***Network Manias White Paper***

# Porting FREE to the MPC860 system

2000년 3월 13일

Written by 유창모

Copyright **Network Manias** All Right Reserved.
URL http://www.netmanias.com

# Porting FREE to the MPC860 system

유창모 (cmyoo@medialincs.com/cmyoo@lycos.co.kr)

본 문서에서는 FREE(Free Realtime Embedded Executive)라는 이름의 RTOS 소스 코드를 MPC860 프로세서가 탑재된 시스템(보드)에 포팅하는 작업 과정 및 FREE 구조에 대해서 설명하도록 하겠다. 본 코드를 상용 제품에 적용하기에는 코드의 안전성 및 견고함에 많은 문제가 있다. 본 작업/문서의 취지는 RTOS 개발/포팅 작업을 위해 필요한 지식 및 개발 환경들을 소개하고, RTOS에 대한 추상화된 개념을 구현을 통해서 좀더 명확히 하자는데 그 의미를 두고자 한다. 본 작업에 있어서, 최대한 Original Source Code를 유지하려고 하였으며, 지면 절약을 위해서 본 문서에 설명된 코드 대부분의 에러 체크 루틴은 제거하였다.

## 1. Introduction

FREE(Free Realtime Embedded Executive)는 Hwa-Jin Bae(VxWorks 관련 뉴스그룹에 많은 글/답변을 올리는 미국 거주 한국인)라는 사람이 Motorola 68030 CPU가 탑재된 MVME 134 보드(Motorola에서 제작한 Evaluation board)에서 동작하도록 제작 및 테스트(테스트까지 해보았는지는 좀 의심스러움)한 매우 작은 사이즈의 RTOS 소스 코드이다. 본 문서에서는 Embedded system에서 많이 사용되고 있는 CPU 중에 하나인 MPC860 프로세서(가 탑재된 보드)에 FREE를 포팅 하는 작업에 대해서 설명하도록 하겠다.

포팅(Porting)이란, A라는 CPU가 장착된 보드에서 동작하는 프로그램을 B라는 CPU가 장착된 보드에서 동작하도록 소스 코드를 수정하는 일련의 작업을 일컫는다.

FREE는 MicroC/OS-II와 매우 유사한 구조를 가지고 있다. 그 기능을 간략히 소개하자면 다음과 같다.

- **Preemptive kernel**: READY상태에 있는 task중 가장 높은 priority의 task가 항상 CPU를 점유/사용할 수 있는 구조이다.
- **Multitasking**: 최대 1024개의 task를 지원(0 – 1023)하며, 같은 priority의 task는 허용하지 않는다(모든 task는 다른 priority를 가져야 함). 같은 priority의 task를 허용하지 않기 때문에, Round-robin scheduling은 지원하지 않는다.
- **Deterministic**: kernel에서 제공하는 시스템 콜(함수)은 task 개수와 무관하게 동일한 execution time을 제공한다. 단, **pick**()이라는 함수내에서 Ready list에 있는 task 중 가장 높은 priority를 선택 시에 task priority 값/개수에 따라서 execution time이 달라진다.
- **Stack**: 각 task 별로 독립된, 서로 다른 크기의 스택을 할당 할 수 있다. 스택이 overflow가 발생했는지를 체크하는 Stack-checking 기능은 지원하지 않는다.
- **Semaphore**: Priority based Binary & Counting Semaphore를 지원한다. Priority based란, N개의 task가 하나의 세마포어를 기다리고 있고, 그 세마포어를 가지고 있던 task가 이를 반환했을 때, N개의 task 중 priority가 가장 높은 task가 세마포어를 가지게 되는 방식을 뜻한다.

- **Message Queue/Mailbox**: 지원하지 않는다.
- **Memory**: 메모리 할당(allocation) 및 해제(release) 함수는 지원하지 않는다.
- **Timer**: 10ms 단위(resolution)의 timer tick을 제공하지만, 타이머를 이용한 서비스(함수)는 지원하지 않는다.
- **I/O**: Serial을 통한 데이터 송수신 기능(blocked/unblocked I/O) 및 이와 관련된 ANSI C library의 일부를 제공한다.
  - blocked I/O : Serial로 문자를 출력하려는 시점에서 Serial device가 그 문자를 보낼 수 없을 때(바뻐서), Serial 출력 함수를 호출한 task가 SUSPENDED상태로 된다. 그리고 Serial 입력 함수를 호출한 task의 경우, Serial로부터 데이터가 수신될 때까지 SUSPENDED상태로 된다.
  - unblocked I/O : Serial device가 문자를 출력할 수 있을 때까지 task는 RUNNING상태에서 무한정 기다리고(무한루프), 데이터 수신시에는 수신 데이터가 없으면 바로 에러 값을 리턴한다.

# 2. Source list

FREE 소스 코드의 디렉토리 구조 및 파일 리스트는 다음과 같다.

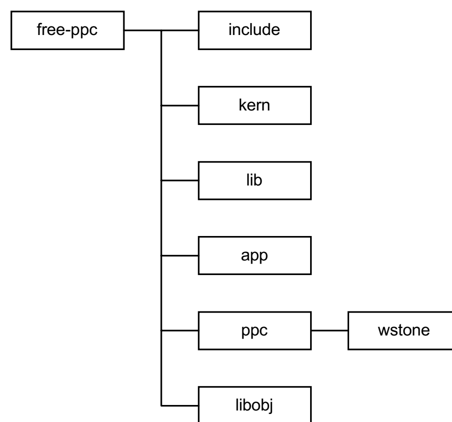


Figure 1: Directory Hierarchy

| Directory | File list | Description |
|---|---|---|
| free-ppc | Makefile README LICENSE | root directory |
| include | makerule common.h kern.h sem.h stdio.h string.h task.h | 각 모듈(디렉토리)에서 사용하는 header 파일들 |
| kern | Makefile init.c kern.c sem.c | kernel core + semaphore 코드가 정의되어 있음 |
| lib | Makefile stdio.c kstdio.c string.c | ANSI C library 중 일부가 정의되어 있음 |
| app | Makefile shell.c task.c test.c | shell, task entry point 및 test 코드가 정의되어 있음 |
| ppc | makedef Makefile cpu.h quicc.h trap.h cpu.c trap.c locore.s | PowerPC 관련 코드(Context Switch, Interrupt Handler)가 정의되어 있음 |
| wstone | Makefile ld.script bsp.h bsp.c boot.s wstone* wstone.bdx* wstone.ab* | target board 관련 코드(timer, serial)가 정의되어 있음 |
| libobj/mpc860 | libc.a* libapp.a* libcpu.a* libkern.a* mpc860.a* | /lib, /app, /ppc, /kern 디렉토리의 소스 코드에 의해서 생성된 library가 본 디렉토리에 저장됨 |

*: output file

# 3. Develope Environments

FREE 개발 환경은 그림 2와 같이 MPC860이 탑재된 target 보드와 코드 개발용 host(Linux), 디버깅용 host(Windows98) 그리고 BDM(Background Debug Mode) 장비로 구성된다.

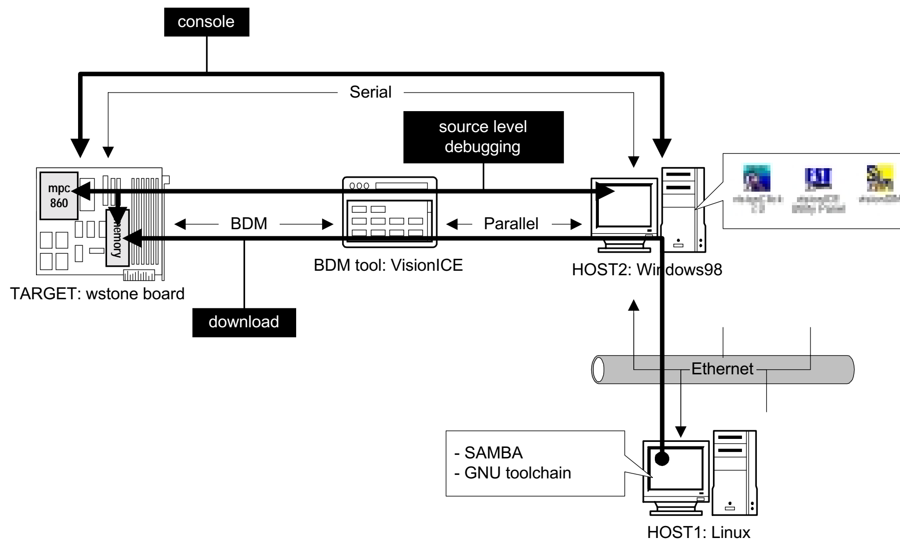


Figure 2: Develope Environments

각 장비의 기능은 다음과 같다.

• wstone board: Target board는 그림 3과 같이 구성되어 있으며, 본 구현에서 Ethernet, Flash memory는 사용되지 않았다.

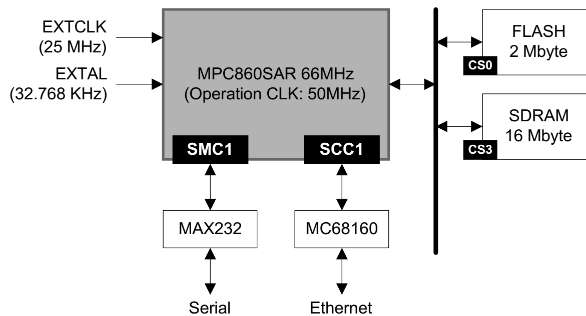


Figure 3: wstone target system

• VisionICE: Target(wstone)과는 10-pin BDM port로, PC(host2)와는 Parallel port로 연결되어 다음과 같은 기능을 지원한다.

  • MPC860 레지스터 값 확인 및 수정 기능

    VisionICE에서 MPC860 internal register를 초기화 해주기 때문에, FREE 코드에서는 SMC1을 이용한 Serial 및 Decrementer 초기화 만을 수행한다.

- Host에서 생성된 target image(wstone target에서 동작하는)를 target board의 메모리로 다운로드(BDX/AB 파일 포맷이 다운로드 됨)하고 소스 레벨 디버깅 지원
- Flash memory(주로 Intel 및 AMD 계열) erase/write 기능

• Windows98: Click7.0(그림 4참조)이라는 EST 제공 응용 프로그램을 이용하여, 위에서 나열한 VisionICE 기능을 제어하고, **convert.exe**(그림 5 참조) 프로그램을 이용하여, ELF 포맷을 BDX(실행 이미지), AB(디버깅 정보가 담긴 이미지) 파일 포맷으로 변형한다. ELF(Executable and Linking Format)는 SYSTEM V ABI에서 규정하는 표준 포맷 파일이고 BDX, AB는 EST의 VisionICE tool 지원을 위한 포맷이다.

• Linux: PowerPC용 GNU toolchain을 이용, FREE 소스 코드를 compile/link하여 ELF 포맷의 최종 파일(wstone)을 생성한다. 그리고 이 시스템에 SAMBA를 인스톨하여 Windows98에서 Linux 파일 시스템을 mount하여 사용하였다.

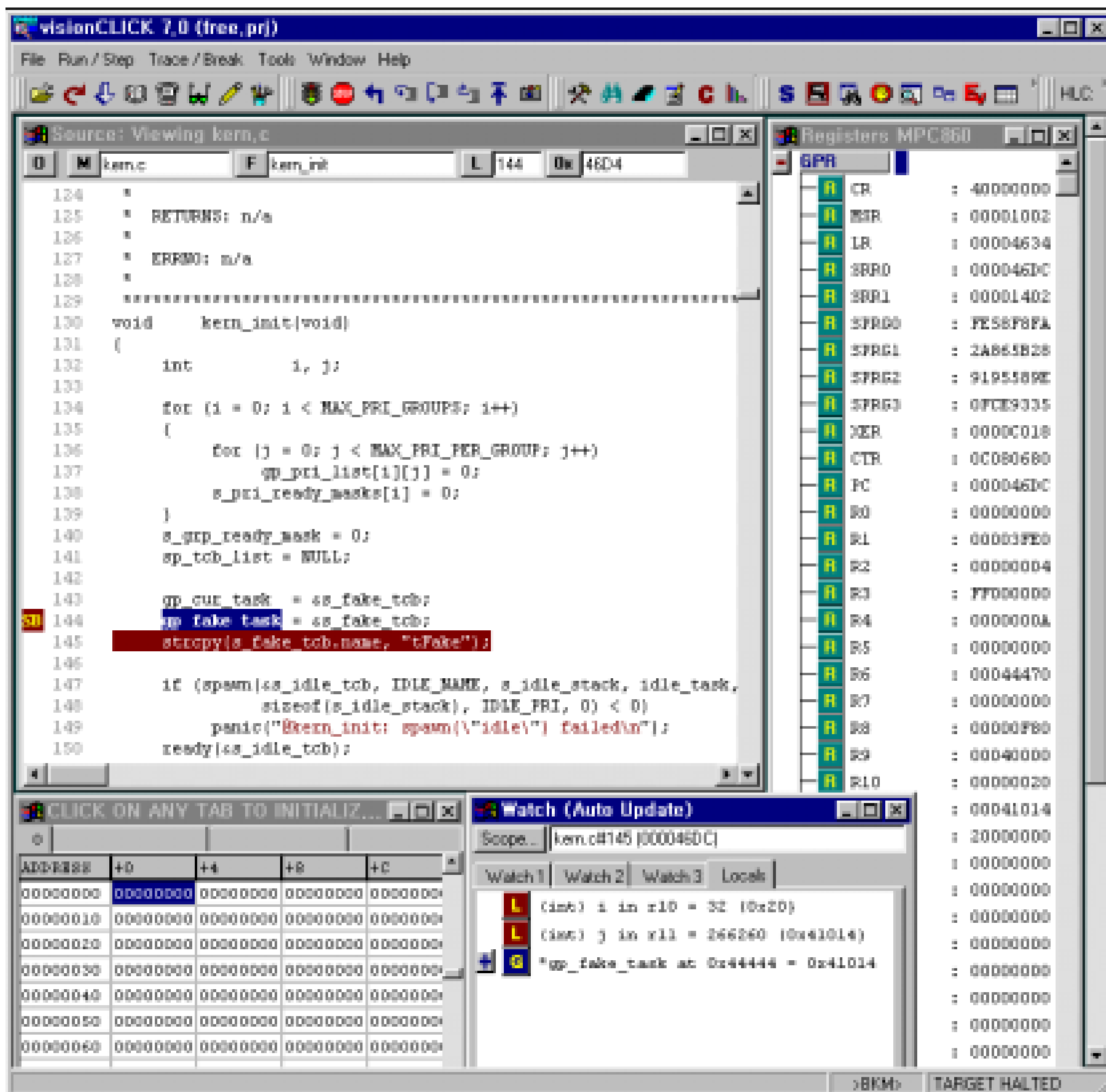


Figure 4: Clock7.0

```
convert v6.1B (32bit) Copyright (c) 1996-1998 Embedded Support Tools Corp.
usage: convert [-e|-s|-d|-h|-w] <input file> -b/-c/-a  <output file>
               <-v> <output message file> <-l/-u/-z/-f/-r>
Switch                                   Description
  -s     ...   <S-Record input file                                      >
  -e     ...   <IEEE-695 input file                                      >
  -d     ...   <COFF input file                                          >
  -h     ...   <a.out input file                                         >
  -w     ...   <ELF  input file                                          >
  -c     ...   <create symbol output file for visionCLICK                >
  -b     ...   <create bdx output file                                   >
  -a     ...   <create flat binary output file                           >
  -v     ...   <create Message output file, default "warning.doc"        >
  -l     ...   <Image Lower Hex Address (included), default 0x0          >
  -u     ...   <Image Upper Hex Address (included), default 0xFFFFFFFF   >
  -m     ...   <C++ demangling (none, arm, or gnu), default none         >
  -f     ...   <Hex Fill Character, default 0xFF                         >
  -z     ...   <wait for key to be pressed before exiting                >
  -q     ...   <quiet operation - no extraneous messages                 >
  -T     ...   <location of a.out segments -Ttext ORG,-Tdata ORG,-Tbss ORG >
  -t     ...   <default target type (ppc or m68k) if not in input file   >
  -r     ...   <replace NOOP's with TRACE enabled code                   >
  -k     ...   <create a compressed BIN or BDX file                      >
```

Figure 5: CONVERT.EXE help

FREE 개발/포팅은 다음과 같은 순서로 이루어졌다.

❶ [Linux] 소스 코드 제작 및 수정

❷ [Linux] GNU toolchain으로 compile/link하여 ELF 포맷의 **wstone** 파일 생성

  **[free-ppc] make** ↵

  **[free-ppc/libobj/mpc860] ls** ↵

  **libc.a  libapp.a  libcpu.a  libkern.a  mpc860.a** (5개의 library file 생성)

  **[free-ppc/ppc/wstone] make** ↵

  **[free-ppc/ppc/wstone] ls** ↵

  **wstone** (최종 파일 생성)

❸ [Win98] **convert.exe**을 이용, **wstone** 파일을 **wstone.bdx**, **wstone.ab** 파일로 변형(생성).

  **Z:\free-ppc\ppc\wstone>convert -w wstone -b -c** ↵

  **Z:\free-ppc\ppc\wstone>dir** ↵

  **wstone  wstone.bdx  wstone.ab**

❹ [VisionICE] VisionICE 장비를 이용하여 MPC860이 동작 가능한 상태가 되도록 레지스터 초기화

❺ [Win98] Click7.0/VisionICE를 이용하여 **wstone.bdx**, **wstone.ab** 파일을 wstone target에 다운로드 및 코드 실행

❻ [Win98] Click7.0/VisionICE를 이용하여 코드 디버깅.

❸❺❻은 Linux의 SAMBA를 이용, Windows98에 Linux file system을 하나의 드라이브로 마운트(네트워크 드라이브 연결) 하여 작업을 진행하였다. 즉, wsteon, wstone.bdx, wstone.ab 및 모든 소스 코드는 Linux에 존재한다.

# 4. FREE architecture

## 4.1 Task States

그림 6은 FREE의 task 상태 천이 및 이와 관련된 kernel 함수를 나타낸 것이다.

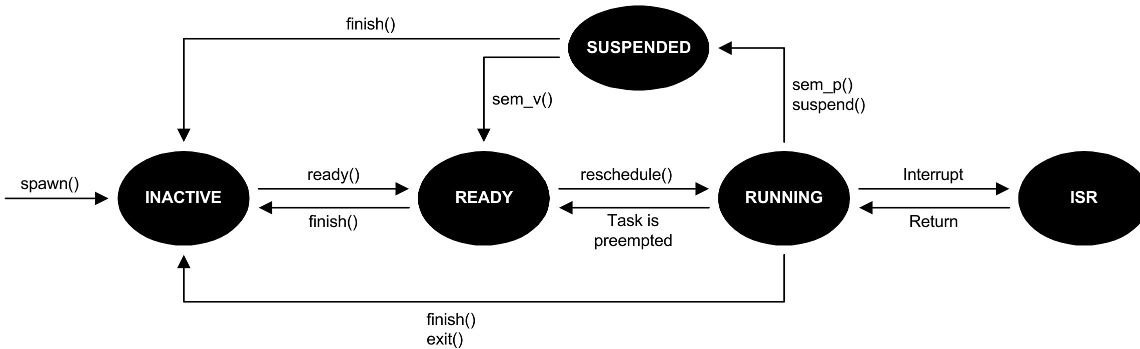


Figure 6: FREE task states

- **spawn()** 함수에 의해서 task가 생성(아직 CPU를 사용할 수 없는 INACTIVE상태임)되며, task가 사용할 TCB(Task Control Block) structure 및 kernel structure를 초기화한다.
- **ready()** 함수에 의해서 task는 READY상태(이제 CPU를 사용할 수 있는 기회를 가짐)가 된다.
- **reschedule()** 함수에 의해서, READY상태에 있는 task 중 가장 높은 priority를 가진 task를 RUNNING상태(CPU를 사용하는 상태)로 만든다. Preemption당한(CPU 제어권을 빼앗긴) task는 READY상태로 된다.
- RUNNING상태에서 CPU를 사용하던 task는 인터럽트 발생에 의해서 수행이 중단되고, ISR (Interrupt Service Routine)이 실행된다. ISR 종료 지점에서 **reschedule()** 함수가 호출되어 ISR 종료 후에, 가장 높은 priority가 수행 할 수 있도록 한다. 즉, ISR 종료 후에 인터럽트 발생 직전에 수행되던 task(interrupted task)가 아닌 다른 task가 RUNNING상태로 수행 될 수 있다. 이 경우 interrupted task는 READY상태가 된다.
- RUNNING상태에 있는 task는 **sem_p()**나 **suspend()** 함수 호출에 의해서 SUSPENDED상태로 될 수 있다.
- SUSPENDED상태에 있는 task는 **sem_v()**에 의해서 READY상태로 된다. SUSPENDED상태에 있는 task는 CPU를 사용할 수 없는 상태이기 때문에 SUSPENDED상태의 task가 **sem_v()**를 호출할 수 없으며, ISR이나 현재 RUNNING상태에 있는 task의 **sem_v()** 호출에 의해서 SUSPENDED상태의 task가 READY상태로 되는 것이다.

## 4.2 Task Control Block(TCB)

**spawn()**에 의해서 task가 생성되면 아래와 같은 TCB structure(**tcb_t**)가 초기화된다. task마다 각각 하나의 TCB structure를 가지며 본 TCB structure에 task 관련 정보들이 저장된다.

free-ppc/include/kern.h

```
01  struct tcb
02  {
03      context_t    context;
04      char         name[16];
05      int          (*entry)();
06      int          (*restart)();
07      int          stkbeg;
08      int          stkend;
09      int          stksiz;
10      int          grp_index;
11      int          pri_index;
12      int          stat;
13      int          arg;
14      char         *pstdio_buf;
15      struct tcb   *pnext;
16  };
17  typedef struct tcb   tcb_t;
```

free-ppc/include/kern.h

03 context: PowerPC register가 저장된다. (4.5 Task Scheduling에서 설명)

04 name: task name이 ASCII character(string)로 저장된다.

05 (*entry)(): task의 시작 함수(entry point) 주소가 저장된다.

06 (*restart)(): restart() 함수에 의해서 task 재실행 시의 entry point 함수가 저장된다.

07 stkbeg: 스택의 시작 위치(top of stack)를 가리킨다.

08 stkend: 스택의 마지막 위치(bottom of stack)를 가리킨다.

09 stksiz: 스택 영역의 크기를 저장한다.

10 grp_index: task priority의 상위 5 bit값이 저장된다. (그림 7 참조)

11 pri_index: task priority의 하위 5 bit값이 저장된다. (그림 7 참조)

12 stat: task 상태가 저장된다. (kern.h에 정의)

13 arg: (*entry)(), (*restart)() 호출시의 아규먼트가 저장된다. 본 구현에서는 사용되지 않았다.

14 pstdio_buf: standard I/O용 buffer 주소가 저장된다.

15 pnext: 각 TCB structure를 single linked-list로 연결하는데 사용된다.

## 4.3 Ready List & Priority List

Multitasking 환경에서 task들이 생성되고, priority 기준에 따라 CPU를 사용하기 위해서는 READY 상태에 있는 task들 정보 관리를 위한 Ready List와 Priority List가 필요하다.

free-ppc/include/kern.h

```
01  #define MIN_PRIORITY              0
02  #define MAX_PRIORITY              ((MAX_PRI_GROUPS * MAX_PRI_PER_GROUP) - 1)
03  #define PRI_TO_GROUP_INDEX(pri)   (((pri) >> 5) & 0x1f)
04  #define PRI_TO_PRI_INDEX(pri)     ((pri) & 0x1f)
```

free-ppc/include/kern.h

01-02 task priority는 0에서 1023 중에 하나의 값을 가지며, 0값은 “idle task”에 할당되어 있기 때문에 다른 task에서는 사용할 수 없다.

03-04 0부터 1023까지의 priority 값은 10 bit로 표현 가능하며, FREE에서는 이 값을 상위 5 bit와 하위 5 bit로 나누어 처리한다. 상위 5 bit로 32개의 task group을 형성하고, 각 그룹에는 32개(5 bit)의 task가 놓인다. (1024명의 학생을 키순으로 정렬시켜, 32개의 교실에 32명씩 배정하는 식으로...)

free-ppc/kern/kern.c

```
01  tcb_t           *gp_pri_list[MAX_PRI_GROUPS][MAX_PRI_PER_GROUP];
02  static u_int    s_grp_ready_mask;
03  static u_int    s_pri_ready_masks[MAX_PRI_GROUPS];
```

free-ppc/kern/kern.c

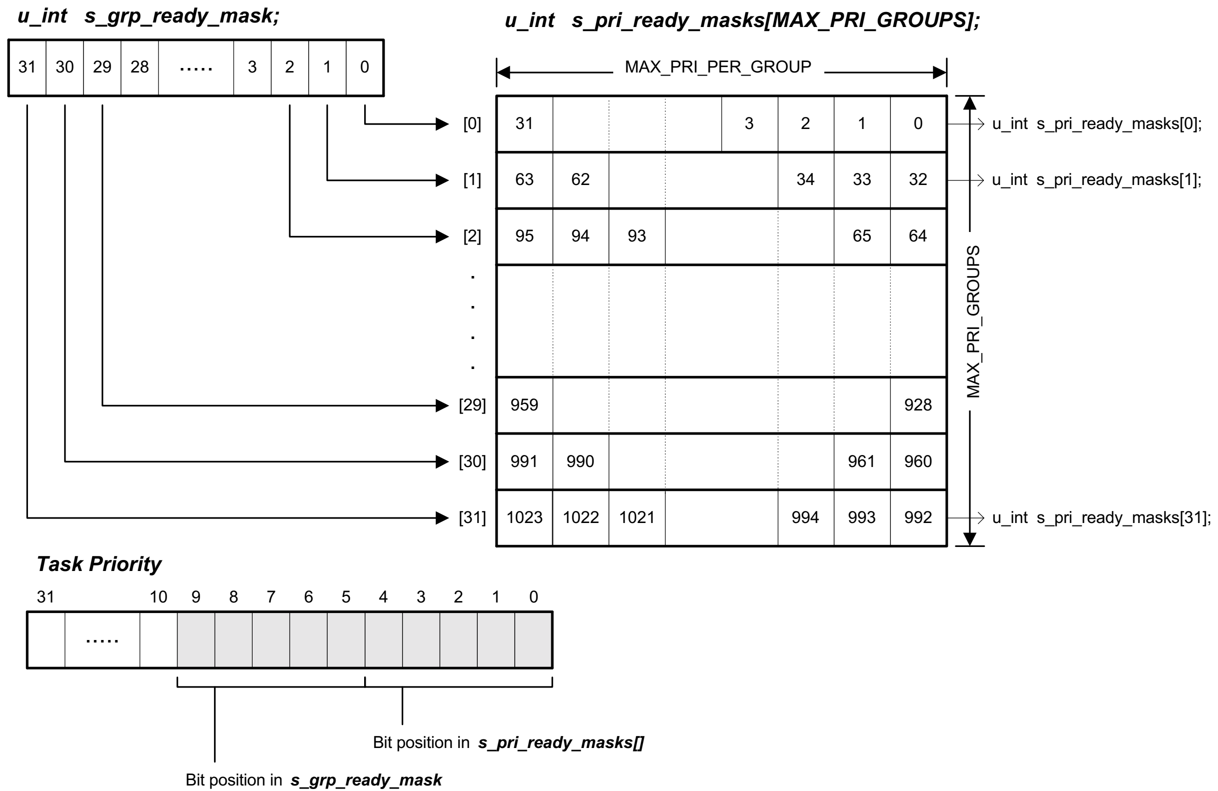


Figure 7: Ready List & Priority List

01 spawn()에 의해서 생성된 TCB structure의 주소가 priority값을 인덱스로 하여 gp_pri_list[][]에 저장된다.

02-03 READY상태에 있는 task들은 Ready list에 저장된다. 최대 1024개의 task가 READY상태에 존재할 수 있기 때문에, 이 Ready list는 1024 bit(32bit * 32개)만큼의 저장공간이 필요해 진다(예를 들어 priority 200인 task가 READY상태가 되면, 1024 bit 중에 200번째 bit를 1로 set). FREE에서는 그림 7과 같이 s_grp_ready_mask와 s_pri_ready_masks[32]라는 2개의 변수를 사용하여, 1024개의 상태를 관리한다. 1024 bit는 32개로 그룹화(0 – 31)되어, 각 그룹은 s_grp_ready_mask의 비트 위치 및 s_pri_ready_masks[] 배열의 인덱스와 매핑 되어지며, 각 그룹에 존재하는 32개의 엔트리는 s_pri_ready_masks[n]의 각 비트에 매핑 되는 구조를 가진다. 그래서 s_pri_ready_masks[n]의 값이 0이면 s_grp_ready_mask의 n비트는 0이 되고, s_pri_ready_masks[n]의 값이 0이 아니면 s_grp_ready_mask의 n비트 역시 0이 아닌 값을 가진다(그 역관계도 성립).

예를 들어서 priority가 100(0x64: 011 00100)인 task가 spawn()되고 READY상태가 된다면,

- gp_pri_list[3][4]에 이 TCB의 주소가 저장되고,

```
ptcb->grp_index = PRI_TO_GROUP_INDEX(priority);
ptcb->pri_index = PRI_TO_PRI_INDEX(priority);
```

```
gp_pri_list[ptcb->grp_index][ptcb->pri_index] = ptcb;
```

- **s_grp_ready_mask**(32bits)는 xxxx xxxx xxxx xxxx xxxx xxxx xxxx 1xxx이 되고,

```
s_grp_ready_mask |= (1 << ptask->grp_index);
```

- **s_pri_ready_masks[3]**(32bits)은 xxxx xxxx xxxx xxxx xxxx xxxx xxx1 xxxx이 된다.

```
s_pri_ready_masks[ptask->grp_index] |= (1 << ptask->pri_index);
```

## 4.4 Task Management

free-ppc/kern/kern.c

```
01  int      spawn(tcb_t *ptcb, char *pname, char *pstack, int (*entry)(),
02                int stksiz, int priority, int arg)
03  {
04      int          s = splhi();
05      context_t    *pctx;

06      ptcb->grp_index = PRI_TO_GROUP_INDEX(priority);
07      ptcb->pri_index = PRI_TO_PRI_INDEX(priority);

08      if (gp_pri_list[ptcb->grp_index][ptcb->pri_index])
09      {
10          kprintf("[e:%s]@spawn: priority slot already occupied by \"%s(%h)\"\n",
11                  t_name(),
12                  gp_pri_list[ptcb->grp_index][ptcb->pri_index]->name,
13                  gp_pri_list[ptcb->grp_index][ptcb->pri_index]);
14          splx(s);
15          return -1;
16      }
17      debug("[%s]@spawn: \"%s(%h)\",stk:%h, code:%h, stksz:%d, pri:%d\n",
18            t_name(), pname, ptcb, pstack, entry, stksiz, priority);

19      /*
20       * After task is spawned, context_swap() is called by reschedule().
21       * When new_context is restored in registers, LR will point to 'task_main' entry.
22       * After context_swap() is returned, task_main() will be executed.
23       * In task_main()'s prologue, it make a stack frame and save LR value.
24       * We must reserved some area to save 'LR'
25       */
26      ptcb->stkbeg = (int)(pstack + stksiz - 32);    /* stack margine */

27      pctx = &ptcb->context;
28      ctx_fill(pctx, (u_int)task_main, MSR_INIT_VAL, ptcb->stkbeg);

29      ptcb->pnext = sp_tcb_list;
30      sp_tcb_list = ptcb;

31      strcpy(ptcb->name, pname);
32      ptcb->entry          = entry;
33      ptcb->restart        = entry;
34      ptcb->stkend         = ptcb->stkbeg;
35      ptcb->stksiz         = stksiz;
36      ptcb->stat           = TASK_INACTIVE;
37      ptcb->pstdio_buf     = 0;
38      ptcb->arg            = arg;
39      gp_pri_list[ptcb->grp_index][ptcb->pri_index] = ptcb;
40      splx(s);
41      return 0;
42  }   /* end of spawn */
```

free-ppc/kern/kern.c

01-02 **spawn()**에 의해서 task가 생성되고 INACTIVE상태가 된다.

▪ ptcb: 생성될 task의 TCB 주소를 가지고 있으며, 이 TCB structure는 **spawn()**을 호출한 측(함수)에서 할당해야 한다. 본 구현에서는 전역 변수로 처리하였다.

▪ pname: task 이름이 저장된 곳의 주소를 가리킨다.

▪ pstack: task가 사용할 스택의 시작 주소가 담긴다. **spawn()** 호출 측에서 스택을 할당해야 하며, 본 구현에서는 전역 변수로 처리하였다.

- entry: task의 entry point(시작 함수)가 담긴다.
- stksiz: 스택 크기가 담긴다.
- priority: 생성할 task의 priority가 담긴다.
- arg: entry() 함수의 아규먼트가 담긴다.

04,40 **kern.c** 및 **sem.c**에 정의되어 있는 모든 kernel 관련 함수들은 함수 실행 첫 부분에서 인터럽트를 disable(**splhi**())하고 함수 종료시점에서 인터럽트를 다시 enable(**splx**())하여, 코드 수행중에 Scheduling 이 발생하지 않도록 한다.

06-16 **spawn**()의 아규먼트로 넘어온 priority값을 상위 5bit와 하위 5bit로 나누어 각각 TCB structure의 **grp_index**와 **pri_index**에 저장한다. 같은 priority의 task를 허용하지 않기 때문에, **gp_pri_list[][]**를 참조하여 같은 priority의 task가 이미 존재할 경우에는 에러를 리턴한다.

19-26 스택의 시작점(top of stack)을 **stkbeg**에 저장하는데, 할당된 메모리 영역의 최상위 주소에서 32(PowerPC EABI 규정에 따라 8도 가능할 것임)를 뺀 주소를 저장한다. 그 이유는 Function Prologue의 Stack frame 형성과 LR(Link Register) 저장 위치에 의한 것으로, 자세한 사항은 PowerPC EABI 문서를 참조하기 바란다.

27-28 본 task의 TCB structure에 저장되는 Context(CPU 레지스터의 값의 모임)중, Program Counter, Status Register, Stack Pointer를 초기화한다.

- Program Counter: PowerPC에는 Program Counter Register가 존재하지 않는 대신(내부적으로 관리하는 듯) LR(Link Register)에 branch(jump)할 곳의 주소가 저장된다. 모든 task의 실행 시작점을 **task_main**() 함수로 설정하기 위해서 LR에 이 함수의 주소를 저장한다.
- Status Register: PowerPC의 MSR(Machine State Register)에 해당하며, 다음 기능들을 enable(set)한다.
  - EE: External interrupt enable
  - ME: Machine check enable
  - RI: Recoverable exception enable
- Stack Pointer: PowerPC EABI 규격에 따라서 r1 레지스터를 stack pointer로 사용하며, r1이 top of stack을 가리키도록 한다.

29-30 새로 생성/초기화된 TCB structure를 TCB linked-list에 삽입한다.

31-39 **spawn**()의 아규먼트로 넘어온 값들을 이용하여 TCB를 초기화하고, **gp_pri_list[][]**에 본 TCB structure의 주소를 저장한다.

free-ppc/kern/kern.c

```
01  void     task_main(void)
02  {
03      debug("[%s]@task_main: gp_cur_task:\"%s(%h)\", entry:%h\n",
04            t_name(), gp_cur_task->name, gp_cur_task, gp_cur_task->entry);
05      (*gp_cur_task->entry)();
06      exit(0);
07  }   /* end of task_main */
```

free-ppc/kern/kern.c

01 앞에서 설명하였듯이, 모든 task의 entry point(task의 시작점)는 **task_main**()이다.

05 task가 처음 실행되면(**task_main**()이 호출되면), **spawn**()에서 저장했던 사용자 코드의 시작 함수(**entry**())를 호출한다.

06 task가 종료되면(**entry**() 함수가 종료되면), **exit**()가 호출되어 본 task는 INACTIVE상태가 된다.

free-ppc/kern/kern.c

```
01  int     idle_task(void)
02  {
03      u_int   secs;
04      u_int   iters;
05      u_int   max_iters;

06      secs        = g_timer_secs;
07      iters       = 0;
08      s_max_usage = 0;
09      max_iters   = 0;

10      for (;;)
11      {
12          if (g_timer_secs != secs)
13          {
14              /* select idle maxinum counter */
15              if (iters > max_iters)
16                  max_iters = iters;

17              s_cpu_usage = ((max_iters - iters) * 100) / max_iters;
18              if (s_cpu_usage > s_max_usage)
19                  s_max_usage = s_cpu_usage;
20              debug("[%s]@idle_task: %d secs since boot, cpu usage %d%%\n",
21                  t_name(), g_timer_secs, s_cpu_usage);
22              secs    = g_timer_secs;
23              iters   = 0;
24          }
25          else
26              iters++;
27      }
28      return 0;
29  }   /* end of idle_task */
```

free-ppc/kern/kern.c

01 READY상태에 존재하는 task가 하나도 없을 경우에(CPU를 사용할 task가 없을 경우) “idle task”가 수행된다. 즉, 최소한 하나의 task는 RUNNING상태에 있어 CPU를 사용해야 한다. 이 task는 가장 낮은 priority(=0)를 가지고 있고, kernel에 의해서 생성(**spawn**())되어 지며, CPU 사용율 계산을 담당한다.

10-27 이 task는 결코 종료되어서는 안되기 때문에, 무한 루프를 돈다.

12-26 다음과 같은 수식을 이용하여 CPU 사용율을 계산하게 된다. **g_timer_secs** 변수는 Timer ISR(**timer_handle**())에 의해서 매초마다 1씩 증가된다.

$$CPUUsage(\%) = 100(1 - \frac{Current\ Idle\ Count\ Value}{Maximum\ Idle\ Count\ Value})$$

## 4.5 Task Scheduling

본 절에서는 FREE의 task scheduling 구조와 PowerPC 레지스터 저장/복귀에 대해서 설명하도록 하겠다.

free-ppc/kern/kern.c

```
01  int     ready(tcb_t *ptask)
02  {
03      int     s = splhi();

04      ptask->stat &= ~TASK_STATE_MASK;
05      ptask->stat |= TASK_READY;

06      s_grp_ready_mask |= (1 << ptask->grp_index);
07      s_pri_ready_masks[ptask->grp_index] |= (1 << ptask->pri_index);

08      debug("[%s]@ready: \"%s(%h)\" marked ready, s_grp_ready_mask:%h\n",
```

```
09              t_name(), ptask->name, ptask, s_grp_ready_mask);
10          splx(s);
11          return 0;
12      }   /* end of ready */
```

free-ppc/kern/kern.c

01 INACTIVE, SUSPENDED상태에 있던 task는 **ready**() 함수에 의해서 READY상태가 된다. 아규먼트 ptask는 READY상태로 만들 task의 TCB structure의 주소이다.

04-07 TCB structure의 stat 필드를 READY상태로 표시하고, Ready list의 해당 비트를 set한다. 이와 같은 작업(TCB의 stat를 READY로 하고, Ready list에 해당 슬롯을 1로 set)에 의해서 task는 READY 상태가 되는 것이고, 만약 Ready list에서 대기하고 있는 task들 중 이 task의 priority가 가장 높다면, Scheduler(**reschedule**() 함수) 및 Context Switch(**context_swap**() 함수)에 의해서 본 task가 RUNNING 상태가 되어 CPU를 사용하게 될 것이다.

free-ppc/kern/kern.c

```
01  int      suspend(tcb_t *ptask)
02  {
03      int      s = splhi();

04      if (!ptask)
05      {
06          if (gp_cur_task == 0)
07              panic("@suspend: gp_cur_task = 0, ptask = 0");
08          ptask = gp_cur_task;
09      }

10      ptask->stat &= ~TASK_STATE_MASK;
11      ptask->stat |=  TASK_SUSPENDED;

12      s_pri_ready_masks[ptask->grp_index] &= ~(1 << ptask->pri_index);
13      if (s_pri_ready_masks[ptask->grp_index] == 0)
14          s_grp_ready_mask &= ~(1 << ptask->grp_index);

15      debug("[%s]@suspend: \"%s(%h)\" suspended\n", t_name(), ptask->name, ptask);
16      splx(s);
17      return 0;
18  }   /* end of suspend */
```

free-ppc/kern/kern.c

01 **suspend**()의 아규먼트는 SUSPENDED상태로 만들 task의 TCB structure 포인터나 0(NULL)이 될 수 있다.

04-09 만약 아규먼트가 0이라면, 현재 RUNNING상태에 있는 task가 스스로 SUSPENDED상태로 되려는 경우로서 전역변수 **gp_cur_task**(현재 RUNNING상태에 있는 task의 TCB structure 포인터를 저장)의 값을 참조하여 TCB 포인터를 얻어온다. 0이 아닌 값 즉, 다른 task의 TCB structure pointer라는 것은 현재 RUNNING상태에 있는 task가 아규먼트로 넘어온 task(RUNNING상태가 아닌 task)를 SUSPENDED상태로 만드는 것이다.

10-11 TCB stat 필드를 SUSPENDED상태로 한다.

12-14 **ready**() 함수에 의해서 1로 set되었던 Ready list의 해당 비트를 0으로 clear한다.

free-ppc/kern/kern.c

```
01  void     reschedule(void)
02  {
03      int      s = splhi();
04      tcb_t    *pnew_task, *pold_task;

05      pnew_task = pick(0);
06      if (pnew_task == gp_cur_task)
07      {
```

```
08          debug("[%s]@reschedule: \"%s(%h)\" highest priority(no context_swap)\n",
09                t_name(), gp_cur_task->name, gp_cur_task);
10          splx(s);
11          return;
12      }
13      pold_task    = gp_cur_task;
14      gp_cur_task  = pnew_task;
15      pold_task->stat &= ~TASK_RUNNING;
16      pnew_task->stat |= TASK_RUNNING;
17      debug("[%s]@reshcedule: \"%s(%h)\" -> \"%s(%h)\"\n",
18            t_name(), pold_task->name, pold_task, pnew_task->name, pnew_task);
19      debug("[%s]@reshcedule: (o)lr/srr0:%h/%h, (n)lr/srr0:%h/%h\n",
20            t_name(), pold_task->context.lr, pold_task->context.srr[0],
21                      pnew_task->context.lr, pnew_task->context.srr[0]);
22      context_swap(&pold_task->context, &pnew_task->context);
23      debug("[%s]@reschedule: after context_swap()\n", t_name());
24      splx(s);
25  }   /* end of reschedule */
```

free-ppc/kern/kern.c

01 **reschdule()** 함수에 의해서 task scheduling이 발생한다.

05-12 **pick()** 함수를 이용하여 가장 높은 prority의 task를 선택하게 되며, 만약 RUNNING상태의 task가 선택되었다면 Scheduling 발생 없이 **reschedule()** 함수는 종료된다(현재 RUNNING상태에 있는 task가 가장 높은 prority의 task인 상황).

13-14 CPU 제어권을 빼앗길 task(지금은 RUNNING상태이나 곧 READY나 SUSPENDED상태가 될) TCB structure 주소를 지역 변수 **pold_task**에 저장하고, RUNNING상태가 될 task의 TCB strucuture 주소를 전역 변수 **gp_cur_task**에 저장한다.

15-16 새로 RUNNING상태가 되는 task와 CPU 제어권을 빼앗기는 task의 상태를 변경한다.

22 **context_swap()**을 호출하여, 새로 RUNNING상태가 된 task가 CPU를 사용하도록 한다.

free-ppc/kern/kern.c

```
01  tcb_t*   pick(void *parg)
02  {
03      int      s = splhi();
04      int      grp_index;
05      int      pri_index;
06      tcb_t    *ptcb;
07      if (parg)
08      {
09          sem_t    *psem = (sem_t *)parg;
10          m_FIRST_BIT(grp_index, psem->grp_index);
11          m_FIRST_BIT(pri_index, psem->pri_index[grp_index]);
12          debug("[%s]@pick<-sem_v(): psem:%h, gmask:%h, gidx:%h, pmask:%h, pidx:%h\n",
13                t_name(), psem, s_grp_ready_mask, grp_index,
14                s_pri_ready_masks[grp_index], pri_index);
15      }
16      else
17      {
18          m_FIRST_BIT(grp_index, s_grp_ready_mask);
19          m_FIRST_BIT(pri_index, s_pri_ready_masks[grp_index]);
20          debug("[%s]@pick<-reschedule(): gmask:%h, gidx:%h, pmask:%h, pidx:%h\n",
21                t_name(), s_grp_ready_mask, grp_index,
22                s_pri_ready_masks[grp_index], pri_index);
23      }
24      ptcb = gp_pri_list[grp_index][pri_index];
25      splx(s);
26      return ptcb;
27  }   /* end of pick */
28  #define m_FIRST_BIT(result, mask) \
29          { \
30              int      i; \
31              u_int    s_mask = mask; \
```

```
32              for (i = 31; i >= 0; i--) \
33              { \
34                  if ((s_mask) & 0x80000000) \
35                      break; \
36                  else \
37                      (s_mask) <<= 1; \
38              } \
39              (result) = i; \
40          }
```

free-ppc/kern/kern.c

01 **pick()** 함수는 RUNNING상태가 될 task를 결정하는 일을 담당하며, 앞에서 보인 **reschedule()**과 뒤에서 설명할 **sem_v()**에서 본 함수를 사용(호출)한다.

07-15 **sem_v()**에서 호출한 경우이며, 세마포어를 기다리고 있는 task들 중에 가장 높은 priority task를 골라 그 결과를 **grp_index**와 **pri_index**에 각각 상/하위 5 bit로 나누어 저장한다.

16-23 **reschedule()**에서 호출한 경우이며, RUNNING상태 task와 READY상태에 있는 task들(Ready list에 비트가 set되어 있는) 중에서 가장 높은 priority를 찾아 그 결과를 **grp_index**와 **pri_index**에 저장한다.

24,26 **spawn()**에 의해서 생성된 task들의 TCB structure 주소가 priority값을 인덱스로 하여 **gp_pri_list[][]**에 저장되어 있으며, 앞에서 계산된 **grp_index**, **pri_index**를 이용하여 RUNNING상태가 될 task의 TCB structure 주소를 리턴한다.

28-40 MSB(Most Significant Bit)에서 LSB(Least Significant Bit) 방향으로, 처음 1로 set되어 있는 비트의 위치를 찾아내는 매크로이다.

[예] priority 100(0x64: 011 00100), 195(0xC3: 110 00011), 200(0xC8: 110 01000)인 3개의 task가 READY상태에 있고, priority 300인 task가 RUNNING상태에 있다가 CPU 사용을 포기(SUSPENDED상태로 됨)해서 **reschedule()**이 호출된다면,

현 상태에서의 값은 다음과 같으며,

```
s_grp_ready_mask = 0b0000 0000 0000 0000 0000 0000 0100 1000
s_pri_ready_masks[3] = 0b0000 0000 0000 0000 0000 0000 0001 0000
s_pri_ready_masks[6] = 0b0000 0000 0000 0000 0000 0001 0000 1000
```

❶ **reschedule()**에서 **pick()**을 호출하고,
❷ **pick()** 함수 line18에 의해서 **grp_index**에는 6이 저장되며
❸ **pick()** 함수 line19에 의해서 **pri_index**에는 8이 저장되고,
❹ **pick()** 함수 line24/26에 의해서 **gp_pri_list[6][8]**에 저장되어 있는 TCB structure 주소가 리턴된다.

본 구현의 문제점은 task 개수 및 priority 값에 따라서 **m_FIRST_BIT()**의 수행 시간이 달라지게 되고, 이에 의해서 task scheduling time이 deterministic하지 않게 되는 것이다. 68030 CPU는 **bfffo**라는 instruction을 제공하여 deterministic하게 **m_FIRST_BIT()** 기능을 제공하지만, PowerPC의 경우 이러한 instruction을 지원하지 않는 이유로 C code로 구현하게 되었고, 이로 인해서 Non-deterministic 문제가 생긴 것이다. MicroC/OS-II에 이 문제에 대한 해결 방법이 제시되어 있으며, 그 구성을 간단히 설명하도록 하겠다.

MicroC/OS-II는 총 64개(0-63) task priority를 지원하며, 이 중 상위 3비트 정보는 **u_char OSRdyGrp**에 저장되고, 하위 3비트는 **u_char OSRdyTbl[8]**에 저장된다. 즉, FREE와 동일한 구조를 가지면서 priority level이 1024개에서 64개로 줄었고, 이로 인해서 이를 표현하는 변수의 크기도 32비트 단

위(u_int s_grp_ready_mask, s_pri_ready_masks[32])에서 8비트 단위(u_char OSRdyGrp, u_char OSRdyTbl[8])로 줄어든 것이다. FREE의 pick()과 동일한 기능 수행을 위해서 제공되는 MicroC/OS-II 함수는 다음과 같다.

OS_CORE.C(uC/OS-II)

```
01  INT8U const OSUnMapTbl[] = {
02      0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
03      4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
04      5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
05      4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
06      6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
07      4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
08      5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
09      4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
10      7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
11      4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
12      5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
13      4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
14      6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
15      4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
16      5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
17      4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0
18  };

19  void    OSSched(void)
20  {
21      INT8U x, y;

        ...

22      /* Get pointer to highest priority task ready to run  */
23      x = OSUnMapTbl[OSRdyGrp];
24      y = OSUnMpaTbl[OSRdyTbl];
25      OSPrioHighRdy = (INT8U)((x << 3) + y);

        ...

26  }
```

OS_CORE.C(uC/OS-II)

01-18 m_FIRST_BIT() 매크로 기능을 대체하는 테이블이며, 다음과 같은 코드에 의해서 생성되어진다.

```
void    main(void)
{
    uint    i, in, tbl[] = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80};

    printf("0x0 -> %d\n", 0);
    for (in = 0x00; in <= 0xFF; in++) {
        for (i = 0; i < 8; i++) {
            if ((in & tbl[i]) != 0) {
                printf("0x%x -> %d\n", in, i);
                break;
            }
        }
    }
}   /* end of main */
```

22-23 pick() 함수의 line18/19에 해당한다. 즉, pick()의 grp_index가 본 함수의 x가 되고, pri_index가 y에 해당하게 된다.

free-ppc/ppc/cpu.h

```
01  typedef struct
02  {
03      u_int   r[32];      /* General Purpose Registers R0~R31 */
04      u_int   srr[2];     /* Save/Restore Registers SRR0, SRR1 */
05      u_int   ctr;        /* Count Register */
06      u_int   xer;        /* Interget Exception Register */
07      u_int   cr;         /* Condition Register */
08      u_int   lr;         /* Link Register */
09      u_int   msr;        /* Machine State Register */
```

```
10  } context_t;
```

free-ppc/ppc/cpu.h

01-10 task간 Context Switch 발생시에 본 PowerPC register의 값이 task의 TCB structure에 저장/복귀된다. **context_t**의 필드는 **cpu.h**에 정의된 레지스터 정의(**R0_OFFSET ~ MSR_OFFSET**)와 그 위치가 정확히 일치해야 한다.

free-ppc/ppc/locore.s

```
01  context_swap:
02      stw     r0, R0_OFFSET(r3)
03      stw     r1, R1_OFFSET(r3)
04      stw     r2, R2_OFFSET(r3)
05      stmw    r5, R5_OFFSET(r3)

06      mfspr   r0, SRR0
07      stw     r0, SRR0_OFFSET(r3)
08      mfspr   r0, SRR1
09      stw     r0, SRR1_OFFSET(r3)
10      mfctr   r0
11      stw     r0, CTR_OFFSET(r3)
12      mfxer   r0
13      stw     r0, XER_OFFSET(r3)
14      mfcr    r0
15      stw     r0, CR_OFFSET(r3)
16      mflr    r0
17      stw     r0, LR_OFFSET(r3)
18      mfmsr   r0
19      stw     r0, MSR_OFFSET(r3)

20      lwz     r0, LR_OFFSET(r4)
21      mtlr    r0
22      lwz     r0, CR_OFFSET(r4)
23      mtcr    r0
24      lwz     r0, XER_OFFSET(r4)
25      mtxer   r0
26      lwz     r0, CTR_OFFSET(r4)
27      mtctr   r0
28      lwz     r0, SRR1_OFFSET(r4)
29      mtspr   SRR1, r0
30      lwz     r0, SRR0_OFFSET(r4)
31      mtspr   SRR0, r0

32      lmw     r5, R5_OFFSET(r4)
33      lwz     r2, R2_OFFSET(r4)
34      lwz     r1, R1_OFFSET(r4)

35      lwz     r0, MSR_OFFSET(r4)
36      mtmsr   r0
37      lwz     r0, R0_OFFSET(r4)

38      blr
        /* end of context_swap */
```

free-ppc/ppc/locore.s

01 **reschdule()**에서 본 함수를 호출하며, r3는 CPU 제어권을 빼앗길 task의 TCB structure의 context_t 필드의 주소가, r4는 CPU를 사용할 task의 TCB structure의 context_t 필드의 주소를 가리킨다. (**context_swap(&pold_task->context, &pnew_task->context)**)

02-19 CPU 사용을 포기할 task가 사용하던 PowerPC 레지스터 값을 r3가 가리키는 TCB structure context_t에 저장한다. (그림 8 참조)

20-37 새로 CPU를 사용할 task가 이전에 사용했던 PowerPC 레지스터 값(r4가 가리키는 TCB structure context_t에 저장되어 있음)을 PowerPC 레지스터로 복귀한다. (그림 8 참조)

38 **blr** 실행 시점에서 새로운 task가 CPU를 사용하게 된다.

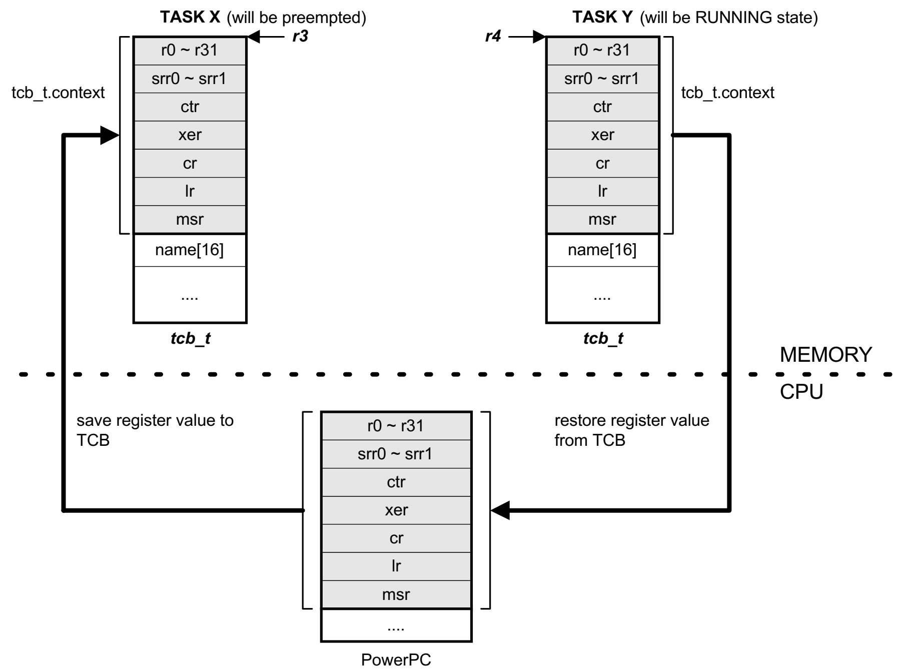


Figure 8: context_swap()

## 4.6 Semaphore

free-ppc/kern/sem.c

```
01  void     sem_p(sem_t *psem)
02  {
03      int      s = splhi();
04      u_int    grp_index, pri_index;
05      if (g_interrupt_nest > 0)
06      {
07          kprintf("[E:%s]@sem_p: can not use in ISR\n", t_name());
08          splx(s);
09          return;
10      }
11      if (--psem->count < 0)
12      {
13          grp_index = gp_cur_task->grp_index;
14          pri_index = gp_cur_task->pri_index;
15          psem->grp_index              |= (1 << grp_index);
16          psem->pri_index[grp_index]   |= (1 << pri_index);
17          debug("[%s]@sem_p: \"%s(%h)\" yields\n",
18                t_name(), gp_cur_task->name, gp_cur_task);
19          suspend(gp_cur_task);
20          reschedule();
21      }
22      splx(s);
23  }   /* end of sem_p */
```

free-ppc/kern/sem.c

01 sem_create()에 의해서 생성된 세마포어를 얻어오는 함수이다. sem_p()의 파라미터를 통해서 여러 개의 세마포어 중에 어떤 세마포어인지를 구별한다.

05-10 세마포어가 없는 상황에서 sem_p()를 호출하면, 이 함수를 호출한 task는 SUSPENDED상태로 되는데, ISR은 SUSPENDED상태가 될 수 없기 때문에 에러를 리턴한다.

11 세마포어 카운드를 먼저 감소하고 이 값이 음수일 경우(세마포어가 없는 상태), 본 함수를 호출한

task를 SUSPENDED상태로 만든다.

13-16 SUSPENDED상태가 될 task의 priority를 세마포어 structure에 저장한다. 이 정보는 **sem_v()**에서 사용된다.

19-20 **suspend()**를 호출하여 현재 RUNNING상태의 task를 SUSPENDED상태로 만들고, **reschedule()**을 통해서 READY상태에 있는 task들 중에 priority가 가장 높은 task가 CPU를 사용하도록 한다.

free-ppc/kern/sem.c

```
01  int     sem_v(sem_t *psem)
02  {
03      tcb_t   *ptask;
04      u_int   grp_index, pri_index;
05      int     s;

06      s = splhi();
07      if (psem->count++ < 0)    /* somebody waiting */
08      {
09          /* pick highest priority task waiting on this semaphore.
10             this task must be SUSPENDED stat */
11          ptask = pick((void *)psem);

12          grp_index = gp_cur_task->grp_index;
13          pri_index = gp_cur_task->pri_index;

14          psem->pri_index[grp_index] &= ~(1 << pri_index);
15          if (psem->pri_index[grp_index] == 0)
16              psem->grp_index &= ~(1 << grp_index);

17          if (g_interrupt_nest == 0 &&
18             (ptask->grp_index > grp_index && ptask->pri_index > pri_index))
19          {
20              debug("[%s]@sem_v: \"%s(%h)\" made to run\n",
21                     t_name(), ptask->name, ptask);
22              ready(ptask);
23              reschedule();
24          }
25          else
26          {
27              debug("[%s]@sem_v: \"%s(%h)\" made to ready\n",
28                     t_name(), ptask->name, ptask);
29              if (g_interrupt_nest > 0)
30                  g_need_resched++;
31              ready(ptask);
32          }
33          splx(s);
34          return 1;
35      }
36      else/* nobody waiting */
37      {
38          splx(s);
39          return 0;
40      }
41  }   /* end of sem_v */
```

free-ppc/kern/sem.c

01 이전에 **sem_p()**를 통해서 얻었던 세마포어를 반환하는 함수로, 세마포어를 얻었던 task가 반드시 세마포어를 반환해야 할 필요는 없다. 즉, task1이 **sem_p()**를 통해서 세마포어를 얻었다 해도, 이 세마포어를 task2나 ISR에서 **sem_v()**를 통해서 반환할 수 있다.

07 세마포어 카운트가 음수라는 것은 본 세마포어를 기다리는 SUSPENDED상태의 task가 있다는 것을 의미한다.

09-16 본 세마포어를 기다리는 task들 중에 priority가 가장 높은 task를 선택한다. 또한 세마포어 structure에서 본 task에 해당하는 비트를 clear하여, 더 이상 세마포어를 기다리고 있지 않음을 표시한다.

17-32 **sem_v()**는 task나 ISR에서 모두 호출할 수 있는 함수이며, 다음과 같은 3가지 경우를 고려해야 한다.

❶ task에서 **sem_v()**를 호출했고, RUNNING상태의 task보다 세마포어를 기다리던 task의 priority가 높은 경우: **ready()**를 통해서 세마포어를 기다리던 task를 READY상태로 만들고, **reschedule()**을 호출하여 세마포어를 기다리던 task가 CPU를 사용할 수 있도록 한다.
❷ task에서 **sem_v()**를 호출했고, RUNNING상태의 task보다 세마포어를 기다리던 task의 priority가 낮은 경우: RUNNING상태의 task보다 priority가 낮기 때문에 **reschedule()**을 호출한다 해도 아직 CPU를 사용할 수 없기 때문에 **reschedule()**을 호출하지 않는다.
❸ ISR에서 **sem_v()**를 호출한 경우: **g_interrupt_nest**가 양수임을 보고 ISR에서 호출했음을 알 수 있으며, **reschedule()**를 호출하는 대신에 **g_need_resched**를 증가시켜 ISR 종료 시점(**trap_catch()**)에서 **reschedule()**을 호출하도록 한다.

## 4.7 Interrupt

하드웨어 인터럽트는 비동기적(asynchronous)으로 발생한다. 즉, 프로그램이 실행 되는 중에 어느 시점에 인터럽트가 발생할 것이지 예측할 수 없는 것이다. 프로그램이 수행된다는 것은 PowerPC의 레지스터들(General/Special purpose register)를 사용한다는 의미이고, 따라서 인터럽트가 발생하면 ISR(Interrupt Service Routine) 코드에서 레지스터값을 수정하기 전에, 프로그램이 사용하던 레지스터들을 스택에 저장하고 ISR 종료 직전에 저장했던 값들을 복원해야 한다.

free-ppc/ppc/locore.s

```
01  trap_default:
02      subi    r1, r1, STACK_FRAME_SZ
03      stw     r0, R0_OFFSET(r1)
04      stw     r1, R1_OFFSET(r1)
05      stmw    r2, R2_OFFSET(r1)

06      mfspr   r0, SRR0
07      stw     r0, SRR0_OFFSET(r1)
08      mfspr   r0, SRR1
09      stw     r0, SRR1_OFFSET(r1)
10      mfctr   r0
11      stw     r0, CTR_OFFSET(r1)
12      mfxer   r0
13      stw     r0, XER_OFFSET(r1)
14      mfcr    r0
15      stw     r0, CR_OFFSET(r1)
16      mflr    r0
17      stw     r0, LR_OFFSET(r1)
18      mfmsr   r0
19      stw     r0, MSR_OFFSET(r1)

20      /* trap_catch(exc_num); */
21      mfspr   r3, SPRG3

22      /* trap_catch() function's prologue saves LR in this frame */
23      subi    r1, r1, C_FRAME_SZ

24      bl      trap_catch

25      /* deallocate C frame */
26      addi    r1, r1, C_FRAME_SZ

27      lwz     r0, LR_OFFSET(r1)
28      mtlr    r0
29      lwz     r0, CR_OFFSET(r1)
30      mtcr    r0
31      lwz     r0, XER_OFFSET(r1)
32      mtxer   r0
33      lwz     r0, CTR_OFFSET(r1)
34      mtctr   r0
35      lwz     r0, SRR1_OFFSET(r1)
36      mtspr   SRR1, r0
37      lwz     r0, SRR0_OFFSET(r1)
```

```
38      mtspr    SRR0, r0

39      lmw      r2, R2_OFFSET(r1)
40      lwz      r1, R1_OFFSET(r1)

41      lwz      r0, MSR_OFFSET(r1)
42      mtmsr    r0
43      lwz      r0, R0_OFFSET(r1)

44      addi     r1, r1, STACK_FRAME_SZ
45      rfi
        /* end of trap_default */
```

free-ppc/ppc/locore.s

02-19 trap_catch() 함수(코드)에서 레지스터 값을 수정하기 전에 레지스터 값들을 스택에 저장한다.

22-26 trap_catch()를 호출하기 전에 C 코드를 위한 스택 프레임을 생성하고, 호출 후에 본 프레임을 제거한다. 본 스택 프레임 생성 및 제거에 대한 사항은 PowerPC EABI(Embedded Application Binary Interface) 문서를 참조하기 바란다.

27-44 스택에 저장했던 레지스터 값들을 복원한다.

그림 9는 스택에 저장되는 레지스터의 형상을 나타낸 것이다.

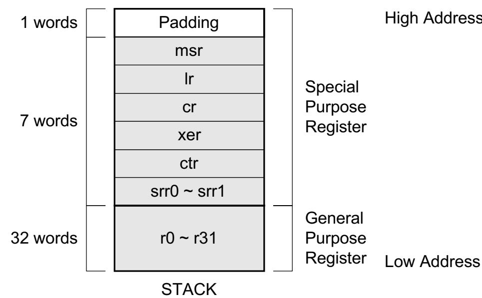


Figure 9: Interrupt Stack Frame

free-ppc/ppc/trap.c

```
01  int      trap_catch(int exc_num)
02  {
03      u_long  immr;
04      int     vec, civr;

05      g_interrupt_nest++;
06      g_exc_num = exc_num;

07      switch (exc_num)
08      {
09          case 0x500:
10              immr = get_immr() & 0xffff0000;
11              vec = m_WORD(immr + SIVEC) >> 26;
12              if (vec == SIU_INTR_LEV4)
13              {
14                  m_HWRD(immr + CIVR) |= 0x0001;
15                  asm("eieio");
16                  civr = m_HWRD(immr + CIVR) >> 11;
17                  vec = CPM_INTR_ERR + civr;
18                  if (s_trap_funcs[vec] != 0)
19                      (*s_trap_funcs[vec])();
20              }
21              break;
22          case 0x900:
23              timer_handle();
24              break;
25      }
26      g_interrupt_nest--;
27      if (g_need_resched > 0)
28      {
29          g_need_resched--;
30          reschedule();
```

```
31        }
32        return 0;
33  }     /* end of trap_catch */
```

free-ppc/ppc/trap.c

05,26 **g_interrupt_nest**를 증가/감소하여 Interrupt nesting(ISR 종료전에 다시 인터럽트가 발생하는 상황) 카운트를 유지한다. 만약 본 값이 양수일 경우에는 ISR 상태임을 나타낸다.

09,22 **exc_num** 0x500은 PowerPC External Interrupt number이고 0x900은 PowerPC Decrementer Interrupt number이다.

26-31 **s_trap_funcs[vec]()** 함수 내에서 **sem_v()**를 호출하면 **g_need_resched**가 양수의 값을 가지게 되고, 이 경우에 ISR에서 **reschedule()**를 호출한다.

## Enable & Disable Interrupt

Kernel 코드의 대부분은 수행 도중에 다른 task에 의해서 CPU 제어권을 빼앗기지 않도록 다음과 같은 구조를 가진다.

```
int s = splhi();
/* start "critical regioin" */
...
/* end "critical regioin" */
splx(s);
```

free-ppc/ppc/locore.s

```
01  splhi:
02      mfmsr    r3
03      mtspr    81, 0
04      blr
        /* end of splhi */
```

free-ppc/ppc/locore.s

03 PowerPC의 MSR[EE] 비트를 clear하여 인터럽트를 금지한다.

02-04 MSR 레지스터의 변경되기 전의 값을 r3 레지스터에 저장하여, 이 함수(**splhi()**)를 호출한 측(함수)에 넘겨준다. 이 리턴 된 값은 **splx()**의 아규먼트로 사용되어 **splhi()** 호출 이전의 MSR로 복구한다.

free-ppc/ppc/locore.s

```
01  splx:
02      mtmsr    r3
03      blr
        /* end of splx */
```

free-ppc/ppc/locore.s

02 **splx()**의 아규먼트로 넘어온 값(바로 앞에서 설명하였듯이 **splhi()**에서 MSR 변경 이전의 MSR 값)으로 MSR 레지스터를 변경한다.

# 5. Results

그림 10-15는 FREE용 shell task의 결과를 보인 것이다.

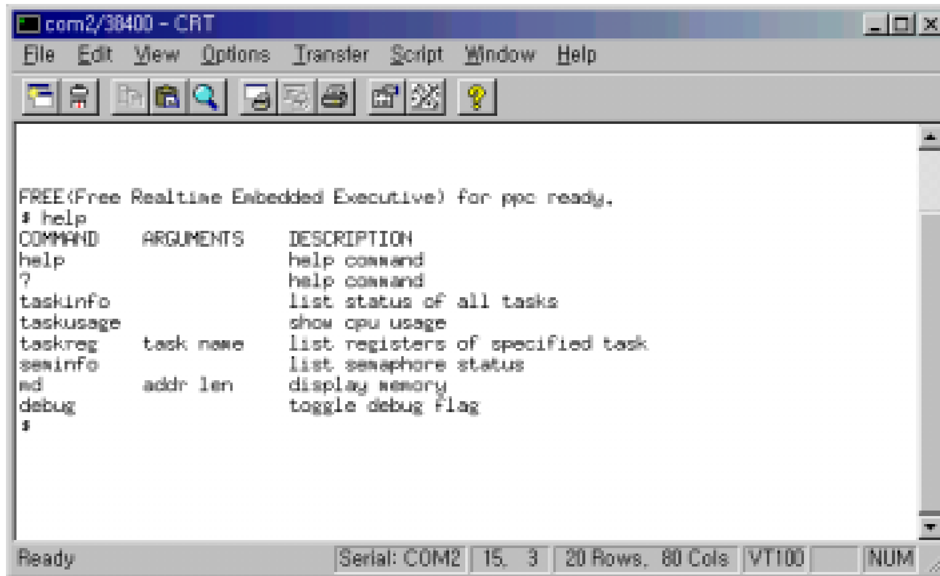


Figure 10: FREE shell

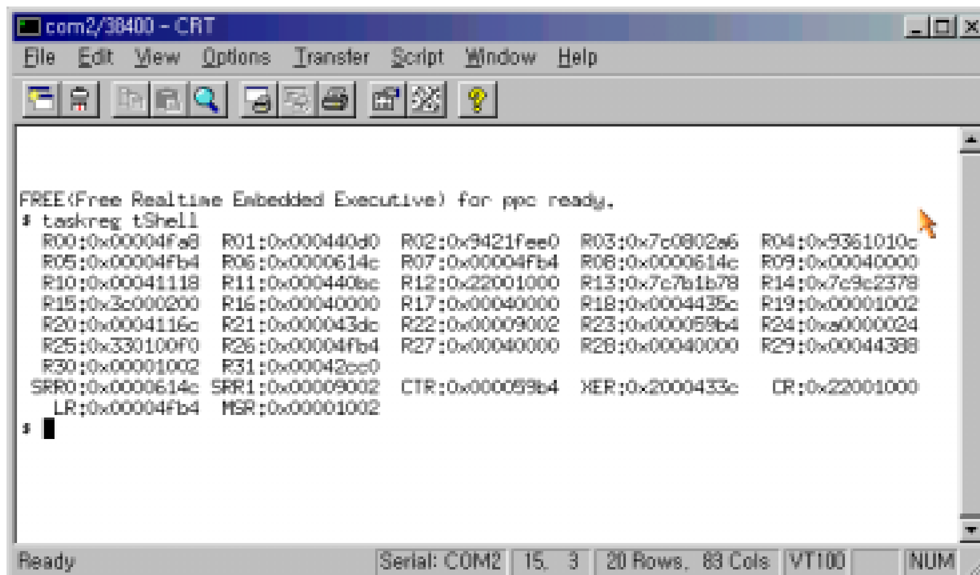


Figure 11: 'taskreg(task register)' command in FREE shell

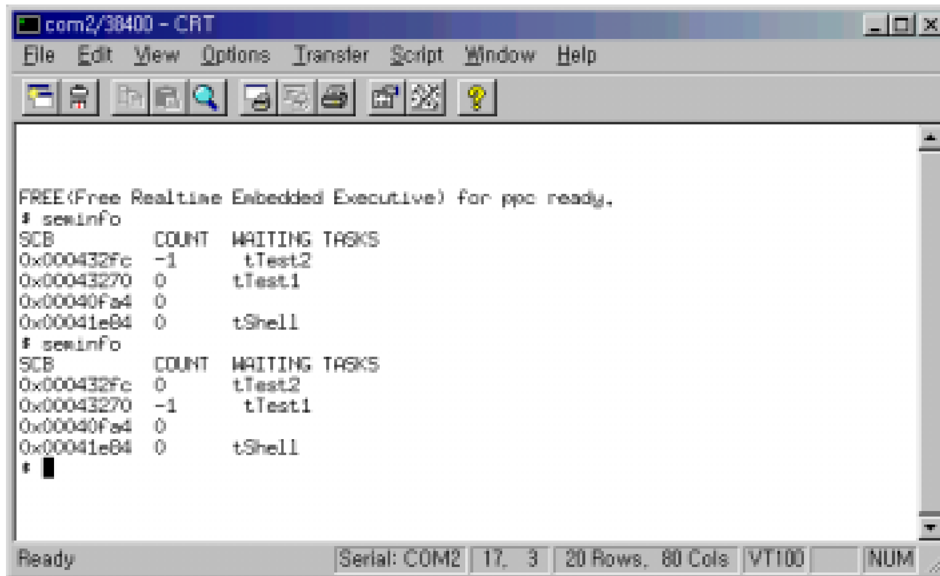


Figure 12: 'seminfo(semaphore information)' command in FREE shell

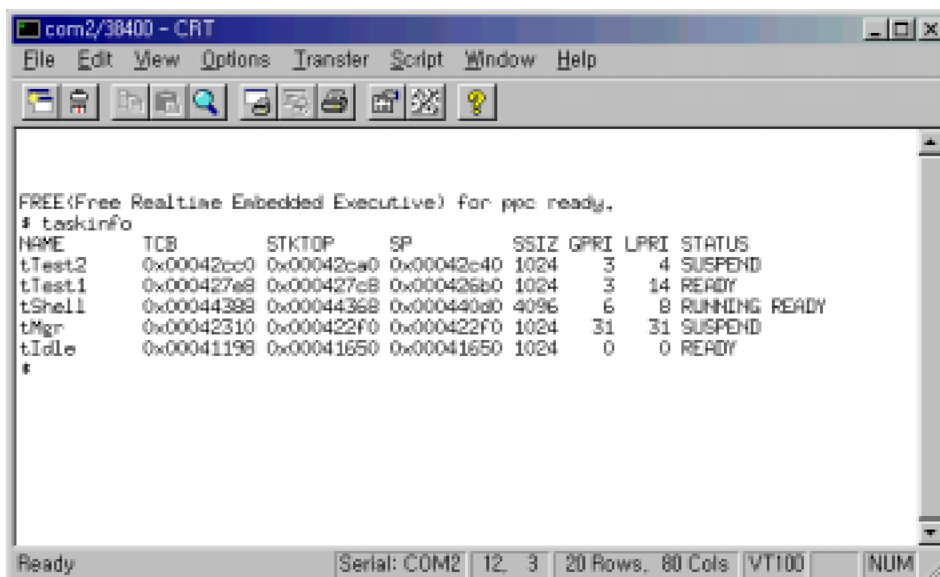


Figure 13: 'taskinfo(task information)' command in FREE shell

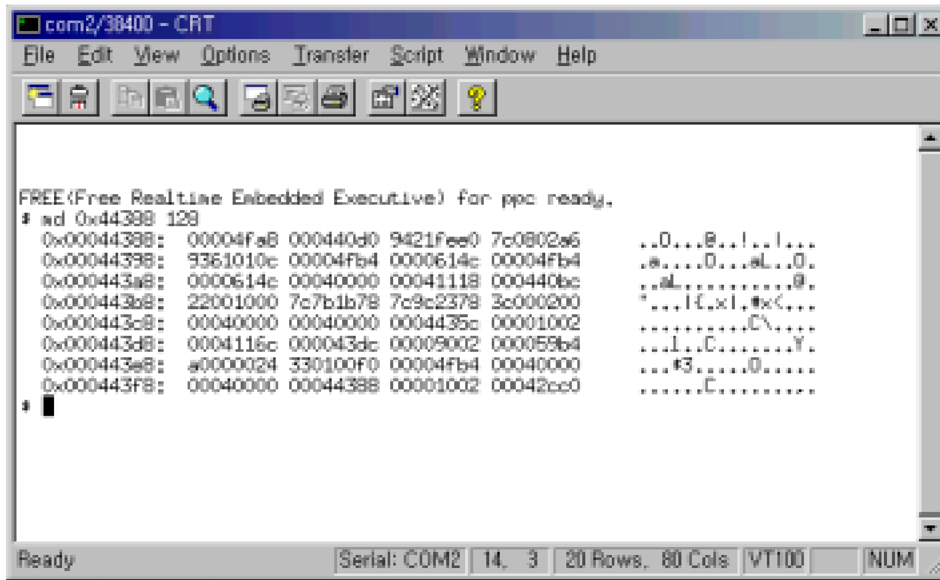


Figure 14: 'md(memory dump)' command in FREE shell

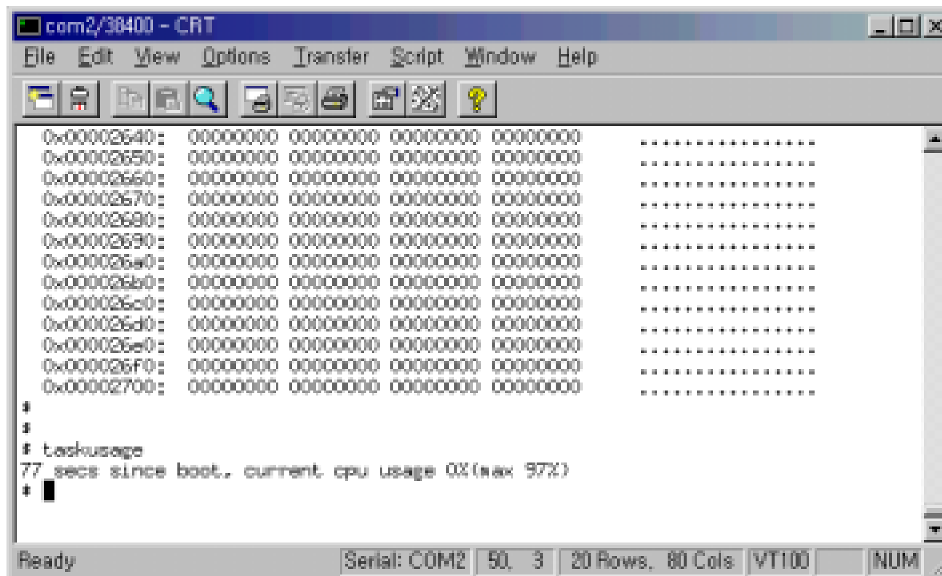


Figure 15: 'taskusage' command in FREE shell

그림 16은 debug mode를 enable시켜, FREE 환경하에서 'tShell', 'tTest1', 'tTest2' task가 어떤 식으로 동작하는지를 보인 것이다.

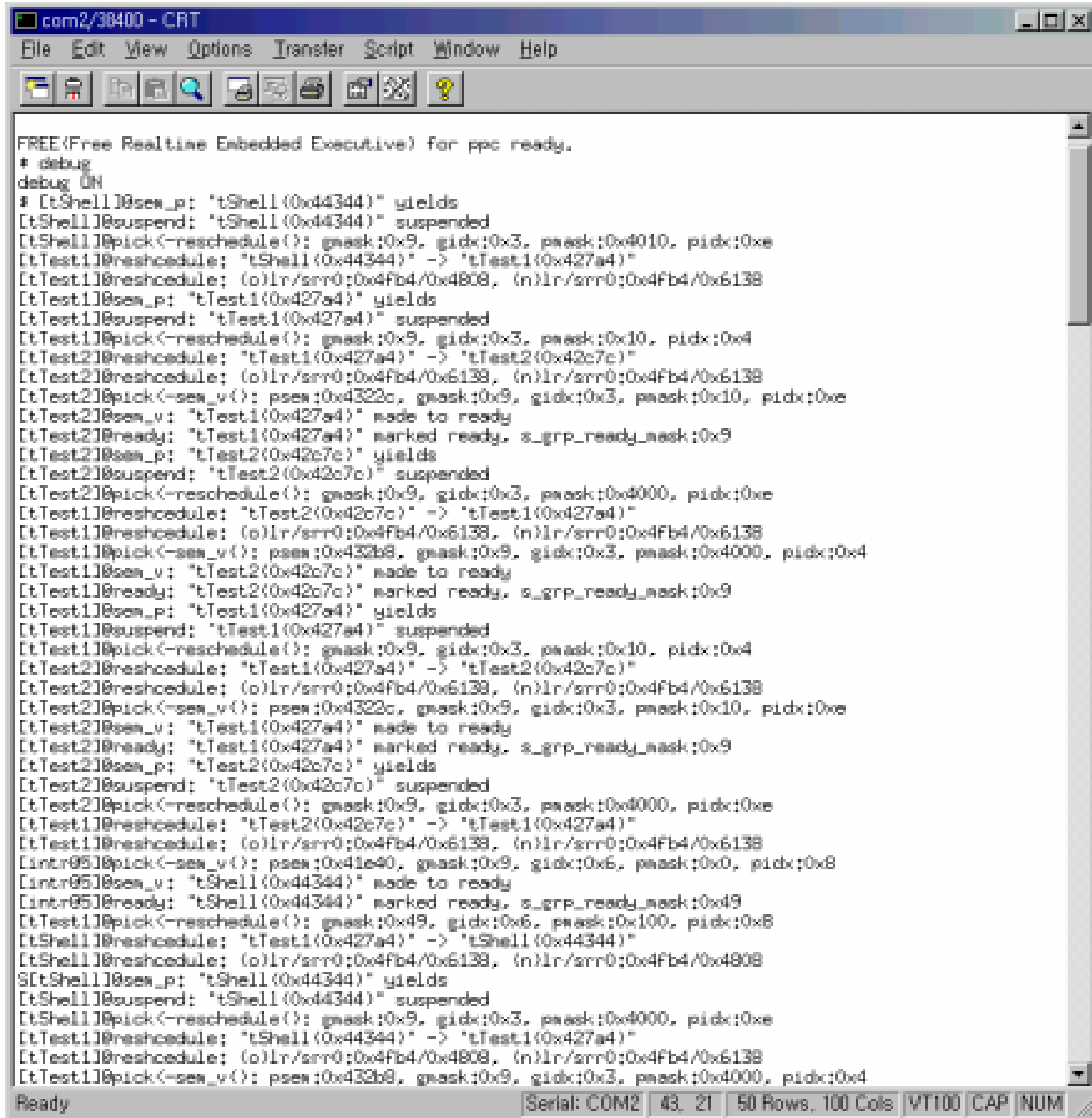


Figure 16: FREE debugging mode

# 6. Conclusion

본 문서에서는 FREE라는 작은 사이즈의 RTOS 코드를 MPC860에 포팅하는 일련의 과정에 대해서 설명하였다. 문서 작업에 있어서 “Context Switch와 Interrupt간의 관계”, “FREE I/O System”에 대한 사항도 정리하려 했지만 시간의 부족함으로 생략하게 되었다. 프로세서에 의존적인 사항들만 수정하려 했던 본 작업이 예상치 않게 그 외에 많은 부분들을 수정하게 되어 Original Source Code와 많은 부분의 모습이 달라지게 되었으며, 본 코드는 최대한 본인의 “C coding standard”에 맞추려고 노력하였다. 아직까지도 많은 부분이 허술한 상태임을 인정하는 바이며, 다시 한번 강조하지만 본 코드를 상용 제품에 탑재하는 일은 절대 권하고 싶지 않다. 마지막으로 MPC860 시스템을 위한 RTOS 제작/포팅 작업에 있어서 필요한 사항들을 정리하면 다음과 같다.

- RTOS basic & advanced concepts
- MPC860 architecture
- PowerPC assembler, register, exception
- PowerPC EABI