Contents

1. Overview...........................................................................................................................................2
2. Background Information..................................................................................................................2
  2.1. About µC/OS-II......................................................................................................................2
  2.2. Task in µC/OS-II.....................................................................................................................2
  2.3. Task Scheduling & Context Switch .......................................................................................3
  2.4. Coding Guidelines for Embedded RTOS..............................................................................4
  2.5. Starting µC/OS-II...................................................................................................................5
  2.6. Setting up the ARMulator .....................................................................................................6
3. Instructions......................................................................................................................................6
  3.1. Building µC/OS-II..................................................................................................................6
  3.2. Porting Program to µC/OS-II..............................................................................................8
  3.3. Building Program with µC/OS-II.........................................................................................11
4. Exercise..........................................................................................................................................13
5. References......................................................................................................................................14
Real-Time OS

1. Overview

This lab is a guide to Real-time Operating System (RTOS) in SoC design. This lab is based on µC/OS-II, a compact but complete RTOS shipped with ARM Firmware Suite (AFS). Internal mechanism of µC/OS-II is beyond the scope of this lab. For more detailed information about µC/OS-II, please refer to the book “MicroC/OS-II, the real-time kernel” by Jean J. Labrosse.

2. Background Information

2.1. About µC/OS-II

- Soft Real-time – tasks are performed as fast as possible
- Portable – runs on architectures ranging from 8-bit to 64 bit
- Scalable – features are configurable at compile time
- Multitasking – support 64 tasks simultaneously; including 8 reserved tasks
- Preemptive – preemptive multi-tasking with priority scheduling
- Kernel Services – provides task, time, memory management API; inter-process communication API; task synchronization API
- Nested Interrupt – up to 255 levels of nested interrupt
- Priority Inversion Problem – does not support priority inheritance
- Not using MMU – no well protected memory space like Unix or Win

2.2. Task in µC/OS-II

- Task is a single instance of program
- Task thinks it has all CPU control itself
- Task has its own stack and own set of CPU registers backup in its stack
- Task is assigned a unique priority (highest 0 ~ lowest 63)
- Task is an infinite loop and never returns
- Task has states (see Figure 1)
- µC/OS-II saves task records in Task Control Block(TCB)
Task States in µC/OS-II:

- Running – task has control of the processor and executing its job
- Ready – task is ready to execute but its priority is less than the running task
- Waiting – task requires the occurrence of an event to continue
- ISR – task is paused because the processor is handling an interrupt
- Dormant – task resides in memory, but not seen by the scheduler

2.3. Task Scheduling & Context Switch

In µC/OS-II, task scheduling is performed on following conditions:

- A task is created/deleted
- A task changes state
  - On interrupt exit
  - On post signal
  - On pending event
  - On task suspension

If the scheduler chooses a new task to run, context switch occurs. First, the context (processor registers) of current running task is saved in its stack. Next, the context of the new task is loaded into the processor. Finally, the processor continues execution. (see Figure 2)
2.4. Coding Guidelines for Embedded RTOS

Coding for a program to run on embedded RTOS is slightly different from coding for PC. In an embedded application, workloads and available resources are known at design time. Hence, the designer should carefully explore the design space and optimize for short latency, small memory footprint, low power…etc.
When writing programs for µC/OS-II, some rules should be noticed:

- **Resources**
  - Use µC/OS-II defined data types for consistency & portability
  - Use statically allocated local variables for preemptive multitasking
  - Use semaphore to protect global variables and resources

- **Data transfer**
  - Inter-task communication can be achieved by mailbox/queue
  - µC/OS-II & user program all run in privileged mode, use share memory with caution!

- **Memory allocation**
  - Use µC/OS-II Kernel API: OSMemCreate(), OSMemGet()

- **Standard C library**
  - Many standard routine works in semi-hosting mode but not in stand-alone mode. (e.g. printf, fopen)

### 2.5. Starting µC/OS-II

µC/OS-II is initialized and started in the main function. The initialization order is important.

1. Initialize ARM target
2. Initialize OS
3. OS create/allocate resources
4. Create an initial task with highest priority
5. Create other user tasks
6. OS start scheduling
7. In the initial task, enable global interrupt
8. the initial task deletes itself
9. Now, all other tasks runs under the control of OS

Note that you must create at least one task before OS start scheduling. Otherwise, you don’t get a chance to start any task and the system remains in idle state.

If an interrupt occurs before OS starts scheduling, the scheduler doesn’t know which tasks to run on interrupt exit and the system crashes. So, we introduce an initial task to
enable global interrupt. This is for safety, but not required.

2.6. Setting up the ARMulator

The ARMulator can function as virtual prototype to various ARM core and development boards. However, the original configuration of ARMulator does not match that of ARM Integrator/AP. To run μC/OS-II on ARMulator, you need to follow the configuration steps in the reference section.

3. Instructions

3.1. Building μC/OS-II

1. Open μC/OS-II project file in C:/lab08/ucos2/ with Code Warrior.
2. Edit OS_CFG.H to customize μC/OS-II.

The configuration of μC/OS-II is done through a number of #define derivatives found in OS_CFG.H. Basically, the default settings in OS_CFG.H work fine. You might want to disable some unused features or decrease some value of settings for more compact memory footprint on final release.

<table>
<thead>
<tr>
<th>Option</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_MAX_EVENTS</td>
<td>default: 20</td>
<td>Maximum number of event control block available. The value should be greater than the total number of mailbox, semaphore and queue required by application.</td>
</tr>
<tr>
<td></td>
<td>range: &gt;=2</td>
<td></td>
</tr>
<tr>
<td>OS_MAX_MEM_PART</td>
<td>default: 10</td>
<td>Maximum partitions to be managed by memory partition manager. Note that OS_MEM_EN must be set to 1 first.</td>
</tr>
<tr>
<td></td>
<td>range: &gt;=2</td>
<td></td>
</tr>
<tr>
<td>OS_MAX_QS</td>
<td>default: 5</td>
<td>Maximum number of queue available in system.</td>
</tr>
<tr>
<td></td>
<td>range: &gt;=2</td>
<td></td>
</tr>
<tr>
<td>OS_MAX_TASKS</td>
<td>default: 62</td>
<td>Maximum number of task can exist at a time. Note that although μC/OS-II can handle 64 tasks but it reserves 2 tasks for itself.</td>
</tr>
<tr>
<td></td>
<td>range: 62~2</td>
<td></td>
</tr>
<tr>
<td>OS_LOWEST_PRIO</td>
<td>default: 63</td>
<td>Specifies the lowest task priority (i.e., highest number) that you intend to use.</td>
</tr>
<tr>
<td></td>
<td>range: 63~1</td>
<td></td>
</tr>
<tr>
<td>OS_TASK_IDLE_STK_SIZE</td>
<td>default: 512</td>
<td>Stack size (in 16-bit entries) for IDLE_TASK. The minimum stack size depends on processor type and deepest interrupt level allowed. It’s better to use default setting.</td>
</tr>
<tr>
<td>OS_TASK_STAT_EN</td>
<td>default: 0</td>
<td>Statistic task computes CPU usage once every second. The priority of statistic task is always set to OS_LOWEST_PRIO-1.</td>
</tr>
<tr>
<td>OS_TASK_STAT_STK_SIZE</td>
<td>default: 512</td>
<td>Stack size (in 16-bit entries) for STAT_TASK. It is suggested to use default value.</td>
</tr>
<tr>
<td>OS_CPU_HOOKS_EN</td>
<td>default: 1</td>
<td>This setting indicates whether hook functions are enabled.</td>
</tr>
</tbody>
</table>
should be included or not. Hook functions are declared in OS_CPU_C.C.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_MBOX_EN</td>
<td>default: 1</td>
<td>This enables or disables code generation of message mailbox service.</td>
</tr>
<tr>
<td>OS_MEM_EN</td>
<td>default: 0</td>
<td>This enables or disables code generation of memory partition manager and its associated data structures.</td>
</tr>
<tr>
<td>OS_Q_EN</td>
<td>default: 1</td>
<td>This enables or disables code generation of message queue service.</td>
</tr>
<tr>
<td>OS_SEM_EN</td>
<td>default: 1</td>
<td>This enables or disables code generation of semaphore manager.</td>
</tr>
<tr>
<td>OS_TASK_CHANGE_PRIO_EN</td>
<td>default: 0</td>
<td>This indicates whether tasks can change priority at runtime.</td>
</tr>
<tr>
<td>OS_TASK_CREATE_EN</td>
<td>default: 1</td>
<td>This enables support for standard task creation function. You can choose either standard or extended version of task creation function. If you wish, you can use both.</td>
</tr>
<tr>
<td>OS_TASK_CREATE_EXT_EN</td>
<td>default: 0</td>
<td>This enables support for extended task creation function. Extended version delivers more powerful control over task.</td>
</tr>
<tr>
<td>OS_TASK_DEL_EN</td>
<td>default: 0</td>
<td>This enables or disables task deletion capability.</td>
</tr>
<tr>
<td>OS_TASK_SUSPEND_EN</td>
<td>default: 1</td>
<td>This enables or disables task suspension capability.</td>
</tr>
<tr>
<td>OS_TICKS_PER_SEC</td>
<td>default: 200</td>
<td>This constant specifies the rate at which OSTimeTick() is called. Better leave this untouched.</td>
</tr>
</tbody>
</table>

Table 1  uC/OS-II configuration options in OS_CFG.H

3. In target settings dialog, turn to Language Settings > ARM C Compiler > Preprocessor. Add INTEGRATOR to List of #DEFINEs to generate specific code for Integrator. (see Figure 3)
4. At last, add 2 directories to the Access Path so header files could be found. In target settings dialog, select **Target > Access Paths**. Add 
${(AFS\_ROOT)}/AFSv1\_4/Source/uHAL/h/ \text{ and} 
${(AFS\_ROOT)}/AFSv1\_4/Source/uHAL/Boards/INTEGRATOR/ \text{ as shown in Figure 4.} 

![Figure 4 Adding access path for ARM uHAL library](image)

5. Press the **Make** button, µC/OS-II library should be built successfully. A static library **ucos2.a** is created. Check the file in your working directory.

### 3.2. Porting Program to µC/OS-II

1. Open the project “**eg1**” at **C:\lab08\eg1** with CodeWarrior.

2. The original code of eg1.c is listed in **Figure 5**. This program asks user for name and age, then it prints greeting message.
```c
#include <stdio.h>

int main(void)
{
    char name[64];
    int age;
    printf("please enter your name: ");
    scanf("%s", name);

    do
    {
        printf("please enter your age: ");
        scanf("%d", &age);
    }while(age < 0);

    printf("Hello, %s. Nice to meet you!
Your age is %d.
", name, age);
    return(0);
}
```

**Figure 5 Original code of eg1.c**

3. In order to port the program to µC/OS-II, include "includes.h" in eg1.c. The header file is an interface for µC/OS-II. (see Figure 6)

```c
#include "includes.h" /* uC/OS-II interface */
#include <stdio.h>
```

**Figure 6 Including µC/OS-II interface in your code**

4. Change the function name from main() to Task1() as shown in Figure 7. In addition, change the return type from int to void because a task never returns. Remove the return statements, too.

```c
void Task1(void *pdata)
{
    ...
    return(0); <= remove this!
}
```

**Figure 7 Changes from Main() to Task1()**

5. A task must receive an argument of type (void*), so change argument list from void to void *pdata as in Figure 7. The purpose of this pointer is to pass initialization value to task.

```c
void Task1(void *pdata)
{
    //local variable declaration
    for(;;)
    {
        // user code
        ..."OSTimeDly(100);
    }
}
```
6. A task is an infinite loop, so wrap the codes with a for loop (see Figure 8). Remember, all task should call at least one kernel service in its code body. Otherwise, the multitasking mechanism of μC/OS-II will not work. You can call OSTimeDly() service to pause for a while after each round of processing, allowing lower priority task to execute.

7. Insert a new main function as shown in Figure 10 at the bottom of eg1.c. In the main function, create an instance for Task1.

```c
OSTaskCreate(void (*task)(void* pd), void *pdata, OS_STK *ptos, INTU8 prio)
Arguments:
*task ⇒ pointer to the task’s code
*pdata ⇒ pointer for passing arguments
*ptos ⇒ pointer to top of stack
prio ⇒ unique priority for each task. smaller number means higher priority
```

```c
int main(int argc, char *argv[])
{
    /* do target (uHAL based ARM system) initialization */
    ARMTargetInit();

    /* needed by uC/OS */
    OSInit();

    /* create the tasks in uC/OS */
    OSTaskCreate(Task1, (void *)0, (void *)&Stack1[STACKSIZE - 1], 3);

    /* Start the (uHAL based ARM system) system running */
    ARMTargetStart();

    /* start the game */
    OSStart();

    /* never reached */
    return(0);
}
```

8. Insert the following code near the top of eg1.c to create a stack for Task1. Each task must have its own stack. The actual stack size needed depends on processor type, depth of interrupt allowed and the work your task is running...etc. System crashes on stack overflow. So, it’s better to allocate a bigger stack first than try to decrease the value. (see Figure 11)
/* allocate memory for tasks' stacks */

#ifdef SEMIHOSTED
#define STACKSIZE (64+SEMIHOSTED_STACK_NEEDS)
#else
#define STACKSIZE 64
#endif

OS_STK Stack1[STACKSIZE];

Figure 11 Define stack size and create resources for task

9. Finally, eg1.c is an executable program at µC/OS-II.

3.3. Building Program with µC/OS-II

1. Open the project “eg1” at C:/lab08/eg1/ with CodeWorrior.

2. Add µC/OS-II as a sub-project. This enables automatic rebuilt of sub-project whenever necessary. This approach is more flexible than add the pre-compiled ucos2.a library file. (see Figure 12)

------ Note ------
When adding sub-projects, a popup message might appear indicating that some Access Path is added. This is ok.
---------------------

3. Add ARM uHAL library as a sub-project. The project file is located in ${AFS_ROOT}\Source\uHAL\Build\Integrator.b\uHALlibrary.mcp. (see Figure 12)

------ Note ------
The uHAL library is board specific. Choose the project file that matches your development board. We choose “Integrator.b” for Integrator/AP.
---------------------
4. Now, specify which target to build and link. In project window, click the **Target** tab to display the Targets view for the project. Then, click the **plus** sign next to a build target containing the subproject to expand the hierarchy. Each build target in the subproject is listed in the hierarchy (see **Figure 13**).

![Figure 13 build target hierarchy view](image)

5. Click on the **Target** icon next to the subproject build targets you want to build along with the main project. The CodeWarrior IDE displays an arrow and target icon for selected build targets (see **Figure 14**).

![Figure 14 Select target to build and link with](image)

6. Click in the **Link Column** next to the subproject build targets. Select the target you want to link with the main project (see **Figure 14**).
------ Note ------
You are free to create link dependencies to any of the build targets in a subproject. However, semi-hosted target must be selected for uHAL in order to support debugging with AXD.

------ Note ------
The µC/OS-II shipped with AFS has been tested, so you can select Release target for to focus on debugging user application only.

7. Define SEMIHOSTED for programs to run in semi-hosted mode. In semi-hosted mode, an extra space of 1K bytes is needed for stack.

8. Build the main project. An executable file that contains both user application and operating system will be created.

9. Press Run button, you can see programs running on µC/OS-II in AXD console window. (see Figure 15)

![Console](image)

*Figure 15 Application running in AXD console*

### 4. Exercise

Write an ID checking engine. The checking rule is in the reference section.

User input:
- The ID numbers

Program output:
5. References

Information about µC/OS-II

♦ “MicroC/OS-II, the real-time kernel” (ISBN: 0-87930-543-6)

ID Checking Rules

- ID number comes in a 10-digit set. The ID starts with an alphabet, followed by 9 digits of numeral.
- Check the first numeral, it should be either “1” or “2”.
- Transform the alphabet into 2 digits. Use Figure 16 for transformation.

```
ID   A 1 2 3 4 5 6 7 8 9
     1 0 1 2 3 4 5 6 7 8 9
```

```
A   10 11 12 13 14 15 16 17 18 19 20 21
B   22 23 24 25 26 27 28 29 30 31 32 33
C   N O P Q R S T U V W X Y Z
D
E
F
G
H
I
J
K
L
M
```

*Figure 16 Transform table for alphabet*

- Multiply each digit with its weighting and sum them up.

```
  1 0 1 2 3 4 5 6 7 8 9
      1 9 8 7 6 5 4 3 2 1 1
Digit
Weighting
```
(1×1 + 0×9 + 1×8 + 2×7 + 3×6 + 4×5 + 5×4 + 6×3 + 7×2 + 8×1 + 9×1) = 130

- The summation of a valid ID should be devisable by 10.

130 mod 10 = 0 ⇒ valid