

Application Note **76**

TCP/IP RTOS Integration Case Study

Document number: ARM DAI 0076

Issued: April 1999

Copyright ARM Limited 1999

ARM

Application Note 76

TCP/IP RTOS Integration Case Study

Copyright © 1999 ARM Limited. All rights reserved.

Release information

The following changes have been made to this Application Note.

Change history

Date	Issue	Change
April 1999	A	First release

Proprietary notice

ARM, the ARM Powered logo, Thumb and StrongARM are registered trademarks of ARM Limited.

The ARM logo, AMBA, Angel, ARMulator, EmbeddedICE, ModelGen, Multi-ICE, ARM7TDMI, ARM9TDMI, TDMI and STRONG are trademarks of ARM Limited.

All other products, or services, mentioned herein may be trademarks of their respective owners

Confidentiality status

This document is Open Access. This document has no restriction on distribution.

Feedback on this Application Note

If you have any comments on this Application Note, please send email to errata@arm.com giving:

- the document title
- the document number
- the page number(s) to which your comments refer
- an explanation of your comments.

General suggestions for additions and improvements are also welcome.

ARM web address

<http://www.arm.com>

Table of Contents

1	Introduction	2
1.1	Overview of μ C/OS	2
1.2	ARM TCP/IP and PPP overview	2
2	Porting TCP/IP	3
2.1	Timers.....	3
2.2	Task control	3
2.3	TCP	7
2.4	Pre-emption and protection	9
2.5	PPP porting.....	10
3	ARM Development Board Device Drivers	11
3.1	Source files	11
3.2	Interrupt service routines.....	11

1 Introduction

This application note is intended as a supplement to the *Porting TCP/IP Programmer's* [ARM DUI 0079B]. It gives more detail on some of the issues which arise when targeting an RTOS, and gives a concrete example of such an implementation on μ C/OS, a small RTOS kernel.

1.1 Overview of μ C/OS

μ C/OS (pronounced *micro C OS*) is a small, portable real-time kernel. It has a fully pre-emptive prioritized task scheduler supporting up to 63 tasks, and mechanisms for passing signals and messages between tasks. Tasks can be dynamically created and deleted, and task priorities can be dynamically changed.

μ C/OS is available for download at <http://www.ucos-ii.com>.

The version used in this example has been ported to the ARM7TDMI version of the ARM Development Board. The API has been extended to allow user-defined interrupt service routines to be installed, and also to selectively enable the various interrupt sources available on the development board.

1.2 ARM TCP/IP and PPP overview

The ARM portable TCP/IP stack is designed to access the underlying system through a simple, generic interface. Most systems can be targeted simply by providing device drivers and some small glue functions to map the stack's interface onto the system API.

ARM provides an example implementation on the ARM Development Board, with drivers for the built-in serial port and an Olicom PCMCIA Ethernet interface, including support for modem and direct cable network connections by way of the serial port. Therefore, the only work necessary in porting to a RTOS on the development board is to write the RTOS-specific glue layer and applications.

Note *The ARM implementation makes use of semihosted SWI operations, particularly to read nonvolatile parameters from a file on the host machine. It is therefore necessary to ensure that the semihosting operations are available. For more information on semihosting, see section 13.7 of the SDT 2.50 User Guide [ARM DUI 0040D]. In particular, if μ C/OS installs a SWI handler, this must be chained onto the semihosting one. See Application Note 31: Using EmbeddedICE for more details.*

2 Porting TCP/IP

Many of the source files in the TCP/IP stack are designed to be portable, and should therefore not need to be modified. ARM's example implementation provides Ethernet and PPP network interfaces that need only minor alterations to work with μ C/OS (see below). Otherwise, all that must be done is to code the glue layer. This consists of simple mappings between the service requests made by the stack and the API functions provided by the target.

2.1 Timers

The stack requires access to a timing variable called `cticks`, which is incremented at regular intervals, and a constant called `TPS` (ticks per second), which gives the number of times per second the timer is incremented. μ C/OS implements a 100Hz timer, which can be accessed by way of API calls. Therefore, the stack can be given access to the timer using the following macros (which should be defined in `ipport.h`):

```
#define TPS 100L

#define cticks OSTimeGet()
```

In addition, the ARM example implementation expects functions to initialize and finalize the clock. These are not required on μ C/OS because the clock is initialized during the OS startup, so these functions should be made null, as follows:

```
#define clock_init()

__inline void clock_c(void) {}
```

Note *The `clock_c` function cannot be implemented as a null macro because a pointer to it is installed in the ARM IP start-up code.*

2.2 Task control

Using the multitasking method, it is necessary to create several tasks when the stack is initialized. Many of the ARM implementation modules (such as the modem handler and echo clients) must be frequently polled. Other functions must be called at regular intervals, and in response to signals from other (possibly interrupt-driven) parts of the implementation.

The polling functions can be called from separate tasks, which should be scheduled in a *round-robin* manner. In the μ C/OS implementation, because all tasks must have a unique priority and only the highest priority active task will run, this is implemented by pending on a synchronization semaphore after each call to a polling function. An extra task, at a lower priority than any other, repeatedly signals this semaphore until all of the polling tasks are active again. Many of the polling functions contain calls to the `tk_yield()` function in order to block for a while and let the rest of the stack run. On μ C/OS, this can be mapped directly to wait on the same synchronization semaphore.

2.2.1 Example

In `main.c`, each polling function could have a task coded in this manner:

```
void Task_Poll(void *Id)
{
    for (;;)
    {
        /* Call the polling function */
        polling_function();

        /* Give the other tasks some time */
        tk_yield();
    }
}
```

The `tk_yield()` function can be mapped onto the μ C/OS API function which waits on a signal semaphore. The easiest way to do this is to define an inline function or macro in `ipport.h`, such as

```
__inline void tk_yield(void)
{
    /* placeholder for semaphore error code */
    uint error;

    /* Wait to be signaled */
    OSSemPend(sem_sync, 0, &error);
}
```

where `sem_sync` is a μ C/OS semaphore created with initial value 0, so that initially, any task which waits on it will block.

A separate task could be used to restart the polling tasks. If this has a lower priority than the polling tasks, it will run after the last polling task yields. Alternatively, it could wait for a short delay after restarting the pending tasks. A simple implementation is as follows:

```

void Task_Restart(void *Id)
{
    for(;;)
    {
        /* Lock scheduling to prevent newly restarted
         * tasks from pre-empting this one */
        OSSchedLock();

        /* While tasks are pending, restart them */
        while (sem_sync->OsEventCnt < 0)
            OSSemPost(sem_sync);

        /* Allow the restarted tasks to run */
        OSSchedUnlock();
    }
}

```

If this task is given the lowest priority of all tasks, it will restart the polling tasks after they have all yielded control.

The timer functions must be called regularly, at intervals of typically 100ms to 1s. This can be easily implemented if the RTOS provides a facility to block for a length of time.

2.2.2 Example

μC/OS provides an API function to block for a given length of time, in multiples of 10ms. A simple implementation is to give each timer a task, in a form such as the following.

```

void Task_Timer(void *Id)
{
    for(;;)
    {
        /* Record the start time */
        long time_start = cticks;

        /* Run the timer function */
        timer_function();

        /* Wait for the next period */
        OSTimeDly(TPS/FREQUENCY - (cticks - time_start));
    }
}

```

A more efficient implementation would use only one task. It would keep a local record of the time at which each timer is due, and would delay until the first one arrives.

Some functions must be run in response to signals from other tasks or interrupt handlers. The signal requests can be mapped onto a mechanism provided by the RTOS, and the functions can be called from tasks which respond to the signal.

2.2.3 Example

μC/OS semaphores provide a simple mechanism to signal tasks. If the semaphore counter is initialized to 0, any task which waits on it will block until the semaphore is signaled.

The most important signal is given whenever an incoming network packet is received by the line driver. The packet is placed on a queue, and then the function `pktdemux()` must be called promptly to unwrap the packet and make the data accessible. The line driver arranges this by calling `SignalPktDemux()`. This could be mapped onto the μC/OS semaphore posting function. The simplest method is to define a macro in `ipport.h`, as follows:

```
#define SignalPktDemux() OSSemPost(sem_pktdemux)
```

There should then be a task which waits for this signal and then calls `pktdemux()`, such as:

```
void Task_PktDemux(void *Id)
{
    uint error;

    for(;;)
    {
        /* Wait for the signal */
        OSSemPend(sem_pktdemux, 0, &error);

        /* and act on it */
        pktdemux();
    }
}
```

If several signals are implemented, it may be more efficient to use a μC/OS mailbox. This allows a message (in the form of a generic pointer) to be passed when signaling the task, and this would therefore allow the pending task to determine which event has occurred.

2.3 TCP

Functions in the TCP layer often have to block until a resource becomes available (for example, while a connection is being established, or when two threads wish to send packets). This is implemented by calling `tcp_sleep()` when a required resource is not available, and calling `tcp_wakeup()` when it should be made available to other threads. They each take an argument which can be used as a key for the required resource so that threads are not woken prematurely. However, it is permissible to wake sleeping threads at any time because the specification of `tcp_sleep()` states that, when a thread is woken, it should retest the condition it was waiting for and go back to sleep if necessary. This allows the mechanism to work without an RTOS, whereby `tcp_sleep()` simply calls each polling function once and then returns. It also allows a simple implementation on an RTOS, whereby `tcp_wakeup()` restarts every thread which is sleeping, and then lets those which cannot proceed go back to sleep.

2.3.1 Example

A simple implementation on μ C/OS uses a semaphore to block the tasks, in a similar way to the synchronization semaphore used earlier to schedule the polling functions. Again, its counter is initialized to 0 so that the first call to `tcp_sleep()` will block. The simplest way to implement `tcp_wakeup()` is then to wake all tasks that are sleeping, as shown below. The code should be placed in `tcpport.c`:

```
/* A signal semaphore, which should be
 * initialized with a count of 0 */
extern OS_EVENT *sem_tcp;

void tcp_sleep(void *ptr)
{
    /* placeholder for the semaphore error code */
    int error;

    /* Ensure the other threads can access the network */
    UNLOCK_NET_RESOURCE(NET_RESID);

    /* Wait for the wakeup call */
    OSSemPend(sem_tcp, 0, &error);

    /* Reclaim access to the network */
    LOCK_NET_RESOURCE(NET_RESID);
}

void tcp_wakeup(void *ptr)
{
    /* prevent woken threads from pre-empting this one */
    OSSchedLock();

    /* while threads are sleeping, wake them*/
    while (sem_tcp->OSEventCnt < 0)
        OSSemPost(sem_tcp);

    /* Allow the system to run again */
    OSSchedUnlock();
}
```

This is quite inefficient because all pending threads are woken on every signal, and many might go back to sleep immediately. A better approach might be to use the function arguments as a key in a hash table of semaphores, so that fewer threads are woken unnecessarily each time.

2.4 Pre-emption and protection

The TCP/IP stack provides two methods for preventing global data structures from being corrupted by simultaneous access by two threads:

- The *critical section* method, whereby sections of code must be run without the possibility of being pre-empted. This typically involves disabling interrupts for short periods, and should therefore be avoided on an RTOS running time-critical applications.
- The *resource locking* method, whereby semaphore locks are placed on critical global resources to allow only one thread to access them. Any other threads which require a locked resource will block until the resource is unlocked by the thread which initially locked it.

2.4.1 Examples

Critical section locking can be implemented by disabling interrupts. The ARM Development Board version of μ C/OS has a facility for disabling and re-enabling all interrupts, which gives a simple implementation. Unfortunately, some parts of the stack contain nested critical sections, so it is important to keep track of the nesting level and only re-enable the interrupts when the outermost critical section is left. The following example illustrates this. The functions should be declared in `ipport.h`. The following is an example of how to implement these functions. It must be coded in one of the files of your application:

```
static int critical_nesting = 0; /* nesting counter */

void ENTER_CRIT_SECTION(void *ptr)
{
    /* Disable all interrupts */
    OSDisableInt();

    /* increment nesting counter */
    critical_nesting++;
}

void EXIT_CRIT_SECTION(void *ptr)
{
    /* If this is the outer section, re-enable interrupts */
    if (--critical_nesting == 0)
        OSEnableInt();
}
```

Resource locking can be easily implemented using μ C/OS semaphores. If the semaphore counter is initially set to 1, the first thread which waits on it will be allowed to proceed. Subsequent threads which wait on it will block until the semaphore is signaled, whereupon one pending thread will be allowed to proceed. The glue layer is simple enough to be defined in `ipport.h` as macros or inline functions, as shown in the following example:

```
/* A table of semaphore handles, which should
 * each be initialised with a count value of 1 */
extern OS_EVENT *sem_resources[];

__inline LOCK_NET_RESOURCE(int resource)
{
    /* placeholder for semaphore error code */
    int error;

    /* Wait for resource*/
    OSSemPend(sem_resources[resource], 0, &error);
}

__inline UNLOCK_NET_RESOURCE(int resource)
{
    /* Signal that the resource is now available */
    OSSemPost(sem_resources[resource])
}
```

2.5 PPP porting

Once the TCP/IP stack has been ported to μ C/OS on the ARM Development Board, the ARM PPP code can be used without modification. However, small modifications will be necessary to the device drivers used by the serial line driver. This is detailed below.

3 ARM Development Board Device Drivers

3.1 Source files

The device drivers provided with the example implementation can be used with μ C/OS with only minor modifications. The following files from the `\pid7tdm` directory are required:

<code>82595.c</code>	driver code for the Intel 82595 device (used with the Olicom driver)
<code>82595.h</code>	state machine and register definitions for the 82595 driver
<code>delay.s</code>	spinloop delay, used by the 82595 and PCMCIA drivers
<code>listopts.h</code>	assembly include file required by <code>delay.s</code>
<code>lolevel.h</code>	assembly include file required by <code>delay.s</code>
<code>nisa.h</code>	definitions of NISA bus peripheral registers
<code>olicom.c</code>	driver code for Olicom PCMCIA Ethernet PC-Card
<code>olicom.h</code>	declarations for Olicom PCMCIA Ethernet PC-Card driver
<code>pcmcia.c</code>	driver code for handling PCMCIA cards
<code>pcmcia.h</code>	definitions of PCMCIA control registers
<code>pcmif.h</code>	function declarations for PCMCIA driver
<code>st16c552.h</code>	register definitions for ST16C552 UART/parallel port device
<code>uart.c</code>	device driver for ST16C552 UART.

These source files should be modified to ensure that no other header files from the `\pid7tdm` directory are included because some contain definitions which conflict with μ C/OS definitions. The μ C/OS header files `ucos.h` and `pid.h` should be included in each C source file.

3.2 Interrupt service routines

The interrupt-related function calls in `pcmcia.c` and `uart.c` should be replaced with the equivalent μ C/OS function calls, as follows:

μ C/OS function call equivalents

Replace...	With...
<code>irq_Enable(source)</code>	<code>IRQEnable(source)</code>
<code>irq_Disable(source)</code>	<code>IRQDisable(source)</code>

μ C/OS uses the same names as the example implementation to identify the interrupt sources, but they are defined as source numbers, rather than bitmasks. Therefore, μ C/OS does not allow more than one interrupt to be (for example) enabled in a single function call.

3.2.1 Example

In the example PCMCIA driver, the two PC card interrupts are enabled as follows:

```
irq_Enable( IRQCardA | IRQCardB );
```

Under μ C/OS, this must be done as follows:

```
IRQEnable( IRQCardA );
```

```
IRQEnable( IRQCardB );
```

Additionally, instead of calling `irqInit()` as the example implementation does, an interrupt service routine must be installed for each interrupt source using the ARM μ C/OS ISR installation function `IRQInstall()`. This is best done in the initialization routine for each driver.

The interrupt sources which need to be handled in the example implementation are the two serial ports (`IRQSerialA` and `IRQSerialB`) and the two PC card interfaces (`IRQCardA` and `IRQCardB`). Additionally, μ C/OS will install its own handler for one of the timers (`IRQTimer1`) when the system is started.

In the example implementation, some minimal decoding of the interrupt source is carried out by the low-level interrupt handler, and extra information (for example, which of the two serial interfaces generated the interrupt) is passed to the ISR. In μ C/OS, this can be carried out by a small glue layer between the low-level handler and the ISR, as follows:

```
/* This is installed as the ISR for
 * the serial port A interrupt source */
void SerialA_ISR(void)
{
    /* Call main serial port ISR, giving a pointer
     * to the UART structure representing Serial Port A,
     * which is used for a modem connection */
    Main_Serial_ISR( &ModemUart );
}

/* This is installed as the ISR for
 * the serial port B interrupt source */
void SerialB_ISR(void)
{
    /* Call main serial port ISR, giving a pointer
     * to the UART structure representing Serial Port B,
     * which is used for debugging information */
    Main_Serial_ISR( &DebugUart );
}
```

Using this approach allows the existing example ISRs to be used with very little modification.