIZE, NULL });
CRE_TSK(TASK2, { TA_HLNG|TA_ACT, (VP_INT) 2,
                 task, MID_PRIORITY, STACK_SIZE, NULL });
CRE_TSK(TASK3, { TA_HLNG|TA_ACT, (VP_INT) 3,
                 task, MID_PRIORITY, STACK_SIZE, NULL });
```



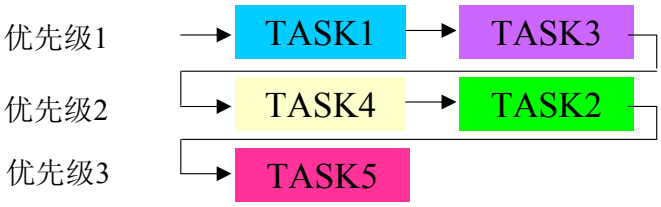

为了自动执行并行执行的任务，在配置文件的CRE_TSK静态API属性中添加TA_ACT就可以在生成时自动执行。

为了增加并行执行的任务数，在配置文件中添加TASK4的CRE_TSK声明。但仅仅是这样还不能实现从主任务对TASK4的执行要求。还需要修改sample1.c文件。将message的空间增加到四个，在main_task函数中追加act_tsk(TASK4);，在switch语句中追加下述内容。

```
case ‘4’:
tskno = 4;
tskid = TASK4;
break;
```

## 调度：就绪队列

- 用于管理处于可执行状态的任务的OS内的队列
- 各优先级的队列按优先级的顺序进行排列
  - 优先级相同的队列按启动的顺序进行排列
- OS执行就绪队列排头的任务（进入执行状态）






TOPPERS

就绪队列是RTOS内用于管理处于可执行状态的任务的队列。就绪队列中各优先级的队列按优先级的顺序进行排列。

在进行调度时，RTOS检查就绪队列，执行队列排头的任务。

## 调度:

- **sample1中并行执行的三个任务是按优先级MID_PRIORITY（10）的初始优先级来执行**
- **运行sample1并从控制台输入命令，然后确认下述内容**
  - **启动时MID_PRIORITY就绪队列的状态**
  - **用“<”命令将TASK1的优先级降为LOW_PRIORITY（11）时MID_PRIORITY就绪队列的状态**
  - **在此状态下用“r”命令将就绪队列倒过来后的状态**
  - **用“=”命令将TASK1的优先级返回到MID_PRIORITY时的MID_PRIORITY就绪队列的状态**




在执行sample1后命令对象变为TASK1的状态下输入“<”命令，并将TASK1的优先级设置为LOW_PRIORITY(11)（“<”命令发行chi_pri(tskid,LOW_PRIORITY)服务调用）。LOG显示将会发生怎样的变化呢？

此后，输入“r”命令来循环就绪队列（“r”命令对HIGH_PRIORITY、MID_PRIORITY、LOW_PRIORITY三个任务等级发行rot_rdq服务调用并要求循环就绪队列）。LOG显示将会发生怎样的变化呢？

输入“=”命令将TASK1优先级设置为MID_PRIORITY(10)。（“=”命令发行chi_pri(tskid, MID_PRIORITY)服务调用）。LOG显示将会发生怎样的变化呢？

【请将课题的任务优先状态填写到下面并完成图。】

・启动时的MID_PRIORITY的就绪队列的状态

MID_PRIORITY → _____ _____ _____

・将TASK1的优先级降为LOW_PRIORITY(11)时

MID_PRIORITY → _____ _____

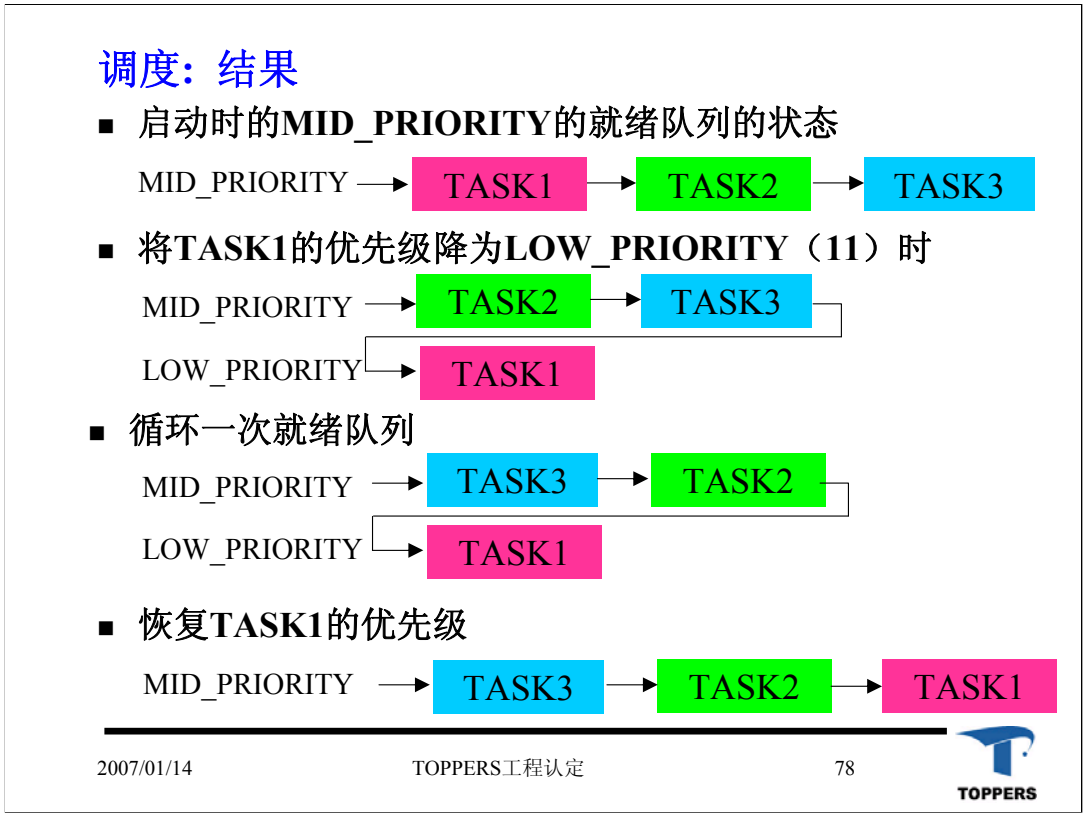
LOW_PRIORITY _____

・循环一次就绪队列

MID_PRIORITY → _____ _____

LOW_PRIORITY _____

・返回TASK1的优先级




检证结果。由于sample1启动后马上运行TASK1，所以会对task1进行LOG显示。

1）如果输入“<”命令，TASK1的优先级就会变为LOW_PRIORITY，所以MID_PRIORITY的就绪队列排头的TASK2就会变为执行状态。因此，LOG显示也变为task2。

2）如果输入“r”命令，MID_PRIORITY的就绪队列就会进行循环，就绪队列的排头变成TASK3。所以LOG显示也变为task3。

3）如果输入“=”命令，TASK1的优先级就会变为MID_PRIORITY，在MID_PRIORITY的就绪队列的最后会与TASK1相连。所以，LOG显示不变。此时如果二次输入“r”命令，TASK1就会变为就绪队列的排头，显示也变为task1。

## 启动要求的排队等待

- 对任务的启动要求、唤醒要求、强制等待要求排队等待
- 用sample1来确认启动要求的排队
  - 用“a”命令可以向TASK1发几次启动要求API act_tsk（）？
  - 在启动要求排队等待的状态下，如果用结束“t”命令（ter_tsk（）的发行）或“e”命令（ext_tsk（）的发行）中止TASK1，TASK1会进行怎样的处理呢?




sample1启动后，输入“a”命令（对相应的任务发行act_tsk服务调用）。即使TASK1正在执行，第一次也不会出现错误，但第二次就会发生E_QOVR错误，不予接收。不管是TOPPERS/JSP还是正在启动，启动要求都要排队等待一次。在这种状态下，即使用“t”命令（ter_tsk服务调用的发行）或“e”命令（ext_tsk服务调用的发行）中止TASK1，也会自动返回到唤醒状态。

# ITRON API解说

1. 任务相关的API
2. 根据程序进行确认
3. 同步附属功能
4. 同步附属功能程序设计



## 任务间的通信与同步

- 任务间的通信・同步的必要性
  - 为了实现任务间的协调运行，需要进行任务间的数据通信（=任务间通信）
  - 在进行任务间的通信时，任务之间需要统一运行的时机（=任务间同步）
  - 即使是在多个任务争夺同一资源时，也需要任务间的同步
- 任务间通信・同步的类型

  如果任务是虚拟化的处理器，那么任务间的通信・同步就是虚拟化的处理器间通信・同步

共享内存通信 **or**消息通信



## 共享内存通信

在可以从多个任务进行访问的内存空间中
存放要传送的数据（共享数据）
- 通过C语言的全局变量来实现

- 在读写共享数据时，需要使用RTOS的功能进行互斥控制
  - 中断/调度禁止
  - 信号量/互斥体
    - ⚠ 注意死锁和优先级颠倒
  - 任务优先级的变更
- 在想快速处理共享数据时使用通知事件发生的功能
  - 任务的启动/唤醒
  - 事件标志/Condition Variable（条件变量）




下面介绍一下在实际的系统设计中构建任务间通信功能的技术。通信方式大致可以分为“共享内存通信”与“消息通信”两种。

在“共享内存通信”中，任务间的通信数据保存在全局内存空间或共享内存上。通过在这些内存空间中读写通信数据来实现任务间的通信。由于各任务、中断非同步运行，所以可能一个任务还没有写完数据别的任务就进行了读取，这样就会接收错误的信息。那么，通过使用信号量等进行互斥控制就可以防止此类事件的发生。

而且，在事件发生了变化或需要马上对应时，就要使用任务的启动、唤醒以及事件标志等使接收方的任务立即运行。

## 消息通信

使用RTOS具备的消息通信的功能
来进行任务间消息的通信

- 消息通信功能的类型
  - 有连接还是没有连接、单向还是双向
    1对1（可以发送的任务是固定的）还是n对1还是n对n
  - 通信对方的指定方法
    - 指定通信对象还是指定对方任务还是……
  - 同步消息通信还是非同步消息通信
  - （在非同步通信中）缓存中没有消息时的处理
    - 等待（拦截）还是不等待（不拦截）
  - （在非同步通信中）缓存已满时的处理
    - 等待（拦截）还是不等待（不拦截）还是覆盖



TOPPERS

在“消息通信”中，使用邮箱或数据队列等消息功能进行构建。消息的内容在进行系统设计时要仔细斟酌。如果互斥控制能够正确进行的话，那么也可以自己创建消息机构。请从整个系统出发创建最合适的方法。

## 消息通信功能的类型（续）

- 消息是固定长度还是可变长度（任意长度）
- （为可变长度时）是否有包的单位
- 传递指针还是拷贝指针
- （在非同步通信中）消息的排队顺序
  - FIFO顺序还是优先级顺序
- 等待接收/发送信息的任务的排队顺序
  - FIFO顺序还是优先级顺序
- 其他特殊功能
  - 多终点传输法/广播传输方式
  - 状态消息（OSEK/VDK规范的unqueued message）

**需要很好地理解RTOS具备的消息通信功能（通常具备多个功能）的性质**



## 任务间通信・同步的设计

- 共享内存vs.消息通信
  - 基本上是用其中一种方式能够实现的用另一种方式也能实现
    - 通常共享内存通信的方式系统开销比较小
    - 在系统验证方面，通信密度大且能够进行在RTOS层进行监视的消息通信比较容易处理
    - 比如很容易进行问题的划分
  - 根据状况，会有某个有优势/方便的情况
    - 信息的流程为1:n（包括广播传输方式）或需要信息的时机不确定等时候，共享内存通信的方式比较有优势
    - 需要信息排队等待的时候消息通信的方式比较方便




试着比较一下用共享内存通信的方式与用消息通信的方式设计的通信同步。基本上是用其中一种方式能够实现的用另一种方式也能实现。但它们各自都有擅长与不擅长的方面。如果要比较通信中产生的系统开销的话，只要设计正确还是共享内存通信的方式系统开销比较小。而且，在想设计成使用中断例序等非任务语境向任务传送要求时，也是共享内存通信的方式比较有优势。但是，如果是想使用RTOS的功能来构建易于监视系统中的通信的系统，则是消息通信的方式比较容易构建。

根据通信同步的设计要求，有时也会出现使用一种方式很难实现的情况。例如，在信息的流程为1对n或信息的传送时机不确定的时候，共享内存通信的方式比较容易进行系统设计。但是，在需要信息（参数）排队等待后再进行传送的系统中，则是消息通信的方式比较容易进行系统设计。

必须考虑系统的要求来选择合适的方式。但是，如果在一个系统中混杂了多种通信方法，那么由于系统过于复杂，扩展以及障碍对应等就会很困难，所以此时请慎重考虑。

## ITRON的同步・通信功能

－在μITRON4.0中，为了实现任务间同步・通信而配备了下述功能

- （主要）用于共享内存通信的功能
  - 信号量（互斥控制等）
  - 事件标志（事件通知）
  - 互斥体（互斥控制）
- 用于消息通信的功能
  - 数据队列（同步・非同步、1Word的消息）
  - 邮箱（非同步、接收发送指针）
  - 消息缓存（同步・非同步、可变长消息）
  - 会合（同步双向通信、可变长消息）




在μITRON中，为了实现任务间同步・通信而配备了下述功能。

・用于共享内存通信中通信、互斥控制的功能

-信号量　CRE_SEM、cre_sem、acre_sem 、del_sem、sig_sem、isem_sig、wai_sem pol_sem、twai_sem、ref_sem

-事件标志　CRE_FLG、cre_flg、acre_flg、del_flg、set_flg、iset_flg、clr_flg、wai_flg pol_flg、twai_flg、ref_flg

-互斥体　CRE_MTX、cre_mtx、acre_mtx、del_mtx、loc_mtx、ploc_mtx、tloc_mtx unl_mtx、ref_mtx

・用于消息通信的功能

-数据队列　CRE_DTQ、cre_dtq、acre_dtq、del_dtq、snd_dtq、psnd_dtq、ipsnd_dtq tsnd_dtq、fsnd_dtq、ifsnd_dtq、rcv_dtq、prcv_dtq、trcv_dtq、ref_dtq

-邮箱　CRE_MBX、cre_mbx、acre_mbx、deLmbx、snd_mbx、rcv_mbx、prcv_dtq trcv_dtq、ref_mbx

-消息缓冲器

CRE_MBF、cre_mbf、acre_mbf、del_mbf、snd_mbf、psnd_mbf、tsnd_mbf rcv_mbf、prcv_mbf、trcv_mbf、ref_mbf

-Rendezvous　CRE_POR、cre_por、acre_por、del_por、cal_por、tcal_por、acp_por pacp_por、tacp_por、fwd_por、rpl_rdv、ref_por、ref_rdv

标准版中有信号量、事件标志、数据队列和邮箱。

## ITRON同步・通信功能的比较

| | 信号量 | 事件标志 | 邮箱 | 数据队列 |
|---|---|---|---|---|
| 功能 | 互斥控制・同步 | 事件通知 | 同步・通信 | 同步・通信 |
| 信息量 | 少 | 中 | 多 | 多 |
| 系统开销 | 小 | 小 | 中～大 | 大 |
| 排队等待的对象 | 资源等待任务 | 事件等待任务 | 接收任务<br>消息 | 发送任务<br>接收任务<br>消息 |
| 服务调用 | CRE_SEM, sig_sem, wai_sem, twai_sem, pol_sem | CRE_FLG, set_flg, clr_flg, wai_flg, pol_flg, twai_flg | CRE_MBX, snd_mbx, rcv_mbx, prcv_mbx, trcv_mbx | CRE_DTQ, snd_dtq, psnd_dtq, tsnd_dtq, fsnd_dtq, rcv_dtq, prcv_dtq, trcv_dtq |

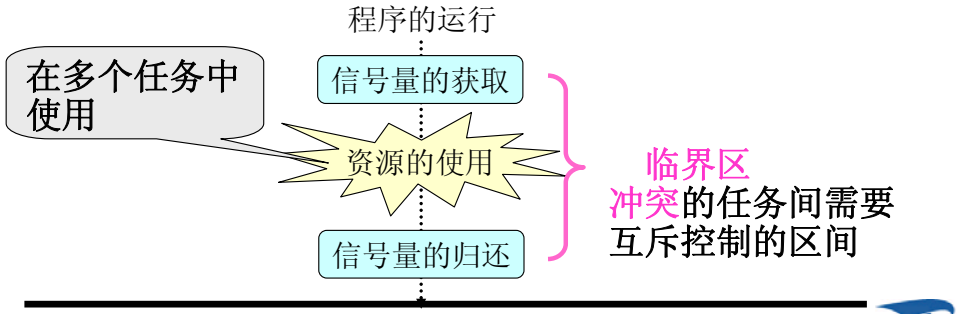


TOPPERS

进行系统设计时，如果一个系统中混杂了多种通信・同步功能，系统就会特别复杂。所以，除非是有特殊原因，否则还是避免此类现象比较好。为了尽量保持单一的系统，我们推荐选择通信・同步功能。

## 信号量（Semaphore）

- 通过将未使用的资源的有无以及数量等作为信号量资源来管理，实现互斥控制
- 使用资源前获取信号量资源，使用完资源后归还信号量资源
- 如果资源已全部使用，在获取信号量资源的时刻进入等待状态






信号量是通过用计数器表示未使用的资源的有无以及数量等来实现使用资源时的互斥控制、同步的对象。释放资源时信号量的计数器加1，获取资源时信号量的计数器减1。

信号量由此计数器和等待获取资源的任务等待队列构成。如果重复获取资源的计数器变为0，那么由于没有未使用的资源，所以要求获取资源的任务就会在信号量资源的等待队列中排队，进入等待状态。

可以对信号量设置资源的初始值以及防止资源归还过多的最大资源数。在归还超过最大资源数的资源时会发生错误。信号量ID是用于识别各信号量的ID。

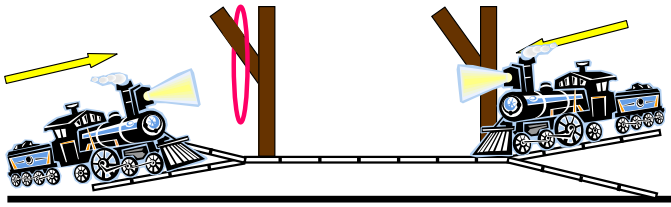
## 信号量（Semaphore）：系统调用&语源

- 系统调用

| | |
|---|---|
| cre_sem | 信号量的生成 |
| del_sem | 信号量的删除 |
| sig_sem, isig_sem | 信号量资源的归还 |
| wai_sem, pol_sem, twai_sem | 信号量资源获取 |

- 语源

铁道的单轨区间内用于控制进入的“圈”
进入单轨区间时，获取这个圈（信号量）



圈 ：信号量
列车 ：任务
单轨区间：资源

2007/01/14 TOPPERS工程认定 89

TOPPERS

*μITRON4.0规范*的信号量的服务・调用如下所示：

1．信号量的生成

CRE_SEM(ID semid, {ATR sematr, UINT isemcnt, UINT maxsem});

| | |
|---|---|
| ID semid | 信号量ID号 |
| ATR sematr | 信号量属性(TA_FIFO \| TA_TPRI) |
| UINT isemcnt | 信号量资源数的初始值 |
| UNIT maxsem | 信号量的最大资源数 |

2．信号量资源的归还

ER ercd = sig_sem(ID semid); ER ercd = iseg_sem(ID semid);

| | |
|---|---|
| ID semid | 资源归还对象的信号量号 |

3．信号量资源的获取

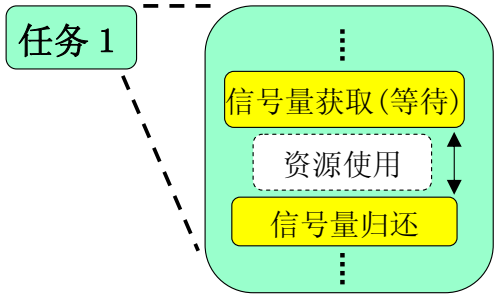
ER ercd = wai_sem(ID semid); ER ercd = pol_sem(ID semid);
ER ercd = twai_sem(ID semid, TMO tmout);

| | |
|---|---|
| ID semid | 资源获取对象的信号量号 |
| TMO tmout | 超时指定（仅限于twai_sem） |

## 信号量（Semaphore）：使用方法

- 以数值（信号量的资源数）的形式管理是否有没使用的资源及数量，由此实现互斥控制
- 系统调用

| | |
|---|---|
| cre_sem | 信号量的生成 |
| del_sem | 信号量的删除 |
| sig_sem, isig_sem | 信号量资源的归还 |
| wai_sem, pol_sem, twai_sem | 信号量资源的获取 |




不用担心其他任务会使用此区间

应保证不会被竞争任务抢走的此区间叫做“临界・区”



TOPPERS

*μITRON4.0规范*的信号量的服务・调用如下所示：

1．信号量的生成

CRE_SEM(ID semid, {ATR sematr, UINT isemcnt, UINT maxsem});

| | |
|---|---|
| ID semid | 信号量ID号 |
| ATR sematr | 信号量属性(TA_FIFO \| TA_TPRI) |
| UINT isemcnt | 信号量资源数的初始值 |
| UNIT maxsem | 信号量的最大资源数 |

2．信号量资源的归还

ER ercd = sig_sem(ID semid); ER ercd = iseg_sem(ID semid);

| | |
|---|---|
| ID semid | 资源归还对象的信号量号 |

3．信号量资源的获取

ER ercd = wai_sem(ID semid); ER ercd = pol_sem(ID semid);

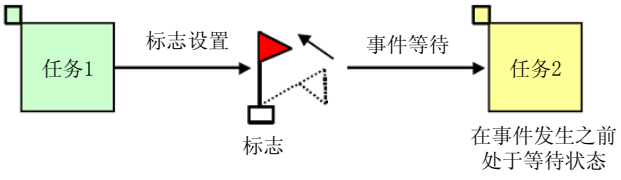
ER ercd = twai_sem(ID semid, TMO tmout);

| | |
|---|---|
| ID semid | 资源获取对象的信号量号 |
| TMO tmout | 超时指定（仅限于twai_sem） |

## 事件标志（EventFlag）

- 通过在任务间传送事件的发生来实现同步的机制。一个事件标志由多个标志构成
- 通过**wai_flg**，可以实现对是设置了所有的指定的标志还是设置了一部分的等待
- 系统调用

| | |
|---|---|
| cre_flg | 事件标志的生成 |
| del_flg | 事件标志的删除 |
| set_flg, iset_flg | 事件标志的设置 |
| clr_flg | 事件标志的清除 |
| wai_flg, pol_flg, twai_flg | 事件标志的等待 |






TOPPERS

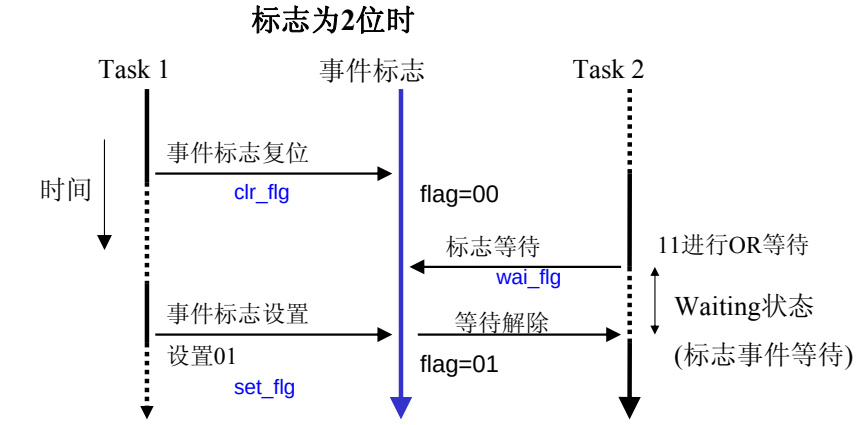
事件标志是通过用各BIT的标志表示事件的有无来实现任务间的同步的对象。

事件标志由对各BIT表示对应事件的有无的BIT类型以及等待设置事件标志的任务等待队列构成。通知事件的一方可以设置或清除指定事件标志的BIT类型的位。另一方面，在设置所有的或一部分指定BIT类型的位之前，等待事件的一方可以使任务进入事件等待状态。要求事件等待状态的任务到要求事件等待队列中排队等待。

事件标志功能由生成或删除事件标志的功能、设置或复位事件标志的功能、等待事件标志的设置的功能、查看事件标志状态的功能构成。

## 事件标志（EventFlag）：任务的执行状态

- 事件的有无通过标志的位类型来通知






TOPPERS

μITRON4.0的事件标志功能服务・调用如下所示。

1．事件标志的生成

CRE_FLG(ID flgid、 {ATR flgatr、 FLGPTN iflgptn});

ID flgid　　事件标志的ID号

ATR flgatr　　事件标志属性((TA_TFIFO||TA_PRI) |
(TA_WSGL||TA_WMUL)| [TA_CLR])

FLGPRN ifigptn　事件标志的BIT类型的初始值

2．事件标志的设置

ER ercd = set_flg(ID flgid、 FLGPTN setptn);

ER ercd = iset_flg(ID flgid、 FLGPTN setptn);

3．事件标志的清除

ER ercd = clr_flg(ID flgid、 FLGPTN clrptn);

4．事件标志等待

ER ercd = wai_flg(ID flgid、 FLGPTN waiptn、 MODE wfmode、 FLGPTN *p_flgptn);

ER ercd = pol_flg(ID flgid、 FLGPTN waiptn、 MODE wfmode、 FLGPTN *p_flgptn);

ER ercd = twai_flg(ID flgid、 FLGPTN waiptn、 MODE wfmode、 FLGPTN *p_flgptn、 TMO tmout);

# ITRON API解说

1. 任务相关的API
2. 根据程序确认
3. 同步附属功能
4. 同步附属功能程序设计



## 同步附属功能程序设计

以sample1为基础学习
同步附属功能的事件标志的程序设计

- 创建程序目录 ./jsp-1.4.2/OBJ/AKIH8_3069F/sync_api，使用configure生成sample1工程
- 变更为通过事件标志实现从主任务（main_task）到并行执行的任务（task）的指令
- 并行执行的任务不使用for()循环，而是通过twai_flg()实现对事件标志更新的1秒钟等待
- 要点
  - 事件标志的生成
  - 位分配与and/or等待
  - 各任务提取自己的事件标志ID的方法




基于sample1创建同步附属功能程序。

变更内容如下：

1）在现行的程序中是通过message数组来传递从主任务(main_task)到并行执行的任务(task)的指令。请修改为用事件标志来传递。

2）并行执行的任务（task）修改为等待事件标志。将通过for语句实现的软件延迟等待修改为通过twai_flg服务调用进行1秒钟的超时等待。

要点：

1）修改配置文件，生成事件标志。

2）确定在任务间进行交换的事件标志的BIT分配，思考是OR等待还是AND等待。

3）确定事件标志ID，并思考在各个任务中如何共享此名称。

## 同步附属功能程序设计

- 事件标志的生成（sampe1.cfg）
  - 对事件标志属性指定TA_CLR，并在解除任务的事件标志等待状态时清除事件标志的BIT类型的所有位

```
CRE_FLG(FLG_TASK1, {(TA_TFIFO|TA_CLR), 0});
CRE_FLG(FLG_TASK2, {(TA_TFIFO|TA_CLR), 0});
CRE_FLG(FLG_TASK3, {(TA_TFIFO|TA_CLR), 0});
```

- BIT分配

```
#define FLG_PTN_e 0x01
#define FLG_PTN_s 0x02
#define FLG_PTN_S 0x04
#define FLG_PTN_d 0x10
```

```
#define FLG_PTN_y 0x20
#define FLG_PTN_Y 0x40
#define FLG_PTN_z 0x80
#define FLG_PTN_Z 0x100
#define FLG_PTN_ALL 0x1ff
```

2007/01/14 TOPPERS工程认定 95

TOPPERS

修改sample1.cfg配置文件，并使用CRE_FLG静态API追加三个并行执行任务(task)用的事件标志的生成。CRE_FLG的属性、TA_TFIFO是C语言接口的指定，TA_CLG是在解除并行执行任务的事件标志等待状态时将事件标志的BIT类型清零的指定。

事件标志定义了8个BIT。此设置在3个标志中是通用的。

1）FLG_PTN_e ‘e’ 命令被指定

2）FLG_PTN_s ‘s’ 命令被指定

3）FLG_PTN_S ‘S’ 命令被指定

4）FLG_PTN_d ‘d’ 命令被指定

5）FLG_PTN_y ‘y’ 命令被指定

6）FLG_PTN_y ‘Y’ 命令被指定

7）FLG_PTN_z ‘z’ 命令被指定

8）FLG_PTN_Z ‘Z’ 命令被指定

为了指定twai_flg的等待状态要定义FLG_PTN_ALL。

## 同步附属功能程序设计

- **标志ID的提取将已设置了标志ID的数组作为静态变量**

```
const ID   flgid[] = {FLG_TASK1, FLG_TASK2, FLG_TASK3};
```

- **通过twai_flg（）提取ID**
  - **等待位类型是通过所有标志都进行了OR运算的FLG_PTN_ALL进行等待**
  - **等待模式是OR等待（TWF_ORW），只要某一个位被设置就会唤醒**

```
FLGPTN flgptn;
        :
flgptn = 0;
twai_flg(flgid[tskno-1], FLG_PTN_ALL, TWF_ORW, &flgptn,
            FLG_WAIT_TIME);
switch (flgptn) {
       case FLG_PTN_e:
```



TOPPERS

定义对应于并行执行任务对应的标志ID的静态数组。此数组用于在主任务和并行执行任务中选择对象的标志ID。为了指定1秒，FLG_WAIT_TIME指定1000。

在并行执行的任务中作为自动变量定义FLGPTN flgptn。在while(1)语句中对事件标志的等待状态追加twai_flg服务调用。等待BIT类型是通过所有标志都进行了OR运算的FLG_PTN_ALL进行等待。将已设置的事件位设置为flgptn变量。并将命令的解析修改为用此flgptn变量来实现。考虑一下同时进行设置，此时用if语句进行BIT解析比用switch语句进行解析更普遍一些。但在此系统中不可能存在同时设置，所以也可以使用switch语句。

## 同步附属功能程序设计

- 主任务
  - 从控制台接收数据并根据数据类型设置事件标志
  - 设置前用**clr_flg**将事件标志清除

```
switch(c){
case 'e':
        syscall(clr_flg(flgid [tskno-1], 0x00));
        syscall(set_flg(flgid[tskno-1], FLG_PTN_e));
        break;
case 's':
        syscall(clr_flg(flgid [tskno-1], 0x00));
        syscall(set_flg(flgid[tskno-1], FLG_PTN_s));
        break;
……
```




在主任务中将命令改为set_flg服务调用并传递给并行执行的任务。

关于syscall宏，请参照./jsp-1.4.2/include/t_services.h。它具有当服务调用的返回值为负值（错误）时显示错误内容的功能。

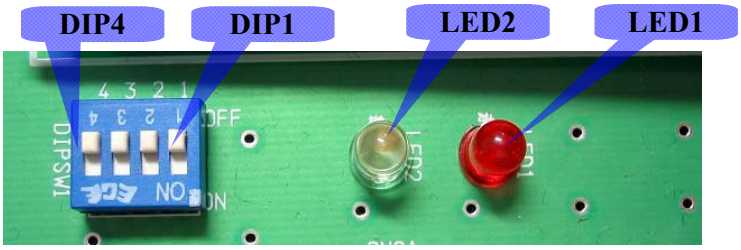
# 碗面定时器的开发

1. 碗面定时器规范
2. 设计方针
3. 设计范例
4. 编码范例



## 碗面定时器的规范

- 概要
  - 定时器启动后通知已超过了设置的时间
  - 时间可以设置为30秒
- 开关与LED的使用方法
  - **LED1：显示电源为ON（程序在运行）**
  - **LED2：显示定时器在运行**
  - **DIP1：启动・停止定时器**
  - **DIP4：设置时间延长30秒**






制作碗面定时器

・简单的规范

在4个DIP-SW中使用DIP1和DIP4。DIP1为ON时30秒启动定时器，为OFF时停止定时器。在启动了定时器时，每次DIP4变为ON时都会将时限延长30秒。

插上电源后，LED1以2秒为周期（1秒亮灯、1秒熄灭）。在启动了定时器时，每隔10秒钟LED2就会以0.5秒的周期（250ms亮灯、250ms熄灭）闪烁1秒钟。到了时限时，LED2以0.5秒的周期闪烁15秒钟。

・开关和LED的功能

1）DIP1-ON时启动定时器，OFF时停止定时器

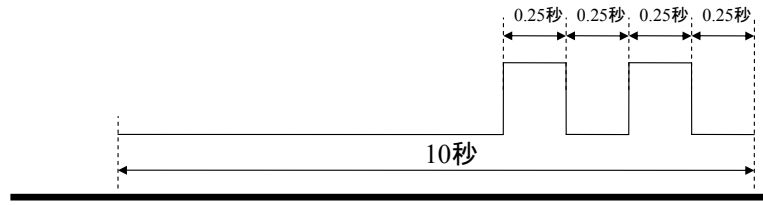
2）DIP4-定时器启动时，每次变为ON都会将超时时间延长30秒

3）LED1-定时器启动时，以2秒为周期进行闪烁

4）LED2-定时器启动时，以10秒钟的单位1秒钟的间隔进行闪烁。超时时以15秒钟的间隔进行闪烁。

## 碗面定时器的规范

- 外部规范
  - 电源为ON时（程序运行时），LED1间隔1秒钟反复亮灯、熄灭
  - 一旦SW1变为ON，就以30秒为设置时间来启动定时器
  - 一旦SW1变为OFF，就停止定时器
  - 在定时器运行时如果SW4变为ON，设置时间就会延长30秒
  - 在定时器运行时，以10秒钟的间隔使LED2闪烁两次
    LED2的两次闪烁是以0.25秒为间隔亮灯・熄灭重复两次
  - 一旦到了设置的时间，LED2闪烁15秒钟后就会熄灭
    LED2的闪烁是以0.25秒为间隔重复亮灯・熄灭






TOPPERS

创建事件列表

| 事件 | Stimulus | 动作 | 响应 |
|---|---|---|---|
| 插上电源 | 开始运行 | 设备的初始化 | 定时器启动准备结束，以2秒为周期闪烁LED1 |
| 启动定时器 | DIP1 ON | 以超时时间30秒启动定时器 | 每10秒、每1秒间隔闪烁LED2，超时时间隔闪烁15秒钟 |
| 追加定时器时间 | DIP4 ON | 将超时时间延长30秒 | 无 |
| 结束定时器 | DIP1 OFF | 停止定时器 | 进入定时器启动准备结束状态 |
| 切断电源 | 停止运行 | | |

创建数据字典

| | |
|---|---|
| 定时器启动准备结束状态 | 硬件和软件的初始化结束，随时可以启动定时器的状态 |
| 超时时间 | 初始超时时间为30秒，每次追加定时器时间时都追加30秒 |

# 碗面定时器的开发

1. 碗面定时器式
2. 设计方针
3. 设计范例
4. 编码范例



## 设计方针

尽可能使用RTOS的功能进行设计

- 有管理定时器的定时器任务和实现LED2闪烁的LED2闪烁任务的两个任务
- 开关的状态获取通过周期句柄实现
- 各种时间生成也通过周期句柄实现
- 周期句柄与任务间以及任务之间的事件通知通过事件标志实现
- 流用在基本程序中创建的设备操作程序




尽可能使用RTOS的功能来设计碗面定时器。

有以下两个任务：

1）定时器用任务（TIMER_TASK）进行定时器的管理

2）闪烁控制用任务（BLINK_TASK)进行LED2的间隔闪烁控制

有4个周期句柄：

1）LED1闪烁周期句柄：控制LED1的闪烁

2）LED2闪烁周期句柄：控制LED2的闪烁

3）开关用周期句柄：检测DIP-SW的状态变化

4）定时器用周期句柄：发行定时器的基本周期

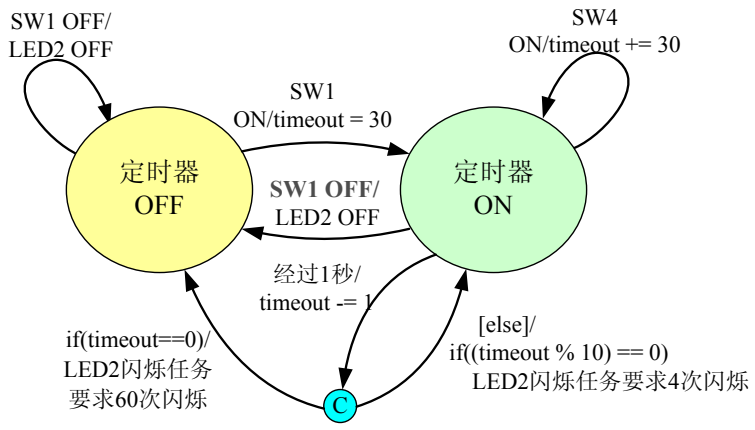
# 碗面定时器的开发

1. 碗面定时器规范
2. 设计方针
3. 设计范例
4. 编码范例



## 设计范例：定时器任务

- 状态迁移图
- 事件
  - **SW1 ON、SW1 OFF、SW4 ON、经过1秒**






定时器任务有2种状态：

1）定时器OFF状态（定时器运行准备结束状态）

定时器OFF状态是由DIP-SW1的状态变化来发生状态变化，在DIP-SW1为ON时迁移到定时器ON状态。

2）定时器ON状态

定时器ON状态由以下两种变化迁移到定时器OFF状态。

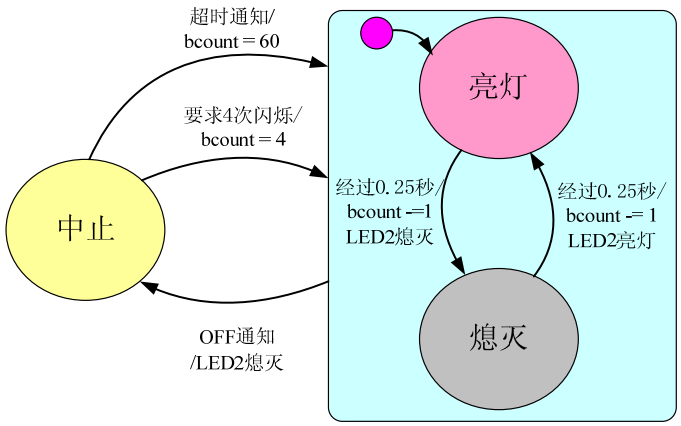
・DIP-SW1变为OFF时

・发生超时时

为了进行超时判定，每10秒钟从定时器周期句柄接收一次通知，并减掉超时值。而且在超时时，每10秒钟进行一次间隔闪烁的指示。

## 设计范例：LED2闪烁任务

- 状态迁移图
- 事件
  - 4次闪烁请求、超时通知、 OFF通知、 经过0.25秒






TOPPERS

为了实现复杂的间隔闪烁，LED2作为LED2闪烁任务进行任务管理。在中止状态与运行状态2种状态下，LED2闪烁任务具有亮灯状态与熄灭状态两种状态。

从中止状态到运行状态的迁移取决于定时器任务的间隔闪烁要求。从运行状态到中止状态的迁移在间隔时间结束时发生。

亮灯状态与熄灭状态间的迁移是在从LED2闪烁周期句柄接到时间经过（250ms）通知时发生。

## 设计范例：周期句柄

- 周期句柄
  - **LED1闪烁周期句柄 ：1秒周期**
  - **LED2闪烁周期句柄 ：250ms周期**
  - **开关用周期句柄 ：10ms周期**
  - **定时器用周期句柄 ：1秒周期**

- 初始化例程
  - **初始化硬件**




有4个周期句柄：
1）LED1闪烁周期句柄（LED1_BLINK_HANDLER）：1秒周期
2）LED2闪烁周期句柄（LED2_BLINK_TIME__HANDLER）：250ms周期
3）开关用周期句柄（SW_SCAN_HANDLER）：10ms周期
4）定时器用周期句柄（BASE_TIME_HANDLER）：1秒周期

有硬件初始化用的函数。
初始化函数（init_cup_timer）进行LED与DIP-SW的初始化。

# 碗面定时器的开发

1. 碗面定时器规范
2. 设计方针
3. 设计范例
4. 编码范例



## 编码范例

- 实装的代码在以下位置
  - 程序目录./jsp-1.4.2/OBJ/AKIH8_3069F/CUP_TIMER
- 定时器任务事件分配

```
#define TIMER_SW1_ON        0x01   /* SW1 ON 通知        */
#define TIMER_SW1_OFF       0x02   /* SW1 OFF  通知      */
#define TIMER_SW4_ON        0x04   /* SW4 ON 通知        */
#define TIMER_BASE_TIME     0x08   /* 经过1秒 通知       */
#define TIMER_ALL           0x0f   /* 事件位的OR         */
```

- LED2闪烁任务事件分配

```
#define BLINK_ACTIVE        0x01   /* 4次闪烁请求 通知   */
#define BLINK_TIMEOUT       0x02   /* 超时 通知          */
#define BLINK_BLINK         0x04   /* 经过0.25秒 通知    */
#define BLINK_OFF           0x08   /* OFF 通知           */
#define BLINK_ALL           0x0f   /* 事件位的OR         */
```



TOPPERS

对两个任务的状态迁移要求通过事件标志进行设计。

定时器任务的事件标志：FLG_TIMER

使用4个标签：

1）TIMER_SW1_ON：DIP-SW1变为ON（从开关用周期句柄）

2）TIMER_SW1_OFF：DIP-SW1变为OFF（从开关用周期句柄）

3）TIMER_SW4_ON：DIP-SW4变为ON（从开关用周期句柄）

4）TIMER_BASE_TIME：定时器经过1秒（从定时器用周期句柄）

LED2闪烁任务的事件标志：FLG_BLINK

使用4个标签：

1）BLINK_ACTIVE：要求1秒间隔闪烁（从定时器任务）

2）BLINK_TIMEOUT： 要求15秒间隔闪烁（从定时器任务）

3）BLINK_BLINK： 要求亮灯、熄灭的迁移（从LED2闪烁周期句柄）

4）BLINK_OFF：要求结束闪烁（从定时器任务）

## 编码范例：定时器任务

- **定时器OFF状态**
  - **接收SW1 ON、SW1 OFF事件**
  - **如果接收SW1_OFF事件，就对LED2任务通知BLINK_OFF**

```
void
timer_task(VP_INT exinf){
    int timer;
    FLGPTN flgptn;
    for (;;) {
        timer = 0;
        do {
            wai_flg(FLG_TIMER, TIMER_ALL, TWF_ORW, &flgptn);
            if ((flgptn & TIMER_SW1_ON) != 0) {
                timer = INIT_TIME;
            }
            if ((flgptn & TIMER_SW1_OFF) != 0) {
                set_flg(FLG_BLINK, BLINK_OFF);
            }
        } while (timer == 0);
```



TOPPERS

定时器任务中的定时器OFF状态是指自动变量的timer值为零的状态。该状态只对TIMER_SW1_ON与TIMER_SW1_OFF两个事件执行处理。接收到TIMER_SW1_ON时，通过将timer值设置为INIT_TIME（30秒），要求向定时器ON状态进行迁移。接收到TIMER_SW1_OFF时，要求LED2周期任务向中止状态进行迁移。

## 编码范例：定时器ON状态

- 如果接收SW1 OFF事件，就会向LED2任务通知BLINK OFF
- 通过SW4 ON延长时间
- 通过1秒经过事件来递减时间，在0秒内通过超时向LED2任务发送60次闪烁事件，以10秒的间隔发送4次闪烁事件

```
sta_cyc(BASE_TIME_HANDLER);
while (timer > 0) {
   wai_flg(FLG_TIMER, TIMER_ALL, TWF_ORW, &flgptn);
   if ((flgptn & TIMER_SW1_OFF) != 0) {
      set_flg(FLG_BLINK, BLINK_OFF);
      timer = 0;
   } else {
      if ((flgptn & TIMER_SW4_ON) != 0)
         timer = timer + EXTRA_UNIT;
      if ((flgptn & TIMER_BASE_TIME) != 0) {
         timer = timer - 1;
         if (timer == 0)
            set_flg(FLG_BLINK, BLINK_TIMEOUT);
         else if ((timer % ACT_INTERVAL) == 0)
            set_flg(FLG_BLINK, BLINK_ACTIVE);
      }
   } }
stp_cyc(BASE_TIME_HANDLER);
```



TOPPERS

在定时器ON状态下，激活定时器用周期句柄，以1秒为单位发行TIMER_BASE_TIME。通过此事件减掉自动变量timer的值，在变为0的时刻通过超时进入定时器OFF状态。timer的值变为0时，通过BLINK_TIMEOUT标志进行15秒钟的间隔闪烁，通过BLINK_ACTIVE标志每10秒进行1秒钟的间隔闪烁。

在定时器ON状态下，如果接收TIMER_SW1_OFF事件，就将timer的值设为零，并进入定时器OFF状态。如果接收TIMER_SW4_ON事件，就将timer值加EXTRA_UNIT（30秒）。

## 编码范例：初始化例程

- 硬件的初始化
- 保存开关的初始状态

```
void
init_cup_timer(void){
    initial_switch();      /* 开关的初始化 */
    initial_led();         /* LED的初始化 */

    sw4 = get_switch(SW4); /* 当前SW4的获取 */
    sw1 = get_switch(SW1); /* 当前SW5的获取 */
}
```




DIP-SW与LED间的接口使用device.h与device.c。硬件的初始化函数调用开关的初始化（initial_switch函数）和LED的初始化（initial_led函数）。在获取DIP-SW的当前状态时通过get_switch函数来进行获取。

Init_cup_timer的执行是通过ATT_INI静态API实现的。

## 编码范例：周期句柄

- 定时器用周期句柄
  - 以1秒钟的周期通知1秒经过事件

```
void
base_time_handler(void){
    syscall(iset_flg(FLG_TIMER, TIMER_BASE_TIME));
}
```

- LED2闪烁周期句柄
  - 以250ms的周期通知0.25秒经过事件

```
void
led2_blink_time_handler(void){
    syscall(iset_flg(FLG_BLINK, BLINK_BLINK));
}
```




每次启动定时器用周期句柄时，都会对定时器任务设置TIMER_BASE_TIME标志。由于周期句柄是在非任务语境下执行的，所以要使用iset_flg服务调用，而不使用set_flg服务调用。执行周期通过CRE_CYC静态API进行指定。启动、结束要求通过定时器任务来实现。

每次启动LED2闪烁周期句柄时，都会对LED2闪烁任务设置BLINK_BLINK标志。执行周期通过CRE_CYC静态API进行指定。启动、结束要求通过LED2闪烁任务来实现。

## 编码范例：开关用周期句柄

- 开关的状态发生变化时通知事件

```
void
sw_scan_handler(void){
   CONDITION sw;
   FLGPTN   flgptn = 0;
   sw = get_switch(SW1);
   if (sw1 != sw) {
      if (sw == ON) {
        flgptn |= TIMER_SW1_ON;
      } else {
        flgptn |= TIMER_SW1_OFF;
      }
      sw1 = sw;
   }
```

```
   sw = get_switch(SW4);
   if (sw4 != sw) {
      if (sw == ON)
        flgptn |= TIMER_SW4_ON;
      sw4 = sw;
   }
   if (flgptn != 0x00)
     syscall(iset_flg(FLG_TIMER, flgptn));
}
```

2007/01/14 TOPPERS工程认定 113

TOPPERS

开关用周期句柄是指监视DIP-SW状态的周期句柄。它监视DIP-SW1和DIP-SW4的状态，当有变化时就设置下述标志。

- TIMER_SW1_ON
- TIMER_SW1_OFF
- TIMER_SW4_ON

驱动器的初始化是通过ATT_INI实现的，所以sw1和sw4的初始值已经设置了。

所以，此周期句柄在系统启动时也可以是启动状态。请通过CRE_CYC静态API在活动属性下执行。程序读取DIP-SW的当前状态，并比较sw1与sw4。如果发生了变化就通过iset_flg服务调用来设置标志。

## 编码范例：LED 1 闪烁周期句柄

– **LED1以1秒为周期闪烁**

```
void
led1_blink_handler(void){
    static CONDITION sta_led = OFF;
    if (sta_led == ON) {
        sta_led = OFF;
    } else {
        sta_led = ON;
    }
    set_led(LED1, sta_led);
}
```



TOPPERS

LED1闪烁周期句柄是以2秒钟为周期闪烁LED1的周期句柄，LED在初始化例程中已经完成了初始化，而且在定时器可启动状态下进行闪烁。所以，也请将此周期句柄设置为在系统启动时可以执行。

LED1闪烁周期句柄是以2秒为周期闪烁LED1的周期句柄，LED已经由初始化例程进行了初始化，在定时器可启动状态下进行闪烁，所以此周期句柄也请设定为系统启动时可以执行。

## 编码范例：配置文件

- 2个任务、 4个周期句柄 、 2个事件标志
- 初始化例程

```
CRE_TSK(TIMER_TASK, { TA_HLNG|TA_ACT, (VP_INT) 0, timer_task,
                DEFAULT_PRIORITY, STACK_SIZE, NULL });
CRE_TSK(BLINK_TASK, { TA_HLNG|TA_ACT, (VP_INT) 0, blink_task,
                DEFAULT_PRIORITY, STACK_SIZE, NULL });
CRE_CYC(LED1_BLINK_HANDLER,{TA_HLNG|TA_STA, (VP_INT)0,
                led1_blink_handler, LED1_BLINK_INTERVAL,0});
CRE_CYC(LED2_BLINK_TIME_HANDLER,{TA_HLNG, (VP_INT)0,
                led2_blink_time_handler, LED2_BLINK_INTERVAL,0});
CRE_CYC(SW_SCAN_HANDLER,{TA_HLNG|TA_STA, (VP_INT)0,
                sw_scan_handler, SW_SCAN_INTERVAL,0});
CRE_CYC(BASE_TIME_HANDLER,{TA_HLNG, (VP_INT)0,
                base_time_handler,  BASE_TIME_INTERVAL,0});
CRE_FLG(FLG_TIMER, {(TA_TFIFO|TA_CLR), 0});
CRE_FLG(FLG_BLINK, {(TA_TFIFO|TA_CLR), 0});
ATT_INI({TA_HLNG, 0, init_cup_timer});
```




在结构中追加2个任务、4个周期句柄、2个时间标志的生成、初始化例程的设置的记述。

CRE_TSK以及CRE_CYC等的属性、TA_HLNG指定C语言的接口记述，TA_ACT指定在系统启动时激活任务、周期句柄。CRE_FLG的属性、TA_TFIFO将等待事件标志的任务等待队列的执行顺序定为FIFO顺序（在此系统中只有一个执行任务，而且TA_WMUL是不允许多个任务等待的，所以在此处无意义）。TA_CLR指定在将任务从事件标志等待状态移到执行状态的时刻事件标志的所有BIT类型都清零。如果进行了此设置，那么在执行wai_flg服务调用后就不需要通过clr_flg服务调用来进行标志的清除了。

如果用ATT_INI静态API指定初始化，就会在系统启动前执行初始化函数。

# 总结

## 总结

学习以下几项：

- AKI-H8/3069F与TOPPERS/JSP Kernel的构建方法
- 外围设备的程序设计方法
- ITRON规范
- 使用ITRON API进行程序设计
  - 任务相关的API
  - 同步附属功能
- 碗面定时器
  - 设计
  - 编码

第2天用TINET进行TCP/IP的程序设计



# 实习环境的构建方法

- 安装PizzaFactory2或Cygwin
  - 使用Cygwin时，要引用安装包中的make、perl。H8用的gcc用源码构建，或下载二进制进行安装
- 安装终端软件（Tera Term等）
- 分别在环境的主目录下（非必须）打开压缩了实习文件的toppers_mid.lzh
- 为了将设备管理器的信息注册到注册表上，执行打开的文件夹下的./rtsys/ToppersDeviceManager/regist.bat。确认“device.dll的DllRegisterServer成功了”的对话框是否显示
- 模拟用各执行文件
  - ExtBoard模拟器 ./rtsys/ExtBoard/ExtBoard.exe,
  - netDevice ./rtsys/ExtBoard/netDevice.exe
  - POT模拟器 ./rtsys/potsim/potsim.exe
- 有些环境需要安装Visual Basic 6 RunTime



# 教材的编写

参考资料

- 《H8/300H系列程序设计手册》
  瑞萨科技
- 《H8/3069 F-ZTAT硬件手册》
  瑞萨科技