Realtime Operating Systems

Concepts and Implementation of Microkernels

for Embedded Systems

Dr. Jürgen Sauermann, Melanie Thelen

Contents

List of Figures

List of Tables

Preface

Every year, millions of microprocessor and microcontroller chips are sold as CPUs for general purpose computers, such as PCs or workstations, but also for devices that are not primarily used as computers, such as printers, TV sets, SCSI controllers, cameras, and even coffee machines. Such devices are commonly called *embedded systems*. Surprisingly, the number of chips used for embedded systems exceeds by far the number of chips used for general purpose computers.

Both general purpose computers and embedded systems (except for the very simple ones) require an operating system. Most general purpose computers (except mainframes) use either UNIX, Windows, or DOS. For these operating systems, literature abounds. In contrast, literature on operating systems of embedded systems is scarce, although many different operating systems for embedded systems are available. One reason for this great variety of operating systems might be that writing an operating system is quite a challenge for a system designer. But what is more, individually designed systems can be extended in exactly the way required, and the developer does not depend on a commercial microkernel and its flaws.

The microkernel presented in this book may not be any better than others, but at least you will get to know how it works and how you can modify it. Apart from that, this microkernel has been used in practice, so it has reached a certain level of maturity and stability. You will learn about the basic ideas behind this microkernel, and you are provided with the complete source code that you can use for your own extensions.

The work on this microkernel was started in summer 1995 to study the efficiency of an embedded system that was mainly implemented in C++. Sometimes C++ is said to be less efficient than C and thus less suitable for embedded systems. This may be true when using a particular C++ compiler or programming style, but has not been confirmed by the experiences with the microkernel provided in this book. In 1995, there was no hardware platform available to the author on which the microkernel could be tested. So instead, the microkernel was executed on a simulated MC68020 processor. This simulation turned out to be more useful for the development than real hardware, since it provided more information about the execution profile of the code than hardware could have done. By mere coincidence, the author joined a project dealing with automated testing of telecommunication systems. In that project, originally a V25 microcontroller had

been used, running a cooperative multitasking operating system. At that time, the system had already reached its limits, and the operating system had shown some serious flaws. It became apparent that at least the operating system called for major redesign, and chances were good that the performance of the microcontroller would be the next bottleneck. These problems had already caused serious project delay, and the most promising solution was to replace the old operating system by the new microkernel, and to design a new hardware based on a MC68020 processor. The new hardware was ready in summer 1996, and the port from the simulation to the real hardware took less than three days. In the two months that followed, the applications were ported from the old operating system to the new microkernel. This port brought along a dramatic simplification of the application as well as a corresponding reduction in source code size. This reduction was possible because serial I/O and interprocess communication were now provided by the microkernel rather than being part of the applications.

Although the microkernel was not designed with any particular application in mind, it perfectly met the requirements of the project. This is neither by accident nor by particular ingenuity of the author. It is mainly due to a good example: the MIRAGE operating system written by William Dowling of Sahara Software Ltd. about twenty years ago. That operating system was entirely written in assembler and famous for its real-time performance. Many concepts of the microkernel presented in this book have been adopted from the MIRAGE operating system.

1 Requirements

1.1 General Requirements

Proper software design starts with analyzing the requirements that have to be fulfilled by the design. For embedded systems, the requirements are defined by the purpose of the system. General definitions of the requirements are not possible: for example, the requirements of a printer will definitely be different from those of a mobile phone. There are, however, a few common requirements for embedded systems which are described in the following sections.

1.2 Memory Requirements

The first PCs of the early eighties had 40 kilobytes of ROM, 256 or 512 kilobytes of RAM, and optionally a hard disk drive with 5 or 10 megabytes capacity. In the mid-nineties, an off-the-shelf PC had slightly more ROM, 32 megabytes of RAM, and a hard disk drive of 2 or 4 gigabytes capacity. Floppy disks with 360 or 720 kilobyte capacity, which were the standard medium for software packages and backups, had been replaced by CD-ROM and tape streamers with capacities well above 500 megabytes. Obviously, capacity has doubled about every two years, and there is no indication that this trend will change. So why bother about memory requirements?

A PC is an open system that can be extended both in terms of memory and peripherals. For a short while, a PC can be kept up to date with technological developments by adding more memory and peripherals until it is ultimately outdated. Anyway, a PC could live for decades; but its actual lifetime is often determined by the increasing memory demands of operating systems and applications rather than by the lifetime of its hardware. So to extend the lifetime of a PC as much as possible and thus to reduce the costs, its configuration has to be planned thoroughly.

For a given embedded system, in contrast, the memory requirements are known in advance; so costs can be saved by using only as much memory as required. Unlike PCs, where the ROM is only used for booting the system, ROM size plays a major role for the memory requirements of embedded systems, because in embedded systems, the ROM is used as program memory. For the ROM, various types of memory are available, and their prices differ dramatically: EEPROMs are most expensive, followed by static RAMs, EPROMs, dynamic RAMs, hard disks,

floppy disks, CD-ROMs, and tapes. The most economical solution for embedded systems is to combine hard disks (which provide non-volatility) and dynamic RAMs (which provide fast access times).

Generally, the memory technology used for an embedded system is determined by the actual application: For example, for a laser printer, the RAM will be dynamic, and the program memory will be either EEPROM, EPROM, or RAM loaded from a hard disk. For a mobile phone, EEPROMs and static RAMs will rather be used.

One technology which is particularly interesting for embedded systems is on-chip memory. Comparatively large on-chip ROMs have been available for years, but their lack of flexibility limited their use to systems produced in large quantities. The next generation of microcontrollers were on-chip EPROMs, which were suitable also for smaller quantities. Recent microcontrollers provide on-chip EEPROM and static RAM. The Motorola 68HC9xx series, for example, offers on-chip EEPROM of 32 to 100 kilobytes and static RAM of 1 to 4 kilobytes.

With the comeback of the Z80 microprocessor, another interesting solution has become available. Although it is over two decades old, this chip seems to outperform its successors. The structure of the Z80 is so simple that it can be integrated in FPGAs (Field Programmable Logic Arrays). With this technique, entire microcontrollers can be designed to fit on one chip, providing exactly the functions required by an application. Like several other microcontrollers, the Z80 provides a total memory space of 64 kilobytes.

Although the memory size provided on chips will probably increase in the future, the capacities available today suggest that an operating system for embedded system should be less than 32 kilobytes in size, leaving enough space for the application.

1.3 Performance

The increase in the PCs' memory size is accompanied by a similar increase in performance. The first PCs had an 8 bit 8088 CPU running at 8 MHz, while today a 32 bit CPU running at 200 MHz is recommended. So CPU performance has doubled about every two years, too. Surprisingly, this dramatic increase in performance is not perceived by the user: today's operating systems consume even more memory and CPU performance than technological development can provide. So the more advanced the operating system, the slower the applications. One reason for the decreasing performance of applications and also of big operating systems might be that re-use of code has become common practice; coding as such is avoided as much as possible. And since more and more code is

executed in interfaces between existing modules, rather than used for the actual problem, performance steadily deteriorates.

Typically, performance demands of embedded systems are higher than those of general purpose computers. Of course, if a PC or embedded system is too slow, you could use a faster CPU. This is a good option for PCs, where CPU costs are only a minor part of the total costs. For embedded systems, however, the cost increase would be enormous. So the performance of the operating system has significant impact on the costs of embedded systems, especially for single-chip systems.

For example, assume an embedded system requiring serial communication at a speed of 38,400 Baud. In 1991, a manufacturer of operating systems located in Redmond, WA, writes in his C/C++ Version 7.0 run-time library reference: “The _bios_serialcom routine may not be able to establish reliable communications at baud rates in excess of 1,200 Baud (_COM_1200) due to the overhead associated with servicing computer interrupts”. Although this statement assumes a slow 8 bit PC running at 8 MHz, *no* PC would have been able to deal with 38,400 baud at that time. In contrast, embedded systems had been able to manage that speed already a decade earlier: using 8 bit CPUs at even lower clock frequencies than the PCs’.

Performance is not only determined by the operating system, but also by power consumption. Power consumption becomes particularly important if an embedded system is operated from a battery, for example a mobile phone. For today’s commonly used CMOS semiconductor technology, the static power required is virtually zero, and the power actually consumed by a circuit is proportional to the frequency at which the circuit is operated. So if the performance of the operating system is poor, the CPU needs to be operated at higher frequencies, thus consuming more power. Consequently, the system needs larger batteries, or the time the system can be operated with a single battery charge is reduced. For mobile phones, where a weight of 140g including batteries and stand-by times of 80 hours are state of the art, both of these consequences would be show stoppers for the product. Also for other devices, power consumption is critical; and last, but not least, power consumption should be considered carefully for any electrical device for the sake of our environment.

1.4 Portability

As time goes by, the demands on products are steadily increasing. A disk controller that was the fastest on the market yesterday will be slow tomorrow. Mainstream CPUs have a much wider performance range than the different microcontroller families available on the market. Thus eventually it will be necessary to change to a different family. At this point, commercial microkernels

can be a problem if they support only a limited number of microcontrollers, or not the one that would otherwise perfectly meet the specific requirements for a product. In any case, portability should be considered from the outset.

The obvious approach for achieving portability is to use high level languages, in particular C or C++. In principle, portability for embedded system is easier to achieve than for general purpose computers. The reason is that complex applications for general purpose computers not only depend on the CPU used, but also on the underlying operating system, the window system used, and the configuration of the system.

A very small part of the microkernel presented in this book was written in Assembler; the rest was written in C++. The part of the kernel which depends on the CPU type and which needs to be ported when a different CPU family is used, is the Assembler part and consists of about 200 Assembler instructions. An experienced programmer, familiar with both the microkernel and the target CPU, will be able to port it in less than a week.

The entire kernel, plus a simple application, fit in less than 16 kilobyte ROM for a MC68020 CPU. Hence it is especially suitable for single chip solutions.

2 Concepts

2.1 Specification and Execution of Programs

The following sections describe the structure of a program, how a program is prepared for execution, and how the actual execution of the program works.

2.1.1 Compiling and Linking

Let us start with a variant of the well known “Hello World!” program:

```
#include <stdio.h>

const char * Text = "Hello World\n";

char Data[] = "Hello Data\n";

int Uninitialized;   // Bad Practice

int main(int argc, char * argv[])
{
     printf(Text);
}
```

This C++ program prints “Hello World”, followed by a line feed on the screen of a computer when it is executed. Before it can be executed, however, it has to be transformed into a format that is executable by the computer. This transformation is done in two steps: *compilation* and *linking*.

The first step, compilation, is performed by a program called *compiler*. The compiler takes the program text shown above from one file, for example **Hello.cc**, and produces another file, for example **Hello.o**. The command to compile a file is typically something like

```
g++ -o Hello.o Hello.cc
```

The name of the C++ compiler, g++ in our case, may vary from computer to computer. The **Hello.o** file, also referred to as *object file*, mainly consists of three sections: TEXT, DATA, and BSS. The so-called *include file* **stdio.h** is simply copied into **Hello.cc** in an early execution phase of the compiler, known as

preprocessing. The purpose of **stdio.h** is to tell the compiler that **printf** is not a spelling mistake, but the name of a function that is defined elsewhere. We can imagine the generation of **Hello.o** as shown in Figure 2.1.[1]

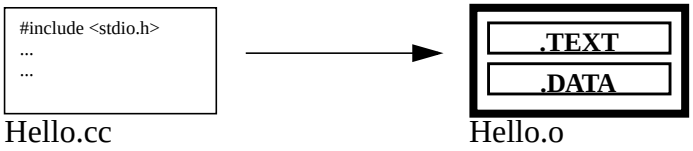

FIGURE 2.1 Hello.o Structure

Several object files can be collected in one single file, a so-called *library.* An important library is **libc.a** (the name may vary with the operating system used): it contains the code for the **printf** function used in our example, and also for other functions. We can imagine the generation of **libc.a** as shown in Figure 2.2.

1. **Note**: The BSS section contains space for symbols that uninitialized when starting the program. For example, the integer variable **Uninitialized** will be included here in order to speed up the loading of the program. However, this is bad programming practice, and the bad style is not weighed up by the gain in speed. Apart from that, the memory of embedded systems is rather small, and thus loading does not take long anyway. Moreover, we will initialize the complete data memory for security reasons; so eventually, there is no speed advantage at all. Therefore, we assume that the BSS section is always empty, which is why it is not shown in Figure 2.1, and why it will not be considered further on.

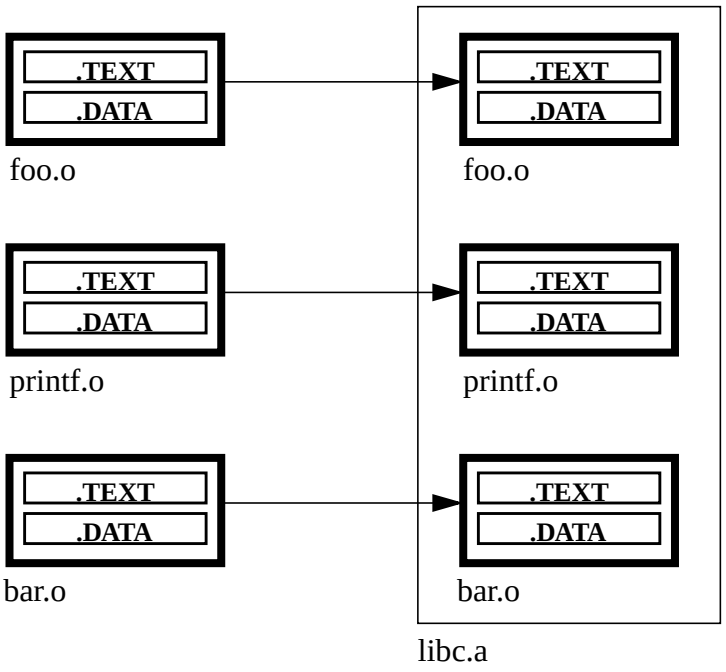

FIGURE 2.2 libc.a Structure

The second step of transforming program text into an executable program is *linking*. A typical link command is e.g.

```
ld -o Hello Hello.o
```

With the linking process, which is illustrated in Figure 2.3, all unresolved references are resolved. In our example, **printf** is such an unresolved reference, as it is used in **main()**, but defined in **printf.o**, which in turn is contained in **libc.a**. The linking process combines the TEXT and DATA sections of different object files in one single object file, consisting of one TEXT and one DTA section only. If an object file is linked against a library, only those object files containing definitions for unresolved symbols are used. It should be noted that a linker can produce different file formats. For our purposes, the so-called Motorola S-record format will be used.

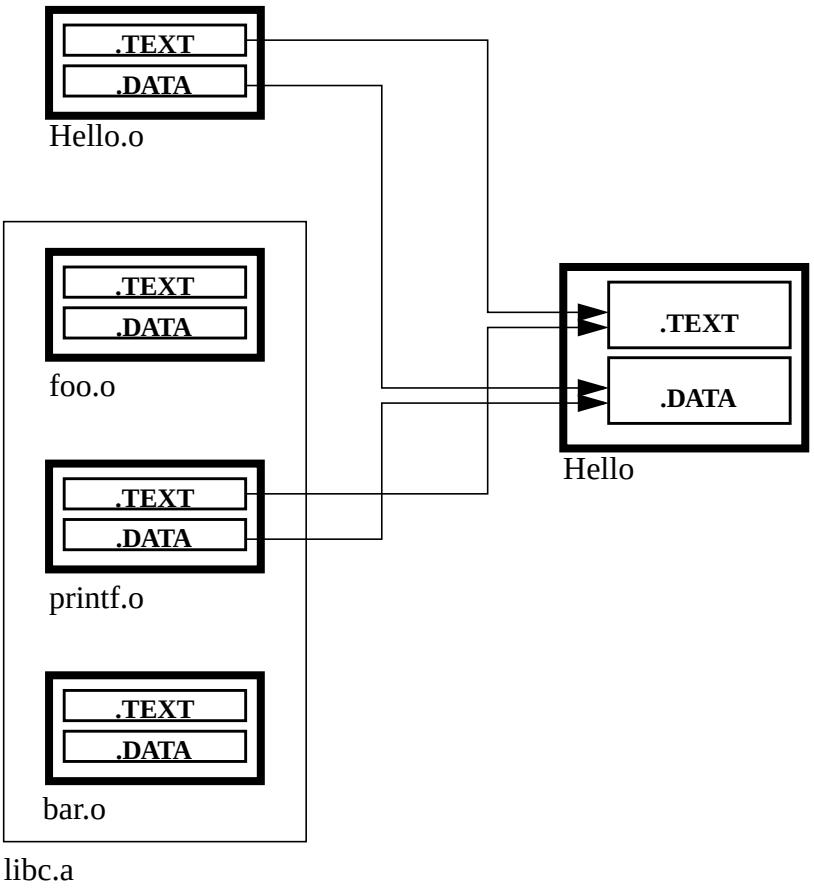

FIGURE 2.3 Hello Structure

2.2 Loading and Execution of Programs

After a program has been compiled and linked, it can be executed. While compilation and linking is basically identical for embedded systems and general purpose computers, there are some differences regarding the execution of programs. Table 2.1 lists the steps performed during program execution and shows the differences between general purpose computers and embedded systems:

	General Purpose Computer	**Embedded System**
1	The TEXT section of the program is loaded into the program memory (part of the computer's RAM).	The TEXT section is already existing in the program memory (EEPROM) of the embedded system.
2	Depending on the object format generated by the linker, the addresses of the TEXT section may need to be relocated. If the compiler produced position independent code (PIC), this step is omitted.	The addresses are computed by the linker.
3	The DATA section of the program is loaded into program memory (part of the computer's RAM).	The DATA section is already in the EEPROM of the embedded system.
4	Depending of the object format generated by the linker, the addresses of the TEXT section may need to be relocated.	The DATA section is copied as a whole to its final address in RAM.

TABLE 2.1 Execution of a program

Obviously, the execution of a program in an embedded system is much easier than in a general purpose computer.

2.3 Preemptive Multitasking

The previous sections described the execution of one program at a time. But what needs to be done if several programs are to be executed in parallel? The method we have chosen for parallel processing is *preemptive multitasking*. By definition, a *task* is a program that is to be executed, and *multitasking* refers to several tasks being executed in parallel. The term *preemptive multitasking* as such may imply a complex concept. But it is much simpler than other solutions, as for example *TSR* (Terminate and Stay Resident) programs in DOS, or *cooperative* multitasking.

To explain the concepts of preemptive multitasking, we developed a model which is described in the following sections.

2.3.1 Duplication of Hardware

Let us start with a single CPU, with a program memory referred to as *ROM* (Read Only Memory), and a data memory, *RAM* (Random Access Memory). The CPU may read from the ROM, as well as read from and write to the RAM. In practice, the ROM is most likely an *EEPROM* (Electrically Erasable Programmable ROM). The CPU reads and executes instructions from the ROM. These instructions comprise major parts of the TEXT section in our example program on page 7. Some of these instructions cause parts of the RAM to be transferred into the CPU, or parts of the CPU to be transferred to the RAM, as shown in Figure 2.4 on page 13. For general purpose computers, the program memory is a RAM, too. But in contrast to embedded systems, the RAM is not altered after the program has been loaded – except for programs which modify themselves, or paged systems where parts of the program are reloaded at runtime.

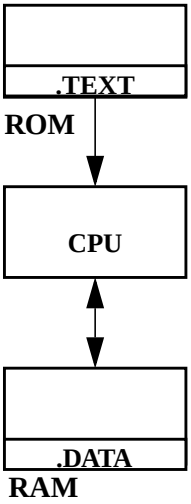

FIGURE 2.4 Program Execution

Now let us assume we have two different programs to be run in parallel. This can be achieved surprisingly easy_ by duplicating the hardware. Thus, one program can be executed on one system, and the second program can be executed on the other system, as shown in Figure 2.5. Note that the TEXT and DATA sections are at different locations in the ROMs and RAMs of Figure 2.5.

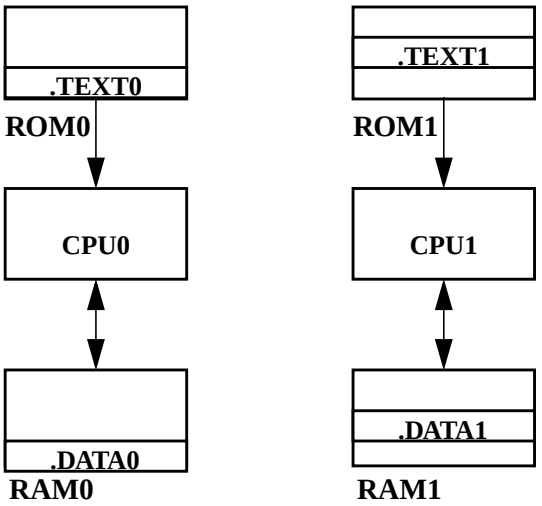

FIGURE 2.5 Parallel execution of two programs

Because of the increased hardware costs, this approach for running different programs in parallel is not optimal. But on the other hand, it has some important advantages which are listed in Table 2.2. Our goal will be to eliminate the disadvantage while keeping the benefits of our first approach.

Advantages	**Disadvantages**
The two programs are entirely protected against each other. If one program crashes the CPU, then the other program is not affected by the crash.	Two ROMs are needed (although the total amount of ROM space is the same).
	Two RAMs are needed (although the total amount of RAM space is the same).
	Two CPUs are needed.
	The two programs cannot communicate with each other.

TABLE 2.2 Duplication of Hardware

2.3.2 Task Switch

The next step in developing our model is to eliminate one of the two ROMs and one of the two RAMs. To enable our two CPUs to share one ROM and one RAM, we have to add a new hardware device: a *clock*. The clock has a single output producing a signal (see Figure 2.5). This signal shall be inactive (*low)* for 1,000 to 10,000 CPU cycles, and active (*high)* for 2 to 3 CPU cycles. That is, the time while the signal is high shall be sufficient for a CPU to complete a cycle.

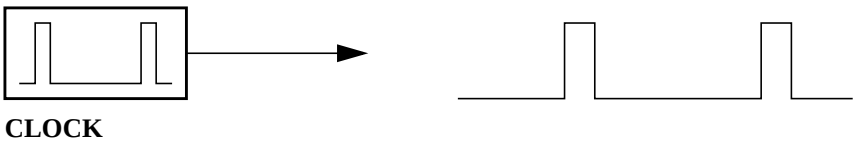

FIGURE 2.6 Clock

The output of the clock is used to drive yet another device: the *task switch* (see Figure 2.7). The task switch has one input and two outputs. The outputs shall be used for turning on and off the two CPUs. The clock (CLK) signal turning from inactive to active is referred to as *task switch event*. On every task switch event, the task switch deactivates the active output, OUT0 or OUT1. Then the task switch waits until the CLK signal becomes inactive again in order to allow the CPU to complete its current cycle. Finally, the task switch activates the other output, OUT0 or OUT1.

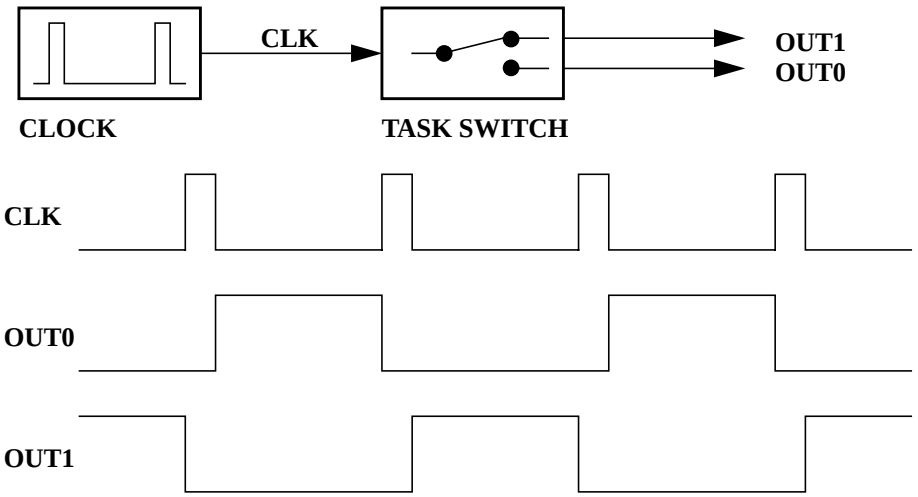

FIGURE 2.7 Task Switch

Each of the CPUs has an input that allows the CPU to be switched on or off. If the input is active, the CPU performs its normal operation. If the input goes inactive, the CPU completes its current cycle and releases the connections towards ROM and RAM. This way, only one CPU at a time is operating and connected to ROM and RAM, while the other CPU is idle and thus not requiring a connection to ROM and RAM. Consequently, we can remove the duplicated ROM and RAM from our model, and the remaining ROM and RAM can be shared by the two CPUs (see Figure 2.8).

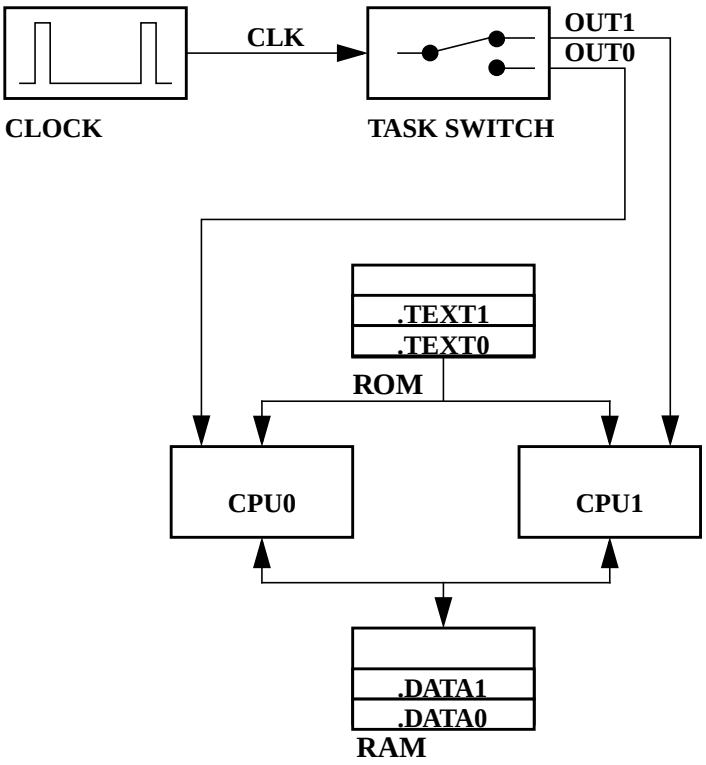

FIGURE 2.8 Shared ROM and RAM

By using the shared RAM, the two CPUs can communicate with each other. We have thus lost one of the advantages listed in Table 2.2: the CPUs are no longer protected against each other. So if one CPU overwrites the DATA segment of the other CPU during a crash, then the second CPU will most likely crash, too. However, the risk of one CPU going into an endless loop is yet eliminated. By the way, when using cooperative multitasking, an endless loop in one task would suspend all other tasks from operation.

2.3.3 Task Control Blocks

The final steps to complete our model are to move the duplicated CPU, and to implement the task switch in software rather than in hardware. These two steps are closely related. The previous step of two CPUs sharing one ROM and one RAM was relatively easy to implement by using different sections of the ROM and RAM. Replacing the two CPUs by a single one is not as easy, since a CPU

cannot be divided into different sections. But before discussing the details, let us have a look at the final configuration which is shown in Figure 2.9:

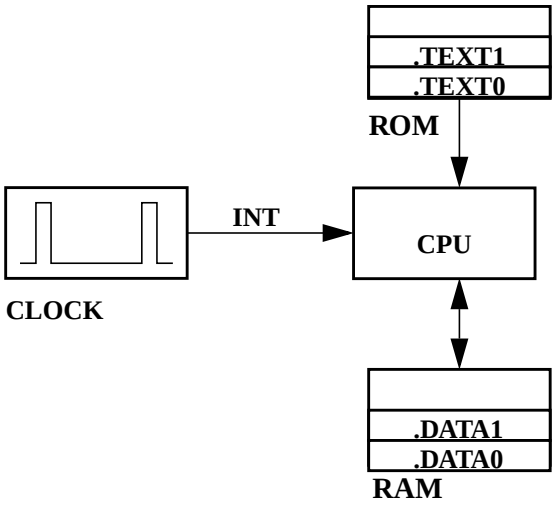

FIGURE 2.9 Final Hardware Model for Preemptive Multitasking

In contrast to the configuration with two CPUs shown in Figure 2.8, the final configuration (see Figure 2.9) has only one CPU and no task switch. Moreover, the CLK signal has been replaced by an INT signal. This signal indicates that in the final model, task switching is initiated by a regular interrupt towards the CPU.

The final configuration is very similar to our initial model shown in Figure 2.4 on page 13. We merely have added the clock device, which is now connected to the interrupt input of the CPU. Note that our final model is able to run more than two programs in parallel.

The main reason why we wanted to remove the duplicated CPU is the following: Think of the two CPUs shown in Figure 2.8 on page 16. At any time, these two CPUs are most likely in different states. The two possible states are represented by the internal registers of the CPU and determined by the programs executed by the CPUs. So to remove the duplicated CPU, we need to replace the hardware task switch by a software algorithm. Upon a task switch event (that is, the time when the clock signal goes inactive, or low), the state of one CPU needs to be saved, and the state of the second CPU needs to be restored. So we obtain the following algorithm:

- **Save the internal registers of CPU0**
- **Restore the internal registers of CPU1**

However, this algorithm does not make much sense, as our final model in Figure 2.9 on page 17 is to have only one CPU. Instead of having two CPUs, we use a data structure called *TCB, Task Control Block*, to represent the CPUs of the system. These TCBs provide space for storing the contents of the CPUs' registers R_0 to R_n. Moreover, each TCB has a pointer to the TCB that represents the next CPU. The task switch of Figure 2.8 on page 16 is replaced by a variable, **CurrentTask**. The TCB concept is illustrated in Figure 2.10.

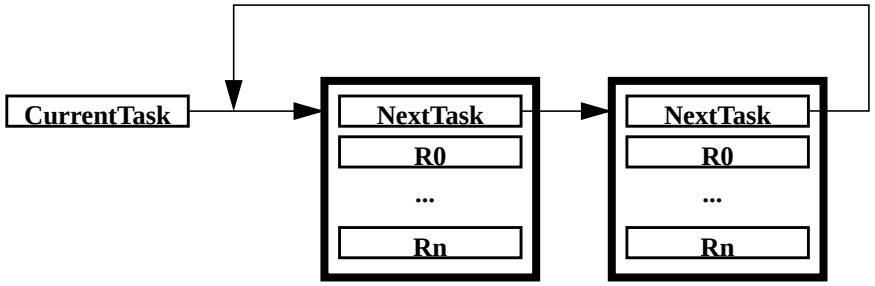

FIGURE 2.10 Task Control Blocks and CurrentTask

As a result, the proper task switch algorithm, which is an *Interrupt Service Routine, ISR*, is as follows:

- **Reset the interrupt, if required**
- **Store the internal CPU registers into the TCB to which CurrentTask is pointing**
- **Replace CurrentTask by NextTask pointer of the TCB to which CurrentTask is pointing**
- **Restore the internal CPU registers from the TCB to which CurrentTask points now**
- **Return from ISR**

Not that the ISR itself does not change the CPU state during the task switch. But this ISR is all we need for preemptive multitasking. By inserting further TCBs in the TCB **NextTask** pointer ring, the model can be extended to perform any number of tasks.

There is an important invariant for this scheme: **Whenever a task examines the variable CurrentTask, it will find this variable pointing to its own TCB**. If **CurrentTask** does not point to some arbitrary task, then this task is not active at

that time, and thus this condition cannot be detected. In brief, **for every task, CurrentTask refers to the tasks's own TCB**.

2.3.4 De-Scheduling

Up to now, our two tasks had equal share of CPU time. As long as both tasks are busy with useful operations, there is no need to change the distribution of CPU time. For embedded systems, however, a typical situation is as follows: each task waits for a certain event. If the event occurs, the task handles this event. Then the task waits for the next event, and so on. For example, assume that each of our tasks monitors one button which is assigned to the relevant task. If one of the buttons is pressed, a *long and involved computation*, *lic*, is called:

```
task_0_main()
{
     for  (;;)
          if (button_0_pressed()) lic_0();
}

task_1_main()
{
     for  (;;)
          if (button_1_pressed()) lic_1();
}
```

As task switching is controlled by our clock device, each task consumes 50 percent of the CPU time, regardless of whether a button is being pressed or not. This situation is described as **busy wait**. So precious CPU time is wasted by the tasks being busy with waiting as long as the **button_x_pressed()** functions return 0. To ensure optimal exploitation of CPU time, we add a **DeSchedule()** function which causes a task to release explicitly its CPU time:

```
task_0_main()
{
     for  (;;)
          if (button_0_pressed()) lic_0();
          else                    DeSchedule();
}

task_1_main()
{
     for  (;;)
          if (button_1_pressed()) lic_1();
          else                    DeSchedule();
}
```

So the **DeSchedule()** function initiates the same activities as our ISR, except that there is no interrupt to be reset. Unless both buttons are pressed simultaneously,

the **DeSchedule()** function allows to assign the CPU time to the task that actually needs it, while still maintaining the simplicity of our model. Note that explicit descheduling should only be used rarely, because … **(ausdrückliche Begründung fehlt!!!)**.

2.4 Semaphores

To further enhance the usage of CPU time and to reduce the time for task switching, we will make use of yet another powerful data structure of preemptive multitasking: *semaphores*. These semaphores allow changing the state of our tasks.

In our current model, the two tasks are permanently running and thus consuming precious CPU capacity. For this purpose, we introduce two new variables in the TCB: **State** and **NextWaiting**. For now, **State** is initially set to the value **RUN,** and **NextWaiting** is set to 0. If required, **State** may be set to the value **BLKD** (that is, blocked). So if we refer to the task as being RUN or BLOCKED, that means that the **State** variable has the corresponding value. As a result, we obtain the TCB and the state machine shown in Figure 2.11. The state machine will be extended later.

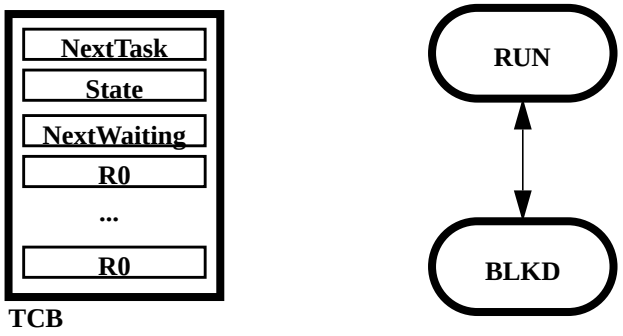

FIGURE 2.11 Task State Machine

Next, we slightly modify our task switching ISR so that it ignores tasks that are not in state RUN:

- **Reset the interrupt, if required**
- **Store the internal CPU registers into the TCB to which CurrentTask is pointing**
- **Repeat**

  Replace CurrentTask by NextTask pointer of the TCB to which CurrentTask is pointing

  until the state of CurrentTask is RUN
- **Restore the internal CPU registers from the TCB to which CurrentTask is pointing now**
- **Return from ISR**

There is an important invariant: **Whenever a task examines the variable State, it will find this variable set to RUN**. **State** may have any value at any time; but if **State** is not set to **RUN**, then this task is not active at that time, and thus the task cannot find itself in another state.

This invariant does not yet have any impact on our model, since our tasks are permanently in state **RUN**. Clearly, if no task were in state **RUN**, the above ISR would loop forever. It will be the semaphores that control the state changes of a task; that is, switch between **RUN** and **BLKD**.

A semaphore represents the number of abstract resources: if resources are available, the semaphore counts the number of resources. If no resources are available, the semaphore counts the number of tasks that are waiting for resources. The latter situation can also be expressed as the “number of resources missing”. If there are resources missing, then the TCBs of the tasks waiting for these resources are appended to a linked list of TCBs of waiting tasks, where the head of the list is part of the semaphore.

The semaphore consists of two variables: a counter and a pointer to a TCB. The TCB pointer **NextWaiting** is only valid if the counter is less than 0; otherwise, it is invalid and set to 0 for clarity. The pointer represents the state of the semaphore as shown in Table 2.3.

Counter Value	**NextWaiting TCB Pointer**	**State**
N > 0	0	N resources available
N = 0	0	No resource available, and no task waiting for a resource
-N < 0	Next task waiting for a resource represented by this semaphore	N tasks waiting for a resource; that is, N resources are missing

TABLE 2.3 Semaphore States

When a semaphore is created, the counter is initialized with the number $N \geq 0$ of resources initially available, and the **NextWaiting** pointer is set to 0. Then tasks may request a resource by calling a function **P()**, or the tasks may release a resource by calling a function **V()**. The names **P** and **V** have been established by Dijkstra, who invented the semaphores concept. In C++, a semaphore is best represented as an instance of a class **Semaphore**, while **P()** and **V()** are public member functions of that class.

The algorithm for the **P()** member function is as follows:

- **If Counter > 0** **(i.e. if resources are available)**
  Decrement Counter (decrement number of resources)
- **Else** **(i.e. if no resources are available)**
  Decrement Counter, (increment number of tasks waiting)
  Set State of CurrentTask to BLKD
  Append CurrentTask at the end of the waiting chain
  DeSchedule()

The **P()** function examines **Counter** in order to verify if there are any resources available. If so, the number of resources is simply decremented and execution proceeds. Otherwise, the number of waiting tasks is increased (which again causes the counter to be decreased, since **-Counter** is increased), the task is blocked and appended to the waiting chain, and finally **DeSchedule()** is called to make the blocking effective. Obviously, **Counter** is decremented in any case. So decrementing the counter can be placed outside the conditional part, thereby changing the comparison from > 0 to ≥ 0. By inverting the condition from ≥ 0 to < 0 and by exchanging the If part (which is empty now) and the Else part, we get the following equivalent algorithm:

- **Decrement Counter**
- **If Counter < 0**
  Set State of CurrentTask to BLKD
  Append CurrentTask at the end of the waiting chain
  DeSchedule()

The V() member function has the following algorithm:

- **If Counter ≥ 0** **(i.e. if there are no tasks waiting)**
  Increment Counter (increment number of resources)
- **Else** **(i.e. if there are tasks waiting)**
  Increment Counter, (decrement number of tasks waiting)
  Set State of first waiting task to RUN
  Remove first waiting task from the head of the waiting chain

The **V()** function examines **Counter**. If **V()** finds that **Counter** is ≥ 0, which means there are no tasks waiting, then it just increments **Counter**, indicating there is one more resource available. If **V()** finds that **Counter** is ≤ 0, there are tasks waiting. The number of waiting tasks is decremented by incrementing the counter, the first task in the waiting chain is then unblocked by setting its state back to RUN, and the task is removed from the waiting chain. The task that is being activated had issued a **P()** operation before and continues execution just after the **DeSchedule()** call it made in the **P()** function. Figure 2.12 shows a

sequence of **P()** function calls performed by a task T0, and **V()** function calls performed by another task or ISR on the same semaphore.

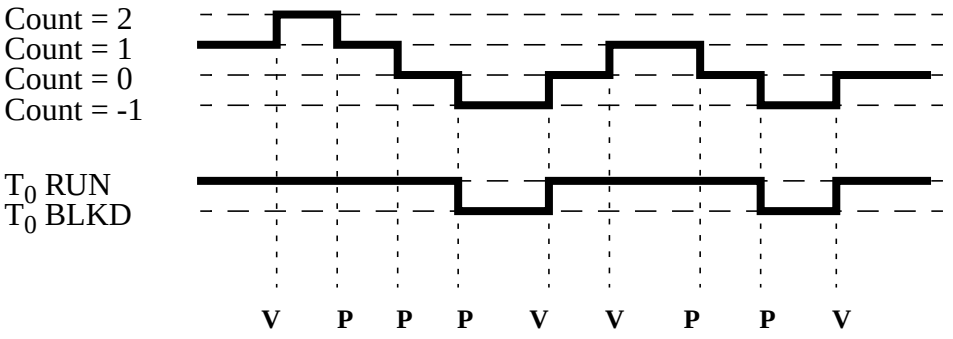

FIGURE 2.12 P() and V() Function Calls

A semaphore is very similar to a bank account. There are no restrictions to pay money into your account (**V()**) whenever you like. In contrast, you can withdraw money (**P()**) only if you have deposited it before. If there is no money left, you have to wait until somebody is kind enough to fill the account again. If you try to cheat the bank by trying to withdraw money from an empty account (**P()** when Counter = 0), you go to jail (get blocked) until there is enough money again. Unfortunately, if you are in jail, there is no way for yourself to fix the problem by depositing money, since in jail you can't do anything at all.

As for the bank account, there are huge differences between the **P()** and **V()** functions, see Table 2.3.

P()	**V()**
P() *must not be called in an ISR*	**V()** may be called from anywhere, including ISR.
A **P()** function call may block the calling task	A **V()** function call may not block any task

TABLE 2.4 P() and V() properties

P()	**V()**
The negative value of Counter is limited by the number of existing tasks, since every task is blocked at a **P()** call with **Counter** ≤ 0.	Any number of **V()** operations may be performed, thus increasing **Counter** to arbitrarily high values.
The **P()** call requires time O(N) if **Counter** < 0; else, **P()** requires time O(1). The time can be made constant by using a pointer to the tail of the waiting chain, but it is usually not worth the effort.	The **V()** call requires constant time

TABLE 2.4 P() and V() properties

Semaphores used some common initial values which have specific semantics, as shown in Table 2.3.

Initial Counter	**Semantic**
N > 1	The semaphore represents a pool of N resources.
N = 1	A single resource that may only be used by one task at a time; for example, hardware devices.
N = 0	One or several resources, but none available initially; for example, a buffer for received characters.

TABLE 2.5 Typical Initial Counter Values

2.5 Queues

Although semaphores provide the most powerful data structure for preemptive multitasking, they are only occasionally used explicitly. More often, they are hidden by another data structure called *queues*. Queues, also called *FIFOs* (first in, first out), are buffers providing at least two functions: **Put()** and **Get()**. The size of the items stored in a queue may vary, thus Queue is best implemented as a template class. The number of items may vary as well, so the constructor of the class will take the desired length as an argument.

2.5.1 Ring Buffers

The simplest form of a queue is a ring buffer. A consecutive part of memory, referred to as Buffer, is allocated, and two variables, the **GetIndex** and the **PutIndex**, are initialized to 0, thus pointing to the beginning of the memory space. The only operation performed on the **GetIndex** and the **PutIndex** is incrementing them. If they happen to exceed the end of the memory, they are reset to the beginning. This wrapping around at the end turns the straight piece of memory into a ring. The buffer is empty if and only if **GetIndex** = **PutIndex**. Otherwise, the **PutIndex** is always ahead of the **GetIndex** (although the **PutIndex** may be less than the **GetIndex** if the **PutIndex** already wrapped around at the end, while the **GetIndex** did not wrap around yet). In Figure 2.13, a ring buffer is shown both as straight memory and as a logical ring.

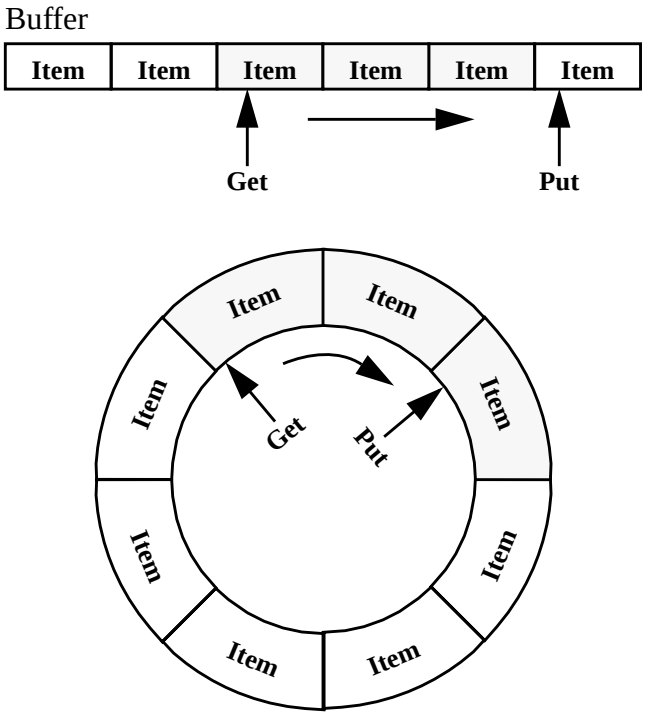

FIGURE 2.13 Ring Buffer

The algorithm for **Put()**, which takes an item as its arguments and puts it into the ring buffer, is as follows:

- **Wait as long as the Buffer is full, or return Error indicating overflow**
- **Buffer[PutIndex] = Item**
- **PutIndex = (PutIndex + 1) modulo BufferSize (increment PutIndex, wrap around at end)**

Get(), which removes the next item from the ring buffer and returns it, has the following algorithm:

- **Wait as long as Buffer is empty, or return Error indicating underflow**
- **Item = Buffer[GettIndex]**
- **GetIndex = (GetIndex + 1) modulo BufferSize(increment GetIndex, wrap around at end)**
- **Return Item**

In practice, an empty buffer is much more likely than a buffer overflow. In embedded systems, an empty buffer is a sign of proper design, while a full buffer usually shows that something is wrong. So **Get()** and **Put()** can also be compared to a bank account, which tends to be empty rather than overflow.

Assume that we don not want to return an error condition on full or empty buffers. There are good reasons not to return an error condition, since this condition is likely to disappear again, and the response to such an error condition will most often be a retry of the **Put()** or **Get()**. That is, we assume we want to wait. The simplest (and worst) approach is again busy wait:

For the Get() function:

- **While GetIndex = PutIndex**
  Do Nothing (i.e. waste time)

For the Put() function:

- **While GetIndex = (PutIndex + 1) modulo BufferSize**
  Do Nothing (i.e. was time)

The note on bank accounts and the term *busy wait* should have reminded you of semaphores.

2.5.2 Ring Buffer with Get Semaphore

The basic idea is to consider the items in a buffer as resources. I have seen this idea for the first time in an operating system called MIRAGE about twenty years ago. It was used for interrupt-driven character I/O.

In addition to the **GetIndex** and **PutIndex** variables, we add a semaphore called **GetSemaphore**, which represents the items in the buffer. As **GetIndex** and **PutIndex** are initialized to 0 (that is, the buffer is initially empty), this semaphore is initialized with its **Counter** variable set to 0.

For each **Put()**, a **V()** call is made to this semaphore **after** the item has been inserted into the buffer. This indicates that another item is available.

- **Wait as long as the Buffer is full, or return Error indicating overflow**
- **Buffer[PutIndex] = Item**
- **PutIndex = (PutIndex + 1) modulo BufferSize(increment PutIndex, wrap around at end)**
- **Call V() for GetSemaphore**

For each **Get()**, a **P()** call is made *before* removing an item from the buffer. If there are no more items in the buffer, then the task performing the **Get()** and thus the **P()** is blocked until someone uses **Put()** and thus **V()** to insert an item.

- **Call P() for GetSemaphore**
- **Item = Buffer[GettIndex]**
- **GetIndex = (GetIndex + 1) modulo BufferSize(increment GetIndex, wrap around at end)**
- **Return Item**

2.5.3 Ring Buffer with Put Semaphore

Instead of considering the items that are already inserted as resources, we could as well consider the free space in the buffer as resources. In addition to the **GetIndex** and **PutIndex** variables for the plain ring buffer, we add a semaphore called **PutSemaphore**, which represents the free space in the buffer. As **GetIndex** and **PutIndex** are initialized to 0 (that is, the buffer is initially empty), this semaphore (in contrast to the **GetSemaphore**) is initialized with its **Counter** variable set to **BufferSize**.

For each **Put()**, a **P()** call is made to this semaphore *before* the item is inserted into the buffer and thus buffer space is reduced. If there is no more free space in the buffer, then the task performing the **Put()** and thus the **P()** is blocked until someone uses **Get()** and thus **V()** to increase the space again.

- **Call P() for PutSemaphore**
- **Buffer[PutIndex] = Item**
- **PutIndex = (PutIndex + 1) modulo BufferSize(increment PutIndex, wrap around at end)**

For each **Get()**, a **P()** call is made *after* removing an item from the buffer, indicating another free position in the buffer.

- **Wait as long as Buffer is empty, or return Error indicating underflow**
- **Item = Buffer[GettIndex]**
- **GetIndex = (GetIndex + 1) modulo BufferSize(increment GetIndex, wrap around at end)**
- **Call V() for PutSemaphore**
- **Return Item**

This scheme is used less often than the ring buffer with Get semaphore. To understand why, let us consider a task which communicates with an interrupt-

driven serial port. For each direction, a buffer is used between the task and the serial port, as shown in Figure 2.14. Assume further that the task shall echo all characters received to the serial port, possibly running at a lower speed. At a first glance, you may expect to have the (upper) receive buffer used with a get semaphore, and the (lower) transmit buffer with a put semaphore. The task will be blocked most of the time on the get semaphore, which is a normal condition. What would happen, however, if the task would block on the put semaphore, i.e. if the transmit buffer is full? This will eventually happen if the transmit data rate is lower than the receive data rate. In this case, one would normally signal the sender at the far end to stop transmission for a while, for example by hardware or software handshake. A blocked task, however, would not be able to do this. This scenario is quite common, and one would use a get semaphore for the upper buffer, but a plain ring buffer for the lower one.

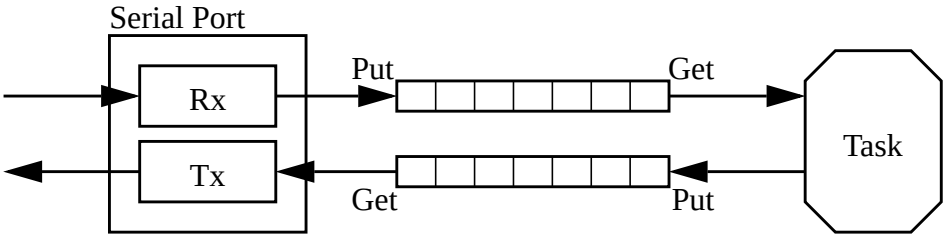

FIGURE 2.14 Serial Communication between a Task and a Serial Port

2.5.4 Ring Buffer with Get and Put Semaphores

The final option is to use both a get and a put semaphore. The buffer and the semaphores are initialized as described in the previous sections.

For each **Put()**, a **P()** call is made to the put semaphore *before* the item is inserted, and a **V()** call is made to the get semaphore *after* the item is inserted:

- **Call P() for PutSemaphore (block until there is space)**
- **Buffer[PutIndex] = Item**
- **PutIndex = (PutIndex + 1) modulo BufferSize**
- **Call V() for GetSemaphore (indicate a new item)**

For each **Get()**, a **V()** call is made on the get semaphore *before* an item is removed, and a **P()** call is made on the put semaphore *after* removing an item from the buffer.

- **Call P() for GetSemaphore (block until there is an item)**
- **Item = Buffer[GettIndex]**
- **GetIndex = (GetIndex + 1) modulo BufferSize**
- **Call V() for PutSemaphore (indicate space available)**
- **Return Item**

This ring buffer with get and put semaphore is optimal in the sense that no time is wasted, and no error condition is returned on either full or empty queues. However, it cannot be used in any ISR, since both sides, **Put()** and **Get()**, use the **P()** call which is forbidden for ISRs. Thus the only application for this scheme would be the communication between tasks. Moreover, the disadvantages of put semaphores apply here as well.

3 Kernel Implementation

3.1 Kernel Architecture

Figure 3.1 shows the overall architecture of the kernel implementation.

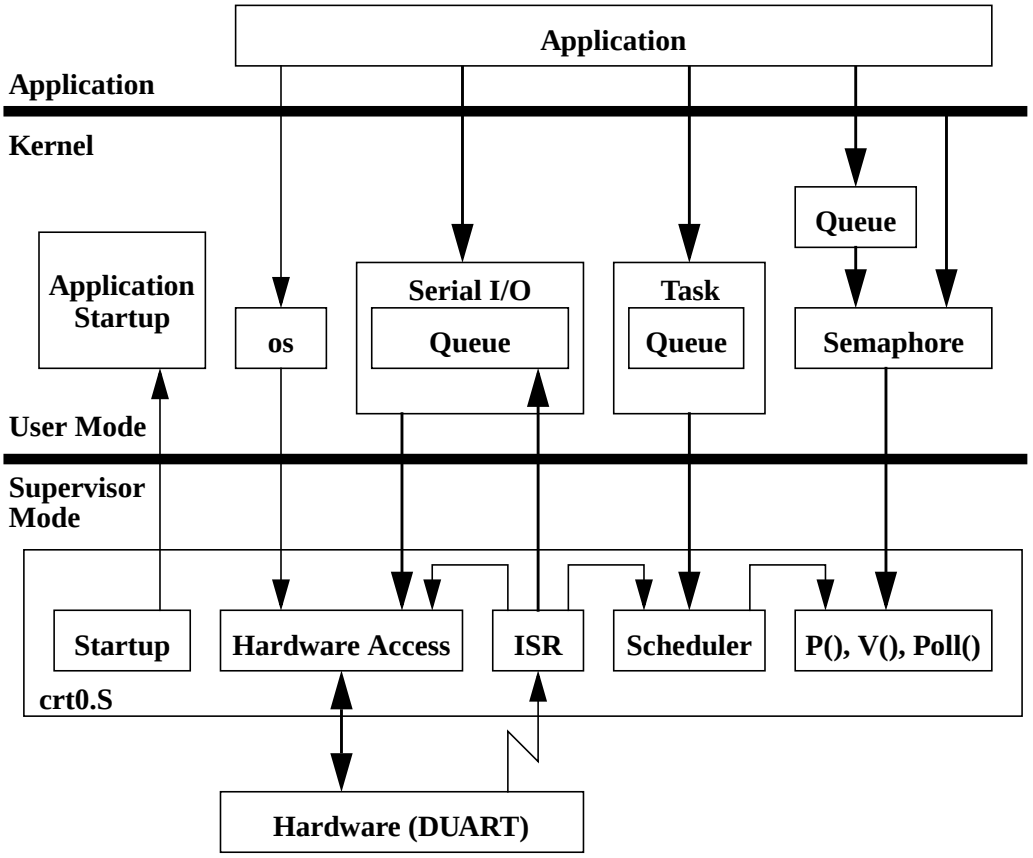

FIGURE 3.1 Kernel Architecture

The bottom part of Figure 3.1 shows the part of the kernel that is (along with the functions called from there) executed in supervisor mode. All code that is

executed in supervisor mode is written in assembler and is contained in the file **crt0.S**. The code in **crt0.S** is divided into the start-up code, functions for accessing the hardware, interrupt service routines, the task switch (scheduler), and the semaphore functions that are written in assembler for performance reasons.

The middle part of Figure 3.1 shows the rest of the kernel, which is executed in user mode. Any call to the code in **crt0.S** requires a change to supervisor mode, i.e. every arrow from the middle to the lower part is related to one or several TRAP instructions which cause a change to supervisor mode. Class **os** contains a collection of wrapper functions with TRAP instructions and enables the application to access certain hardware parts. The classes **SerialIn** and **SerialOut**, referred to as **Serial I/O**, require hardware access and are also accessed from the interrupt service routine. Class **Task** contains anything related to task management and uses the supervisor part of the kernel for (explicit) task switching. Task switching is also caused by the interrupt service routine. Class **Semaphore** provides wrapper functions to make the implementation of its member functions available in user mode. Several **Queue** classes are used inside the kernel and are also made available to the application; most of them use class **Semaphore**.

Normally, an application is not concerned with the internal kernel interfaces. The relevant interfaces towards the kernel are those defined in classes **os**, **SerialIn**, **SerialOut**, **Task**, **Queue,** and sometimes **Semaphore.**

3.2 Hardware Model

In order to understand the kernel implementation, we need some information about the underlying hardware:

- **Which processor type is used?**
- **How is the memory of the processor mapped?**
- **Which peripherals are used?**
- **Which interrupt assignment of the peripherals are used?**
- **How do the peripherals use the data bus?**

For the implementation discussed here, the hardware described in the following sections is assumed.

3.2.1 Processor

We assume that any processor of the Motorola MC68000 family is used. The implementation works for the following processors:

- **MC68000**
- **MC68008**
- **MC68010**
- **MC68012**
- **MC68020**
- **MC68030**
- **MC68040**
- **CPU32**

Note that out of this range of processors, only the MC68020 has been tested. For use of other chips, see also Section 3.2.5.

3.2.2 Memory Map

We assume the following memory map for the processor:

- **(E)EPROM at address 0x00000000..0x0003FFF**
- **RAM at address 0x20000000..0x2003FFF**
- **DUART at address 0xA0000000..A000003C**

The EPROM and RAM parts of the memory map are specified in the **System.config** file.

```
1  #define ROMbase 0x00000000
2  #define ROMsize 0x00040000
3  #define RAMbase 0x20000000
4  #define RAMsize 0x00040000
```

3.2.3 Peripherals

We assume a MC68681 DUART with two serial ports, a timer, and several general purpose input and output lines.

The DUART base address, along with the addresses of the various DUART registers, is contained in the file **duart.hh**.

```
5  #define DUART           0xA0000000
```

3.2.4 Interrupt Assignment

We assume the DUART may issue interrupts at level 2 to the CPU. We further assume that the interrupt vector is determined by the interrupt level (i.e. the vector is a so called autovector) rather than by the DUART.

3.2.5 Data Bus Usage

We assume the DUART is connected to data lines D16..D23 of a MC68020, and that it indicates WORD size for read accesses because of the considerable turn-off time of 150 nS for the data bus of the MC68681 as well as for many other peripherals. For a MC68020 running at 20 MHz, the timing to deal with is as shown in Figure 3.2.

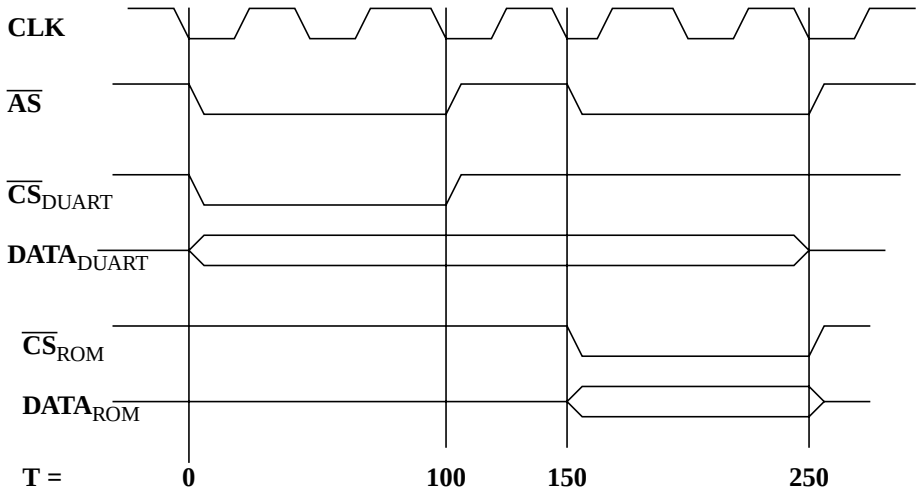

FIGURE 3.2 Data Bus Contention

After deasserting the DUART's chip select, the DUART needs a long time to three-state its data bus. This causes contention on the data bus between the DUART and the device addressed with the next cycle, which is usually a ROM or RAM. Adding wait states does not help here: this way, the $\overline{CS}_{DUART}$ would merely be extended, while the contention remains as it is. The standard way of dealing with this contention is to separate the DUART from the CPU's data bus by means of a bidirectional driver, which is switched on with the DUART's chip select $\overline{CS}_{DUART}$. But this solution requires an additional driver, and frequently cost limits, PCB space, or components do not allow for this.

For the MC68000 family, this problem can also be solved in a different way: by generating two read cycles towards the DUART instead of one read cycle only. However, only in the first cycle, a $\overline{CS}_{DUART}$ is generated, while the second is a dummy cycle allowing the DUART to completely three-state its data bus. For higher speeds, the dummy cycle can be extended by wait states.

As the CPUs of the MC68000 family have different memory interfaces, the way to implement such a dummy cycle depends on the CPU used.

For MC68020, MC68030, and MC68040 CPUs, the CPU executes a LONG move from the peripheral. This causes the CPU's SIZ0 and SIZ1 to request a LONG read cycle from the peripheral. The peripheral would, however, indicate a WORD size at the end of the cycle. The CPU then automatically initiates another cycle with size WORD in order to get the missing data. This second cycle is the dummy cycle. The actual value read by the CPU contains only one valid byte from the peripheral (in D23..D16 or D31..D24, depending on where the peripheral is located on the data bus). The remaining three bytes read are invalid. If the SIZ0 and SIZ1 lines are properly decoded, generating a bus error for all other sizes, this method is safe even in the case of software faults.

For the MC68000, MC68010 and MC68012, such dynamic bus resizing is not possible. However, the data bus size of the peripheral is limited to WORD size anyway for these CPUs. Unfortunately, these CPUs do not provide SIZ0 and SIZ1 lines to indicate the size of a cycle. Instead, the A1 address line has to be decoded in order to distinguish between the first cycle towards the peripheral and the following dummy cycle. This method is not entirely safe though: by mistake, one might access the address for the dummy cycle first.

Finally, for the MC68008, which has a 8 bit data bus only, the same method as for the MC68000 can be used, except that a WORD read cycle instead of a LONG read cycle is executed, and address line A0 is used instead of A1. The CPU splits this WORD read cycle into two BYTE read cycles automatically. Surprisingly, this method is safe again, because a word read to an odd address causes an address error trap.

In any case, some part of the data bus is undefined. The CPUs of the MC68000 family may change their Z (zero) and N (negative) flag depending on the data read. There is a negligeable chance that these flags become metastable inside the CPU when the floating part of the data bus changes just in the moment when the data lines are latched by the CPU. Although most likely this has no effect in practice, one should use a **move** instruction that does not change any status bits, for example MOVEM. It is primarily intended for moving several registers, but can serve for this particular purpose as well. In the case of a MC68008 CPU, i.e when using MOVEM.W, be aware of a strange inconsistency of the MOVEM

instruction that causes the lower word of a data register to be sign-extended into the upper word. That is, .W refers to the source size only. Failing to save the upper word of the register is a common mistake that is hard to detect, since it usually occurs in an interrupt service routine.

As a result, **crt0.S** contains the following two lines for all CPUs of the MC68000 family except for MC68008:

```
136           MOVEM.L rDUART_ISR, D7          | get interrupt sources
137           SWAP    D7                      |
```

For the MC68008, the above lines need to be replaced by the following code:

```
              MOVEM.W rDUART_ISR, D7          | CCAUTION: D7.W is sign-extended !!!
              ASR.W   #8, D7                  |
```

3.3 Task Switching

The MC68000 family of microprocessors which is used for our implementation provides two basic modes of operation: the *user mode* and the *supervisor mode*. (The 68020 microprocessors and higher also feature a sub-mode of the supervisor mode, the *master mode*, which allows for a cleaner implementation of interrupt handling. But for compatibility reasons, we did not use it here.) In user mode, only a subset of the instructions provided by the microprocessor can be executed. An attempt to execute a *privileged instruction* (that is, an instruction not allowed in user mode) causes a *privilege violation exception* to be executed instead of the instruction. Usually, C++ compilers do no generate any privileged instructions. The microprocessor indicates its present mode also to the hardware by its FC2 output. This way, certain hardware parts, such as the DUART in our implementation, are protected against inadvertent accesses from user mode.

One could ignore the user mode feature and run the whole system in supervisor mode. A task could then e.g. write to a hardware register at address **reg** directly from C++:

```
*(unsigned char *)reg = data;
```

This method is commonly used for processors that have no separate user and supervisor modes. But the price paid for this simplicity is a considerable loss of protection.

The MC68000 family evolved in such a way that the distinction between user and supervisor mode could be applied to memory accesses also by using a hardware memory management unit (MMU). From the MC68040 on, this MMU was even integrated in the microprocessor chip. By using a MMU, tasks are completely protected against each other. Therefore, we chose not to take the easy way, but to used the separate user and supervisor modes: regular task code is run in user mode, while code accessing critical resources is run in supervisor mode. Such critical resources are peripherals as for example our DUART, or the interrupt mask of the processor.

Sometimes, plotting the mode (*U* is user mode, *S* is supervisor mode) together with the interrupt level against time proves to be useful. A typical plot is shown in Figure 3.3. In our system, we use only one interrupt at level 2. Thus the only interrupt mask levels that make sense in our system are 0 (all interrupts will be served), 2 (only interrupts above level 2 will be served), and 7 (only non-maskable interrupts will be served). Regular task code runs in user mode, with all interrupts enabled (indicated by *U0*). In some cases, in particular when performing operations on queues, interrupt service routines must be prevented from changing a queue’s variables. The can be easily achieved by disabling interrupts even in user mode, *U7*. In user mode, other interrupt levels than the

ones cited above are rarely used, because one would have to analyze carefully which data structures could be modified at which interrupt level. Changing interrupt levels would then mean repeating this analysis, which is an error-prone procedure.

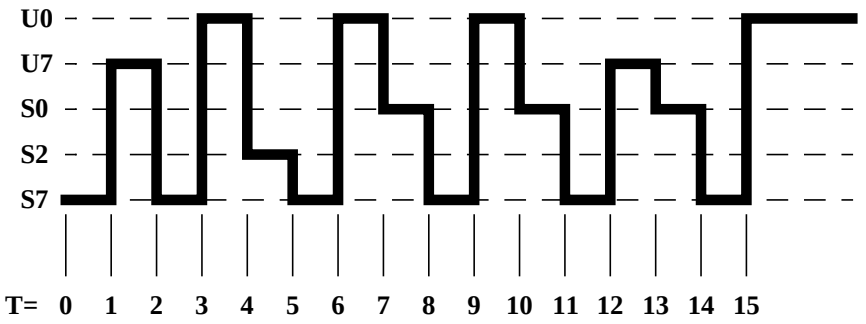

FIGURE 3.3 Modes and Interrupts vs. Time

As shown in the above figure, the system starts at *T=0* in supervisor mode, with all interrupts disabled. After initialization, the first task (which is the idle task explained later) starts execution at *T=1*, with interrupts still disabled. The idle task sets up other tasks and enables interrupts in the hardware. At *T=2*, the idle task wants to lower the interrupt mask to 0. Since this is a privileged instruction, it has to enter supervisor mode, change interrupt mask and return to user mode with interrupts enabled at *T=3*. At this point, that is at *T=4*, interrupts from the hardware are accepted by the CPU. The interrupt changes to supervisor mode and automatically sets the interrupt level to 2. As we will see later, in our implementation we will always check for possible task switches before returning to user mode. This check is made with interrupts disabled. Hence every return to user mode is from *S7*. Thus at *T=5*, the interrupt processing is finished, and a check for task switching is made with interrupts disabled. At *T=6*, this check is finished, and the CPU returns to user mode, which may be code of the same task or a different one. At *T=7*, a task performs a protected operation in supervisor mode, such as writing to a hardware register. Like before, it returns to user mode (via *S7* at *T=8*) at *T=9*. Next, we see a task intending to raise the interrupt level in order to modify a critical data structure. It does so by entering supervisor mode at *T=10* and returning to user mode in the usual way (via *S7* at *T=11*), but with interrupts disabled, at *T=12*. After finishing the critical section, it enters supervisor mode again at *T=13* and returns to user mode with interrupts enabled (via *S7* at *T=14*) at *T=15*.

As already mentioned, we check for tasks switches at every return to user mode. Instead, it would also be possible to switch tasks immediately, whenever desired. However, it is of no use to switch tasks while in supervisor mode, as the task switch would come into effect only at return to user mode. Switching tasks immediately could lead to several task switches while in supervisor mode, but only one of these task switches would have any effect. It is thus desirable to avoid unnecessary task switches and delay the decision whether to switch tasks until returning to user mode. Since task switching affects critical data structures, interrupts are disabled when tasks are actually switched.

As explained in Section 2.3, each task is represented by a Task Control Block, *TCB*. This TCB is implemented as an instance of the class **Task**. This class contains all functions necessary for managing tasks. For task switching, the following members of class **Task** are relevant:

```
 25   class Task
 26   {
...
 30      Task * next;                                        // 0x00
...
 32      unsigned long Task_D0, Task_D1, Task_D2, Task_D3;   // 0x08..
 33      unsigned long Task_D4, Task_D5, Task_D6, Task_D7;   // 0x18..
 34      unsigned long Task_A0, Task_A1, Task_A2, Task_A3;   // 0x28..
 35      unsigned long Task_A4, Task_A5, Task_A6;            // 0x38..
 36      unsigned long * Task_USP;                           // 0x44..
 37      void (*Task_PC)();                                  // 0x48
 38      unsigned long TaskSleep;                            // 0x4C
...
 40      unsigned short  priority;                           // 0x54
 41      unsigned char Task_CCR;                             // 0x56
 42      unsigned char TaskStatus;                           // 0x57
...
 71      static void Dsched()
 72         { asm("TRAP #1"); };
...
108      enum { RUN        = 0x00,
109             BLKD       = 0x01,
110             STARTED    = 0x02,
111             TERMINATED = 0x04,
112             SLEEP      = 0x08,
113             FAILED     = 0x10,
114           };
...
132      static Task *              currTask;
...
139   };
```

The variables **Task_D0**..**Task_D7**, **Task_A0**..**Task_A6**, **Task_USP**, **Task_PC** and **Task_CCR** provide space for saving the corresponding CPU registers when a task is swapped out.

The **Task** pointer **next** is used to find the next TCB, while the task's priority and status are analyzed in order to find the next task to be run at a task switch.

currTask points to the task currently running. This variable is static, i.e. it is shared by all instances of the class **Task**.

The easiest way to trigger a task switch is to explicitly de-schedule a task, which is implemented as the inline function **Dsched()**. This function merely executes a **Trap #1** instruction. This instruction causes the CPU to enter supervisor mode and to continue execution at an address specified by a vector associated with the instruction (see also **crt0.S** in Appendix A.1).

```
 58           .LONG   _deschedule             | 33     TRAP #1 vector
...
228   |-------------------------------------------------------------------------|
229   |               TRAP #1 (SCHEDULER)                                       |
230   |-------------------------------------------------------------------------|
231                                           |
232   _deschedule:                            |
233           ST      _consider_ts            | request task switch
234                                           |
235   _return_from_exception:                 | check for task switch
...
418   _consider_ts:   .BYTE   0              | true if task switch need be checked
```

So executing **Trap #1** causes the CPU to proceed in supervisor mode at label **_deschedule**. There, a flag called **_consider_ts** is set, and the common code for all returns to user mode is executed. It is this common code that may actually perform the task switch.

Upon entering supervisor mode, the CPU automatically creates an *exception stack frame* on its *supervisor stack*, as shown in Figure 3.4:

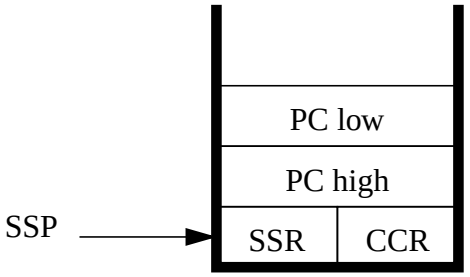

FIGURE 3.4 Exception Stack Frame

Let us have a closer look at the code after label **_return_from_exception**. First of all, all interrupts are disabled, so that this code is not interrupted before the exception is completely handled:

```
235   _return_from_exception:                | check for task switch
236           OR.W    #0x0700, SR            | disable interrupts
```

Then the stack frame is analyzed to determine in which mode the exception occurred. If the supervisor bit is set (0x2000 in the SR), then the exception occurred in supervisor mode, and the task switch shall thus be deferred until returning to user mode. If the exception occurred in user mode, but with nonzero interrupt level (SR & 0x0700) in user mode, then the task switch shall be deferred as well, since the task has disabled interrupts. That is, whenever (SR & 0x2700) is nonzero, the task switch shall not be performed, and the CPU directly returns from the exception:

```
237           MOVE.W  (SP), -(SP)           | get status register before exception
238           AND.W   #0x2700, (SP)+          | supervisor mode or ints disabled ?
239           BNE     L_task_switch_done      | yes dont switch task
...
304   L_task_switch_done:                    |
305           RTE                            |
```

Otherwise, it is checked whether a task switch is required at all. In our case, this was true, since we have unconditionally set **_consider_ts**. In certain situations, **_consider_ts** is not set; for example when unblocking a task that has a lower priority than the current task. Then case the CPU merely returns from the exception:

```
240           TST.B   _consider_ts           | task switch requested ?
241           BEQ     L_task_switch_done      | no
```

At this point, we initiate a task switch. First, **_consider_ts** is reset to prevent further task switches. Then the CPU registers are stored in the current TCB. Since we may not destroy any CPU registers here, we save A6 onto the stack and restore it back to the TCB afterwards:

```
242           CLR.B   _consider_ts           | reset task switch request
243                                          |
244   |--------------------------------------|
245   |       swap out current task by saving
246   |       all user mode registers in TCB
247   |--------------------------------------|
248                                          |
249           MOVE.L  A6, -(SP)              | save A6
250           MOVE.L  __4Task$currTask, A6   |
251           MOVEM.L D0-D7/A0-A5, Task_D0(A6)| store D0-D7 and A0-A5 in TCB
252           MOVE.L  (SP)+, Task_A6(A6)     | store saved A6        in TCB
```

Swapping out the task is completed by saving the USP (i.e., A7 when in user mode), the CCR, and the PC of the current task into the TCB:

```
253           MOVE    USP, A0                |
254           MOVE.L  A0, Task_USP(A6)       | save USP              in TCB
255           MOVE.B  1(SP), Task_CCR(A6)    | save CCR from stack   in TCB
256           MOVE.L  2(SP), Task_PC(A6)     | save PC  from stack   in TCB
257                                          |
```

Now all data belonging to the current task are saved in their TCB. We are free to use the CPU registers from here on. The next step is to find the next task to run: by chasing the **next** pointer of the current task, until the current task is reached again. We use A2 to mark where the search started. The task we are looking for is the one with the highest priority in state RUN (i.e. 0). If the current task is in state RUN, then we need not consider tasks with lower priority, which speeds up the search loop. Otherwise we make sure that at least the idle task will run in case no other task can:

```
258   |--------------------------------------|
257   |       find next task to run
260   |       A2: marker for start of search
261   |       A6: best candidate found
262   |       D6: priority of task A6
263   |       A0: next task to probe
264   |       D0: priority of task A0
265   |--------------------------------------|
266                                          |
267           MOVE.L  __4Task$currTask, A2   |
268           MOVE.L  A2, A6                 |
269           MOVEQ   #0, D6                 |
270           TST.B   TaskStatus(A6)         | status = RUN ?
271           BNE     L_PRIO_OK              | no, run at least idle task
272           MOVE.W  TaskPriority(A6), D6   |
273   L_PRIO_OK:                             |
274           MOVE.L  TaskNext(A6), A0       | next probe
275           BRA     L_TSK_ENTRY            |
```

The search loop skips all tasks which are not in state RUN or have a lower priority than the last suitable task found. If several tasks in state RUN have the same priority, the first of these tasks is chosen. The best candidate found is stored in A6:

```
276   L_TSK_LP:                              |
277           TST.B   TaskStatus(A0)         | status = RUN ?
278           BNE     L_NEXT_TSK             | no, skip
277           MOVEQ   #0, D0                 |
280           MOVE.W  TaskPriority(A0), D0   |
281           CMP.L   D0, D6                 | D6 higher priority ?
282           BHI     L_NEXT_TSK             | yes, skip
283           MOVE.L  A0, A6                 |
284           MOVE.L  D0, D6                 |
285           ADDQ.L  #1, D6                 | prefer this if equal priority
286   L_NEXT_TSK:                            |
287           MOVE.L  TaskNext(A0), A0       | next probe
288   L_TSK_ENTRY:                           |
289           CMP.L   A0, A2                 |
290           BNE     L_TSK_LP               |
291                                          |
```

Here, A6 points to the TCB of the next task which is to run and which is set as current task. In the same way as the previous task was swapped out, the new current task is swapped in. First, the CCR and PC in the exception stack frame are replaced by that of the new current task:

```
292   |--------------------------------------|
293   |       next task found (A6)
294   |       swap in next task by restoring
295   |       all user mode registers in TCB
296   |--------------------------------------|
297                                          |
298           MOVE.L  A6, __4Task$currTask   | task found.
299           MOVE.L  Task_PC(A6), 2(SP)     | restore PC  on stack
300           MOVE.B  Task_CCR(A6), 1(SP)    | restore CCR on stack
```

Then the USP and registers for the new current task are restored, and the CPU returns from exception processing. This way, the execution would normally be continued where the former current task was interrupted. But since we have replaced the return address and CCR of the stack frame by that of the new current task, execution proceeds where the new current task was interrupted instead:

```
301           MOVE.L  Task_USP(A6), A0       |
302           MOVE    A0, USP                | restore USP
303           MOVEM.L Task_D0(A6), D0-D7/A0-A6| restore D0-D7, A0-A5 (56 bytes)
304   L_task_switch_done:                    |
305           RTE                            |
```

3.4 Semaphores

Semaphores are declared in file **Semaphore.hh**. Although they could be implemented in C++, we will see that they are best implemented in assembler. Thus, there is no Semaphore.cc file. The interface to the assembler routines is specified inline in **Semaphore.hh**.

3.4.1 Semaphore Constructors

One of the most common special cases for semaphores are semaphores representing a single resource that is available from the outset. We have chosen this case for the default constructor. Semaphores representing 0 or more than one resources initially can be constructed by providing the initial count:

```
13      Semaphore()        : count(1),   nextTask(0) {};
14      Semaphore(int cnt) : count(cnt), nextTask(0) {};
```

3.4.2 Semaphore Destructor

There is no destructor for semaphores. In general, it is dangerous to destruct semaphores at all. If a semaphore with a counter value < 0 is deleted, then the tasks in the waiting queue would either be unblocked (although most likely the resource they are waiting for would not be available), or blocked forever. In the first case, the semaphore would need to return an error code which would need to be checked after any **P()** operation. This is not very handy, so we made **P()** a function returning no value at all. Generally, semaphores should have an infinite lifetime, i.e. they should be static.

However, sometimes dynamic semaphores can be useful. In these cases, it is the responsibility of the programmer to make sure that the semaphore dies in the correct way.

3.4.3 Semaphore P()

The **P()** member function could be written in C++. While the semaphore and possibly the chain of waiting tasks is modified, interrupts must be disabled:

```
void Semaphore::P()
{
   oldIntMask = os::set_INT_MAK(7);   // disable interrupts

   counter --;

   if (counter < 0)                     // if no resource available
      {
```

```
        consider_ts = 1;                // task switch required
        CurrentTask->Status |= BLKD;    // block current task
        CurrentTask->nextWaiting = 0;   // current task is  end of waiting chain

        if (nextTask == 0)      // no other task waiting
           {
              nextTask = CurrentTask;         // head of task waiting chain
           }
        else
           {
             Task * t = nextTask;

             // find end of task waiting chain...
             while (t->nextWaiting;)   t = t->nextWaiting;

             // here t is the last task in the waiting chain
             t->nextWaiting = CurrentTask;
           }
      }

   os::set_INT_MASK(oldIntMask);   // restore interrupt level
   return;
}
```

Note that the actual task switch would happen at the second **set_INT_MASK()** call, when the corresponding exception processing changes back to user mode. Disabling and enabling interrupts would cause two TRAP instructions for the **set_INT_MASK()** calls and for the relevant check for task switches at the end of exception processing. Compared to an assembler implementation, this would be a significant overhead. Considering that semaphores are used by higher level data structures, such as queues, as well as in every character I/O interrupt service routine (**V()** only), this overhead should be avoided by implementing all **Semaphore** member functions in assembler (see also **crt0.S** in Appendix A.1). For the **P()** function, we use TRAP #3 to switch to supervisor mode, passing the semaphore in register A0 and telling the compiler that D0 might be changed, so that we do not need to save it.

```
15      void P()    {
16                    asm volatile ("MOVE.L %0, A0
17                                   TRAP   #3" : : "g"(this) : "d0", "a0");
18                  };
```

In **crt0.S**, the TRAP #3 vector points to the actual assembler code for **P()**:

```
60           .LONG   _Semaphore_P            | 35     TRAP #3 vector
```

The assembler code is only slightly longer than the C++ code. Since this is an exception handling routine, we do not need to restore the interrupt level at the end.

```
307   |-------------------------------------------------------------------------|
308   |               TRAP #3 (Semaphore P operation)                           |
309   |-------------------------------------------------------------------------|
310                                         |
311   _Semaphore_P:                         | A0 -> Semaphore
```

```
312           OR      #0x0700, SR             | disable interrupts
313           SUBQ.L  #1, SemaCount(A0)       | count down resources
314           BGE     _return_from_exception  | if resource available
315           ST      _consider_ts            | request task switch
316           MOVE.L  SemaNextTask(A0), D0    | get waiting task (if any)
317           BNE.S   Lsp_append              | got a waiting task
318           MOVE.L  __4Task$currTask, D0    | get current Task
319           MOVE.L  D0, SemaNextTask(A0)    | store as first waiting
320           MOVE.L  D0, A0                  |
321           BSET    #0, TaskStatus(A0)      | block current task
322           CLR.L   TaskNextWaiting(A0)     | say this is last waiting
323           BRA     _return_from_exception  | done
324                                           |
325   Lsp_append:                             | goto end of waiting list
326           MOVE.L  D0, A0                  |
327           MOVE.L  TaskNextWaiting(A0), D0 | get next waiting (if any)
328           BNE.S   Lsp_append              | if not last waiting
329                                           |
330           MOVE.L  __4Task$currTask, D0    | get current task
331           MOVE.L  D0, TaskNextWaiting(A0) | store as last waiting
332           MOVE.L  D0, A0                  |
333           BSET    #0, TaskStatus(A0)      | block current task
334           CLR.L   TaskNextWaiting(A0)     | say this is last waiting
335           BRA     _return_from_exception  | done
336                                           |
```

3.4.4 Semaphore Poll()

The **Poll()** member function is the simplest semaphore. In C++ we would have the following lines of code:

```
void Semaphore::Poll()
{
int result = 1;   // assume no resource avaliable

   oldIntMask = os::set_INT_MAK(7);   // disable interrupts

   if (counter > 0)
      {
        counter--;
        result = 0;
      }

   os::set_INT_MASK(oldIntMask);   // restore interrupt level
   return result;
}
```

Like for **P()**, we implement this in assembler, using TRAP #5:

```
 23      int Poll()  {
 24                    int r;
 25
 26                    asm volatile ("MOVE.L %1, A0
 27                                   TRAP   #5
 28                                   MOVE.L D0, %0"
 29                                   : "=g"(r) : "g"(this) : "d0", "a0");
 30                    return r;
 31                  };
```

In **crt0.S**, the TRAP #5 vector points to the actual assembler code for **Poll()**:

```
 62           .LONG   _Semaphore_Poll         | 37     TRAP #5 vector
```

And the code is straightforward:

```
363   |-------------------------------------------------------------------------|
364   |               TRAP #5 (Semaphore Poll operation)                        |
365   |-------------------------------------------------------------------------|
366                                           |
367   _Semaphore_Poll:                        | A0 -> Semaphore
368           OR      #0x700, SR              | disable interrupts
369           MOVEQ   #1, D0                  | assume failure
370           TST.L   SemaCount(A0)           | get count
371           BLE     _return_from_exception  | failure
372           SUBQ.L  #1, SemaCount(A0)       |
373           MOVEQ   #0, D0                  | success
374           BRA     _return_from_exception  | check for task switch
375                                           |
```

3.4.5 Semaphore V()

The last member function required is **V()**. Again, we provide a C++ implementation first to understand the assembler code:

```
void Semaphore::V()
{
   oldIntMask = os::set_INT_MAK(7);   // disable interrupts

   counter ++;

   if (counter <= 0)                     // if any task waiting
      {
        Task * head = nextTask

        nextTask = head->nextWaiting;   // remove head of waiting chain
        head>Status &= ~BLKD;           // unblock head of waiting chain

        if (CurrentTask->priority < head->priority)
           consider_ts = 1;                // task switch required
      }

   os::set_INT_MASK(oldIntMask);   // restore interrupt level
   return;
}
```

The comparison `(CurrentTask->priority < head->priority)` is crucial for the entire system performance. If we always set **consider_ts**, then e.g. any character received, for which a lower priority task is waiting, would swap out and in again every higher priority task. In contrast to **P()**, **V()** may be used in interrupt service routines. Thus performance is even more critical, and **V()** is implemented in assembler:

```
19      void V()    {
20                    asm volatile ("MOVE.L %0, A0
21                                   TRAP   #4" : : "g"(this) : "d0", "a0");
22                  };
```

This time, TRAP #4 is used:

```
61            .LONG   _Semaphore_V            | 36     TRAP #4 vector
```

The assembler code for **V()** is as follows:

```
337   |-----------------------------------------------------------------------|
338   |               TRAP #4 (Semaphore V operation)                         |
339   |-----------------------------------------------------------------------|
340                                           |
341   _Semaphore_V:                           | A0 -> Semaphore
342           OR      #0x0700, SR             | disable interrupts
343           ADDQ.L  #1, SemaCount(A0)       |
344           BLE.S   Lsv_unblock             | unblock waiting task
345           CLR.L   SemaNextTask(A0)        |
346           BRA     _return_from_exception  | done
347                                           |
348   Lsv_unblock:                            |
349           EXG     D0, A1                  |
350           MOVE.L  SemaNextTask(A0), A1    | get next waiting task
351           MOVE.L  TaskNextWaiting(A1), SemaNextTask(A0)
352           MOVE.L  A1, A0                  |
353           EXG     D0, A1                  |
354           BCLR    #0, TaskStatus(A0)      | unblock the blocked task
355           CLR.L   TaskNextWaiting(A0)     | just in case
356           MOVE.W  TaskPriority(A0), D0    | get priority of unblocked Task
357           MOVE.L  __4Task$currTask, A0    | get current Task
358           CMP.W   TaskPriority(A0), D0    | current prio >= unblocked prio ?
359           BLS     _return_from_exception  | yes, done
360           ST      _consider_ts            | no, request task switch
361           BRA     _return_from_exception  | done
362                                           |
```

Up to now, we have presented almost all of the code written in assembler. So it is time to relax by looking at some simple C++ code.

3.5 Queues

As we already saw, there are different kinds of queues, depending on where semaphores are used. But common to all queues is a ring buffer. Hence we implement ring buffers as a separate class from which the different queues are derived. Since a ring buffer may contain any kind of items, we make a template class called **RingBuffer**.

```
  1   // Queue.hh
...
 12   template <class Type> class RingBuffer
 13   {
 14   public:
 15      RingBuffer(unsigned int Size);
 16      ~RingBuffer();
 17
 18      int IsEmpty() const { return (count)        ? 0 : -1; };
 19      int IsFull()  const { return (count < size) ? 0 : -1; };
 20
 21      int Peek(Type & dest) const;
 22
 23   protected:
 24      enum { QUEUE_OK = 0, QUEUE_FAIL = -1 };
 25
 26      virtual int  PolledGet(Type & dest) = 0;
 27      virtual int  PolledPut(const Type & dest) = 0;
 28      inline  void GetItem(Type & source);
 29      inline  void PutItem(const Type & src);
 30
 31      unsigned int size;
 32      unsigned int count;
 33
 34   private:
 35      Type *       data;
 36      unsigned int get;
 37      unsigned int put;
 38   };
```

3.5.1 Ring Buffer Constructor and Destructor

The constructor initializes the **put** and **get** indices to 0, the **count** of items in the buffer to 0, and stores the **size** of the buffer. Then the constructor allocates a buffer big enough to store **size** instances of class **Type**.

```
  1   // Queue.cc
...
  9   template <class Type> RingBuffer<Type>::RingBuffer(unsigned int Size)
 10      : size(Size), get(0), put(0), count(0)
 11
 12   {
 13      data = new Type[size];
 14   :
```

The destructor releases the memory allocated for the buffer.

```
  1   // Queue.cc
```

```
...
 16   template <class Type> RingBuffer<Type>::~RingBuffer()
 17   {
 18      delete [] data;
 19   }
```

3.5.2 RingBuffer Member Functions

The member functions **IsEmpty()** and **IsFull()** are self-explanatory. **Peek(Type & dest)** returns **QUEUE_FAIL** (i.e. nonzero) if the queue is empty. Otherwise, it stores the next item in the queue in **dest**, but without removing it from the queue. The **Peek()** function is useful for scanners which usually require a single character look-ahead. Traditionally, a character looked ahead is pushed back into a queue by means of a function **unput(char)** if the character is not required. But this solution causes several problems. ??? Which problems ??? So providing a look-ahead function like **Peek()** is the better solution, as it does not remove any item from the queue.

```
  1   // Queue.cc
...
 21   template <class Type> int RingBuffer<Type>::Peek(Type & dest) const
 22   {
 23   int ret = QUEUE_FAIL;
 24
 25      {
 26        os::INT_MASK old_INT_MASK = os::set_INT_MASK(os::NO_INTS);
 27        if (count)   { dest = data[get];   ret = QUEUE_OK; }
 28        os::set_INT_MASK(old_INT_MASK);
 29      }
 30      return ret;
 31   }
```

The member function **PutItem()** inserts, and **GetItem()** removes an item from the queue. However, **PutItem()** assumes that the queue is not full when it is called, and **GetItem()** assumes that the queue is not empty. This condition is not checked, because the check as such is different for queues that use semaphores and queues that do not use semaphores. Apart from that, interrupts are in general to be disabled when these functions are called. To avoid direct usage of these functions, they are made protected so that only classes derived from **RingBuffer** can use them.

```
 33   template <class Type> inline void RingBuffer<Type>::GetItem(Type & dest)
 34   {
 35      dest = data[get++];
 36      if (get >= size)   get = 0;
 37      count--;
 38   }
...
```

```
40   template <class Type> inline void RingBuffer<Type>::PutItem(const Type &src)
41   {
42     data[put++] = src;
43     if (put >= size)   put = 0;
44     count++;
45   }
```

Finally, it has shown to be useful to provide polled access to both ends of a queue, even if semaphores are used. For this purpose, the member functions **PolledGet()** and **PolledPut()** are used. Their implementation depends on where semaphores are used; thus they are purely virtual.

3.5.3 Queue Put and Get Functions

The polled and semaphore-controlled **Put()** and **Get()** for the four possible types of queues result in a total of 12 functions. Rather than explaining them all in detail, we only present the basic principles:

- **Interrupts are disabled while the ring buffer is accessed.**
- **For polled operation, if a semaphore is used at the polled end of the queue, the semaphore is polled as well in order to keep the semaphore synchronized with the item count.**
- **It is always checked if the queue is full before PutItem is called, and if the queue is empty before GetItem is called. This check is explicit if no semaphore is used at the respective ends, or implicit by polling the semaphore.**

3.5.4 Queue Put and Get Without Disabling Interrupts

In the implementation shown, the manipulation of the queue is always performed with interrupts enabled. Considering the short code, this causes a significant overhead. Often interrupts are already disabled anyway, e.g. in interrupt service routines. In those cases, one can derive other queue classes from RingBuffer that do not disable interrupts.

It should also be noted that the get and put ends of the queue are more or less independent of each other. As we have seen in **PutItem()** and **GetItem()**, the count is always modified **after** putting or getting an item. If incrementing or decreasing **count** is atomic (which is the case for most compilers), and if there is only one task or interrupt service routine at all (which is the case for most queues), then it is not necessary at all to disable interrupts. It may as well be the case that interrupts need to be disabled only at one end of a queue, e.g. for one task that receives messages from several other tasks. A good candidate for such optimizations are the character input and output queues for serial I/O.

3.6 Interprocess Communication

So far, we have considered different tasks as being independent of each other. Most often, however, some of the tasks in an embedded system have to exchange information. The simplest way for the tasks to enable this exchange is to share memory. One task updates a variable in the memory while another task reads that variable. Although shared memory is considered as the fastest way of exchanging information, this is only true for the information exchange as such. In addition to exchanging the information, the tasks have to coordinate when the information is valid (i.e. when it is provided by the sending task) and how long it is processed by the receiving task. This coordination could be implemented as a valid flag, which is initially set to invalid. After a task has provided information, it sets the flag to valid. The receiving task then processes the information and sets the flag back to invalid, so that the memory can be used again. Obviously, this procedure means busy wait for both tasks involved and is thus inefficient.

A much better way is to use queues containing messages for exchanging information. To avoid busy waiting at either end, both put and get semaphores are used. If the queue is full, the sending task is blocked until the receiving task has removed items. For small information quantities, such as characters or integers, the information can be stored in the message itself; for larger quantities, pointers to the information are used. This way, the performance of shared memory for the information exchange as such can be maintained. Using pointers is tricky in detail, since it needs to be defined whether the receiver or the sender must release the memory. For example, the receiver must release the memory if the memory is allocated with the **new** operator. The sender has to release the memory, e.g. if the memory is allocated on the senders stack; in this case, the sender needs to know when the receiver has finished processing of the message. If the memory is released by the sender, then the receiver typically sends an acknowledgment back to the sender to indicate that the memory is no longer needed. As a consequence, the receiver needs to know which task has sent the message.

Rather than defining a specific queue for each particular purpose, it is convenient to have the same data structure for messages in the whole system, as defined in **Message.hh** (see also Appendix A.9).

```
  1   // Message.hh
...
  5   class Message
  6   {
  7   public:
  8      Message()                : Type(0), Body(0), Sender(0) {};
  9      Message(int t, void * b) : Type(t), Body(b), Sender(0) {};
 10      int    Type;
 11      void * Body;
 12      const Task * Sender;
 13   };
```

This data structure contains a type that indicates the kind of message, a body that is optionally used for pointers to larger data structures, and a task pointer identifying the sender of the task.

Communication between tasks being so common, every task is provided with a message queue:

```
     // Task.hh
 25  class Task
 26  {
...
138     Queue_Gsem_Psem<Message>    msgQ;
139  };
```

The size of the message queue can be specified individually for each task in order to meet the task's communication requirements.

```
  1  // Task.cc
...
 33  Task::Task(void (*main)(),
...
 35              unsigned short  qsz,
...
 38             )
 39        : US_size(usz),
...
 44          msgQ(qsz),
```

As we know by now, every task executing code must be the current task. Thus a message sent is always sent by **CurrentTask**. Since **Message** itself is a very small data structure, we can copy the Type, Body and Sender members without loosing much of the performance. This copy is made by the **Put()** function for queues. The code for sending a message becomes so short that it makes sense to have it inline.

```
     // Task.hh
 96     void SendMessage(Message & msg)
 97        { msg.Sender = currTask;   msgQ.Put(msg); };
```

Note that **SendMessage()** is a non-static member function of class task. That is, the instance of the class for which **SendMessage()** is called is the receiver of the message, not the sender. In the simplest case, only a message type is sent, e.g. to indicate that an event has occurred:

```
void informReceiver(Task * Receiver, int Event)
{
   Message msg(Event, 0);
   Receiver->SendMessage(msg);
}
```

The sender may return from **informReceiver()** before the receiver has received the message, since the message is copied into the message queue. It is also safe to

send pointers to the **.TEXT** section of the program to the receiver (unless this is not prevented by hardware memory management):

```
void sayHello(Task * Receiver)
{
   Message msg(0, "Hello");
   Receiver->SendMessage(msg);
}
```

This ??? structure/function/code ??? is valid since “Hello” has infinite lifetime. It is illegal, however, to send dangling pointers to the receiver; as it is illegal to use dangling pointers in general:

```
void DONT_DO_THIS(Task * Receiver)
{
   char hello[6] = "Hello";
   Message msg(0, hello);
   Receiver->SendMessage(msg);   // DON’T DO THIS !!!
}
```

After the above function has returned, the pointer sent to the receiver points to the stack of the sender which is not well defined when the receiver gets the message.

The receiving task may call **GetMessage()** in order to get the next message it has been sent. This function is even shorter, so it is declared inline as well:

```
   // Task.hh
56    static void GetMessage(Message & msg)
57       { currTask->msgQ.Get(msg); };
```

The receiver uses **GetMessage()** as follows:

```
void waitForMessage()
{
   Message msg();
   Task::GetMessage(msg);
   switch(msg.Type)
      {
         ...
      }
}
```

This usage pattern of the **Message** class explains its two constructors: the constructor with **Type** and **Body** arguments is used by the sender, while the receiver uses the default constructor without any arguments that is updated by **GetMessage()** later on. A scenario where the sender allocates memory which is released by the receiver could be as follows: the sender sends integers 0, 1 and 2 to the receiver. The memory is allocated by new, rather than ??? pointing ??? on the stack like in the bad example above.

```
void sendData(Task * Receiver)
{
```

```
   int * data = new int[3];

   data[0] = 0;   data[1] = 1;   data[2] = 2;
   Message msg(0, data);
   Receiver->SendMessage(msg);
}
```

The receiver would then release the memory after having received the message:

```
void receiveData()
{
   Message msg();
   Task::GetMessage(msg);
    ...
    delete [] (int *)(msg.Body);
}
```

If a system uses hardware memory management (which is rarely the case for embedded systems today, but may be used more frequently in the future), the data transmitted must of course be accessible by both tasks.

The last scenario using new/delete is safe and provides sufficient flexibility for large data structures. Unfortunately, using new/delete is a bad idea for embedded systems in general. While resetting a PC twice a day is not uncommon, resets cannot be accepted for a robot on the mars. The safest but least flexible way of allocating memory is by means of static variables. Automatic allocation on the stack is a bit more risky, because the stack might overflow; but this solution is much more flexible. The ultimate flexibility is provided by new/delete, but it is rather difficult to determine the memory requirements beforehand, which is partly due to the fragmentation of the memory. The problem in the bad example above was the lifetime of the variable **hello**, which was controlled by the sender. This problem can be fixed by using a semaphore that is unlocked by the receiver after having processed the message:

```
class DataSemaphore
 {
   public:
      DataSemaphore() : sem(0) {};   // resource not available
      int data[3];
      Semaphore sem;
 }

void sendMessageAndWait(Task * Receiver)
{
   DataSemaphore ds;
   Message msg(0, ds);
   ds.data[0] = 0;   ds.data[1] = 1;   ds.data[2] = 2;
   Receiver->SendMessage(msg);
   ds.sem.P();
}
```

The sender is blocked as soon as it has sent the message, since the semaphore was initialized with its counter set to 0, indicating that the resource (i.e. the data) is not available. The receiver processes the message and unlocks it, which causes the sender to proceed:

```
void receiveDataAndUnlock()
{
   Message msg();
   Task::GetMessage(msg);
    ...
    ((DataSemaphore *)msg.Body).V();
}
```

Unfortunately, blocking the sender is a disadvantage of this otherwise perfect method. The sender may, however, proceed its operation as long as it does not return from the function. This is also one of the very few examples where a semaphore is not static. It does work here because both sender and receiver cooperate in the right way. Although we have not shown any perfect solution for any situation of interprocess communication, we have at least seen a set of different options with different characteristics. Chances are good that one of them will suit the particular requirements of your application.

3.7 Serial Input and Output

The basic model for serial input and output has already been discussed in Section 2.5.3 and presented in Figure 2.14. In principle, the input and output directions are completely independent of each other, except for the software flow control (e.g. XON/XOFF protocol) at the hardware side of the receive buffer, and possibly line editing functions (e.g. echoing of characters) at the task side of the receive buffer.

This section deals with the task side of both the receive and transmit buffers; the hardware side is discussed in Section 3.8. Strictly speaking, the aspects of serial input and output discussed here are not part of the operating system itself. But they are so commonly used that it is appropriate to include them in the kernel.

Several tasks sharing one serial input or output channel is a common source of trouble. A typical example is a router that receives data packets on several serial ports and transmits them (after possibly modifying them) on other serial ports. ??? What is the trouble ???An implementation with three serial ports could be as shown in Figure 3.5.

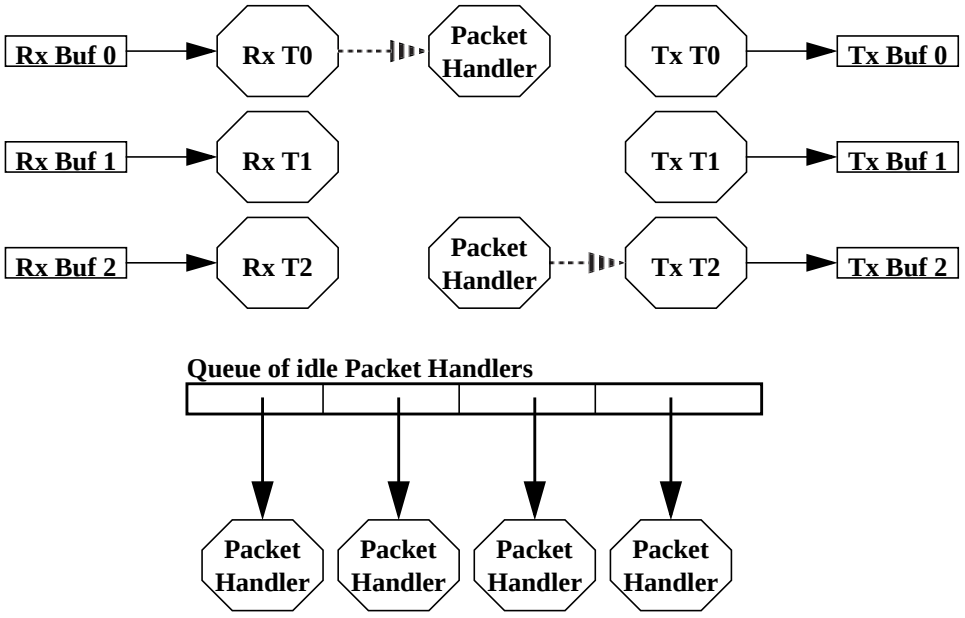

FIGURE 3.5 Serial Router (Version A)

For each serial port, there is a receive task (**RX T0..2**) that receives characters from its serial port. If a complete packet is received, the receive task fetches a pointer to an idle packet handler task and sends a message containing the packet to that task. The packet handler task processes the packet and may create other packets that are sent as messages to some of the transmit tasks (**Tx T0..2**). When a packet handler has finished processing a packet, it puts itself back into the queue of idle packet handlers. The transmit tasks merely send the packets out on their respective serial outputs. In this implementation, each serial input is handled by one task **Rx Ti**, and each serial output is handled by a task **Tx Ti** dedicated to that port. The main purpose of these tasks is to maintain atomicity at packet level. That is, these tasks are responsible for assembling and de-assembling sequences of characters into packets and vice versa. Since the receive and transmit tasks are statically bound to their serial ports, there is no conflict between tasks with regard to ports.

Now assume there is some mechanism by which a task can temporarily claim a serial input and output port for a period of time so that no other task can use that port at the same time. Then the number of tasks involved could be reduced as shown in Figure 3.6.

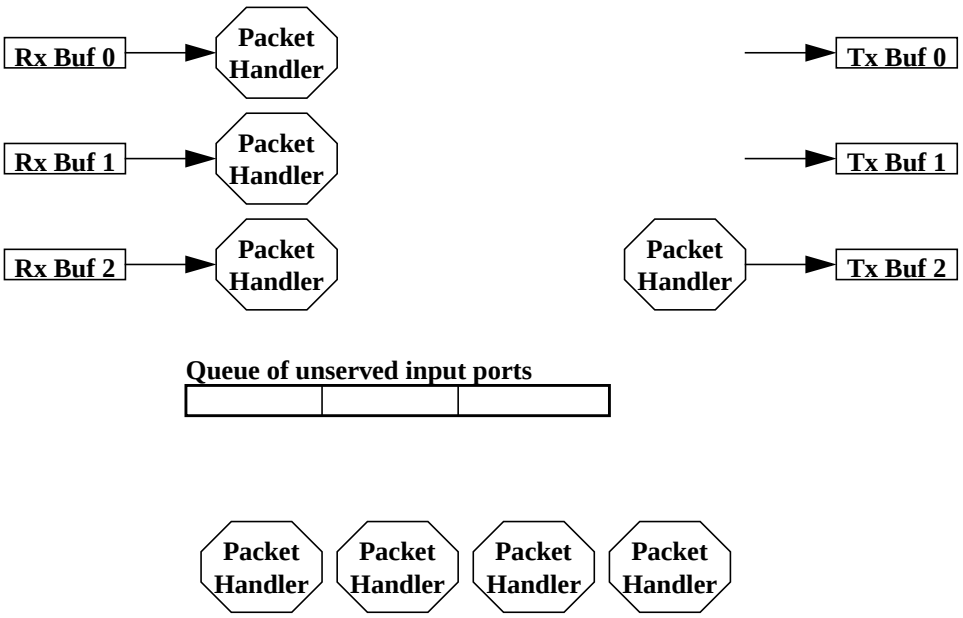

FIGURE 3.6 Serial Router (Version B)

At the output side, a packet handler merely claims a serial output port when it needs to transmit a packet. The queue of idle packet handlers has been replaced by a queue of input ports that have no packet handlers assigned; this queue initially contains all serial input ports. A packet handler first gets an unserved input port, so that shortly after start-up of the system each input port is served by a packet handler; the other packet handlers are blocked at the queue for unserved inputs. A packet handler serving an input first claims that input port and starts collecting the characters of the next packet. When a complete packet is received, the packet handler releases the input port (which causes the next idle packet server to take over that port), puts it back into the queue of unserved input ports, and continues processing of the packet. Like in router version A, this scheme schedules the packet handlers between the ports in a fair way. Sometimes, in particular if the serial ports need to have different priorities (e.g. due to different communication speeds), a scheduling on a per-port basis is required. This leads to an even simpler implementation shown in Figure 3.7.

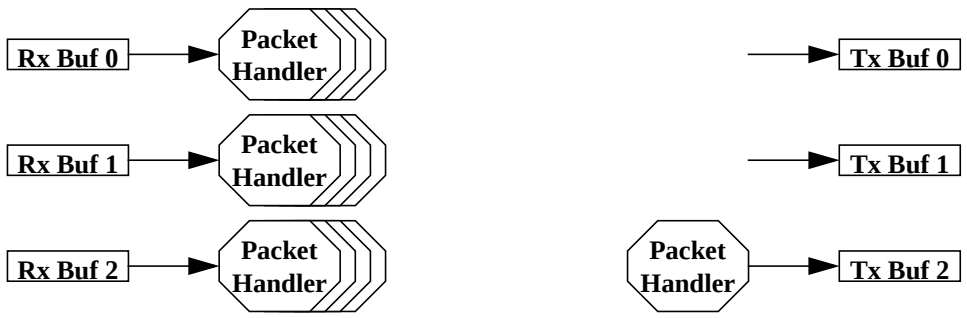

FIGURE 3.7 Serial Router (Version C)

With this implementation, one can e.g. assign different priorities to each input port and use different numbers of packet servers. The packet servers queue themselves by claiming the input port, so that the queue of unserved input ports used in version B becomes obsolete. As a consequence, no initialization of that queue is required. The code for the packet handler becomes as simple as that:

```
Semaphore Port_0_Input, Port_0_Output;
Semaphore Port_1_Input, Port_1_Output;
Semaphore Port_2_Input, Port_2_Output;

void packet_handler_main(Semaphore & Port_i_Input)
{
     for (;;)
          {
             Port_i_Input.P();
```

```
            char * Packet = getPacket(port);
            Port_i_Input.V();
            handlePacket(Packet);    // deletes Packet
         }
}
```

The semaphores control the claiming and releasing of the serial input and output ports. Using semaphores explicitly is not very elegant though. First, it must be assured that any task using a serial port is claiming and releasing the corresponding semaphore. Also it is often desirable to have a "dummy" port (such as */dev/nul* in UNIX) that behaves like a real serial port. Such a dummy port could be used e.g. to turn logging information on and off. But claiming and releasing dummy ports makes little sense. In general, the actual implementation of a port should be hidden from the interface using the port. Thus for a clean object-oriented design, the semaphores should be maintained by the port rather than by an application using the port. This leads to the kernel implementation of serial input and output described in the following sections.

3.7.1 Channel Numbers

It is convenient to refer to serial ports by channel numbers. In our hardware model, we assumed one DUART with two serial ports, which we call **SERIAL_0** and **SERIAL_1**. These are normally operated in an interrupt-driven manner. Sometimes however, it is required to have a polled operation available, in particular before the interrupt system has been initialized, and in the case of fatal system errors. For achieving this polled operation, the channels **SERIAL_0_POLLED** and **SERIAL_1_POLLED** are provided. Finally, the **DUMMY_SERIAL** channel is used when the actual serial output needs to be suppressed.

```
 1   // Channels.hh
...
 5   enum Channel {
 6                 SERIAL_0           = 0,
 7                 SERIAL_1           = 1,
 8                 SERIAL_0_POLLED    = 4,
 9                 SERIAL_1_POLLED    = 5,
10                 DUMMY_SERIAL       = 8,
11               };
```

Often, one would like to turn the serial output on and off, e.g. for debugging purposes. Therefore, channel variables rather than explicit channels are used:

```
 1   // Channels.hh
...
13   extern Channel MonitorIn;
14   extern Channel MonitorOut;
15   extern Channel ErrorOut;
```

```
16  extern Channel GeneralOut;
```

If the variable **ErrorOut** is used for e.g. debugging information, then this output can be suppressed or directed to any serial port by setting the **ErrorOut** variable to **DUMMY_SERIAL** or **SERIAL_0**/**1**. This can be done in a dynamic way and can be extended to several debugging levels by introducing new **Channel** variables in accordance with the various debugging levels.

3.7.2 SerialIn and SerialOut Classes and Constructors/Destructors

Since the serial input and output are mainly independent of each other, they are implemented as separate classes. The constructors and destructors are so similar, however, that they are described together.

As we already saw, a mechanism allowing a task to exclusively claim a serial (input or output) port for a certain period of time is required. Clearly, this mechanism will be based on a semaphore. A particularly elegant implementation of this mechanism is to create an object with a lifetime that is exactly the period during which the port is being claimed. The lifetime of an object is the time between the construction and the destruction of the object. Thus if we perform the semaphore **P()** operation inside the constructor and the **V()** operation inside the destructor, ??? was dann ???. For the **SerialOut** class, we get the following constructor:

```
 1  /* SerialOut.cc */
...
16  Semaphore SerialOut::Channel_0;
17  Semaphore SerialOut::Channel_1;
...
20  SerialOut::SerialOut(Channel ch) : channel(ch)
21  {
22     switch(channel)
23        {
24          case SERIAL_0:
25               if (Task::SchedulerRunning())   Channel_0.P();
26              else                            channel = SERIAL_0_POLLED;
27               return;
28
29          case SERIAL_1:
30               if (Task::SchedulerRunning())   Channel_1.P();
31              else                            channel = SERIAL_1_POLLED;
32               return;
33
34          case SERIAL_0_POLLED:
35          case SERIAL_1_POLLED:
36               return;
37
38          default:
39               channel = DUMMY_SERIAL;        // dummy channel
```

```
40              return;
41          }
42  }
```

Basically, the constructor performs a **P()** operation on the **Channel_0/1** semaphore associated with the channel. If another task tries to create a **SerialOut** object, then that task is blocked until the task that created the **SerialOut** object first has destroyed it again. The **SerialOut** object also stores for which channel it has been constructed, so that subsequent changes e.g. of a channel variable do not affect a **SerialOut** object. Note that the **P()** operation is only performed for those channels that are subject to access conflicts. If multitasking is not yet in effect (i.e. during system start-up), the construction is creating a polled serial port. Thus the code creating a **SERIAL_0/1** object will work even at system start-up.

The semaphores must be static and private to prevent abuse of the semaphores:

```
 1  /* SerialOut.hh */
...
12  class SerialOut
13  {
...
23  private:
...
36     static Semaphore Channel_0;
37     static Semaphore Channel_1;
...
44  };
```

The destructor performs the **V()** operation only for those ports for which the constructor has performed a **P()** operation. Thus if a **SERIAL_0/1** object is created before multitasking has started, then **channel** is mapped to a polled port in the constructor, and the destructor will not perform a **V()** operation on the semaphore later on.

```
 1  /* SerialOut.cc */
...
44  SerialOut::~SerialOut()
45  {
46     switch(channel)
47        {
48          case SERIAL_0:   Channel_0.V();   return;
49          case SERIAL_1:   Channel_1.V();   return;
50        }
51  }
```

The constructor and destructor for the **SerialIn** class are conceptionally identical to those of the **SerialOut** class, so that we do not repeat them here. The only difference is a simplification in the **SerialIn** constructor: it does not check whether multitasking is already running, because during system start-up, there is typically no serial input, while serial output for debugging purposes is quite

common. It would do no harm, however, to make the **SerialIn** constructor identical to that of **SerialOut**.

3.7.3 Public SerialOut Member Functions

The simplest public member function of the **SerialOut** class is **Putc(int character)**. The purpose of **Putc()** is to transmit its argument character on the channel. Since the way how this transmission has to be done is different for the channels (interrupt driven for **SERIAL_0/1**, polled for **SERIAL_0/1_POLLED**, or just discarding the character for **DUMMY_SERIAL**), **Putc()** simply decodes the channel and then calls the appropriate function that actually transmits the character.

```
  1  /* SerialOut.cc */
...
104  void SerialOut::Putc(int c)
105  {
106     switch(channel)
107        {
108          case SERIAL_0:         Putc_0(c);           return;
109          case SERIAL_1:         Putc_1(c);           return;
110          case SERIAL_0_POLLED:  Putc_0_polled(c);    return;
111          case SERIAL_1_POLLED:  Putc_1_polled(c);    return;
112          case DUMMY_SERIAL:                          return;
113          default:                                    return;
114        }
115  }
```

Thus **Putc()** provides a unified interface towards the different channels.

If a channel is interrupt driven (as for **SERIAL_0/1**), then the character is put into the corresponding output buffer. As we will see in Section 3.8, transmit interrupts need to be disabled if the output queue becomes empty. If this situation is indicated by the **TxEnabled_0/1** variable, then the interrupts must be turned on again by writing a certain command into the DUART.

```
  1  /* SerialOut.cc */
...
 53  void SerialOut::Putc_0(int c)
 54  {
 55  unsigned char cc = c;
 56
 57     outbuf_0.Put(cc);
 58     if (!TxEnabled_0)
 59        {
 60          TxEnabled_0 = 1;
 61          os::writeRegister(wDUART_CR_A, CR_TxENA);   // enable Tx
 62        }
 63  }
```

If a channel is polled, then the polled **Putc()** function makes sure that the initialization of the hardware has reached a sufficient level (**Polled_IO**, i.e. the DUART has been initialized, but interrupts are not yet enabled), and then it polls the DUART's status register until it is able to accept a new character.

```
  1  /* SerialOut.cc */
...
 77  void SerialOut::Putc_0_polled(int c)
 78  {
 79     if (os::initLevel() < os::Polled_IO)   os::init(os::Polled_IO);
 80
 81     while (!(os::readDuartRegister(rDUART_SR_A) & SR_TxRDY))   /**/ ;
 82
 83     os::writeRegister(wDUART_THR_A, c);
 84
 85     while (!(os::readDuartRegister(rDUART_SR_A) & SR_TxRDY))   /**/ ;
 86  }
```

In the case of the **DUMMY_SERIAL** channel, the corresponding **Putc()** function does not do anything.

```
  1  /* SerialOut.cc */
...
 99  void SerialOut::Putc_dummy(int)
100  {
101     // dummy Putc to compute length
102  }
```

Although **Putc_dummy()** is not called in **Putc()**, it will be required later on, where any of the above specific **Putc_()** functions will be passed as an argument to a print function discussed below.

Note that in the case of interrupt-driven serial output, the **Putc()** function may return long before the character has been transmitted by the DUART, since the **Putc()** only places the character into the output buffer. Sometimes we also want to know if the character has indeed been transmitted. For this purpose, the **IsEmpty()** function returns true if the output buffer of a channel is empty.

Based on the **Putc()** function, we can implement more sophisticated functions for formatted output similar to the **fprintf()** in C libraries. There are both a static **Print()** function taking a channel as an argument and a non-static **Print()** function.

```
  1  /* SerialOut.hh */
...
 12  class SerialOut
 13  {
...
 18     static int Print(Channel, const char *, ...);
...
 21     int Print(const char *, ...);
```

```
...
 44  };
```

The static **Print()** function creates a **SerialOut** object for the channel and then proceeds exactly like the non-static **Print()** function.

```
  1  /* SerialOut.cc */
...
132  int SerialOut::Print(Channel channel, const char * format, ...)
133  {
134  SerialOut so(channel);
...
```

The **SerialOut** object is automatic in the static **Print()** function so that it is automatically destructed when **Print()** returns. This way it is ensured that anything being printed is not interrupted by other tasks calling a **Print()** function for the same channel.

The non-static **Print()** function selects the proper **Putc_()** function for its channel and either calls this **Putc_()** function (for those characters of the format string that are to be copied to the output), or calls **print_form()** for format characters. The implementation of **print_form()** is straightforward, but somewhat lengthy, so that we skip it here and refer to Appendix A.12. Any of the **Print()** functions return the number of characters printed on the channel.

```
  1  /* SerialOut.cc */
...
159  int SerialOut::Print(const char * format, ...)
160  {
161  void (*putc)(int);
162  const unsigned char ** ap = (const unsigned char **)&format;
163  const unsigned char * f = *ap++;
164  int len = 0;
165  int cc;
166
167     switch(channel)
168        {
169          case SERIAL_0:         putc = Putc_0;           break;
170          case SERIAL_1:         putc = Putc_1;           break;
171          case SERIAL_0_POLLED:  putc = Putc_0_polled;    break;
172          case SERIAL_1_POLLED:  putc = Putc_1_polled;    break;
173          case DUMMY_SERIAL:     putc = Putc_dummy;       break;
174          default:               return 0;
175        }
176
177     while (cc = *f++)
178        if (cc != '%')    { putc(cc);   len++; }
179        else              len += print_form(putc, ap, f);
180
181     return len;
182  }
```

So, why are two different **Printf()** functions needed? The reason is that sometimes not all information to be printed together is easily available beforehand. Consider two tasks running the same code and using the same channel:

```
void task_main(Channel ch)
{
     for (;;)
          {
            Message msg;
            char * p = (char *)(msg.Body);
            Task::GetMessage(msg);
            for (unsigned int i = 0; msg.Body[i]; i++)
                SerialOut::Print(ch,"%c ",p[i]);
          }
}
```

In this example, each message character with its trailing blank from any task is printed as a whole, since the lifetime of the **SerialOut** objects created automatically by the static **Print()** function is basically the time it takes for the print function to execute. If one task receives “AAA” and the other tasks receives “BBB” as the body of a message at the same time, then the lines of both tasks may be intermixed, producing e.g. the following output:

A A B B B A

In contrast, the output

A AB B B A

would never be produced, since the trailing blank is always “bound” to its preceding character by the single invocation of the static **Print()** function. If we want to print a whole message, i.e. produce e.g. A A A B B B instead of A A B B B A, then we have to extend the lifetime of the **SerialOut** object. This is where the non-static **Print()** function is used, like in the following code:

```
void task_main(Channel ch)
{
     for (;;)
          {
            Message msg;
            char * p = (char *)(msg.Body);
            Task::GetMessage(msg);
```

```
            {
              SerialOut so(ch);
              for (unsigned int i = 0; msg.Body[i]; i++)
              so.Print(ch,"%c ",p[i]);
            }
        }
}
```

Now there is only one **SerialOut** object instead of one for each message character which causes an entire message to be printed. Thus the static **Print()** is typically used when the item to be printed can be expressed by a single format string, while the non-static **Print()** is used otherwise.

3.7.4 Public SerialIn Member Functions

The simplest public member function of the **SerialIn** class is **Getc()** which returns the next character received on a channel. If no characters are available, then the task calling **Getc()** is blocked until the next character is received. In contrast to the **SerialOut** class, **Getc()** returns useful results only for interrupt driven I/O and indicates EOF (-1) otherwise. **Getc()** returns **int** rather than **char** in order to distinguish the EOF condition from the regular **char** 0xFF (i.e. -1).

```
 1  /* SerialIn.cc */
...
34  int SerialIn::Getc()
35  {
36  unsigned char cc;
37
38     switch(channel)
39        {
40          case SERIAL_0:  inbuf_0.Get(cc);   return cc;
41          case SERIAL_1:  inbuf_1.Get(cc);   return cc;
42          default:        return -1;
43        }
44  }
```

If it is not desired to block the task, **Pollc()** can be used instead. **Pollc()** returns EOF when **Putc()** would block the task.

```
 1  /* SerialIn.cc */
...
46  int SerialIn::Pollc()
47  {
48  unsigned char cc;
49
50     switch(channel)
51        {
52       case SERIAL_0:  return inbuf_0.PolledGet(cc)  ? -1 : cc;
53       case SERIAL_1:  return inbuf_1.PolledGet(cc)  ? -1 : cc;
54       default:        return -1;
55        }
```

```
 56  }
```

Often one wants to receive characters up to, but not including a terminating character; e.g. if decimal numbers of unknown length are entered. UNIX has a **unputc()** function which undoes the last **putc()**. We have not adopted this scheme, but instead provide a function **Peekc()** which works like **Pollc()**, but does not remove the character from the receive queue. Both the **unputc()** approach and the **Peekc()** approach have their advantages and disadvantages, and one can easily implement **unputc()** in the SerialIn class.

```
  1  /* SerialIn.cc */
...
 58  int SerialIn::Peekc()
 59  {
 60  unsigned char cc;
 61
 62     switch(channel)
 63        {
 64          case SERIAL_0:   return inbuf_0.Peek(cc)  ? -1 : cc;
 65          case SERIAL_1:   return inbuf_1.Peek(cc)  ? -1 : cc;
 66          default:         return -1;
 67        }
 68  }
```

GetDec() and **GetHex()** are based on the **Pollc()** and **Peekc()** functions and collect decimal (’0’..’9’) or hexadecimal (’0’..’9’,’A’..’F’ and ’a’..’f’) sequences of characters, and return the resulting integer value. These functions do not necessarily belong to an operating system, but are provided since they are commonly required.

For serial output, characters can never get lost, since tasks performing output would block before the transmit buffer overflows. For serial input however, the receive buffer may overflow, e.g. if no task is performing **Getc()** for some time. The function **getOverflowCounter()** returns the number of characters lost due to buffer overflow, and 0 for polled or dummy serial input where this condition can not be easily detected.

3.8 Interrupt Processing

As shown in Section 3.2.4, the only device generating interrupts is the DUART using interrupt level 2, which corresponds to autovector #2 in the CPU's vector table. After reset, interrupts from the DUART are disabled in the DUART, and in addition, the CPU's interrupt mask is set to level 7, thus preventing interrupts from the DUART. Before discussing the interrupt processing, we shall have a look at the hardware initialization.

3.8.1 Hardware Initialization

Hardware initialization is performed in two steps, which are controlled by the variable **os::init_level** and by the function **os::init()** which performs initialization up to a requested level.

```
  1   /* os.hh */
...
 18   class os
 19   {
...
 30      enum INIT_LEVEL {
 31                        Not_Initialized = 0,
 32                        Polled_IO       = 1,
 33                        Interrupt_IO    = 2
 34                      };
 35
 36      static void init(INIT_LEVEL new_level);
...
 49      static INIT_LEVEL init_level;
...
 88   };
```

After RESET, the **init_level** is **Not_initialized**. The **Polled_IO** level refers to a hardware state, where the DUART is initialized, but interrupts are masked. The final level is **Interrupt_IO**, where interrupts are also enabled. If an initialization to **Interrupt_IO** is requested, then the initialization for level **Polled_IO** is automatically performed by the **os:init()** function. During normal system start-up, the **Polled_IO** level is never requested; instead, the initialization jumps directly from **Not_initialized** to **Interrupt_IO**. This happens at a rather late stage in the start-up of the system. If debugging printouts are inserted during system start-up, then the **Putc_0**/**1_polled**() functions request initialization to level **Polled_IO**.

```
128   void os::init(INIT_LEVEL iLevel)
129   {
130   enum { green = 1<<7 };   // green LED, write to BCLR turns LED on
131
132      if (init_level < Polled_IO)
133         {
134           initDuart(DUART, CSR_9600, CSR_9600);
135           init_level = Polled_IO;
136         }
137
```

```
138      if (iLevel == Interrupt_IO && init_level < Interrupt_IO)
139         {
140           readDuartRegister (rDUART_STOP);                 // stop timer
141           writeRegister(xDUART_CTUR, CTUR_DEFAULT);   // set CTUR
142           writeRegister(xDUART_CTLR, CTLR_DEFAULT);   // set CTLR
143           readDuartRegister(rDUART_START);                 // start timer
144
145           writeRegister(wDUART_IMR, INT_DEFAULT);
146           init_level = Interrupt_IO;
147         }
148   }
```

Initialization to level **Polled_IO** basically sets the baud rate and data format for both DUART channels to 9600 Baud, 8 data bits, two stop bits, and enables the receivers and transmitters of both serial channels. Thus after reaching this initialization level, the DUART can be operated in a polled mode.

Initialization to level **Interrupt_IO** programs the DUART timer to generate interrupts every 10ms. This is the rate at which task scheduling is performed. Then interrupts from all internal interrupt sources of the DUART that are used are enabled: the timer interrupt as well as receive and transmit interrupts for all channels. These interrupts are never turned off afterwards. If a transmit buffer gets empty, then the corresponding transmit interrupt is disabled by disabling the transmitter rather than masking its interrupt (otherwise, one would need to maintain a copy of the interrupt mask register, which would be less elegant).

At this point, the interrupts are enabled in the DUART, but the CPU's interrupt mask is still at level 7, so that interrupts have no effect yet.

```
 1   // Task.cc
78   void main()
79   {
80      if (Task::SchedulerStarted)   return -1;
81
82      for (int i = 0; i < TASKID_COUNT; i++)   Task::TaskIDs[i] = 0;
83      setupApplicationTasks();
84
85      for (Task * t = Task::currTask->next; t != Task::currTask; t = t->next)
86          t->TaskStatus &= ~Task::STARTED;
87
88      Task::SchedulerStarted = 1;
89      os::init(os::Interrupt_IO);   // switch on interrupt system
90      os::set_INT_MASK(os::ALL_INTS);
91
92      Task::Dsched();
93
94      for (;;)   os::Stop();
95
96      return 0;   /* not reached */
97   }
```

The initialization to level **Interrupt_IO** is done in function **main()**. This function first sets up all tasks that are supposed to run after systems start-up, initializes the hardware to level **Interrupt_IO,** and finally lowers the CPU's interrupt mask so

that all interrupts are accepted. The **main()** function is actually executed by the idle task, which deschedules itself and then enters an infinite loop. Since the idle task has the lowest priority of all tasks, it only executes if all other tasks are blocked. It thus stops the CPU until the next interrupt occurs.

3.8.2 Interrupt Service Routine

As we already saw, the only interrupt that could occur in our system is an autolevel 2 interrupt. Of course, the system can be easily extended to support more peripherals. Thus if an interrupt occurs, the CPU fetches the corresponding interrupt vector and proceeds at the address **_duart_isr**, where the interrupt service routine for the DUART starts. The CPU is in supervisor mode at this point.

```
  1   | crt0.S
...
 52           .LONG   _duart_isr              | 26     level 2 autovector
...
```

The CPU first turns on a LED. This LED is turned off each time the CPU is stopped. The brightness of the LED thus shows the actual CPU load, which is very useful sometimes. The CPU then saves its registers onto the system stack and reads the interrupt status from the DUART which indicates the source(s) of the interrupt.

```
...
133   _duart_isr:                           |
134           MOVE.B  #LED_YELLOW, wLED_ON    | yellow LED on
135           MOVEM.L D0-D7/A0-A6, -(SP)      | save all registers
136           MOVEM.L rDUART_ISR, D7          | get interrupt sources
137           SWAP    D7                      |
138           MOVE.B  D7, _duart_isreg        |
139                                           |
...
```

If the interrupt is caused by the receiver for **SERIAL_0**, then the received character is read from the DUART and put into the receive queue of **SERIAL_0**. This queue has a get semaphore, so that as a consequence, a task blocked on the receive queue may be unblocked. Reading the received character from the DUART automatically clears this interrupt.

```
...
140           BTST    #1, _duart_isreg        | RxRDY_A ?
141           BEQ     LnoRxA                  | no
142           MOVEM.L rDUART_RHR_A, D0        | get char received
143           MOVE.L  D0, -(SP)               |
144           PEA     1(SP)                   | address of char received
145           PEA     __8SerialIn$inbuf_0     | inbuf_0 object
146           JSR     _PolledPut__t10Queue_Gsem1ZUcRCUc
147           LEA     12(SP), SP              | cleanup stack
148   LnoRxA:                                 |
149                                           |
```

```
...
```

The same applies for an interrupt from the receiver for **SERIAL_1**.

```
...
150           BTST    #5, _duart_isreg        | RxRDY_B ?
151           BEQ     LnoRxB                  | no
152           MOVEM.L rDUART_RHR_B, D0        | get char received
153           MOVE.L  D0, -(SP)               |
154           PEA     1(SP)                   | address of char received
155           PEA     __8SerialIn$inbuf_1     | inbuf_1 object
156           JSR     _PolledPut__t10Queue_Gsem1ZUcRCUc
157           LEA     12(SP), SP              | cleanup stack
158   LnoRxB:                                 |
159                                           |
...
```

If the interrupt is caused by the transmitter for **SERIAL_0**, then the next character from the transmit queue for **SERIAL_0** is fetched. The transmit queue may be empty, however; in this case, the transmitter is disabled to clear the interrupt. This is also indicated towards the **Putc_0()** function by the **SerialOut::TxEnabled_0** variable (see also Section 3.7.3). If the queue is not empty, then the next character is written to the DUART which clears this interrupt.

```
...
160           BTST    #0, _duart_isreg        | TxRDY_A ?
161           BEQ     LnoTxA                  | no
162           LEA     -2(SP), SP              | space for next char
163           PEA     1(SP)                   | address of char received
164           PEA     __9SerialOut$outbuf_0   | outbuf_0 object
165           JSR     _PolledGet__t10Queue_Psem1ZUcRUc
166           LEA     8(SP), SP               | cleanup stack
167           MOVE.W  (SP)+, D1               | next output char (valid if D0 = 0)
168           TST.L   D0                      | char valid ?
169           BEQ     Ld1i11                  | yes
170           CLR.L   __9SerialOut$TxEnabled_0| no, disable Tx
171           MOVE.B  #0x08, wDUART_CR_A      | disable transmitter
172           BRA     LnoTxA                  |
173   Ld1i11: MOVE.B  D1, wDUART_THR_A        | write char (clears int)
174   LnoTxA:                                 |
175                                           |
...
```

The same is true for an interrupt from the transmitter for **SERIAL_1**.

```
...
176           BTST    #4, _duart_isreg        | TxRDY_B ?
177           BEQ     LnoTxB                  | no
178           LEA     -2(SP), SP              | space for next char
179           PEA     1(SP)                   | address of char received
180           PEA     __9SerialOut$outbuf_1   | outbuf_1 object
181           JSR     _PolledGet__t10Queue_Psem1ZUcRUc
182           LEA     8(SP), SP               | cleanup stack
183           MOVE.W  (SP)+, D1               | next output char (valid if D0 = 0)
184           TST.L   D0                      | char valid ?
185           BEQ     Ld1i21                  | yes
186           CLR.L   __9SerialOut$TxEnabled_1| no, disable Tx
```

```
187           MOVE.B  #0x08, wDUART_CR_B      | disable transmitter
188           BRA     LnoTxB                  |
189   Ld1i21: MOVE.B  D1, wDUART_THR_B        | write char (clears int)
190   LnoTxB:                                 |
191                                           |
...
```

The last option is a timer interrupt. In this case, the interrupt is cleared by writing to the DUART's stop/start registers. Next, a pair of variables indicating the system time since power on in milliseconds is updated. This implements a simple system clock:

```
...
192           BTST    #3, _duart_isreg        | timer ?
193           BEQ     LnoTim                  | no
194           MOVEM.L rDUART_STOP, D1         | stop timer
195           MOVEM.L rDUART_START, D1        | start timer
196                                           |
197                                           | increment system time
198           ADD.L   #10, _sysTimeLo         | 10 milliseconds
199           BCC.S   Lsys_time_ok            |
200           ADDQ.L  #1, _sysTimeHi          |
201   Lsys_time_ok:                           |
202                                           |
...
```

A common problem is to poll a peripheral (e.g. a switch) in regular intervals or to wait for certain period of time. Neither blocking a task or busy wait is appropriate for this purpose. Instead, we implement a function **Task::Sleep()** which will be explained later on. This **Sleep()** function uses a variable **TaskSleepCount** for each task which is decremented with every timer interrupt. If the variable reaches 0, the task return to state **RUN** by clearing a particular bit in the task's status register.

```
...
203           MOVE.L  __4Task$currTask, D1    |
204           MOVE.L  D1, A0                  |
205   L_SLEEP_LP:                             | decrement sleep counters...
206           SUBQ.L  #1, TaskSleepCount(A0)  |
207           BNE     L_NO_WAKEUP             |
208           BCLR    #3, TaskStatus(A0)      | clear sleep state
209   L_NO_WAKEUP:                            |
210           MOVE.L  TaskNext(A0), A0        |
211           CMP.L   A0, D1                  |
212           BNE     L_SLEEP_LP              |
213           ST      _consider_ts            | request task switch anyway
214   LnoTim:                                 |
215                                           |
...
```

Now all interrupt sources causing the present interrupt are cleared. During this process, new interrupts may have occurred. In that case, the interrupt service routine will be entered again when returning from exception processing. The interrupt processing is finished by restoring the interrupts saved at the beginning.

The variable **_consider_ts** may or may not have been set during the interrupt service routine. The final step is to proceed at label **_return_from_exception**.

```
...
216           MOVEM.L (SP)+, D0-D7/A0-A6      | restore all registers
217           BRA     _return_from_exception  |
```

The processing at label **_return_from_exception** has already been described in Section 3.3, i.e. it will be checked whether a task switch is required. Note that for the code starting at **_return_from_exception** it makes no difference whether a task switch was caused by an interrupt or not.

3.9 Memory Management

As we will see in Section 6.4, a library **libgcc2** has to be provided in order to link the kernel. This library contains in particular the code for the global C++ operators **new** and **delete**. The code in **libgcc2** basically calls two functions, **malloc()** (for operator **new**) and **free()** (for operator **delete**).

One way to provide these functions is to compile the GNU malloc package and to link it to the kernel. But this method consumes considerable memory space. It should also be noted that the malloc package contains uninitialized variables and would thus result in a non-empty BSS section. Since we do not use the BSS section, the source code of the malloc package needs to be modified by initializing all uninitialized variables to 0.

As you may have noticed, we never used the **new** operator in the kernel code, except for creating new tasks and their associated stacks. The main reason for not using this operator is that in an embedded system, there is most likely no way to deal with the situation where **new** (i.e. **malloc()**) fails due to lack of memory. The malloc package allocates memory in pages (e.g. 4kByte; the page size can be adjusted) and groups memory requests of similar size (i.e. rounded up to the next power of 2) in the same page. Thus if there are requests for different sizes, a significant number of pages could be allocated. For conventional computers with several megabytes of memory this is a good strategy, since the waste of memory in partly used pages is comparatively small. For embedded systems, however, the total amount of memory is typically much smaller, so that the standard **malloc()** is not the right choice.

We actually used the standard **malloc()** in the early kernel versions, but replaced it later on by the following.

```
  1   /* os.cc */
...
 17   extern int edata;
 18   char * os::free_RAM = (char *)&edata;
```

The label **edata** is computed by the linker and indicates the end of the .DATA section; i.e. past the last initialized variable. The char pointer **free_RAM** is thus initialized and points to the first unused RAM location.

```
 21   extern "C" void * sbrk(unsigned long size)
 22   {
 23   void * ret = os::free_RAM;
 24
 25      os::free_RAM += size;
 26
 27      if (os::free_RAM > (char *)RAMend)   // out of memory
 28         {
 29           os::free_RAM -= size;
 30           ret = (void *) -1;
```

```
31         }
32
33      return ret;
34   }
```

The function **sbrk(unsigned long size)** increases the **free_RAM** pointer by **size** and returns its previous value. That is, a memory block of size **size** is allocated and returned by **sbrk()**.

```
36   extern "C" void * malloc(unsigned long size)
37   {
38   void * ret = sbrk((size+3) & 0xFFFFFFFC);
39
40      if (ret == (void *)-1)   return 0;
41      return ret;
42   }
```

Our **malloc()** implementation rounds the memory request size up to a multiple of four bytes so that the memory is aligned to a long word boundary.

```
45   extern "C" void free(void *)
46   {
47   }
```

Finally, our **free()** function *does not* free the memory returned. As a consequence, **delete** must not be used. As long as tasks are not created dynamically and **new** is not used elsewhere, this scheme is most efficient and adequate. Otherwise, one should use the standard malloc package or write an own version meeting specific requirements. A better solution than the global **new** operator is to overload the **new** operator for specific classes. For example, memory for certain classes could be allocated statically and the class specific new operator (which defaults to the global **new** operator) could be overloaded. This gives more control over the memory allocation.

Finally, it should be noted that embedded systems with hardware memory management need a memory management scheme that is written specifically for the memory management unit used.

3.10 Miscellaneous Functions

So far, we have discussed most of the code comprising the kernel. What is missing is the code for starting up tasks (which is described in Section 4.3) and some functions that are conceptually of minor importance but nevertheless of certain practical use. They are described in less detail in the following sections.

3.10.1 Miscellaneous Functions in Task.cc

The **Monitor** class uses member functions that are not used otherwise. **Current()** returns a pointer to the current task. **Dsched()** explicitly deschedules the current task. **MyName()** returns a string for the current task that is provided as an argument when a task is started; **Name()** returns that string for any task. **MyPriority()** returns the priority of the current task, **Priority()** returns the priority for any task. **userStackBase()** returns the base address of the user stack; **userStackSize()** returns the size of the user stack**;** and **userStackUsed()** returns the size of the user stack that has already been used by a task. When a task is created, its user stack is initialized to contain characters 'U'. **userStackUsed()** scans the user stack from the bottom until it finds a character which differs from 'U' and so computes the size of the used part of the stack. **Status()** returns the task status bitmap.

Next() returns the next task in the ring of all existing tasks. If we need to perform a certain function for all tasks, we could do it as follows:

```
for (const Task * t = Task::Current();;)
     {
        ...
        t = t->Next();
        if (t == Task::Current())   break;
     }
```

Sleep(unsigned int ticks) puts the current task into sleep mode for **ticks** timer interrupts. That is, the task does not execute for a time of **ticks***10ms without wasting CPU time.

When a task is created, its state is set to **STARTED**; i.e. the task is not in state **RUN**. This allows for setting up tasks before multitasking is actually enabled. **Start()** resets the task state to **RUN**.

Terminate() sets a task's state to **TERMINATED**. This way, the task is prevented from execution without the task being deleted.

GetMessage(Message & dest) copies the next message sent to the current task into **dest** and removes it from the task's message queue (**msgQ**).

3.10.2 Miscellaneous Functions in os.cc

getSystemTime() returns the time in millisecond since system start-up (more precisely since multitasking was enabled) as a **long long**. **initChannel()** initializes the data format (data bits, stop bits) of a DUART channel, **setBaudRate()** sets ??? What ???. **Panic()** disables all interrupts, turns on the red LED and then continuously dumps an exception stack frame on **SERIAL_0**. This function is used whenever an exception for which no handler exists is taken (label **_fatal**). That is, if a fatal system error occurs, the red LED is turned on, and we can connect a terminal to **SERIAL_0**. The exception stack frame can then be analyzed, together with the map file created by the linker, to locate the fault in the source code. **readDuartRegister()** is called to read a DUART register. **writeRegister()** is used to write into a hardware (i.e. DUART) register.

4 Bootstrap

4.1 Introduction

In this chapter, the start-up of the kernel is described. It contains two phases: the initialization of the system after RESET, and the initialization of the tasks defined in the application.

4.2 System Start-up

The compilation of the various source files and the linking of the resulting object files results in two files containing the .TEXT and ..DATA sections of the final system (see also Section 2.1.1). The linker has generated addresses referring to the .DATA section, which normally starts at the bottom of the system's RAM. After RESET, however, this RAM is not initialized. Thus the .DATA section must be contained in the system's ROM and copied to the RAM during system start-up, ??? as shown in Figure 4.1 ???

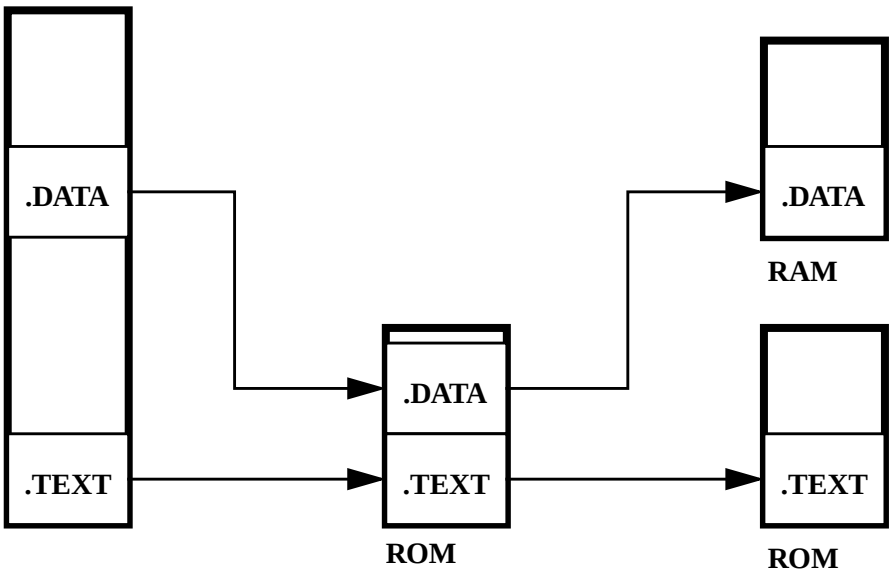

FIGURE 4.1 ??? .DATA and .TEXT during System Start-Up ???

The .TEXT section, in contrast, does not need any special handling. Figure 4.1 shows the output of the linker on the left. The ROM image for the system is created by appending the .DATA section after the .TEXT section. The address of the .DATA section in ROM can be computed from the end of the .TEXT section; this address is provided by the linker (symbol **_etext**). Depending on the target system for which the linker has been installed, **_etext** may need to be rounded up (e.g. to the next 2Kbyte boundary) to determine the exact address of the .DATA section in RAM. Although it is not strictly necessary, it is generally a good idea to initialize the unused part of the RAM to 0. This allows to reproduce faults created by uninitialized variables.

After RESET, the CPU loads its supervisor stack pointer with the vector at address 0 and its program counter with the next vector. In our implementation, the vector for the supervisor stack pointer is somewhat abused, as it contains a branch to the start of the system initialization. This allows for issuing a JMP 0 (in supervisor mode) to restart the system, although this feature is not used yet. These two vectors are followed by the other exception vectors. Most of them are set to label **_fatal**, which is the handler for all fatal system errors.

```
 1   | crt0.S
37   _null:  BRA     _reset                  |  0     initial SSP (end of RAM)
38           .LONG   _reset                  |  1     initial PC
39           .LONG   _fatal, _fatal          |  2, 3  bus error, adress error
40           .LONG   _fatal, _fatal          |  4, 5  illegal instruction, divide/0
41           .LONG   _fatal, _fatal          |  6, 7  CHK, TRAPV instructions
42           .LONG   _fatal, _fatal          |  8, 9  privilege violation, trace
43           .LONG   _fatal, _fatal          | 10,11  Line A,F Emulators
44                                           |
45           .LONG   _fatal,_fatal,_fatal    | 12...  (reserved)
46           .LONG   _fatal,_fatal,_fatal    | 15...  (reserved)
47           .LONG   _fatal,_fatal,_fatal    | 18...  (reserved)
48           .LONG   _fatal,_fatal,_fatal    | 21...  (reserved)
49                                           |
50           .LONG   _fatal                  | 24     spurious interrupt
51           .LONG   _fatal                  | 25     level 1 autovector
52           .LONG   _duart_isr              | 26     level 2 autovector
53           .LONG   _fatal                  | 27     level 3 autovector
54           .LONG   _fatal, _fatal          | 28,29  level 4,5 autovector
55           .LONG   _fatal, _fatal          | 30,31 level 6,7 autovector
56                                           |
57           .LONG   _stop                   | 32     TRAP #0 vector
58           .LONG   _deschedule             | 33     TRAP #1 vector
59           .LONG   _fatal                  | 34     TRAP #2 vector
60           .LONG   _Semaphore_P            | 35     TRAP #3 vector
61           .LONG   _Semaphore_V            | 36     TRAP #4 vector
62           .LONG   _Semaphore_Poll         | 37     TRAP #5 vector
63           .LONG   _fatal, _fatal          | 38,39  TRAP #6, #7 vector
64           .LONG   _fatal, _fatal          | 40,41  TRAP #8, #9 vector
65           .LONG   _fatal, _fatal          | 42,43  TRAP #10,#11 vector
66           .LONG   _fatal                  | 44     TRAP #12 vector
67           .LONG   _set_interrupt_mask     | 45     TRAP #13 vector
68           .LONG   _readByteRegister_HL    | 46     TRAP #14 vector
69           .LONG   _writeByteRegister      | 47     TRAP #15 vector
...
```

Thus after RESET, processing continues at label **_reset**. The supervisor stack pointer is initialized to point to the top of the RAM. This is necessary because the vector for this purpose was abused for the branch to **_reset**. Next the vector base register (VBR) is set to the beginning of the vector table. This applies only for MC68020 chips and above and allows for relocation of the vector table. Actually, the branch to **_reset** is intended for jumping to the content of the VBR so that the system can be restarted with a relocated .TEXT section, provided that the VBR points to the proper vector table. For processors such as the MC68000 that do not provide a VBR, this instruction must be removed. After setting the VBR, the LEDs are turned off.

```
 81   _reset:                                |
 82            MOVE.L  #RAMend, SP           | since we abuse vector 0 for BRA.W
 83            LEA     _null, A0             |
 84            MOVEC   A0, VBR               | MC68020++ only
 85                                          | enable cache
 86            MOVE.B  #0, wDUART_OPCR       | all outputs via BSET/BCLR
 87            MOVE.B  #LED_ALL, wLED_OFF    | all LEDs off
```

Then the RAM is initialized to 0. The end of the .TEXT section is rounded up to the next 2Kbyte boundary (assuming the linker was configured to round up the .TEXT section to a 2Kbyte boundary), which yields the start of the .DATA section in ROM. The size of the .DATA section is computed, and the .DATA section is then copied from ROM to the RAM.

```
 89            MOVE.L  #RAMbase, A1          | clear RAM...
 90            MOVE.L  #RAMend, A2           |
 91   L_CLR:   CLR.L   (A1)+                 |
 92            CMP.L   A1, A2                |
 93            BHI     L_CLR                 |
 94                                          | relocate data section...
 95            MOVE.L  #_etext, D0           | end of text section
 96            ADD.L   #0x00001FFF, D0       | align to next 2K boundary
 97            AND.L   #0xFFFFE000, D0       |
 98            MOVE.L  D0, A0                | source (.data section in ROM)
 99            MOVE.L  #_sdata, A1           | destination (.data section in RAM)
100            MOVE.L  #_edata, A2           | end of .data section in RAM
101   L_COPY:  MOVE.L  (A0)+, (A1)+          | copy data section from ROM to RAM
102            CMP.L   A1, A2                |
103            BHI     L_COPY                |
```

At this point, the .TEXT and .DATA sections are located at those addresses to which they had been linked. The supervisor stack pointer is set to the final supervisor stack, and the user stack pointer is set to the top of the idle task's user stack (the code executed here will end up as the idle task).

```
105            MOVE.L  #_SS_top, A7          | set up supervisor stack
106            MOVE.L  #_IUS_top, A0         |
107            MOVE    A0, USP               | set up user stack
```

Finally (with respect to **crt0.S**), the CPU enters user mode and calls function **_main()**. It is not intended to return from this call; if this would happen, then it would be a fatal system error.

```
108                                         |
109           MOVE    #0x0700, SR           | user mode, no ints
110           JSR     _main                 |
111                                         |
112   _fatal:                               |
```

If for any reason label **_fatal** is reached, then all interrupts are disabled, the red LED is turned on, and the **SERIAL_1** transmitter is enabled to allow for polled serial output. Then the present supervisor stack pointer, which points to the exception stack frame created for the fatal system error, is saved and the supervisor stack pointer is set to the end of the RAM. Then **os::Panic()** is called forever with the saved exception stack frame as its argument. **os::Panic()** prints the stack frame in a readable format on the **SERIAL_1** channel, so that the cause of the fault can easily be determined. It ??? what is it ??? is called forever, so that a terminal can be connected to **SERIAL_1** even after a fatal system error and the stack frame is not lost, but repeated forever.

```
112   _fatal:                               |
113           MOVE.W  #0x2700, SR           |
114           MOVE.B  #LED_RED, wLED_ON     | red LED on
115           MOVE.B  #0x04, wDUART_CR_B    | enable transmitter
116           MOVE.L  SP, A0                | old stack pointer
117           MOVE.L  #RAMend, SP           |
118   _forever:                             |
119           MOVE.L  A0, -(SP)             | save old stack pointer
120           MOVE.L  A0, -(SP)             | push argument
121           JSR     _Panic__2osPs         | print stack frame
122           LEA     2(SP), SP             | remove argument
123           MOVE.L  (SP)+, A0             | restore old stack pointer
124           BRA     _forever              |
125                                         |
126   _on_exit:                             |
127           RTS                           |
```

In general, a function name in assembler refers to a C function, whose name is the same except for the leading underscore. This would mean that “JSR **_main**” would call **main()**, which is defined in **Task.cc**. For the GNU C++ compiler/linker, the **main()** function is handled in a special way. In this case, a function **__main()** is automatically created and called just before **main()**. This **__main()** function basically calls the constructors for all statically defined objects so that these are initialized properly. The way this is done may change in future, so special attention should be paid to the compiler/linker release used. The **__main** function also calls **on_exit()** (i.e. label **_on_exit** above), which just returns. So the call of **main()** in **crt0.S** basically initializes the static objects and proceeds in the real **main()**.

Now the CPU is in user mode, but interrupts are still disabled. First, the variable **SchedulerStarted** is checked to ensure **main()** is not called by mistake; in our case **SchedulerStarted** is 0.

```
  1 // Task.cc
...
```

```
78    void main()
 79   {
 80      if (Task::SchedulerStarted)   return -1;
```

Then a vector containing all tasks known at system start-up is initialized to 0 and **setupApplicationTasks()** is called. In **setupApplicationTasks()**, all tasks required by the application are created (see also Section 4.3). All tasks created have their status set to STARTED. That is, the task ring is completely set up, but no task is in state RUN. Next, the status for each task is set from STARTED to RUN.

```
 82      for (int i = 0; i < TASKID_COUNT; i++)   Task::TaskIDs[i] = 0;
 83      setupApplicationTasks();
 84
 85      for (Task * t = Task::currTask->next; t != Task::currTask; t = t->next)
 86          t->TaskStatus &= ~Task::STARTED;
```

Here all tasks are in state RUN, but interrupts are still disabled. In the next step, variable **SchedulerStarted** is set to prevent subsequent calls to **main()** (which would have disastrous effects). Then the hardware is initialized to level **Interrupt_IO**, and finally interrupts are enabled. The idle task then de-schedules itself, which causes the task with the highest priority to execute. The idle task itself goes into an infinite loop. Whenever the idle task is swapped in (i.e. no other task is in state RUN), it calls **os::Stop()**.

```
 88      Task::SchedulerStarted = 1;
 89      os::init(os::Interrupt_IO);   // switch on interrupt system
 90      os::set_INT_MASK(os::ALL_INTS);
 91
 92      Task::Dsched();
 93
 94      for (;;)   os::Stop();
 95
 96      return 0;   /* not reached */
 97   }
```

Function **os::Stop()** merely executes TRAP #0.

```
  1   /* os.cc */
...
 67   void os::Stop()
 68   {
 69      asm("TRAP #0");
 70   }
```

The CPU thus enters supervisor mode, fetches the corresponding vector and proceeds at label **_stop**.

```
  1   | crt0.S
...
 57           .LONG   _stop                   | 32     TRAP #0 vector
```

At label **_stop**, the yellow LED (which is turned on at every interrupt) is turned off. The CPU then stops execution with all interrupts enabled until an interrupt

occurs. That is, the yellow LED is turned on whenever the CPU is not in stopped mode, thus indicating the CPU load. After an interrupt occurred, the CPU proceeds at label **_return_from_exception**, where it checks if a task switch is required. Note that the interrupt itself cannot cause a task switch directly, since the interrupt occurs while the CPU is in supervisor mode.

```
223   _stop:                                    |
224           MOVE.B  #LED_YELLOW, wLED_OFF     | yellow LED off
225           STOP    #0x2000                   |
226           BRA     _return_from_exception    | check for task switch
227                                             |
```

After having left supervisor mode, the idle task is again in its endless loop and stops the CPU again, provided that no other task with higher priority is in state RUN.

4.3 Task Start-up

As already mentioned in Section 4.2, a task is started in two steps. First, a task control block (i.e. an instance of class **Task**) is created and inserted into the task ring. At this point, the task status is set to **STARTED** (i.e. not **RUN**) so that the task exists, but may not yet execute. In the second step, the task status is set to **RUN**. The main reason for this two-step approach is that tasks often set up in groups that cooperate by sending messages to each other. Suppose, for instance, that a task *T0* sets up two other tasks *T1* and *T2*. Suppose further that both tasks *T1* and *T2* send messages to each other directly after being created. It then might happen that task *T1*, provided its priority is higher than the priority of *T0*, executes before task *T2* is created by task *T0*. Sending a message from *T0* to *T1* would then fail. In our two-step approach, however, *T2* would exist already, but would not yet execute. Thus the message from *T1* to *T2* would be delivered correctly.

4.3.1 Task Parameters

The creation of a task is controlled by a number of parameters. A task is created by creating an instance of class **Task**:

```
     // Task.hh
...
 25   class Task
 26   {
...
 49      Task( void              (* main)(),
 50            unsigned long        userStackSize,
 51            unsigned short       queueSize,
 52            unsigned short       priority,
 53            const char *         taskName
 54          );
...
139   };
```

The parameters are the function to be executed by the task, the size of the stack for the task, the size of the task's message queue, the priority at which the task shall run, and a character string specifying the name of the task. The task name is useful for debug messages generated by the task and can be retrieved by the function **Task::MyName()** which returns this string:

```
SerialOut::Print(SERIAL_0, "\nTask %s started", Task::MyName());
```

So far, tasks have only been referred to by **Task** pointers, since the name is only used for printing purposes. But sometimes it is convenient to refer to tasks by an integer task ID rather than by task pointers. Assume we want to send a message to all tasks. One way of doing this is the following:

```
for (const Task * t = Current(); ; t = t->Next())
```

```
        {
          Message msg("Hello");
          t->SendMessage(msg);
          if (t->Next() == Current()   break;
        }
```

Unfortunately, this approach has some drawbacks. First, the order in which this loop is performed is different when executed by different tasks. Second, it is assumed that all tasks are present in the task chain. Although this is the case in our implementation, one may consider to remove tasks that are not in state **RUN** temporarily from the task chain in order to speed up task switching. In this case, only tasks in state **RUN** would receive the message which is probably not what was desired. A better approach is to maintain a table of task pointers, which is indexed by an integer task ID. The task IDs could be defined as follows:

```
1  // TaskId.hh
2
3  enum { TASKID_IDLE = 0,
4         TASKID_MONITOR,
5         TASKID_COUNT                  // number of Task IDs
6       };
```

More task IDs can be added before the **TASK_ID_COUNT**, so that **TASK_ID_COUNT** always reflects the proper number of tasks handled this way. Task IDs and task pointers are mapped by a table:

```
1     // Task.cc
...
  13  Task *              Task::TaskIDs[TASKID_COUNT];
```

As a matter of convenience, the task pointers can now be defined as macros:

```
  1   // TaskId.hh
...
  8   #define IdleTask            (Task::TaskIDs[TASKID_IDLE])
  9   #define MonitorTask         (Task::TaskIDs[TASKID_MONITOR])
```

This is nearly equivalent to defining e.g **MonitorTask** directly as a task pointer:

```
      Task * MonitorTask;
```

The difference between using a table and direct declaration of **Task** pointers is basically that for a table, all pointers are collected while for the direct declaration, they are spread over different object files. For a reasonably smart compiler, the macros can be resolved at compile time so that no overhead in execution time or memory space is caused by the table. Instead, the code of our example is even simplified:

```
      for (int t_ID = 0; t_ID < TASKID_COUNT; t_ID++)
            {
              Message msg("Hello");
              TaskIDs[t_ID]->SendMessage(msg);
            }
```

The **TaskIDs** table is initialized to zero in the idle task's **main()** function.

4.3.2 Task Creation

As a matter of style, for each task a function that starts up the task should be provided. This way, the actual parameters for the task are hidden at the application start-up level, thus supporting modularity. The function **setupApplicationTasks()**, which is called by the idle task in its **main()** function, sets the serial channels to their desired values (**SERIAL_1** in this case) and then calls the start-up function(s) for the desired tasks. In this example, there is only one application task; its start-up function is defined in class **Monitor** (see also Chapter 5).

```
 1  // ApplicationStart.cc
...
22  void setupApplicationTasks()
23  {
24     MonitorIn  = SERIAL_1;
25     MonitorOut = SERIAL_1;
26     ErrorOut   = SERIAL_1;
27     GeneralOut = SERIAL_1;
28
29     Monitor::setupMonitorTask();
30  }
```

The function **setupMonitorTask()** creates a new instance of class **Task** with task function **monitor_main**, a user mode stack of 2048 bytes, a message queue of 16 messages, a priority of 240, and the name of the task set to "Monitor Task".

```
 1  // Monitor.cc
...
13  void Monitor::setupMonitorTask()
14  {
15     MonitorTask = new Task        (
16                      monitor_main,    // function
17                      2048,            // user stack size
18                      16,              // message queue size
19                      240,             // priority
20                      "Monitor Task");
21  }
```

The priority (240) should be higher than that of other tasks (which do not exist in the above example) so that the monitor executes even if another task does not block. This allows for identifying such tasks ??? What tasks ???. Creating a new instance of class **Task** (i.e **new Task(...)**) returns a **Task** pointer which is stored in the **TaskIDs** table, remembering that **MonitorTask** was actually a macro defined as **TaskIDs[TASKID_MONITOR]**. With the **Task::Task(...)** constructor, a new task which starts the execution of a function **monitor_main()** is created. The function **monitor_main()** itself is not of particular interest here. It

should be noted, however, that **monitor_main()** may return (although most task functions will not) and that this requires special attention. For task creation, we assume that a hypothetical function **magic()** exists. This function does not actually exist as code, but only for the purpose of explaining the task creation. Function **magic()** is defined as follows:

```
void magic()
{
      Task::Terminate_0( monitor_main() );
      /* not reached */
}
```

Note that **Terminate_0()** is actually defined to have no arguments, but since **magic()** is only hypothetically, this does no harm.

```
  1   // Task.cc
...
 99   void Task::Terminate_0()
100   {
101      Terminate(0);
102   }
...
104   void Task::Terminate(int ex)
105   {
106      {
107        SerialOut so(ErrorOut);
108        so.Print("\n%s Terminated", currTask->name);
109      }
110      currTask->ExitCode = ex;
111      currTask->TaskStatus |= TERMINATED;
112      Dsched();
113   }
```

magic() calls the task's main function, which is provided when the task is created (in this case **monitor_main()**), as well as **Terminate_0()** in case the main function returns. Normally tasks do not return from their main functions; but if they do, then this return is handled by the **Terminate_0()** function, which merely calls **Terminate(0)**. The functions **Terminate_0()** and **Terminate(int ex)** may also be called explicitly by a task in order to terminate a task; e.g. in the case of errors. If these functions are called explicitly, then a message is printed, an exit code is stored in the TCB, and the task's state is set to **TERMINATED**. This causes the task to refrain from execution forever. The TCB, however, is not deleted, and the exit code TCB may be analyzed later on in order to determine why the task died. Setting the task status to **TERMINATED** does not immediately affect the execution of the task; hence it is followed by a **Dsched()** call which causes the task to be swapped out.

Now task creation mainly means setting up the TCB and the user stack of the task. The user stack is created as if the task had been put in state **STARTED** after calling **Terminate_0()** in **magic**, but before the first instruction of the task's main function. First, several variables in the TCB are set up according to the parameters

supplied to the constructor. At this point, the TCB is not yet linked into the task chain.

```
  1   // Task.cc
...
 33   Task::Task(void (*main)(),
 34              unsigned long   usz,
 35              unsigned short  qsz,
 36              unsigned short  prio,
 37              const char *    taskName
 38             )
 39         : US_size(usz),
 40           priority(prio),
 41           name(taskName),
 42           TaskStatus(STARTED),
 43           nextWaiting(0),
 44           msgQ(qsz),
 45           ExitCode(0)
...
```

Then the user stack of the task is allocated and initialized to the character **userStackMagic** (’U’). This initialization allows to determine the stack size used by the task later on.

```
 46   {
 47   int i;
 48
 49      Stack = new char[US_size];   // allocate stack
 50
 51      for (i = 0; i < US_size;)   Stack[i++] = userStackMagic;
```

The task’s program counter is set to the first instruction of its main function. If the task is swapped in later on, the execution proceeds right at the beginning of the task’s main function. Also all other registers of the CPU in the TCB are initialized. This is not necessary, but improves reproducibility of faults, e.g. due to dangling pointers.

```
 53      Task_A0  = 0xAAAA5555; Task_A1 = 0xAAAA4444;
 54      Task_A2  = 0xAAAA3333; Task_A3 = 0xAAAA2222;
 55      Task_A4  = 0xAAAA1111; Task_A5 = 0xAAAA0000;
 56      Task_A6  = 0xAAAA6666;
 57      Task_D0  = 0xDDDD7777; Task_D1 = 0xDDDD6666;
 58      Task_D2  = 0xDDDD5555; Task_D3 = 0xDDDD4444;
 59      Task_D4  = 0xDDDD3333; Task_D5 = 0xDDDD2222;
 60      Task_D6  = 0xDDDD1111; Task_D7 = 0xDDDD0000;
 61      Task_PC  = main;
 62      Task_CCR = 0x0000;
```

The user stack pointer of the task is set to the top of the user stack. Then the address of **Terminate_0()** is pushed on the user stack. **Task::Terminate_0()** is called in case the task’s main function returns.

```
 64      Task_USP    = (unsigned long *)(Stack + US_size);
 65      *--Task_USP = (unsigned long)Terminate_0;
```

If **currTask** is not set yet (i.e. if this is the first task that is created), then a TCB for the idle task is created, and **currTask** is set to that TCB. For this purpose, a **Task** constructor without arguments is used. In view of this code, it seems more reasonable to create the idle task from the outset rather than when the first application task is created.

```
67      if (!currTask)
68         currTask = new Task();
```

Finally, the TCB is linked into the task chain directly after **currTask** (which may be the idle task, as in our example, or another task). This operation must not be interrupted, so interrupts are masked here.

```
70      {
71        os::INT_MASK old_INT_MASK = os::set_INT_MASK(os::NO_INTS);
72        next = currTask->next;
73        currTask->next = this;
74        os::set_INT_MASK(old_INT_MASK);
75      }
76   }
```

The TCB of the newly created task is in a state as if it were put in state STARTED just before executing the first instruction of its main function.

4.3.3 Task Activation

After creating a number of tasks, these tasks need to be activated. This is done by changing the tasks' state from **STARTED** to **RUN**.

```
  1 // Task.cc
...
 78   void main()
 79   {
...
 85      for (Task * t = Task::currTask->next; t != Task::currTask; t = t->next)
 86          t->TaskStatus &= ~Task::STARTED;
```

If an application task (rather than the idle task) creates new tasks, it should activate the tasks after creating them in a similar way.

4.3.4 Task Deletion

If a task terminates, its TCB still exists. Deleting TCBs largely depends on the actual application and requires great care. Since TCBs have been allocated with the **new** operator, they need to be deleted with the **delete** operator. Also, if the **TaskIDs** table is used for a task (which is probably not the case for dynamically created tasks), the **Task** pointer needs to be removed from the table as well. In addition, it must be assured that no other task maintains a pointer to the deleted

task. Finally, use of the **delete** operator requires use of the **malloc** package, in contrast to the simple allocation mechanism we used by default.

An alternative to deleting tasks (which is likely to be a risk due to memory management as discussed in Section 3.9) is to provide a pool of static tasks which put themselves in a queue when they are idle. A task requiring a dynamic task would get such a task out of the queue and send a message containing a function to be performed to it. ??? Hä ??? This leads to structures similar to those discussed for the serial router in Section 3.7. In principle, static TCB can be used instead of the **new** operator for TCBs. The reason why we used **new** rather than static TCBs has historical reasons. The first application for which our kernel was used had a DIP switch that selected one of several applications. The kernel was the same for all applications, and the actual application was selected in **setupApplicationTasks()** by starting different tasks depending on the DIP switch setting. Static TCB allocation would have wasted RAM for those tasks not used for a particular DIP switch setting, while allocation by **new** used only those TCBs actually required, thus saving a significant amount of RAM.

5 An Application

5.1 Introduction

In this chapter, we present a simple application: a monitor program that receives commands from a serial port, executes them, and prints the result on the same serial port. The commands are mainly concerned with retrieving information about the running system, such as the status of tasks, or the memory used. This monitor has shown to be quite useful in practice, so it is recommended to include it in any application. In order to use the monitor, a terminal or a computer running a terminal emulation, for example the kermit program, is connected to the serial port used by the monitor.

5.2 Using the Monitor

The monitor supports a collection of commands that are grouped in menus: the main menu, the info menu, the duart menu, the memory menu, and the task menu. Other menus can easily be added if required. The only purpose of the main menu is to enter one of the other menus.

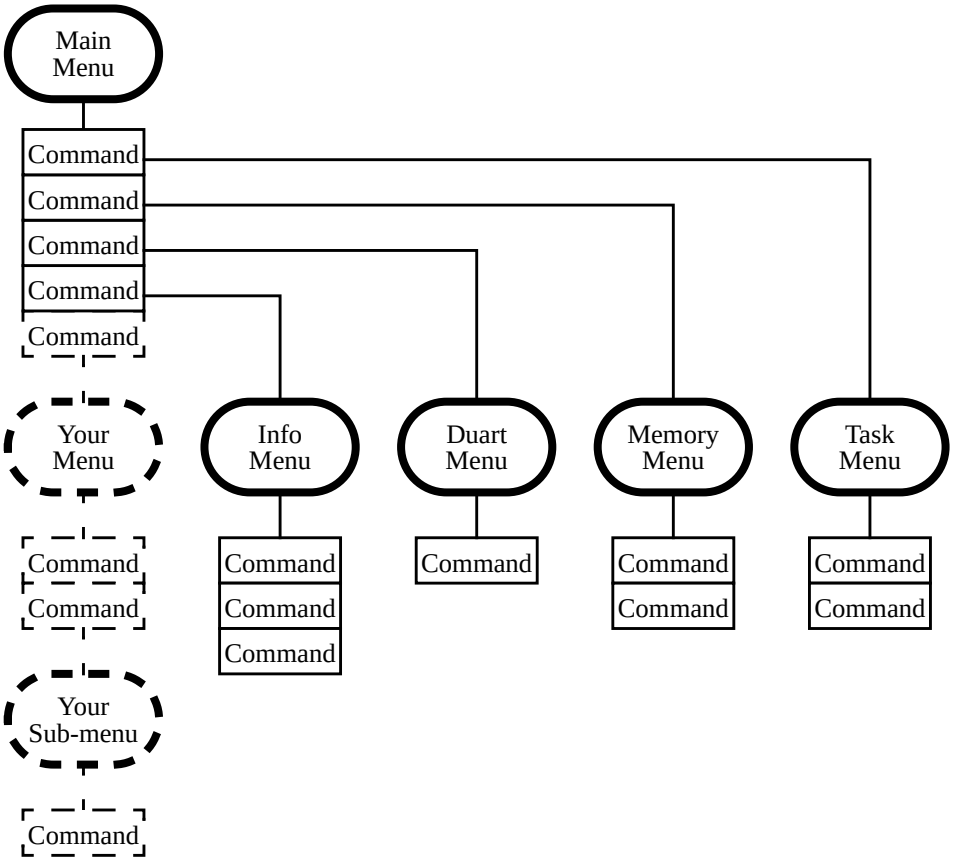

FIGURE 5.1 Monitor Menu Structure

In each menu, the monitor prints a prompt, such as "**Main** >" when the monitor is ready to accept a command. A command consists of a single character and, for some commands, of an additional argument. Some commands may be activated by different characters (e.g. H or ? for help), and commands are not case-sensitive. It is not possible to edit commands or arguments.

The two commands shown in Table 1 are valid for all menus:

Command	Action
H h ?	Print Help on commands available in menu.
Q q ESC	Return from this menu (ignored in main menu).

TABLE 1. Commands available in all menus

The remaining commands shown in Table 2 are only valid in their specific menus.

Menu	Command	Action	Argument
Main	I i	Enter Info Menu	-
Main	D d	Enter Duart Menu	-
Main	M m	Enter Memory Menu	-
Main	T t	Enter Task Menu	-
Info	O s	Display Overflows	-
Info	S s	Display Top of Memory	-
Info	T t	Display System Time	-
Duart	B b	Set Baud Rate	Baud Rate
Duart	C c	Change Channel	-
Duart	M m	Set Serial Mode	Data bits and Parity
Duart	T t	Transmit Character	Character (hex)
Memory	D	Display Memory	Address (hex)
Memory	\n	Continue Display Memory	-
Task	S s	Display all Tasks	-
Task	T t	Display particular Task	Task number
Task	P p	Set Task Priority	Priority (decimal)

TABLE 2. Specific commands

5.3 A Monitor Session

The commands of the monitor are best understood by looking at a commented monitor session. Commands and arguments entered are shown in bold font. When the monitor is started, it prints a start-up message:

```
Monitor started on channel 1.
Type H or ? for help.
Main Menu [D I M T H]
Main >
```

`H` (or `?`) shows the options available in the (main) menu:

```
Main > h
D - Duart Menu
I - Info Menu
M - Memory Menu
T - Task Menu
```

`D` enters the duart menu and `h` shows the options available:

```
Main > d
Duart Menu [B C M T H Q]
Duart_A > ?
B - Set Baud Rate
C - Change Channel
M - Change Mode
T - Transmit Character
```

`B` sets the baud rate of the duart channel A (SERIAL_0), `M` sets the data format. The monitor itself is running on SERIAL_1 so that this setting does not disturb the monitor session.

```
Duart_A > b
Baud Rate ? 9600
Duart_A >
Duart_A > m
Data Bits (5-8) ? 8
Parity (N O E M S) ? n
Databits = 8 / Parity = n set.
```

`C` toggles the duart channel, which changes the prompt of the duart menu.

```
Duart_A > c
Duart_B >
```

`T` transmits a character. The character is entered in hex (0x44 is ASCII ’D’).

```
Duart_B > t 44
Sending 0x44D
Duart_B >
```

The last character (’D’) in the line above is the character transmitted. Q exits the duart menu and i enters the info menu.

```
Duart_B > q
Main > i
Info > ?
O - Overflows
S - System Memory
T - System Time
Info Menu [O S T H Q]
```

O displays the overflows of the serial input queues.

```
Info > o
Ch 0 in  : 0
Ch 1 in  : 0
```

S displays the top of the system RAM used. Since the RAM is starting at address 0x20000, the total amount of RAM required is slightly more than 4 kBytes:

```
Info > s
Top of System Memory: 20001050
```

T shows the time since system start-up in milliseconds (i.e. 23 seconds) and q leaves the info menu.

```
Info > t
System Time: 0:23140
Info > q
```

M enters the memory menu and h shows the available options.

```
Main > m
Memory Menu [D H Q]
Memory > h
D - Dump Memory
```

D dumps the memory from the address specified. The memory dump may be continued after the last address by typing return (not shown). Here, the address is 0; thus dumping the vector table at the beginning of **crt0.S**. Q leaves the memory menu.

```
Memory > d Dump Mamory at address 0x0
00000000: 6000 00FE 0000 0100 0000 0172 0000 0172 ‘..........r...r
00000010: 0000 0172 0000 0172 0000 0172 0000 0172 ...r...r...r...r
00000020: 0000 0172 0000 0172 0000 0172 0000 0172 ...r...r...r...r
00000030: 0000 0172 0000 0172 0000 0172 0000 0172 ...r...r...r...r
00000040: 0000 0172 0000 0172 0000 0172 0000 0172 ...r...r...r...r
00000050: 0000 0172 0000 0172 0000 0172 0000 0172 ...r...r...r...r
00000060: 0000 0172 0000 0172 0000 01A4 0000 0172 ...r...r.......r
00000070: 0000 0172 0000 0172 0000 0172 0000 0172 ...r...r...r...r
```

```
00000080: 0000 02F6 0000 0306 0000 0172 0000 03AC ...........r....
00000090: 0000 03FE 0000 0444 0000 0172 0000 0172 .......D...r...r
000000A0: 0000 0172 0000 0172 0000 0172 0000 0172 ...r...r...r...r
000000B0: 0000 0172 0000 0458 0000 046A 0000 0474 ...r...X...j...t
000000C0: FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF ................
000000D0: FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF ................
000000E0: FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF ................
000000F0: FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF ................
Memory > q
```

`T` enters the task menu and `h` shows the available options.

```
Main > t
Task Menu [P S T H Q]
Task > h
P - Set Task Priority
S - Show Tasks
T - Show Task
```

`S` displays a list of all tasks. The current task is marked with an arrow:

```
Task > s Show Tasks:
-----------------------------------------------------
    TCB      Status Pri TaskName        ID  US Usage
-----------------------------------------------------
--> 20000664 RUN    240 Monitor Task     1   0000014C
    20000FB4 RUN      0 Idle Task        0   000000A0
=====================================================
```

`T` shows details of a particular task. The task number entered is the position of the task in the display of the previous command, starting at 0, rather than the task ID. Thus entering 1 displays the idle task rather than the monitor task.

```
Task > t Show Task:
Task number = 1
Task Name:   Idle Task
Priority:    0
TCB Address: 20000FB4
Status:      RUN
US Base:     2000020C
US Size:     00000200
US Usage:    000000A0 (31%)
Task >
```

Apparently the user stack of 512 bytes for the idle task could be reduced to 160 bytes. Finally, `p` sets the monitor task priority and `q` returns to the main menu:

```
Task > p Set Task Priority:
Task number = 0
Task priority = 200
```

```
Set Monitor Task Priority to 200
Task >
Task > q
Main >
```

In some cases, an additional prompt is printed after having entered numbers. The function accepting numbers waits until a non-digit, such as carriage return, is entered. If this carriage return is not caught, then it is interpreted as a command. Except for the memory menu, carriage return is not a valid command; it is ignored and a new prompt is displayed.

5.4 Monitor Implementation

The different monitor commands and menus are contained in a class **Monitor**, see Section A.19 for details. The monitor is included in the system by creating a task for the monitor in **setupApplicationStart()** and setting the channels **MonitorIn** and **MonitorOut** to the desired serial channel, in our case **SERIAL_1**.

```
  1  // ApplicationStart.cc
...
 22  void setupApplicationTasks()
 23  {
 24     MonitorIn  = SERIAL_1;
 25     MonitorOut = SERIAL_1;
 26     ErrorOut   = SERIAL_1;
 27     GeneralOut = SERIAL_1;
 28
 29     Monitor::setupMonitorTask();
 30  }
```

With **Monitor::setupMonitorTask()**, the monitor task is created:

```
  1  // Monitor.cc
...
 13  void Monitor::setupMonitorTask()
 14  {
 15     MonitorTask = new Task        (
 16                      monitor_main,    // function
 17                      2048,            // user stack size
 18                      16,              // message queue size
 19                      240,             // priority
 20                      "Monitor Task");
 21  }
```

Function **setupMonitorTask()** creates a task with main function **monitor_main**, a user stack of 2048 bytes, a message queue for 16 messages (which is actually not used), a task name of “Monitor Task”, and a priority of 240. The monitor should have a priority higher than that of all other tasks. This allows the monitor to display all tasks even if some task (of lower priority) is in busy wait (e.g by mistake) of some kind and to identify such tasks.

Function **monitor_main()**, which is the code executed by the monitor task, prints a message that the task has started and creates an instance of class **Monitor** using **MonitorIn** and **MonitorOut** as channels for the serial port and enters the main menu of the monitor.

```
  1  // Monitor.cc
...
 23  void Monitor::monitor_main()
 24  {
 25     SerialOut::Print(GeneralOut,
 26                      "\nMonitor started on channel %d.",
 27                      MonitorOut);
```

```
 28
 29     Monitor Mon(MonitorIn, MonitorOut);
 30     Mon.MonitorMainMenu();
 31  }
```

The constructor for class **Monitor** creates a **SerialIn** object **si** for its input channel. In contrast, the output channel is merely stored, but no **SerialOut** object is created. As a result, the input channel is reserved for the monitor forever, while the output channel can be used by other tasks as well. This explains why **ErrorOut** and **GeneralOut** could have been set to **SERIAL_1** as well. The remaining data members of class **Monitor** are used to remember the state of sub-menus even if the monitor returns from the menus.

```
  1  // Monitor.hh
...
 11  class Monitor
 12  {
 13  public:
 14     Monitor(Channel In, Channel Out)
 15        :  si(In), channel(Out), currentChannel(0), last_addr(0) {};
...
 48  };
```

The code for the menus is straightforward and basically the same for all menus. For instance, the main menu prints a prompt, receives the next character (command), and calls the function corresponding to the command (if any).

```
  1  // Monitor.cc
...
 59  //-------------------------------------------------------------------
 60  void Monitor::MonitorMainMenu()
 61  {
 62     SerialOut::Print(channel, "\nType H or ? for help.");
 63     SerialOut::Print(channel, "\nMain Menu [D I M T H]\n");
 64
 65     for (;;)    switch(getCommand("Main"))
 66         {
 67           case 'h': case 'H': case '?':
 68                {
 69                  SerialOut so(channel);
 70                  so.Print("\nD - Duart Menu");
 71                  so.Print("\nI - Info Menu");
 72                  so.Print("\nM - Memory Menu");
 73                  so.Print("\nT - Task Menu");
 74                }
 75                continue;
 76
 77           case 'd': case 'D': DuartMenu();    continue;
 78           case 'i': case 'I': InfoMenu();     continue;
 79           case 'm': case 'M': MemoryMenu();   continue;
 80           case 't': case 'T': TaskMenu();     continue;
 81         }
 82  }
```

The same ??? structure/code ??? applies for all other menus. However, we should focus on an interesting situation in the duart menu: here, the user can toggle the duart channel to which the commands of the duart menu apply with the command `C;` i.e. toggle between channels **SERIAL_0** and **SERIAL_1**. The actual channel chosen is displayed as the prompt of the duart menu. Now consider the `T` command, which reads a character to transmit (in hex), prints the character to be transmitted, and finally transmits the character on the duart channel selected. A naive implementation would be the following:

```
            case 't': case 'T':
                 {
                   SerialOut so(channel);
                   currentChar = si.Gethex(so);

                   so.Print("\nSending 0x%2X", currentChar & 0xFF);

                   Channel bc;

                   if (currentChannel)   bc = SERIAL_1;
                   else                  bc = SERIAL_0;

                   SerialOut::Print(bc, "%c", currentChar);
                 }
                 continue;
```

Function **getCurrentChannel()** simply returns **SERIAL_0** or **SERIAL_1**, depending on what has been selected with the `C` command. This works fine if **SERIAL_0** is selected. But what happens otherwise, i.e. if **getCurrentChannel()** returns **SERIAL_1**? In this case, we have already created a **SerialOut** object **so** for **channel** (which is **SERIAL_1**), and we are about to perform a **SerialOut::Print(bc,...)** with **bc** set to **SERIAL_1** as well. This print will try to create another **SerialOut** object for **SERIAL_1**. As we are already using **SERIAL_1**, the task blocks itself forever, because it claims a resource it already owns. This is a nice example of a deadlock. The proper way of handling the situation is as follows:

```
226           case 't': case 'T':
227                {
228                  SerialOut so(channel);
229                  currentChar = si.Gethex(so);
230
231                  so.Print("\nSending 0x%2X", currentChar & 0xFF);
232                }
233                {
234                  Channel bc;
235
236                  if (currentChannel)   bc = SERIAL_1;
237                  else                  bc = SERIAL_0;
238
239                  SerialOut::Print(bc, "%c", currentChar);
240                }
241                continue;
```

The lifetime of the **so** object is simply limited to merely getting the parameter and printing the message about the character that is about to be transmitted. The **so** object is then destructed, making channel **so** available again. The **SerialOut::Print(bc, ...)** can then use channel **bc** (whether it happens to be **SERIAL_1** or not) without deadlocking the monitor task.

6 Development Environment

6.1 General

In this chapter, we specify a complete development environment. This environment is based on the GNU C++ compiler **gcc** which is available for a large number of target systems (i.e. CPU families for the embedded system in this context). The **gcc** is available on the WWW and several CD-ROM distributions, particularly for Linux.

6.2 Terminology

In the following sections, two terms are frequently used: a *host* is a computer system used for developing software, while a *target* is a computer system on which this software is supposed to run, in our case an embedded system. In this context, a computer system is characterized by a CPU type or family, a manufacturer, and an operating system. Regarding the target, the manufacturer and the operating system are of little concern, since we are building this operating system ourselves. The basic idea here is to find an already existing target system that is supported by **gcc** and as similar as possible to our embedded system. This helps to reduce the configuration effort to the minimum.

Thus we are looking for a development environment that exactly matches our host (e.g. a workstation or a PC running DOS or Linux) and the CPU family of our embedded system (e.g. the MC68xxx family). All of the programs required and described below will run on the host, but some of them need to be configured to generate code for the target.

A program for which host and target are identical is called *native*; if host and target are different, the prefix *cross-* is used. For instance, a C++ compiler running on a PC under DOS and generating code to be executed under DOS as well is a native C++ compiler. Another C++ compiler running on a PC under DOS, but generating code for MC68xxx processors is a cross-compiler.

Due to the large number of possible systems, there are many more cross-compilers possible than native compilers. For this reason, native compilers are often available as executable programs in various places, while cross-compilers usually need to be made according to the actual host/target combination required.

It is even possible to create the cross-environment for the host on yet another system called the *build* machine. But in most cases, the host is the same as the build machine.

6.3 Prerequisites

In order to create the development environment, the following items are required on the host machine:

- A suitable native C compiler, preferably **gcc**
- Sufficient support for native program development
- A make program, preferably **gmake**

The term *suitable* refers to the requirements of the **binutils** and **gcc** packages which are stated in the **README** and **INSTALL** files provided with these packages. The **INSTALL** file for **gcc** says that "You cannot install GNU C by itself on **MSDOS**; it will not compile under any **MSDOS** compiler except itself". In such cases, you will need a native **gcc** in binary form; see Section 6.3.2.

Depending on your actual host, there are mainly three scenarios which are described in the following sections.

6.3.1 Scenario 1: UNIX or Linux Host

With a UNIX or Linux host, you already have a suitable native C compiler which may or may not be **gcc**. You also have several other programs such as **tar**, **sed**, and **sh** installed as part of the normal UNIX installation.

You also have a make program installed, but it might not be the GNU make program. In this case, you should consider to install GNU make as well and use it for building the cross-environment. GNU make is by default installed as a program called **make**, which may conflict with an already existing **make** program. In the following, we assume that GNU make is installed as **gmake** rather than **make**.

To install GNU make, proceed as follows:

- Get hold of a file called **make-3.76.1.tar.gz** and store it in a separate directory. You can get this file either from a CD-ROM, e.g. from a Linux distribution, or from the WWW:
  ftp://prep.ai.mit.edu/pub/gnu/make-3.76.1.tar.gz or
  ftp://ftp.funet.fi/pub/gnu/gnu/make-3.76.1.tar.gz
- In the separate directory, unpack the file:
  > **tar -xvzf make-3.76.1.tar.gz** or
  > **zcat make-3.76.1.tar.gz | tar -xvf -** if your **tar** program does not support the -z option
- Change to the directory created by the tar program:

> **cd make-3.76.1**

- Read the files **README** and **INSTALL** for instructions particular for your host
- Configure the package:
  > **./configure**
- Build the packet. This takes about 5 minutes:
  > **make**
- Install the packet. This may require root privileges, depending on where you want it to be installed. At this point, consider the name conflicts with the existing make program. Make sure that GNU make is installed as **gmake**:
  > **make install**

6.3.2 Scenario 2: DOS Host

The simplest way for a **DOS** host is to fetch binary versions of **gcc** and **gmake**. Please refer to

ftp://prep.ai.mit.edu/pub/gnu/MicrosPorts/MSDOS.gcc

for links to sites providing such binaries.

The **gcc** and **binutils** packages provide special means for building the cross-environment for **DOS**. The **gmake** is not strictly required, since it is not needed for building the cross-environment, and you will have to modify the **Makefile** for the embedded system anyway, since most **UNIX** commands are not available under **DOS**. You should fetch the **gmake** nevertheless, because this requires less changes for the target **Makefile**.

6.3.3 Scenario 3: Other Host or Scenarios 1 and 2 Failed

If none of the above scenarios discussed above succeeds, you can still survive:

- Get hold of a machine satisfying one of the above scenarios. This machine is called the **build** machine.
- On the build machine, install **gmake** (not required for scenario 2) and **gcc** as a native C compiler for the build machine.
- On the build machine, build the cross-environment as described later on. Observe the README and INSTALL files particularly carefully. When configuring the packets, set the **--build**, **--host** and **--target** options accordingly.

- Copy the cross-environment to your host.

After that, the build machine is no longer needed.

6.4 Building the Cross-Environment

In the following, we assume that the cross-environment is created in a directory called /**CROSS** on a **UNIX** or **Linux** host, which is also the build machine. In order to perform the "**make install**" steps below, you either need to be **root** or the **/CROSS** directory exists and you have write permission for it.

Since we assume a MC68020 CPU for the embedded system, we choose a **sun3** machine as target. This machine has a CPU of the MC68000 family and is referred to as m68k-sun-sunos4.1 when specifying targets. The general name for a target has the form CPU-Manufacturer-OperatingSystem.

For a DOS host, please follow the installation instructions provided with the **binutils** and **gcc** packages instead.

6.4.1 Building the GNU cross-binutils package

The GNU **binutils** package contains a collection of programs, of which some are essential. The absolute minimum required is the cross-assembler **as** (which is required by the GNU C++ cross-compiler) and the cross-linker **ld**. The **Makefile** provided in this book also uses the cross-archive program **ar**, the name utility **nm** and the **objcopy** program.

```
 1  # Makefile for gmake
 2  #
 3
 4  # Development environment.
 5  # Replace /CROSS by the path where you installed the environment
 6  #
 7  AR         := /CROSS/bin/m68k-sun-sunos4.1-ar
 8  AS         := /CROSS/bin/m68k-sun-sunos4.1-as
 9  LD         := /CROSS/bin/m68k-sun-sunos4.1-ld
10  NM         := /CROSS/bin/m68k-sun-sunos4.1-nm
11  OBJCOPY    := /CROSS/bin/m68k-sun-sunos4.1-objcopy
12  CC         := /CROSS/bin/m68k-sun-sunos4.1-gcc
13  MAKE       := gmake
```

Since the **Makefile** provided with the **binutils** package builds all these programs by default, there is no use at all to build only particular programs instead of the complete **binutils** suite.

To install the GNU **binutils** package, proceed as follows:

- Get hold of a file called **binutils-2.8.1.tar.gz** and store it in a separate directory, for instance **/CROSS/src**. You can get this file either from a CD-ROM, e.g. from a Linux distribution, or from the WWW: **ftp://prep.ai.mit.edu/pub/gnu/binutils-2.8.1.tar.gz** or

ftp://ftp.funet.fi/pub/gnu/gnu/binutils-2.8.1.tar.gz

- In the **/CROSS/src** directory, unpack the file:
  > **cd /CROSS/src**
  > **tar -xvzf binutils-2.8.1.tar.gz** or
  > **zcat binutils-2.8.1.tar.gz | tar -xvf -** if your **tar** program does not support the -z option
- Change to the directory created by the **tar** program:
  > **cd binutils-2.8.1**
- Read the file **README** for instructions particular for your host
- Configure the package. There is a period of a few minutes during which no screen output is generated. If your build machine is not the host, you need to specify a **--host=** option as well:
  > **./configure --target=m68k-sun-sunos4.1 **
  **> --enable-targets=m68k-sun-sunos4.1 **
  --prefix=/CROSS
- Build the packet, which takes about 20 minutes:
  > **gmake all-gcc**
- Install the packet, either as root or with write permission to /CROSS.
  > **gmake install**

6.4.2 Building the GNU cross-gcc package

To install the GNU **gcc** package, proceed as follows:

- Get hold of a file called **gcc-2.8.1.tar.gz** and store it in a separate directory, for instance. **/CROSS/src**. You can get this file either from a CD-ROM, e.g. from a Linux distribution, or from the WWW:
  ftp://prep.ai.mit.edu/pub/gnu/gcc-2.8.1.tar.gz or
  ftp://ftp.funet.fi/pub/gnu/gnu/gcc-2.8.1.tar.gz
- In the **/CROSS/src** directory, unpack the file:
  > **cd /CROSS/src**
  > **tar -xvzf gcc-2.8.1.tar.gz** or
  > **zcat gcc-2.8.1.tar.gz | tar -xvf -** if your **tar** program does not support the -z option
- Change to the directory created by the **tar** program:
  > **cd gcc-2.8.1**
- Read the file **INSTALL** for instructions particular for your host
- Configure the package. If your build machine is not the host, you need to specify a **--host=** option as well:
  > **./configure --target=m68k-sun-sunos4.1 **
  **--prefix=/CROSS **

**--with-gnu-ld **
--with-gnu-as

- Build the C and C++ compilers, which takes about 30 minutes. This make is supposed to fail when making **libgcc1.cross**. This is on purpose, since we have not supplied a **libgcc1.a** at this point:
  > **make LANGUAGES="C C++"**
- Install the compilers, either as root or with write permission to **/CROSS**:
  > **make LANGUAGES="c c++" install-common**
  > **make LANGUAGES="c c++" install-driver**
- You may optionally install man pages and/or info files as root:
  > **make LANGUAGES="c c++" install-man**
  > **make LANGUAGES="c c++" install-info**

Note: There are some dependencies between the actual **gcc** compiler version and the **libgcc.a** library used with it. There are also dependencies between the compiler version and the source code for the target, in particular regarding template class instantiation and support for C++ exceptions. It might therefore be necessary to change the source code provided in this book for different compiler versions.

6.4.3 The libgcc.a library

The **gcc** compiler requires a library that contains functions generated by the compiler itself. This library is usually called **libgcc.a**. The default installation procedure of **gcc** requires that a library **libgcc1.a** is provided beforehand and creates another library **libgcc2.a** itself. These two libraries **libgcc1.a** and **libgcc2.a** are then merged into the library **libgcc.a**. Since we have not provided a **libgcc1.a**, the build was aborted when building the make target **libgcc1.cross** as described in Section 6.4.2. The difference between **libgcc1.a** and **libgcc2.a** (besides the fact that they contain entirely different functions) is that **libgcc2.a** can be compiled with **gcc**, while **libgcc1.a** functions usually cannot, at least not without in-line assembly code.

The final step in setting up the cross-environment is to create **libgcc.a**:

- Change to the **gcc** build directory:
  > **cd /CROSS/gcc-2.8.1**
- Build the **libgcc2** library:
  > **make LANGUAGES="c c++" libgcc2.a**
- Rename **libgcc2.a** to **libgcc.a**:
  > **mv libgcc2.a libgcc.a**

At this point, you have a **libgcc.a**, but it still lacks the functions of **libgcc1.a** . The functions in **libgcc1.a** provide multiplication, division, and modulo operations for 32 bit and 64 bit integers. For the MC68020 and higher CPUs, these operations are directly supported by the CPU, and the **gcc** will use them if the **-mc68020** flag is present. In this case, there is nothing more to do and you may decide to leave the **libggc.a** as it is. If you do so, you should always check the final **Target.td** file for undefined symbols.

If you want to do it the proper way because you do not have a MC68020 CPU, or if you want to make sure that your cross-environment works under all circumstances, you have to provide the functions for **libgcc1.a** yourself. In order to get them compiled with **gcc**, you are of course not allowed to use the functions you are implementing.

As an example, we consider the function **_mulsi3**, which is supposed to multiply two signed 32 bit integers and to return the result. You may implement it as follows (not tested): ??? sollte das nicht besser doch getested sein ???

```
long _mulsi3(long p1, long p2)
{
long result;
int negative = 0;

   if (p1 < 0)   { p1 = -p1; negative++; }
   if (p2 < 0)   { p2 = -p2; negative++; }
   asm("
        MOVE.L %1,D1        | D1.L == p1
        MOVE.L %2,D2        | D2.L == p2
        MOVE.W D2,D0        | D0.W == p1_low
        MULU   D1,D0        | D0.L == p1_low * p2_low
        MOVE.L D2,D3        | D3.L == p2
        SWAP   D3           | D3.W == p2_high
        MULU   D1,D3        | D3.L == p1_low * p2_high
        SWAP   D1           | D1.W == p1_high
        MULU   D2,D1        | D1.L == p1_high * p2_low
        ADD.L  D1,D3        | D3.L == p1_low * p2_high + p1_high * p2_low
        SWAP   D3           | shift D3.L 16 bits, D3.W dirty
        CLR.W  D3           | D3.L == (p1_low * p2_high + p1_high * p2_low) << 16
        ADD.L  D3,D0        | D0.L == p1 * p2
        MOVE.L D0,%0        | store result
      " : =g(result) : "g"(p1), "g"(p2) : "d0", "d1",  "d2",  "d3" );

   if (negative & 1)   return -result;
   else                return result;
}
```

The **libgcc.a** contains several modules for C++ exception support. For an embedded system, you will most probably not use any exceptions at all, since exceptions are fatal errors in this context. When compiling C++ programs, the **gcc** enables exception processing by default. This will increase the size of the ROM image by about 9 kilobytes, which is slightly less than the whole operating system

without applications. You should therefore disable exception handling with the **gcc** option **-fno-exceptions**.

6.5 The Target Environment

The target environment is created by installing all files listed in the appendices in a separate directory on the host. In that directory, you can compile the sources in order to build the final ROM image, which can then be burned into an EPROM for the embedded system. Building the ROM image is achieved by entering

- > **gmake**

This command invokes the build process, which is controlled by the **Makefile**, and creates the ROM image both in binary format (file **Tartget.bin**) and in Srecord format (file **Target**).

6.5.1 The Target Makefile

The whole process of creating the ROM image is controlled by the **Makefile** which is explained in this section. The **Makefile** is used by **gmake** to start compilers, linkers, and so on as required for building the final ROM image. The **Makefile** starts with the locations where the cross-compiler and cross-binutils are installed. In our case, the **gcc** and **binutils** packages have been installed with **prefix=/CROSS**, which installed them below the /**CROSS** directory.

```
 1  # Makefile for gmake
 2  #
 3
 4  # Development environment.
 5  # Replace /CROSS by where you installed the cross-environment
 6  #
 7  CROSS-PREFIX:= /CROSS
 8  AR         := $(CROSS-PREFIX)/bin/m68k-sun-sunos4.1-ar
 9  AS         := $(CROSS-PREFIX)/bin/m68k-sun-sunos4.1-as
10  LD         := $(CROSS-PREFIX)/bin/m68k-sun-sunos4.1-ld
11  NM         := $(CROSS-PREFIX)/bin/m68k-sun-sunos4.1-nm
12  OBJCOPY    := $(CROSS-PREFIX)/bin/m68k-sun-sunos4.1-objcopy
13  CC         := $(CROSS-PREFIX)/bin/m68k-sun-sunos4.1-gcc
14  MAKE       := gmake
15
```

Then the target addresses for ROM and RAM are specified. These addresses are used by the linker. **ROM_BASE** is where the **.TEXT** section is to be linked, and **RAM_BASE** is where the **.DATA** section is to be linked.

```
16  # Target memory mapping.
17  #
18  ROM_BASE:= 0
19  RAM_BASE:= 20000000
```

The command line options for the assembler, linker, and compiler follow. The assembler is instructed to allow the additional MC68020 opcodes and addressing modes. The compiler is also told to use maximum optimization and not to use a frame pointer if none is required. The linker is instructed not to use standard libraries (remember that we did not build standard libraries for our environments), to use the target addresses specified above for the **.TEXT** and **.DATA** sections, and to create a map file. The map file should be checked after the build is completed.

```
21  # compiler and linker flags.
22  #
23  ASFLAGS    := -mc68020
24  CCFLAGS    := -mc68020 -O2 -fomit-frame-pointer -fno-exceptions
25
26  LDFLAGS    := -i -nostdlib \
27                -Ttext $(ROM_BASE) -Tdata $(RAM_BASE) \
28                -Xlinker -Map -Xlinker Target.map
```

Our source files are the assembler start-up file **crt0.S** and all files *.cc, assuming that no other files with extension .cc are stored in the directory where the ROM image is made.

```
30  # Source files
31  #
32  SRC_S      := $(wildcard *.S)
33  SRC_CC     := $(wildcard *.cc)
34  SRC        := $(SRC_S) $(SRC_CC)
```

For each .cc file, the compiler creates a .d file later on, using the -MM option. Rather than making a .cc file dependent of all header (.hh) files, which would lead to re-compiling all .cc files when any header file is changed, this ??? -MM option ??? only causes those .cc files to be compiled that include changed .hh files, which speeds up compilation.

```
36  # Dependency files
37  #
38  DEP_CC     := $(SRC_CC:.cc=.d)
39  DEP_S      := $(SRC_S:.S=.d)
40  DEP        := $(DEP_CC) $(DEP_S)
```

The object files to be created by the assembler or the compiler:

```
42  # Object files
43  #
44  OBJ_S      := $(SRC_S:.S=.o)
45  OBJ_CC     := $(SRC_CC:.cc=.o)
46  OBJ        := $(OBJ_S) $(OBJ_CC)
```

The files that are created by the build process and that may thus be deleted without harm:

```
48  CLEAN      := $(OBJ) $(DEP) libos.a \
```

```
49              Target Target.bin \
50              Target.td Target.text Target.data \
51              Target.map Target.sym
```

The default target (**all**) for the Makefile is the ROM image (**Target**) and the corresponding map and symbol files. Other targets are **clean**, which removes all non-source files (should also be used if entire source files are deleted), and **tar**, which creates a tar file containing the source files and the **Makefile**.

Note: Lines containing a command, like line 66, *must* start with a tab, rather than spaces.

```
53  # Targets
54  #
55  .PHONY:    all
56  .PHONY:    clean
57  .PHONY:    tar
58
59  all:       Target Target.sym
60
61  clean:
62             /bin/rm -f $(CLEAN)
63
64  tar:       clean
65  tar:
66             tar -cvzf ../src.tar *
```

The dependency files are included to create the proper dependencies between the included .cc files and .hh files:

```
68  include    $(DEP)
```

How are object and dependency files made? An object file is made by compiling a .cc or .S file, using the compiler flags discussed above. A dependency file is made by compiling a .cc file using the -MM option additionally. The dependency file itself has the same dependencies as the object file, but the dependency of the dependency file is not maintained automatically by the compiler. For this reason, the left side of a dependency (e.g. **file.o:**) is extended by the corresponding dependency file (resulting in **file.o file.d:**). This method will not work for DOS, because DOS does not have essential commands such as **sed**.

```
70  # Standard Pattern rules...
71  #
72  %.o:       %.cc
73             $(CC) -c $(CCFLAGS) $< -o $@
74
75  %.o:       %.S
76             $(CC) -c $(ASFLAGS) $< -o $@
77
78  %.d:       %.cc
79             $(SHELL) -ec '$(CC) -MM $(CCFLAGS) $< \
80                   | sed '\''s/$*\.o/$*\.o $@/'\'' > $@'
```

```
81
82  %.d:        %.S
83              $(SHELL) -ec '$(CC) -MM $(ASFLAGS) $< \
84                    | sed '\''s/$*\.o/$*\.o $@/'\'' > $@'
```

All object files are placed in a library called **libos.a**. Consequently, only the code that is actually required is included in the ROM image. If code size becomes an issue, then one can break down the source files into smaller source files, containing for instance only one function each. Linking is usually performed at file level, so that for files containing both used and unused functions, the unused functions are included in the final result as well. Splitting larger source files into smaller ones can thus reduce the final code size.

```
86  libos.a:$(OBJ)
87              $(AR) -sr libos.a $?
```

The final ROM image, **Target**, is made by converting the corresponding binary file, **Target.bin**, into Srecord format. Most EPROM programmers accept both binary and Srecord files. However, Srecord files are more convenient to read or to send by mail, and they also contain checksums.

```
89  Target:     Target.bin
90              $(OBJCOPY)  -I binary -O srec $< $@
```

The file **Target.text** contains the **.TEXT** section of the linker's output **Target.td** in binary format. It is created by instructing the **objcopy** to remove the **.DATA** segment and to store the result in binary format.

```
92  Target.text:Target.td
93              $(OBJCOPY)  -R .data -O binary $< $@
```

The file **Target.data** contains the **.DATA** section of the linker's output **Target.td** in binary format. It is created by instructing the **objcopy** to remove the **.TEXT** segment and to store the result in binary format.

```
95  Target.data:Target.td
96              $(OBJCOPY)  -R .text -O binary $< $@
```

For the target configuration we have chosen (aout format), a 32 byte header created is created if the **.TEXT** segment is linked to address 0. This header must be removed, e.g. by a small utility **skip_aout** which is described below. The file **Target.bin** is created by removing this header from **Target.text** and appending **Target.data**:

```
98  Target.bin:Target.text Target.data
99              cat Target.text | skip_aout | cat - Target.data > $@
```

The map file **Target.sym** is created by the **nm** utility with the linker's output. The **nm** is instructed to create a format easier to read by humans then the default output by the option **--demangle**. From this output, several useless symbols are

removed. The map file is useful to translate absolute addresses (e.g. in stack dumps created in the case of fatal errors) to function names.

```
101  Target.sym:Target.td
102             $(NM) -n --demangle $< \
103             | awk '{printf("%s %s\n", $$1, $$3)}' \
104             | grep -v compiled | grep -v "\.o" \
105             | grep -v "_DYNAMIC" | grep -v "^U" > $@
```

The object file **crt0.o** for the start-up code **crt0.S** is linked with **libos.a** (containing all object files for our sources) and with **libgcc** (containing all object files required by the **gcc** compiler).

```
108  Target.td:crt0.o libos.a libgcc.a
109             $(CC) -o $@ crt0.o  -L. -los -lgcc $(LDFLAGS)
```

6.5.2 The skip_aout Utility

As already mentioned, the **.TEXT** segment extracted from **Target.td** by **objcopy** starts with a 32 byte header if the link address is 0. This header can be removed by the following utility **skip_aout**, which simply discards the first 32 bytes from **stdin** and copies the remaining bytes to **stdout**.

```
// skip_aout.cc
#include <stdio.h>

enum { AOUT_OFFSET = 0x20 };   // 32 byte aout header to skip

int main(int, char *[])
{
int count, cc;

   for (count = 0; (cc = getchar()) != EOF; count++)
       if (count >= AOUT_OFFSET)   putchar(cc);

   exit(count < AOUT_OFFSET ? 1 : 0);
}
```

7 Miscellaneous

7.1 General

This chapter covers topics that do not fit in the previous chapters in any natural way.

7.2 Porting to different Processors

So far, a MC68020 has been assumed as target CPU. For using a different CPU, the assembler part of the kernel has to be rewritten. Since most of the code is specified in C++, the amount of code to be rewritten is fairly small. The files concerned are **crt0.S** and the files containing in-line assembler code, i.e. **os.cc**, **os.hh**, **Task.hh**, and **Semaphore.hh**.

7.2.1 Porting to MC68000 or MC68008 Processors

If the target CPU is a MC68000 or MC68008, then only one instruction in **crt0.S** needs to be removed. The start-up code **crt0.S** has been written so that it can be linked not only to base address 0 (i.e. assuming the code is executed directly after a processor RESET) but also to other addresses. In this case, a jump to the start of **crt0.S** is required:

```
  1   | crt0.S
...
 37   _null:  BRA     _reset                  |  0     initial SSP (end of RAM)
 38           .LONG   _reset                  |  1     initial PC
```

Normally, exception vector 0 contains the initial supervisor stack pointer, but since the supervisor stack pointer is not required from the outset, we have inserted a branch to label **_reset** instead. Thus a **BRA _null** has the same effect as a processor RESET. The CPU needs to know, however, where the vector table (starting at label **_null**) is located in the memory. For MC68010 CPUs and above, a special register, the vector base register **VBR**, has been implemented. After RESET, the **VBR** is set to 0. If **crt0.S** is linked to a different address, then the **VBR** has to be set accordingly. In **crt0.S**, the vector base address is computed automatically so that the user is not concerned with this matter:

```
  1   | crt0.S
```

```
...
 81   _reset:                                  |
 82            MOVE.L  #RAMend, SP             | since we abuse vector 0 for BRA.W
 83            LEA     _null, A0               |
 84            MOVEC   A0, VBR                 | MC68010++ only
```

The first instruction after label **_reset** sets up the SSP, which fixes the abuse of vector 0. Then the VBR is set to point to the actual vector table. For a MC68000 or a MC68008, there is no **VBR** and the instruction would cause an illegal instruction trap at this point. For a MC68000 or MC68008 CPU, the move instruction to the **VBR** must be removed. Clearly, for such CPUs it is impossible to locate the vector table (i.e. **crt0.S**) to anywhere else than address 0.

7.2.2 Porting to Other Processor families

The only specific feature of the MC68000 family we used was the distinction between supervisor mode and user mode. At the end of an exception processing routine, it was checked whether a change back to user mode would happen. If so, a pending task switch was executed.

```
235   _return_from_exception:                 | check for task switch
236            OR.W    #0x0700, SR            | disable interrupts
237           MOVE.W  (SP), -(SP)            | get status register before exception
238            AND.W   #0x2700, (SP)+          | supervisor mode or ints disabled ?
239            BNE     L_task_switch_done      | yes dont switch task
```

If a processor, e.g a Z80, does not provide different modes, then these modes can be emulated by a counter which is initialized to 0. For every exception, i.e. interrupts and also the function calls using the TRAP interface such as **Semaphore::P()**, this counter is incremented. At the end of every exception processing, the counter is decremented, and reaching 0 is equivalent to returning to user mode.

7.3 Saving Registers in Interrupt Service Routines

An interrupt service routine must not alter any registers. For a simple interrupt service routine, this can be achieved by saving those registers that the interrupt service routine uses and by restoring them after completion.

```
   1 | crt0.S
...
133   _duart_isr:                           |
134           MOVE.B  #LED_YELLOW, wLED_ON    | yellow LED on
135           MOVEM.L D0-D7/A0-A6, -(SP)      | save all registers
...
216           MOVEM.L (SP)+, D0-D7/A0-A6      | restore all registers
...
```

This is a safe way, but not the most efficient one. Considering the code between line 135 and 216, only registers D0, D1, D7, and A0 are modified by the interrupt service routine. So it would be sufficient to save and restore only these registers. However, the interrupt service routine calls other functions which may alter other registers, and these need to be saved as well. In order to save only those registers changed by the interrupt service routine and the functions it calls, one needs to know which registers are altered by the functions generated by the compiler. For some compilers, there is a convention such as “any function generated by the compiler may alter registers D0 through D3 and A0 through A3 and leaves all other registers intact”. The register preserving convention is usually documented for a compiler in a chapter like “function calling conventions”. In case of **gcc**, there is a file config/<machine>/<machine>.h in the directory where the compiler sources are installed, where <machine> stands for the target for which the compiler was configured. In our case, this would be the file config/m68k/m68k.h. In this file, a macro **CALL_USED_REGISTERS** is defined, which marks those registers with 1 that are changed by a function call. The first line refers to data registers, the next line to address registers and the third line to floating point registers.

```
// config/m68k/m68k.h
...
#define CALL_USED_REGISTERS \
 {1, 1, 0, 0, 0, 0, 0, 0,   \
  1, 1, 0, 0, 0, 0, 0, 1,   \
  1, 1, 0, 0, 0, 0, 0, 0 }
```

That is, if the compiler is configured to use the file m68k.h, then registers D0, D1, A0, A1, A7, and floating point registers FP0 and FP1 may be altered by function calls generated by the compiler. If the compiler uses other registers, it saves and restores them automatically. Although A7 (i.e. the SP) is altered, it is restored by the function call mechanism. With this knowledge, one could safely write

```
   1 | crt0.S
...
133   _duart_isr:                           |
```

```
134           MOVE.B  #LED_YELLOW, wLED_ON    | yellow LED on
135           MOVEM.L D0/D1/D7/A0/A1, -(SP)   | save registers used later on
...
216           MOVEM.L (SP)+, D0/D1/D7/A0/A1   | restore registers
...
```

This causes only 5 instead of 15 registers to be saved and restored. Since compilers tend to choose lower register numbers (D0, D1, A0, A1, FP0, and FP1) for registers that they may destroy, we chose a high register (D7) for the interrupt status so that it does not need to be saved before C++ function calls.

7.4 Semaphores with time-out

So far, the state machine shown in Figure 7.1 is used for the state of a task.

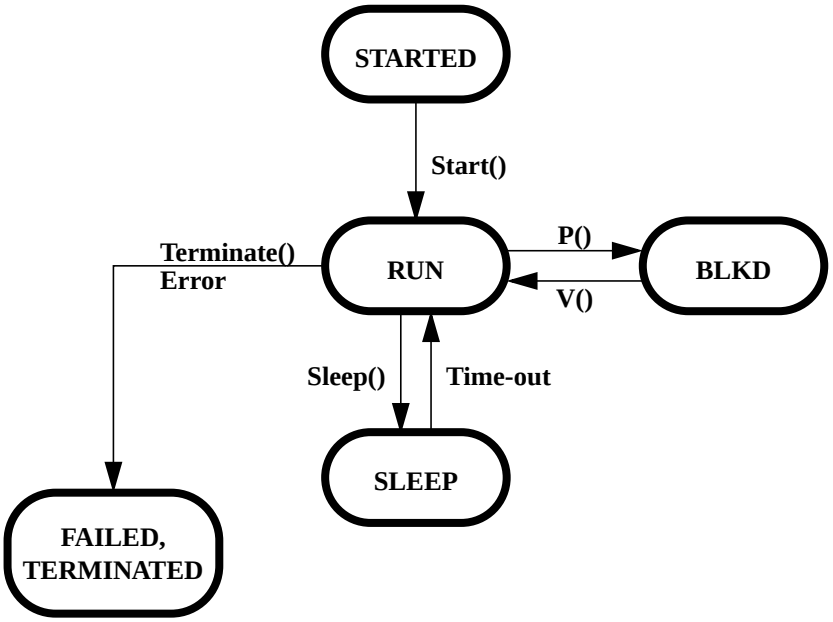

FIGURE 7.1 Task State Machine

Sometimes a combination of the states **SLEEP** and **BLKD** is required. One example is waiting for a character, but indicating a failure if the character is not received within a certain period of time. With the present state machine, there are several possibilities to achieve this, but none is perfect. We could, for instance, first **Sleep()** for the period and then **Poll()** to check if a character has arrived during **Sleep()**. This would lead to bad performance, in particular if the period is long and if time-out rarely occurs. One could increase the performance by performing **Sleep()** and **Poll()** in a loop with smaller intervals, but this would cost extra processing time. Another alternative would be to use two additional tasks: one that is responsible for receiving characters, and the other for sleeping. Any of these additional tasks would send an event to the task that is actually waiting for a character or time-out, indicating that the character has been received or that time-out has occurred. All this is significant effort for an otherwise simple problem. The best solution is to extend the task state machine by a new state **S_BLKD**, as shown in Figure 7.2.

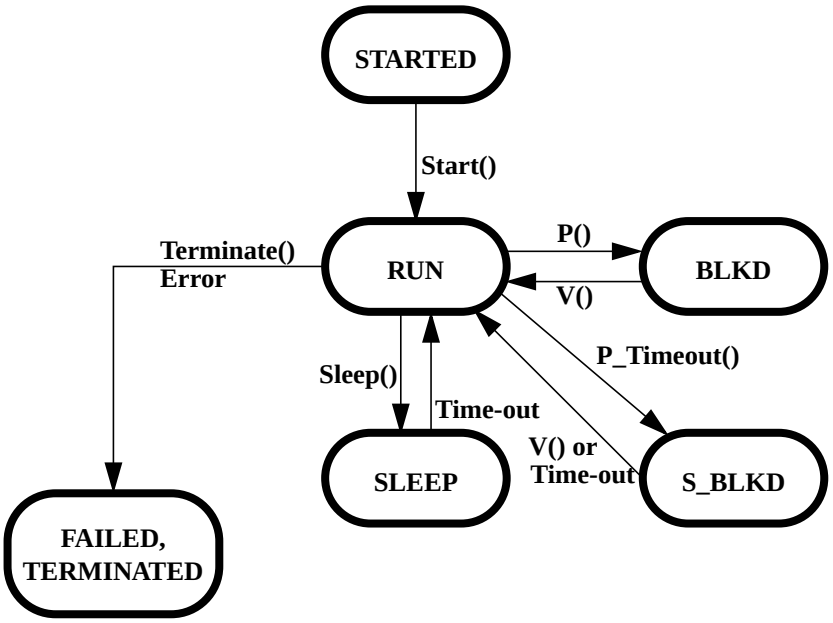

FIGURE 7.2 Task State Machine with new State S_BLKD

The new state **S_BLKD** combines the properties of states **SLEEP** and **BLKD** by returning the task to state **RUN** if either the resource represented by a semaphore is available (the character is received in our example) or the time-out provided with the call **Semaphore::P_Timeout(unsigned int time)** has expired. The task calling **P_Timeout()** must of course be able to determine whether the resource is available or time-out has occurred. That is, **P_Timeout()** will return e.g. an **int** indicating the result rather than **Semaphore::P()**, which returns **void**. The new state can be implemented as follows, where the details are left as an exercise to the reader. ??? willst Du die Lösung nicht verraten ???

- The class **Task** gets two new data members **int P_Timeout_Result** and **Semaphore * P_Timeout_Semaphore**.
- The class **Semaphore** is extended by a new member function **int P_Timeout(unsigned long time)**. This function is similar to **P()** with the following differences: If a resource is available, **P_Timeout()** returns 0 indicating no time-out. Otherwise it sets the current task's member **P_Timeout_Semaphore** to the semaphore on which **P_Timeout** is performed, sets the current task's TaskSleep to **time**, and blocks the task by setting both the **BLKD** and the **SLEEP** bits in the current task's **TaskStatus**. After the task has been unblocked by either a **V()** call or time-out, it returns **P_Timeout_Result** of the current task.

- **Semaphore::V()** is modified so that it sets the **P_Timeout_Result** of a task that is unblocked to 0, indicating no time-out. That task will then return 0 as the result of its **P_Timeout()** function call. It also clears the **SLEEP** bit of the task that is unblocked.
- If the sleep period of a task has expired (after label **L_SLEEP_LP** in **crt0.S**), then the **BLKD** bit is examined besides clearing the **SLEEP** bit of the task. If it is set, i.e. if the task is in state **S_BLKD**, then this bit is cleared as well, the task is removed from the semaphore waiting chain (using the **P_Timeout_Semaphore** member of the task) and **P_Timeout_Result** is set to nonzero, indicating time-out.

After the semaphore class has been extended this way, the queue classes are extended accordingly, implementing member functions like **Get_Timeout()** and **Put_Timeout()**. Since all these changes require considerable effort, they should only be implemented when needed. As a matter of fact, we have implemented quite complex applications without the need for time-outs in semaphores.

A Appendices

A.1 Startup Code (crt0.S)

```
 1  | crt0.S
 2
 3  #define ASSEMBLER
 4
 5  #include "Duart.hh"
 6  #include "Task.hh"
 7  #include "Semaphore.hh"
 8  #include "System.config"
 9                                          |
10          .global _null                   |
11          .global _on_exit                |
12          .global _reset                  |
13          .global _fatal                  |
14          .global _deschedule             |
15          .global _consider_ts            |
16          .global _return_from_exception  |
17          .global _stop                   |
18          .global _sdata                  |
19          .global _idle_stack             |
20          .global _IUS_top                |
21          .global _sysTimeHi              |
22          .global _sysTimeLo              |
23                                          |
24          .text                           |
25                                          |
26  wLED_ON         =       wDUART_BCLR     |
27  wLED_OFF        =       wDUART_BSET     |
28  LED_GREEN       =       0x80            |
29  LED_YELLOW      =       0x40            |
30  LED_RED         =       0x20            |
31  LED_ALL         =       0xE0            |
32                                          |
33  |=======================================================================|
34  |               VECTOR TABLE                                            |
35  |=======================================================================|
36                                          | Vector
37  _null:  BRA     _reset                  |  0     initial SSP (end of RAM)
38          .LONG   _reset                  |  1     initial PC
39          .LONG   _fatal, _fatal          |  2, 3  bus error, adress error
40         .LONG   _fatal, _fatal         |  4, 5  illegal instruction, divide/0
41          .LONG   _fatal, _fatal          |  6, 7  CHK, TRAPV instructions
42          .LONG   _fatal, _fatal          |  8, 9  privilege violation, trace
43          .LONG   _fatal, _fatal          | 10,11  Line A,F Emulators
44                                          |
45          .LONG   _fatal,_fatal,_fatal    | 12...  (reserved)
46          .LONG   _fatal,_fatal,_fatal    | 15...  (reserved)
47          .LONG   _fatal,_fatal,_fatal    | 18...  (reserved)
48          .LONG   _fatal,_fatal,_fatal    | 21...  (reserved)
49                                          |
50          .LONG   _fatal                  | 24     spurious interrupt
51          .LONG   _fatal                  | 25     level 1 autovector
52          .LONG   _duart_isr              | 26     level 2 autovector
53          .LONG   _fatal                  | 27     level 3 autovector
54          .LONG   _fatal, _fatal          | 28,29  level 4,5 autovector
55          .LONG   _fatal, _fatal          | 30,31 level 6,7 autovector
56                                          |
57          .LONG   _stop                   | 32     TRAP #0 vector
58          .LONG   _deschedule             | 33     TRAP #1 vector
```

```
 59          .LONG   _fatal                  | 34     TRAP #2 vector
 60          .LONG   _Semaphore_P            | 35     TRAP #3 vector
 61          .LONG   _Semaphore_V            | 36     TRAP #4 vector
 62          .LONG   _Semaphore_Poll         | 37     TRAP #5 vector
 63          .LONG   _fatal, _fatal          | 38,39  TRAP #6, #7 vector
 64          .LONG   _fatal, _fatal          | 40,41  TRAP #8, #9 vector
 65          .LONG   _fatal, _fatal          | 42,43  TRAP #10,#11 vector
 66          .LONG   _fatal                  | 44     TRAP #12 vector
 67          .LONG   _set_interrupt_mask     | 45     TRAP #13 vector
 68          .LONG   _readByteRegister_HL    | 46     TRAP #14 vector
 69          .LONG   _writeByteRegister      | 47     TRAP #15 vector
 70                                          |
 71          .FILL   16, 4, -1               | 48 .. 63 (reserved)
 72                                          |
 73  |=======================================================================|
 74  |               CODE                                                    |
 75  |=======================================================================|
 76                                          |
 77  |-----------------------------------------------------------------------|
 78  |               STARTUP  CODE                                           |
 79  |-----------------------------------------------------------------------|
 80                                          |
 81  _reset:                                 |
 82          MOVE.L  #RAMend, SP             | since we abuse vector 0 for BRA.W
 83          LEA     _null, A0               |
 84          MOVEC   A0, VBR                 | MC68010++ only
 85                                          |
 86          MOVE.B  #0, wDUART_OPCR         | all outputs via BSET/BCLR
 87          MOVE.B  #LED_ALL, wLED_OFF      | all LEDs off
 88                                          |
 89          MOVE.L  #RAMbase, A1            | clear RAM...
 90          MOVE.L  #RAMend, A2             |
 91  L_CLR:  CLR.L   (A1)+                   |
 92          CMP.L   A1, A2                  |
 93          BHI     L_CLR                   |
 94                                          | relocate data section...
 95          MOVE.L  #_etext, D0             | end of text section
 96          ADD.L   #0x00001FFF, D0         | align to next 2K boundary
 97          AND.L   #0xFFFFE000, D0         |
 98          MOVE.L  D0, A0                  | source (.data section in ROM)
 99          MOVE.L  #_sdata, A1             | destination (.data section in RAM)
100          MOVE.L  #_edata, A2             | end of .data section in RAM
101  L_COPY: MOVE.L  (A0)+, (A1)+            | copy data section from ROM to RAM
102          CMP.L   A1, A2                  |
103          BHI     L_COPY                  |
104                                          |
105          MOVE.L  #_SS_top, A7            | set up supervisor stack
106          MOVE.L  #_IUS_top, A0           |
107          MOVE    A0, USP                 | set up user stack
108                                          |
109          MOVE    #0x0700, SR             | user mode, no ints
110          JSR     _main                   |
111                                          |
112  _fatal:                                 |
113          MOVE.W  #0x2700, SR             |
114          MOVE.B  #LED_RED, wLED_ON       | red LED on
115          MOVE.B  #0x04, wDUART_CR_B      | enable transmitter
116          MOVE.L  SP, A0                  | old stack pointer
117          MOVE.L  #RAMend, SP             |
118  _forever:                               |
119          MOVE.L  A0, -(SP)               | save old stack pointer
120          MOVE.L  A0, -(SP)               | push argument
```

```
121           JSR     _Panic__2osPs           | print stack frame
122           LEA     2(SP), SP               | remove argument
123           MOVE.L  (SP)+, A0               | restore old stack pointer
124           BRA     _forever                |
125                                           |
126   _on_exit:                               |
127           RTS                             |
128                                           |
129   |-----------------------------------------------------------------------|
130   |               Duart interrupt                                         |
131   |-----------------------------------------------------------------------|
132                                           |
133   _duart_isr:                             |
134           MOVE.B  #LED_YELLOW, wLED_ON    | yellow LED on
135           MOVEM.L D0-D7/A0-A6, -(SP)      | save all registers
136           MOVEM.L rDUART_ISR, D7          | get interrupt sources
137           SWAP    D7                      |
138           MOVE.B  D7, _duart_isreg        |
139                                           |
140           BTST    #1, _duart_isreg        | RxRDY_A ?
141           BEQ     LnoRxA                  | no
142           MOVEM.L rDUART_RHR_A, D0        | get char received
143           MOVE.L  D0, -(SP)               |
144           PEA     1(SP)                   | address of char received
145           PEA     __8SerialIn$inbuf_0     | inbuf_0 object
146           JSR     _PolledPut__t10Queue_Gsem1ZUcRCUc
147           LEA     12(SP), SP              | cleanup stack
148   LnoRxA:                                 |
149                                           |
150           BTST    #5, _duart_isreg        | RxRDY_B ?
151           BEQ     LnoRxB                  | no
152           MOVEM.L rDUART_RHR_B, D0        | get char received
153           MOVE.L  D0, -(SP)               |
154           PEA     1(SP)                   | address of char received
155           PEA     __8SerialIn$inbuf_1     | inbuf_1 object
156           JSR     _PolledPut__t10Queue_Gsem1ZUcRCUc
157           LEA     12(SP), SP              | cleanup stack
158   LnoRxB:                                 |
159                                           |
160           BTST    #0, _duart_isreg        | TxRDY_A ?
161           BEQ     LnoTxA                  | no
162           LEA     -2(SP), SP              | space for next char
163           PEA     1(SP)                   | address of char received
164           PEA     __9SerialOut$outbuf_0   | outbuf_0 object
165           JSR     _PolledGet__t10Queue_Psem1ZUcRUc
166           LEA     8(SP), SP               | cleanup stack
167           MOVE.W  (SP)+, D1               | next output char (valid if D0 = 0)
168           TST.L   D0                      | char valid ?
169           BEQ     Ld1i11                  | yes
170           CLR.L   __9SerialOut$TxEnabled_0| no, disable Tx
171           MOVE.B  #0x08, wDUART_CR_A      | disable transmitter
172           BRA     LnoTxA                  |
173   Ld1i11: MOVE.B  D1, wDUART_THR_A        | write char (clears int)
174   LnoTxA:                                 |
175                                           |
176           BTST    #4, _duart_isreg        | TxRDY_B ?
177           BEQ     LnoTxB                  | no
178           LEA     -2(SP), SP              | space for next char
179           PEA     1(SP)                   | address of char received
180           PEA     __9SerialOut$outbuf_1   | outbuf_1 object
181           JSR     _PolledGet__t10Queue_Psem1ZUcRUc
182           LEA     8(SP), SP               | cleanup stack
```

```
183           MOVE.W  (SP)+, D1               | next output char (valid if D0 = 0)
184           TST.L   D0                      | char valid ?
185           BEQ     Ld1i21                  | yes
186           CLR.L   __9SerialOut$TxEnabled_1| no, disable Tx
187           MOVE.B  #0x08, wDUART_CR_B      | disable transmitter
188           BRA     LnoTxB                  |
189   Ld1i21: MOVE.B  D1, wDUART_THR_B        | write char (clears int)
190   LnoTxB:                                 |
191                                           |
192           BTST    #3, _duart_isreg        | timer ?
193           BEQ     LnoTim                  | no
194           MOVEM.L rDUART_STOP, D1         | stop timer
195           MOVEM.L rDUART_START, D1        | start timer
196                                           |
197                                           | increment system time
198           ADD.L   #10, _sysTimeLo         | 10 milliseconds
199           BCC.S   Lsys_time_ok            |
200           ADDQ.L  #1, _sysTimeHi          |
201   Lsys_time_ok:                           |
202                                           |
203           MOVE.L  __4Task$currTask, D1    |
204           MOVE.L  D1, A0                  |
205   L_SLEEP_LP:                             | decrement sleep counters...
206           SUBQ.L  #1, TaskSleepCount(A0)  |
207           BNE     L_NO_WAKEUP             |
208           BCLR    #3, TaskStatus(A0)      | clear sleep state
209   L_NO_WAKEUP:                            |
210           MOVE.L  TaskNext(A0), A0        |
211           CMP.L   A0, D1                  |
212           BNE     L_SLEEP_LP              |
213           ST      _consider_ts            | request task switch anyway
214   LnoTim:                                 |
215                                           |
216           MOVEM.L (SP)+, D0-D7/A0-A6      | restore all registers
217           BRA     _return_from_exception  |
218                                           |
219   |-----------------------------------------------------------------------|
220   |               TRAP #0 (STOP PROCESSOR)                                |
221   |-----------------------------------------------------------------------|
222                                           |
223   _stop:                                  |
224           MOVE.B  #LED_YELLOW, wLED_OFF   | yellow LED off
225           STOP    #0x2000                 |
226           BRA     _return_from_exception  | check for task switch
227                                           |
228   |-----------------------------------------------------------------------|
229   |               TRAP #1 (SCHEDULER)                                     |
230   |-----------------------------------------------------------------------|
231                                           |
232   _deschedule:                            |
233           ST      _consider_ts            | request task switch
234                                           |
235   _return_from_exception:                 | check for task switch
236           OR.W    #0x0700, SR             | disable interrupts
237          MOVE.W  (SP), -(SP)             | get status register before exception
238           AND.W   #0x2700, (SP)+          | supervisor mode or ints disabled ?
239           BNE     L_task_switch_done      | yes dont switch task
240           TST.B   _consider_ts            | task switch requested ?
241           BEQ     L_task_switch_done      | no
242           CLR.B   _consider_ts            | reset task switch request
243                                           |
244   |---------------------------------------|
```

```
245   |       swap out current task by saving
246   |       all user mode registers in TCB
247   |-------------------------------------|
248                                         |
249           MOVE.L  A6, -(SP)             | save A6
250           MOVE.L  __4Task$currTask, A6  |
251           MOVEM.L D0-D7/A0-A5, Task_D0(A6)| store D0-D7 and A0-A5 in TCB
252           MOVE.L  (SP)+, Task_A6(A6)    | store saved A6        in TCB
253           MOVE    USP, A0               |
254           MOVE.L  A0, Task_USP(A6)      | save USP from stack   in TCB
255           MOVE.B  1(SP), Task_CCR(A6)   | save CCR from stack   in TCB
256           MOVE.L  2(SP), Task_PC(A6)    | save PC  from stack   in TCB
257                                         |
258   |-------------------------------------|
259   |       find next task to run
260   |       A2: marker for start of search
261   |       A6: best candidate found
262   |       D6: priority of task A6
263   |       A0: next task to probe
264   |       D0: priority of task A0
265   |-------------------------------------|
266                                         |
267           MOVE.L  __4Task$currTask, A2  |
268           MOVE.L  A2, A6                |
269           MOVEQ   #0, D6                |
270           TST.B   TaskStatus(A6)        | status = RUN ?
271           BNE     L_PRIO_OK             | no, run at least idle task
272           MOVE.W  TaskPriority(A6), D6  |
273   L_PRIO_OK:                            |
274           MOVE.L  TaskNext(A6), A0      | next probe
275           BRA     L_TSK_ENTRY           |
276   L_TSK_LP:                             |
277           TST.B   TaskStatus(A0)        | status = RUN ?
278           BNE     L_NEXT_TSK            | no, skip
279           MOVEQ   #0, D0                |
280           MOVE.W  TaskPriority(A0), D0  |
281           CMP.L   D0, D6                | D6 higher priority ?
282           BHI     L_NEXT_TSK            | yes, skip
283           MOVE.L  A0, A6                |
284           MOVE.L  D0, D6                |
285           ADDQ.L  #1, D6                | prefer this if equal priority
286   L_NEXT_TSK:                           |
287           MOVE.L  TaskNext(A0), A0      | next probe
288   L_TSK_ENTRY:                          |
289           CMP.L   A0, A2                |
290           BNE     L_TSK_LP              |
291                                         |
292   |-------------------------------------|
293   |       next task found (A6)
294   |       swap in next task by restoring
295   |       all user mode registers in TCB
296   |-------------------------------------|
297                                         |
298           MOVE.L  A6, __4Task$currTask  | task found.
299           MOVE.L  Task_PC(A6), 2(SP)    | restore PC  on stack
300           MOVE.B  Task_CCR(A6), 1(SP)   | restore CCR on stack
301           MOVE.L  Task_USP(A6), A0      |
302           MOVE    A0, USP               | restore USP
303           MOVEM.L Task_D0(A6), D0-D7/A0-A6| restore D0-D7, A0-A5 (56 bytes)
304   L_task_switch_done:                   |
305           RTE                           |
306                                         |
```

```
307   |-------------------------------------------------------------------------|
308   |               TRAP #3 (Semaphore P operation)                           |
309   |-------------------------------------------------------------------------|
310                                           |
311   _Semaphore_P:                           | A0 -> Semaphore
312           OR      #0x0700, SR             | disable interrupts
313           SUBQ.L  #1, SemaCount(A0)       | count down resources
314           BGE     _return_from_exception  | if resource available
315           ST      _consider_ts            | request task switch
316           MOVE.L  SemaNextTask(A0), D0    | get waiting task (if any)
317           BNE.S   Lsp_append              | got a waiting task
318           MOVE.L  __4Task$currTask, D0    | get current Task
319           MOVE.L  D0, SemaNextTask(A0)    | store as first waiting
320           MOVE.L  D0, A0                  |
321           BSET    #0, TaskStatus(A0)      | block current task
322           CLR.L   TaskNextWaiting(A0)     | say this is last waiting
323           BRA     _return_from_exception  | done
324                                           |
325   Lsp_append:                             | goto end of waiting list
326           MOVE.L  D0, A0                  |
327           MOVE.L  TaskNextWaiting(A0), D0 | get next waiting (if any)
328           BNE.S   Lsp_append              | if not last waiting
329                                           |
330           MOVE.L  __4Task$currTask, D0    | get current task
331           MOVE.L  D0, TaskNextWaiting(A0) | store as last waiting
332           MOVE.L  D0, A0                  |
333           BSET    #0, TaskStatus(A0)      | block current task
334           CLR.L   TaskNextWaiting(A0)     | say this is last waiting
335           BRA     _return_from_exception  | done
336                                           |
337   |-------------------------------------------------------------------------|
338   |               TRAP #4 (Semaphore V operation)                           |
339   |-------------------------------------------------------------------------|
340                                           |
341   _Semaphore_V:                           | A0 -> Semaphore
342           OR      #0x0700, SR             | disable interrupts
343           ADDQ.L  #1, SemaCount(A0)       |
344           BLE.S   Lsv_unblock             | unblock waiting task
345           CLR.L   SemaNextTask(A0)        |
346           BRA     _return_from_exception  | done
347                                           |
348   Lsv_unblock:                            |
349           EXG     D0, A1                  |
350           MOVE.L  SemaNextTask(A0), A1    | get next waiting task
351           MOVE.L  TaskNextWaiting(A1), SemaNextTask(A0)
352           MOVE.L  A1, A0                  |
353           EXG     D0, A1                  |
354           BCLR    #0, TaskStatus(A0)      | unblock the blocked task
355           CLR.L   TaskNextWaiting(A0)     | just in case
356           MOVE.W  TaskPriority(A0), D0    | get priority of unblocked Task
357           MOVE.L  __4Task$currTask, A0    | get current Task
358           CMP.W   TaskPriority(A0), D0    | current prio >= unblocked prio ?
359           BLS     _return_from_exception  | yes, done
360           ST      _consider_ts            | no, request task switch
361           BRA     _return_from_exception  | done
362                                           |
363   |-------------------------------------------------------------------------|
364   |             TRAP #5 (Semaphore Poll operation)
|
365   |-------------------------------------------------------------------------|
366                                           |
367   _Semaphore_Poll:                        | A0 -> Semaphore
```

```
368           OR      #0x700, SR              | disable interrupts
369           MOVEQ   #1, D0                  | assume failure
370           TST.L   SemaCount(A0)           | get count
371           BLE     _return_from_exception  | failure
372           SUBQ.L  #1, SemaCount(A0)       |
373           MOVEQ   #0, D0                  | success
374           BRA     _return_from_exception  | check for task switch
375                                           |
376   |-----------------------------------------------------------------------|
377   |               TRAP #13 (SET INTERRUPT MASK)                           |
378   |-----------------------------------------------------------------------|
379                                           |
380   _set_interrupt_mask:                    |
381           MOVEQ   #7, D0                  |
382           AND.B   (SP), D0                | get old status register
383           AND.B   #7, D1                  | interrupt bits only
384           AND.B   #0xF8, (SP)             | clear interrupt bits
385           OR.B    D1, (SP)                | set interrupt bits from D1
386           BRA     _return_from_exception  | check for task switch
387                                           |
388   |-----------------------------------------------------------------------|
389   |               TRAP #14 (READ DUART REGISTER)                          |
390   |-----------------------------------------------------------------------|
391                                           |
392   _readByteRegister_HL:                   | (emulated)
393           MOVEM.L (A0), D0                | .L to force dummy cycle
394           SWAP    D0                      | D23..D16 -> D7..D0
395           BRA     _return_from_exception  | check for task switch
396                                           |
397   |-----------------------------------------------------------------------|
398   |               TRAP #15 (WRITE HARDWARE REGISTER)                      |
399   |-----------------------------------------------------------------------|
400                                           |
401   _writeByteRegister:                     | (emulated)
402           MOVE.B  D0, (A0)                |
403           BRA     _return_from_exception  | check for task switch
404                                           |
405   |=======================================================================|
406   |               DATA                                                    |
407   |=======================================================================|
408                                           |
409           .data                           |
410                                           |
411   _sdata:         .LONG   0               |
412   _sysTimeHi:     .LONG   0               | system time high
413   _sysTimeLo:     .LONG   0               | system time low
414   _super_stack:   .FILL   512, 1, 'S'     | supervisor stack
415   _SS_top:                                | top of supervisor stack
416   _idle_stack:    .FILL   512, 1, 'U'     | idle task user stack
417   _IUS_top:                               | top of idle task user stack
418   _consider_ts:   .BYTE   0              | true if task switch need be checked
419   _duart_isreg:   .BYTE   0               |
420                                           |
421           .ALIGN  2                       |
422           .END
```

A.2 Task.hh

```
#ifdef ASSEMBLER

#define TaskNext
#define TaskNextWaiting   0x04
#define Task_D0           0x08
#define Task_A6           0x40
#define Task_USP          0x44
#define Task_PC           0x48
#define TaskSleepCount    0x4C
#define TaskHitCount      0x50
#define TaskPriority      0x54
#define Task_CCR          0x56
#define TaskStatus        0x57

#else

#ifndef __TASK_HH_DEFINED__
#define __TASK_HH_DEFINED__
#include "Semaphore.hh"
#include "Message.hh"
#include "Queue.hh"

void setupApplicationTasks();

class Task
{
   friend class Monitor;
private:
   // Make sure the following locations match the assembler defs above !!!
   Task * next;                                      // 0x00
   Task * nextWaiting;                               // 0x04
   unsigned long Task_D0, Task_D1, Task_D2, Task_D3;  // 0x08..
   unsigned long Task_D4, Task_D5, Task_D6, Task_D7;  // 0x18..
   unsigned long Task_A0, Task_A1, Task_A2, Task_A3;  // 0x28..
   unsigned long Task_A4, Task_A5, Task_A6;           // 0x38..
   unsigned long * Task_USP;                          // 0x44..
   void (*Task_PC)();                                 // 0x48
   unsigned long TaskSleep;                           // 0x4C
   unsigned long TaskHitCount;                        // 0x50
   unsigned short  priority;                          // 0x54
   unsigned char Task_CCR;                            // 0x56
   unsigned char TaskStatus;                          // 0x57
   // End of definitions also used in assembler

   friend main();
   friend class Semaphore;

public:
   Task( void              (* main)(),
         unsigned long        userStackSize,
         unsigned short       queueSize,
         unsigned short       priority,
         const char *         taskName
       );

   static void GetMessage(Message & msg)
      { currTask->msgQ.Get(msg); };

   static int PolledGetMessage(Message & msg)
      { return currTask->msgQ.PolledGet(msg); };
```

```
 61
 62      static const char * const MyName()
 63         { return currTask->name; };
 64
 65      static unsigned short MyPriority()
 66         { return currTask->priority; };
 67
 68      static Task * Current()
 69         { return currTask; };
 70
 71      static void Dsched()
 72         { asm("TRAP #1"); };
 73
 74      static int SchedulerRunning()   { return SchedulerStarted; };
 75      static unsigned int Sleep(unsigned int);
 76      static void Terminate(int);
 77
 78      const char * const Name() const
 79         { return name; };
 80
 81      unsigned short Priority() const
 82         { return priority; };
 83
 84      void setPriority(unsigned short newPriority)
 85         { priority = newPriority; };
 86
 87      Task * Next() const
 88         { return next; };
 89
 90      unsigned char Status() const
 91         { return TaskStatus; };
 92
 93      void Start()
 94         { TaskStatus &=  ~STARTED; };
 95
 96      void SendMessage(Message & msg)
 97         { msg.Sender = currTask;   msgQ.Put(msg); };
 98
 99      int checkStacks();
100      unsigned int userStackUsed() const;
101
102      unsigned int userStackBase() const
103         { return (unsigned int)Stack; };
104
105      unsigned int userStackSize() const
106         { return US_size; };
107
108      enum { RUN        = 0x00,
109             BLKD       = 0x01,
110             STARTED    = 0x02,
111             TERMINATED = 0x04,
112             SLEEP      = 0x08,
113             FAILED     = 0x10,
114           };
115
116      static Task * TaskIDs[];
117   private:
118      Task();
119      ~Task();
120
121      void clearHitCount()
122         { TaskHitCount = 0; };
```

```
123
124      unsigned int HitCount() const
125         { return TaskHitCount; };
126
127
128      enum { userStackMagic = 'U', superStackMagic = 'S' };
129
130      static void Terminate_0();
131      static int                  SchedulerStarted;
132      static Task *               currTask;
133
134      char *                      Stack;           // user stack base
135      const unsigned long         US_size;         // user stack size
136      const char *                name;
137      int                         ExitCode;
138      Queue_Gsem_Psem<Message>    msgQ;
139   };
140
141   #endif  __TASK_HH_DEFINED__
142
143   #endif  ASSEMBLER
```

A.3 Task.cc

```
 1  // Task.cc
 2
 3  #include "Task.hh"
 4  #include "TaskId.hh"
 5  #include "System.config"
 6  #include "os.hh"
 7  #include "SerialOut.hh"
 8
 9  //------------------------------------------------------------------------
10  int                 Task::SchedulerStarted = 0;
11
12  Task *              Task::currTask = 0;
13  Task *              Task::TaskIDs[TASKID_COUNT];
14
15  //========================================================================
16  extern char idle_stack;
17  extern char IUS_top;
18
19  Task::Task()
20        : US_size(&IUS_top - &idle_stack),
21          priority(0),
22          name("Idle Task"),
23          TaskStatus(RUN),
24          next(this),
25          nextWaiting(0),
26          Stack(&idle_stack),
27          msgQ(1),
28          ExitCode(0)
29  {
30     TaskIDs[TASKID_IDLE] = this;
31  }
32  //------------------------------------------------------------------------
33  Task::Task(void (*main)(),
34             unsigned long   usz,
35             unsigned short  qsz,
36             unsigned short  prio,
37             const char *    taskName
38            )
39        : US_size(usz),
40          priority(prio),
41          name(taskName),
42          TaskStatus(STARTED),
43          nextWaiting(0),
44          msgQ(qsz),
45          ExitCode(0)
46  {
47  int i;
48
49     Stack = new char[US_size];   // allocate stack
50
51     for (i = 0; i < US_size;)   Stack[i++] = userStackMagic;
52
53     Task_A0  = 0xAAAA5555; Task_A1 = 0xAAAA4444;
54     Task_A2  = 0xAAAA3333; Task_A3 = 0xAAAA2222;
55     Task_A4  = 0xAAAA1111; Task_A5 = 0xAAAA0000;
56     Task_A6  = 0xAAAA6666;
57     Task_D0  = 0xDDDD7777; Task_D1 = 0xDDDD6666;
58     Task_D2  = 0xDDDD5555; Task_D3 = 0xDDDD4444;
59     Task_D4  = 0xDDDD3333; Task_D5 = 0xDDDD2222;
60     Task_D6  = 0xDDDD1111; Task_D7 = 0xDDDD0000;
```

```
 61      Task_PC  = main;
 62      Task_CCR = 0x0000;
 63
 64      Task_USP    = (unsigned long *)(Stack + US_size);
 65      *--Task_USP = (unsigned long)Terminate_0;
 66
 67      if (!currTask)
 68         currTask = new Task();
 69
 70      {
 71        os::INT_MASK old_INT_MASK = os::set_INT_MASK(os::NO_INTS);
 72        next = currTask->next;
 73        currTask->next = this;
 74        os::set_INT_MASK(old_INT_MASK);
 75      }
 76   }
 77   //========================================================================
 78   void main()
 79   {
 80      if (Task::SchedulerStarted)   return -1;
 81
 82      for (int i = 0; i < TASKID_COUNT; i++)   Task::TaskIDs[i] = 0;
 83      setupApplicationTasks();
 84
 85      for (Task * t = Task::currTask->next; t != Task::currTask; t = t->next)
 86          t->TaskStatus &= ~Task::STARTED;
 87
 88      Task::SchedulerStarted = 1;
 89      os::init(os::Interrupt_IO);   // switch on interrupt system
 90      os::set_INT_MASK(os::ALL_INTS);
 91
 92      Task::Dsched();
 93
 94      for (;;)   os::Stop();
 95
 96      return 0;   /* not reached */
 97   }
 98   //========================================================================
 99   void Task::Terminate_0()
100   {
101      Terminate(0);
102   }
103   //========================================================================
104   void Task::Terminate(int ex)
105   {
106      {
107        SerialOut so(ErrorOut);
108        so.Print("\n%s Terminated", currTask->name);
109      }
110      currTask->ExitCode = ex;
111      currTask->TaskStatus |= TERMINATED;
112      Dsched();
113   }
114   //========================================================================
115   int Task::checkStacks()
116   {
117      if ((char *)Task_USP <  Stack          )   return 1;
118      if ((char *)Task_USP >= Stack + US_size)   return 2;
119      return 0;
120   }
121   //
==========================================================================
```

```
122   unsigned int Task::Sleep(unsigned int ticks)
123   {
124      if (!SchedulerStarted)   return 0;
125      if (ticks == 0)          ticks++;
126
127      {
128        os::INT_MASK old_INT_MASK = os::set_INT_MASK(os::NO_INTS);
129        currTask->TaskStatus |= SLEEP;
130        currTask->TaskSleep = ticks;
131        os::set_INT_MASK(old_INT_MASK);
132      }
133      Dsched();
134      return ticks;
135   }
136   //============================================================================
137   unsigned int Task::userStackUsed() const
138   {
139      for (int i = 0; Stack[i] == userStackMagic; i++)  /* empty */ ;
140      return US_size - i;
141   }
142   //============================================================================
```

A.4 os.hh

```
/* os.hh */

#include "Channels.hh"

#ifndef __OS_HH_DEFINED__
#define __OS_HH_DEFINED__

extern "C" void * sbrk(unsigned long);
template <class Type> class RingBuffer;
template <class Type> class Queue;
template <class Type> class Queue_Gsem;
template <class Type> class Queue_Psem;
template <class Type> class Queue_Gsem_Psem;
class Semaphore;

typedef unsigned long HW_ADDRESS;

class os
{
public:
   friend class Monitor;
   friend class SerialIn;
   friend class SerialOut;
   friend void * sbrk(unsigned long);

   static void Stop();                        // for Idle Task only

   static unsigned long long getSystemTime();   // system time in ms

   enum INIT_LEVEL {
                     Not_Initialized = 0,
                     Polled_IO       = 1,
                     Interrupt_IO    = 2
                    };

   static void init(INIT_LEVEL new_level);
   static int  setBaudRate(Channel, int);
   static int  setSerialMode(Channel, int databits, int parity);
   static INIT_LEVEL  initLevel()    { return init_level; };
   static void *      top_of_RAM()   { return free_RAM;   };

private:
   os();   // dont instantiate

   static char * free_RAM;

   static void Panic(short * SP);

   static INIT_LEVEL init_level;
   static void initDuart(HW_ADDRESS base, int baudA, int baudB);
   static void initChannel(HW_ADDRESS base, int baud);
   static void resetChannel(HW_ADDRESS base);

   static unsigned int readDuartRegister(HW_ADDRESS reg)
      {
        int result;
        asm volatile (
                     "MOVE.L %1, A0
                      TRAP   #14
```

```
60                       MOVE.L D0, %0" : "=g"(result) : "g"(reg) : "d0", "a0"
61                      );
62          return result;
63        };
64
65     static void writeRegister(HW_ADDRESS reg, int val);
66
67   public:
68     enum INT_MASK {
69                     NO_INTS  = 0x07,
70                     ALL_INTS = 0x00
71                   };
72
73     static INT_MASK set_INT_MASK(INT_MASK new_INT_MASK)
74        {
75          INT_MASK old_INT_MASK;
76
77          asm volatile (
78                       "MOVE.B %1, D1
79                        TRAP #13
80                        MOVE.B D0, %0"
81                        : "=g"(old_INT_MASK)
82                        :  "g"(new_INT_MASK)
83                        : "d0", "d1"
84                       );
85
86          return old_INT_MASK;
87        };
88   };
89
90   #endif __OS_HH_DEFINED__
91
```

A.5 os.cc

```
/* os.cc */
#include "System.config"
#include "os.hh"
#include "Task.hh"
#include "Semaphore.hh"
#include "SerialOut.hh"
#include "Channels.hh"
#include "Duart.hh"

os::INIT_LEVEL os::init_level = Not_Initialized;

//========================================================================
//
// functions required by libgcc2.a...
//

extern int edata;
char * os::free_RAM = (char *)&edata;

//------------------------------------------------------------------------
extern "C" void * sbrk(unsigned long size)
{
void * ret = os::free_RAM;

   os::free_RAM += size;

   if (os::free_RAM > *(char **)0)   // out of memory
      {
        os::free_RAM -= size;
        ret = (void *) -1;
      }

   return ret;
}
//------------------------------------------------------------------------
extern "C" void * malloc(unsigned long size)
{
void * ret = sbrk((size+3) & 0xFFFFFFFC);

   if (ret == (void *)-1)   return 0;
   return ret;
}

//------------------------------------------------------------------------
extern "C" void free(void *)
{
}
//------------------------------------------------------------------------
extern "C" void write(int, const char *text, int len)
{
SerialOut so(SERIAL_1);
   so.Print(text, len);
}
//------------------------------------------------------------------------
extern "C" void _exit(int ex)
{
   Task::Terminate(ex);
   /* not reached */
   for (;;);
}
```

```
 61
 62   //========================================================================
 63   //
 64   // crt0.S interface functions...
 65   //
 66
 67   void os::Stop()
 68   {
 69      asm("TRAP #0");
 70   }
 71   //------------------------------------------------------------------------
 72   void os::writeRegister(HW_ADDRESS reg, int v)
 73   {
 74      asm("MOVE.L %0,A0; MOVE.L %1,D0; TRAP #15" : : "g"(reg), "g"(v) :
"d0","a0");
 75   }
 76   //------------------------------------------------------------------------
 77   // return time since power on (or reload) in milliseconds
 78   //
 79
 80   extern volatile unsigned long sysTimeLo;   // in crt0.S
 81   extern volatile unsigned long sysTimeHi;   // in crt0.S
 82
 83   unsigned long long os::getSystemTime()
 84   {
 85      for (;;)
 86          {
 87            unsigned long sys_high_1 = sysTimeHi;
 88            unsigned long sys_low    = sysTimeLo;
 89            unsigned long sys_high_2 = sysTimeHi;
 90
 91            // sys_low overflows every 49.86 days. If this function is
 92            // hit by that event (very unlikely) then it may be that
 93            // sys_high_1 != sys_high_2. If so, we repeat reading
 94            // the system time.
 95            if (sys_high_1 != sys_high_2)   continue;
 96
 97            unsigned long long ret = sys_high_1;
 98            ret <<= 32;
 99            return ret + sys_low;
100          }
101   }
102   //------------------------------------------------------------------------
103   // print stack frame in case of fatal errors
104   //
105   void os::Panic(short * SP)
106   {
107   SerialOut so(SERIAL_0_POLLED);
108   int i;
109
110      so.Print("\n\n====================================");
111      so.Print("\nFATAL ERROR STACK DUMP: SP=%8X", SP);
112      so.Print("\n====================================");
113   //   for (i = -5; i < 0; i++)
114   //       so.Print("\n[SP - 0x%2X] : %4X" , -2*i, SP[i] & 0xFFFF);
115      so.Print("\n[SP + 0x00] : %4X      (SR)" , SP[0] & 0xFFFF);
116      so.Print("\n[SP + 0x02] : %4X%4X  (PC)" , SP[1] & 0xFFFF, SP[2] & 0xFFFF);
117      so.Print("\n[SP + 0x06] : %4X      (FType/Vector)" , SP[3] & 0xFFFF);
118      for (i = 4; i < 10; i++)
119          so.Print("\n[SP + 0x%2X] : %4X" , 2*i, SP[i] & 0xFFFF);
120      so.Print("\n====================================\n");
121   }
```

```
122
123   //=========================================================================
124   //
125   // hardware initialization functions...
126   //
127
128   void os::init(INIT_LEVEL iLevel)
129   {
130   enum { green = 1<<7 };   // green LED, write to BCLR turns LED on
131
132      if (init_level < Polled_IO)
133         {
134           initDuart(DUART, CSR_9600, CSR_9600);
135           init_level = Polled_IO;
136         }
137
138      if (iLevel == Interrupt_IO && init_level < Interrupt_IO)
139         {
140           readDuartRegister (rDUART_STOP);               // stop timer
141           writeRegister(xDUART_CTUR, CTUR_DEFAULT);   // set CTUR
142           writeRegister(xDUART_CTLR, CTLR_DEFAULT);   // set CTLR
143           readDuartRegister(rDUART_START);               // start timer
144
145           writeRegister(wDUART_IMR, INT_DEFAULT);
146           init_level = Interrupt_IO;
147         }
148   }
149   //-------------------------------------------------------------------------
----
150   void
151   os::initDuart(HW_ADDRESS base, int baudA, int baudB)
152   {
153      // setup outputs
154      writeRegister((HW_ADDRESS)(base + w_OPCR), OPCR_DEFAULT);
155
156      resetChannel(base + _A);
157      resetChannel(base + _B);
158
159      writeRegister(base + w_ACR,  ACR_DEFAULT);
160
161      initChannel(base + _A, baudA);
162      initChannel(base + _B, baudB);
163   }
164   //-------------------------------------------------------------------------
165   void os::resetChannel(HW_ADDRESS channel_base)
166   {
167   const HW_ADDRESS cr  = channel_base + w_CR;
168
169      writeRegister(cr, CR_RxRESET);    // reset receiver
170      writeRegister(cr, CR_TxRESET);    // reset transmitter
171   }
172   //-------------------------------------------------------------------------
173   void os::initChannel(HW_ADDRESS channel_base, int baud)
174   {
175   const HW_ADDRESS mr  = channel_base + x_MR;
176   const HW_ADDRESS cr  = channel_base + w_CR;
177   const HW_ADDRESS csr = channel_base + w_CSR;
178
179      writeRegister(cr,  CR_MR1);        // select MR1
180      writeRegister(mr,  MR1_DEFAULT);   // set MR1
181      writeRegister(mr,  MR2_DEFAULT);   // set MR2
182      writeRegister(csr, baud);            // set baud rate
```

```
183      writeRegister(cr,  CR_TxENA);      // enable transmitter
184      writeRegister(cr,  CR_RxENA);      // enable receiver
185   }
186   //-------------------------------------------------------------------------
187   int os::setSerialMode(Channel ch, int databits, int parity)
188   {
189   int mr1 = MR1_DEFAULT & ~(MR1_P_MASK | MR1_BITS_mask);
190
191      switch(databits)
192         {
193           case 5:  mr1 |= MR1_BITS_5;   break;
194           case 6:  mr1 |= MR1_BITS_6;   break;
195           case 7:  mr1 |= MR1_BITS_7;   break;
196           case 8:  mr1 |= MR1_BITS_8;   break;
197           default:   return -1;
198         }
199
200      switch(parity)
201         {
202           case 0:   mr1 |= MR1_P_EVEN   ; break;
203           case 1:   mr1 |= MR1_P_ODD    ; break;
204           case 2:   mr1 |= MR1_P_LOW    ; break;
205           case 3:   mr1 |= MR1_P_HIGH   ; break;
206           case 4:   mr1 |= MR1_P_NONE   ; break;
207           default:   return -1;
208         }
209
210      switch(ch)
211         {
212           case SERIAL_0:
213                writeRegister(wDUART_CR_A,  CR_MR1);        // select MR1
214                writeRegister(xDUART_MR_A,  mr1);             // set MR1
215                return 0;
216
217           case SERIAL_1:
218                writeRegister(wDUART_CR_B,  CR_MR1);        // select MR1
219                writeRegister(xDUART_MR_B,  mr1);             // set MR1
220                return 0;
221         }
222
223      return -1;
224   }
225   //-------------------------------------------------------------------------
226   int os::setBaudRate(Channel ch, int baud)
227   {
228   int csr;
229
230      switch(baud)
231         {
232           case 38400:  if ( ACR_DEFAULT & ACR_BRG_1)   return -1;
233                        csr = CSR_38400;   break;
234           case 19200:  if (~ACR_DEFAULT & ACR_BRG_1)   return -1;
235                        csr = CSR_19200;   break;
236           case  9600:  csr = CSR_9600;    break;
237           case  4800:  csr = CSR_4800;    break;
238           case  2400:  csr = CSR_2400;    break;
239           case  1200:  csr = CSR_1200;    break;
240           case   600:  csr = CSR_600;     break;
241           default:   return -1;
242         }
243
244      switch(ch)
```

```
245         {
246           case SERIAL_0: writeRegister(wDUART_CSR_A, csr);   return 0;
247           case SERIAL_1: writeRegister(wDUART_CSR_B, csr);   return 0;
248         }
249       return -1;
250   }
```

A.6 Semaphore.hh

```
#ifdef ASSEMBLER
#define SemaCount
#define SemaNextTask   4
#else  !ASSEMBLER
#ifndef __SEMAPHORE_HH_DEFINED__
#define __SEMAPHORE_HH_DEFINED__

class Task;

class Semaphore
{
public:
   Semaphore()        : count(1),   nextTask(0) {};
   Semaphore(int cnt) : count(cnt), nextTask(0) {};
   void P()    {
                 asm volatile ("MOVE.L %0, A0
                                TRAP   #3" : : "g"(this) : "d0", "a0");
               };
   void V()    {
                 asm volatile ("MOVE.L %0, A0
                                TRAP   #4" : : "g"(this) : "d0", "a0");
               };
   int Poll()  {
                 int r;

                 asm volatile ("MOVE.L %1, A0
                                TRAP   #5
                                MOVE.L D0, %0"
                                : "=g"(r) : "g"(this) : "d0", "a0");
                 return r;
               };
private:
   long   count;
   Task * nextTask;
};
#endif  __SEMAPHORE_HH_DEFINED__
#endif  ASSEMBLER

```

A.7 Queue.hh

```
 1  // Queue.hh
 2
 3  #ifndef __QUEUE_HH_DEFINED__
 4  #define __QUEUE_HH_DEFINED__
 5
 6  #include "os.hh"
 7  #include "Semaphore.hh"
 8
 9  #pragma interface
10
11  //-------------------------------------------------------------------------
12  template <class Type> class RingBuffer
13  {
14  public:
15     RingBuffer(unsigned int Size);
16     ~RingBuffer();
17
18     int IsEmpty() const { return (count)        ? 0 : -1; };
19     int IsFull()  const { return (count < size) ? 0 : -1; };
20
21     int Peek(Type & dest) const;
22
23  protected:
24     enum { QUEUE_OK = 0, QUEUE_FAIL = -1 };
25
26     virtual int  PolledGet(Type & dest) = 0;
27     virtual int  PolledPut(const Type & dest) = 0;
28     inline  void GetItem(Type & source);
29     inline  void PutItem(const Type & src);
30
31     unsigned int size;
32     unsigned int count;
33
34  private:
35     Type *       data;
36     unsigned int get;
37     unsigned int put;
38  };
39  //-------------------------------------------------------------------------
40  template <class Type> class Queue : public RingBuffer<Type>
41  {
42  public:
43     Queue(unsigned int sz)
44        : RingBuffer<Type>(sz), overflow(0), underflow(0)
45        {};
46
47     unsigned int getUnderflowCount() const   { return underflow; };
48     void         clearUnderflowCounter()     { underflow = 0;    };
49     unsigned int getOverflowCount() const    { return overflow;  };
50     void         clearOverflowCounter()      { overflow = 0;     };
51
52     int PolledGet(Type & dest);
53     int PolledPut(const Type & dest);
54
55  private:
56     unsigned int underflow;
57     unsigned int overflow;
58  };
59  //-------------------------------------------------------------------------
60  template <class Type> class Queue_Gsem : public RingBuffer<Type>
```

```
61   {
62   public:
63      Queue_Gsem(unsigned int sz)
64         : RingBuffer<Type>(sz), overflow(0), GetSemaphore(0)
65         {};
66
67      unsigned int getOverflowCount() const    { return overflow; };
68      void clearOverflowCounter()              { overflow = 0;    };
69
70      int PolledGet(Type & dest);
71      int PolledPut(const Type & dest);
72      void Get(Type & dest);
73
74   private:
75      Semaphore    GetSemaphore;
76      unsigned int overflow;
77   };
78   //---------------------------------------------------------------------------
79   template <class Type> class Queue_Psem : public RingBuffer<Type>
80   {
81   public:
82      Queue_Psem(unsigned int sz)
83         : RingBuffer<Type>(sz),
84           PutSemaphore(sz),
85           underflow(0)
86         {};
87
88      unsigned int getUnderflowCount() const   { return underflow; };
89      void clearUnderflowCounter()             { underflow = 0;    };
90
91      int PolledGet(Type & dest);
92      int  PolledPut(const Type & dest);
93      void Put(const Type & dest);
94
95   private:
96      unsigned int underflow;
97      Semaphore    PutSemaphore;
98   };
99   //---------------------------------------------------------------------------
100  template <class Type> class Queue_Gsem_Psem : public RingBuffer<Type>
101  {
102  public:
103     Queue_Gsem_Psem(unsigned int sz)
104        : RingBuffer<Type>(sz), PutSemaphore(sz), GetSemaphore(0)
105        {};
106
107     int  PolledGet(Type & dest);
108     int  PolledPut(const Type & dest);
109     void Get(Type & dest);
110     void Put(const Type & dest);
111
112  private:
113     Semaphore    GetSemaphore;
114     Semaphore    PutSemaphore;
115  };
116  //---------------------------------------------------------------------------
117  #endif __QUEUE_HH_DEFINED__
```

A.8 Queue.cc

```
 1  // Queue.cc
 2
 3  #pragma implementation "Queue.hh"
 4
 5  #include "Queue.hh"
 6  #include "Message.hh"
 7
 8  //========================================================================
 9  template <class Type> RingBuffer<Type>::RingBuffer(unsigned int Size)
10     : size(Size), get(0), put(0), count(0)
11
12  {
13     data = new Type[size];
14  }
15  //------------------------------------------------------------------------
16  template <class Type> RingBuffer<Type>::~RingBuffer()
17  {
18     delete [] data;
19  }
20  //------------------------------------------------------------------------
21  template <class Type> int RingBuffer<Type>::Peek(Type & dest) const
22  {
23  int ret = QUEUE_FAIL;
24
25     {
26       os::INT_MASK old_INT_MASK = os::set_INT_MASK(os::NO_INTS);
27       if (count)   { dest = data[get];   ret = QUEUE_OK; }
28       os::set_INT_MASK(old_INT_MASK);
29     }
30     return ret;
31  }
32  //------------------------------------------------------------------------
33  template <class Type> inline void RingBuffer<Type>::GetItem(Type & dest)
34  {
35     dest = data[get++];
36     if (get >= size)   get = 0;
37     count--;
38  }
39  //------------------------------------------------------------------------
40  template <class Type> inline void RingBuffer<Type>::PutItem(const Type &src)
41  {
42    data[put++] = src;
43    if (put >= size)   put = 0;
44    count++;
45  }
46  //========================================================================
47  template <class Type> int Queue<Type>::PolledGet(Type & dest)
48  {
49  int ret;
50
51     {
52       os::INT_MASK old_INT_MASK = os::set_INT_MASK(os::NO_INTS);
53       if (count)   { GetItem(dest);   ret = QUEUE_OK; }
54       else         { underflow++;     ret = QUEUE_FAIL; }
55       os::set_INT_MASK(old_INT_MASK);
56     }
57     return ret;
58  }
59  //------------------------------------------------------------------------
60  template <class Type> int Queue<Type>::PolledPut(const Type & dest)
```

```
 61   {
 62   int ret;
 63
 64      {
 65        os::INT_MASK old_INT_MASK = os::set_INT_MASK(os::NO_INTS);
 66        if (count < size)   { PutItem(dest);  ret = QUEUE_OK;   }
 67        else                { overflow++;     ret = QUEUE_FAIL; }
 68        os::set_INT_MASK(old_INT_MASK);
 69      }
 70      return ret;
 71   }
 72   //============================================================================
 73   template <class Type> void Queue_Gsem<Type>::Get(Type & dest)
 74   {
 75      GetSemaphore.P();
 76      {
 77        os::INT_MASK old_INT_MASK = os::set_INT_MASK(os::NO_INTS);
 78        GetItem(dest);
 79        os::set_INT_MASK(old_INT_MASK);
 80      }
 81   }
 82   //----------------------------------------------------------------------------
 83   template <class Type> int Queue_Gsem<Type>::PolledGet(Type & dest)
 84   {
 85      if (GetSemaphore.Poll())   return QUEUE_FAIL;
 86      {
 87        os::INT_MASK old_INT_MASK = os::set_INT_MASK(os::NO_INTS);
 88        GetItem(dest);
 89        os::set_INT_MASK(old_INT_MASK);
 90      }
 91      return QUEUE_OK;
 92   }
 93   //----------------------------------------------------------------------------
 94   template <class Type> int Queue_Gsem<Type>::PolledPut(const Type & dest)
 95   {
 96   int ret = QUEUE_FAIL;
 97
 98      {
 99        os::INT_MASK old_INT_MASK = os::set_INT_MASK(os::NO_INTS);
100        if (count < size)
101           {
102             PutItem(dest);
103             GetSemaphore.V();
104             ret = QUEUE_OK;
105           }
106        os::set_INT_MASK(old_INT_MASK);
107      }
108      return ret;
109   }
110   //============================================================================
111   template <class Type> int Queue_Psem<Type>::PolledGet(Type & dest)
112   {
113   int ret = QUEUE_FAIL;
114
115      {
116        os::INT_MASK old_INT_MASK = os::set_INT_MASK(os::NO_INTS);
117        if (count)
118           {
119             GetItem(dest);
120             PutSemaphore.V();
121             ret =  QUEUE_OK;
122           }
```

```
123       else
124          {
125            underflow++;
126            ret = QUEUE_FAIL;
127          }
128       os::set_INT_MASK(old_INT_MASK);
129     }
130     return ret;
131   }
132   //-------------------------------------------------------------------------
133   template <class Type> void Queue_Psem<Type>::Put(const Type & dest)
134   {
135      PutSemaphore.P();
136      {
137        os::INT_MASK old_INT_MASK = os::set_INT_MASK(os::NO_INTS);
138        PutItem(dest);
139        os::set_INT_MASK(old_INT_MASK);
140      }
141   }
142   //-------------------------------------------------------------------------
143   template <class Type> int Queue_Psem<Type>::PolledPut(const Type & dest)
144   {
145      if (PutSemaphore.Poll())  return QUEUE_FAIL;
146      {
147        os::INT_MASK old_INT_MASK = os::set_INT_MASK(os::NO_INTS);
148        PutItem(dest);
149        os::set_INT_MASK(old_INT_MASK);
150      }
151      return QUEUE_OK;
152   }
153   //=========================================================================
154   template <class Type> void Queue_Gsem_Psem<Type>::Get(Type & dest)
155   {
156      GetSemaphore.P();
157      {
158        os::INT_MASK old_INT_MASK = os::set_INT_MASK(os::NO_INTS);
159        GetItem(dest);
160        os::set_INT_MASK(old_INT_MASK);
161      }
162      PutSemaphore.V();
163   }
164   //-------------------------------------------------------------------------
165   template <class Type> int Queue_Gsem_Psem<Type>::PolledGet(Type & dest)
166   {
167      if (GetSemaphore.Poll())   return QUEUE_FAIL;
168      {
169        os::INT_MASK old_INT_MASK = os::set_INT_MASK(os::NO_INTS);
170        GetItem(dest);
171        os::set_INT_MASK(old_INT_MASK);
172      }
173      return QUEUE_OK;
174   }
175   //-------------------------------------------------------------------------
176   template <class Type> void Queue_Gsem_Psem<Type>::Put(const Type & dest)
177   {
178      PutSemaphore.P();
179      {
180        os::INT_MASK old_INT_MASK = os::set_INT_MASK(os::NO_INTS);
181        PutItem(dest);
182        os::set_INT_MASK(old_INT_MASK);
183      }
184      GetSemaphore.V();
```

```
185   }
186   //-------------------------------------------------------------------------
187   template <class Type> int Queue_Gsem_Psem<Type>::PolledPut(const Type &
dest)
188   {
189      if (PutSemaphore.Poll())   return QUEUE_FAIL;
190      {
191        os::INT_MASK old_INT_MASK = os::set_INT_MASK(os::NO_INTS);
192        PutItem(dest);
193        os::set_INT_MASK(old_INT_MASK);
194      }
195      GetSemaphore.V();
196      return QUEUE_OK;
197   }
198   //=========================================================================
199   typedef Queue_Gsem_Psem<Message>   MessageQueue;
200   typedef Queue_Gsem<unsigned char>  serialInQueue;
201   typedef Queue_Psem<unsigned char>  serialOutQueue;
202   //=========================================================================
```

A.9 Message.hh

```
 1  // Message.hh
 2
 3  #ifndef __MESSGAE_HH_DEFINED__
 4  #define __MESSGAE_HH_DEFINED__
 5  class Message
 6  {
 7  public:
 8     Message()                : Type(0), Body(0), Sender(0) {};
 9     Message(int t, void * b) : Type(t), Body(b), Sender(0) {};
10     int    Type;
11     void * Body;
12     const Task * Sender;
13  };
14
15  #endif  __MESSGAE_HH_DEFINED__
```

A.10 Channels.hh

```
 1  // Channels.hh
 2  #ifndef __CHANNELS_HH_DEFINED__
 3  #define __CHANNELS_HH_DEFINED__
 4
 5  enum Channel {
 6                SERIAL_0            = 0,
 7                SERIAL_1            = 1,
 8                SERIAL_0_POLLED     = 4,
 9                SERIAL_1_POLLED     = 5,
10                DUMMY_SERIAL        = 8,
11              };
12
13  extern Channel MonitorIn;
14  extern Channel MonitorOut;
15  extern Channel ErrorOut;
16  extern Channel GeneralOut;
17
18  #endif __CHANNELS_HH_DEFINED__
```

A.11 SerialOut.hh

```
 1  /* SerialOut.hh */
 2
 3  #ifndef __SERIALOUT_HH_DEFINED__
 4  #define __SERIALOUT_HH_DEFINED__
 5
 6  #include "Channels.hh"
 7
 8  // forward declarations...
 9  class Semaphore;
10  template <class Type> class Queue_Psem;
11
12  class SerialOut
13  {
14  public:
15     SerialOut(Channel);
16     ~SerialOut();
17
18     static int Print(Channel, const char *, ...);
19     static int IsEmpty(Channel);
20
21     int Print(const char *, ...);
22     void Putc(int character);
23  private:
24     static int print_form(void (*)(int),
25                           const unsigned char **&,
26                           unsigned const char * &);
27
28     static void Putc_0(int c);
29     static void Putc_1(int c);
30     static void Putc_0_polled(int c);   // Putc_0 before scheduler is
running
31     static void Putc_1_polled(int c);   // Putc_1 before scheduler is
running
32     static void Putc_dummy(int c);      // dummy Putc to compute
length
33
34     Channel channel;
35
36     static Semaphore Channel_0;
37     static Semaphore Channel_1;
38
39     static Queue_Psem<unsigned char> outbuf_0;
40     static Queue_Psem<unsigned char> outbuf_1;
41
42     static int TxEnabled_0;
43     static int TxEnabled_1;
44  };
45
46  #endif  __SERIALOUT_HH_DEFINED__
```

A.12 SerialOut.cc

```
  1  /* SerialOut.cc */
  2
  3  #include "System.config"
  4  #include "os.hh"
  5  #include "Task.hh"
  6  #include "SerialOut.hh"
  7  #include "Duart.hh"
  8
  9  //=====================================================================
 10  Queue_Psem<unsigned char> SerialOut::outbuf_0 (OUTBUF_0_SIZE);
 11  Queue_Psem<unsigned char> SerialOut::outbuf_1 (OUTBUF_1_SIZE);
 12
 13  int SerialOut::TxEnabled_0 = 1;   // pretend Transmitter is enabled
at startup
 14  int SerialOut::TxEnabled_1 = 1;
 15
 16  Semaphore SerialOut::Channel_0;
 17  Semaphore SerialOut::Channel_1;
 18
 19  //=====================================================================
 20  SerialOut::SerialOut(Channel ch) : channel(ch)
 21  {
 22     switch(channel)
 23        {
 24          case SERIAL_0:
 25               if (Task::SchedulerRunning())   Channel_0.P();
 26              else                            channel = SERIAL_0_POLLED;
 27               return;
 28
 29          case SERIAL_1:
 30               if (Task::SchedulerRunning())   Channel_1.P();
 31              else                            channel = SERIAL_1_POLLED;
 32               return;
 33
 34          case SERIAL_0_POLLED:
 35          case SERIAL_1_POLLED:
 36               return;
 37
 38          default:
 39               channel = DUMMY_SERIAL;        // dummy channel
 40               return;
 41           }
 42  }
 43  //---------------------------------------------------------------------
 44  SerialOut::~SerialOut()
 45  {
 46     switch(channel)
 47        {
 48          case SERIAL_0:   Channel_0.V();   return;
 49          case SERIAL_1:   Channel_1.V();   return;
 50        }
 51  }
 52  //=====================================================================
 53  void SerialOut::Putc_0(int c)
```

```
 54  {
 55  unsigned char cc = c;
 56
 57     outbuf_0.Put(cc);
 58     if (!TxEnabled_0)
 59        {
 60          TxEnabled_0 = 1;
 61          os::writeRegister(wDUART_CR_A, CR_TxENA);   // enable Tx
 62        }
 63  }
 64  //-------------------------------------------------------------------
 65  void SerialOut::Putc_1(int c)
 66  {
 67  unsigned char cc = c;
 68
 69     outbuf_1.Put(cc);
 70     if (!TxEnabled_1)
 71        {
 72          TxEnabled_1 = 1;
 73          os::writeRegister(wDUART_CR_B, CR_TxENA);   // enable Tx
 74        }
 75  }
 76  //-------------------------------------------------------------------
 77  void SerialOut::Putc_0_polled(int c)
 78  {
 79     if (os::initLevel() < os::Polled_IO)   os::init(os::Polled_IO);
 80
 81     while (!(os::readDuartRegister(rDUART_SR_A) & SR_TxRDY))   /**/ ;
 82
 83     os::writeRegister(wDUART_THR_A, c);
 84
 85     while (!(os::readDuartRegister(rDUART_SR_A) & SR_TxRDY))   /**/ ;
 86  }
 87  //-------------------------------------------------------------------
 88  void SerialOut::Putc_1_polled(int c)
 89  {
 90     if (os::initLevel() < os::Polled_IO)   os::init(os::Polled_IO);
 91
 92     while (!(os::readDuartRegister(rDUART_SR_B) & SR_TxRDY))   /**/ ;
 93
 94     os::writeRegister(wDUART_THR_B, c);
 95
 96     while (!(os::readDuartRegister(rDUART_SR_B) & SR_TxRDY))   /**/ ;
 97  }
 98  //-------------------------------------------------------------------
 99  void SerialOut::Putc_dummy(int)
100  {
101     // dummy Putc to compute length
102  }
103  //-------------------------------------------------------------------
104  void SerialOut::Putc(int c)
105  {
106     switch(channel)
107        {
108          case SERIAL_0:         Putc_0(c);           return;
109          case SERIAL_1:         Putc_1(c);           return;
```

```
110           case SERIAL_0_POLLED:  Putc_0_polled(c);    return;
111           case SERIAL_1_POLLED:  Putc_1_polled(c);    return;
112           case DUMMY_SERIAL:                          return;
113           default:                                    return;
114         }
115   }
116   //================================================================
117
118   const char * const hex = "0123456789abcdef";
119   const char * const HEX = "0123456789ABCDEF";
120
121   //----------------------------------------------------------------
122   int SerialOut::IsEmpty(Channel channel)
123   {
124      switch(channel)
125         {
126           case 0:  return outbuf_0.IsEmpty();
127           case 1:  return outbuf_1.IsEmpty();
128         }
129      return 1;  // Polled, dummy and remote IO is always empty
130   }
131   //----------------------------------------------------------------
132   int SerialOut::Print(Channel channel, const char * format, ...)
133   {
134   SerialOut so(channel);
135
136   void (*putc)(int);
137   const unsigned char ** ap = (const unsigned char **)&format;
138   const unsigned char * f   = *ap++;
139   int len = 0;
140   int cc;
141
142      switch(channel)
143         {
144           case SERIAL_0:         putc = Putc_0;           break;
145           case SERIAL_1:         putc = Putc_1;           break;
146           case SERIAL_0_POLLED:  putc = Putc_0_polled;    break;
147           case SERIAL_1_POLLED:  putc = Putc_1_polled;    break;
148           case DUMMY_SERIAL:     putc = Putc_dummy;       break;
149           default:               return 0;
150         }
151
152      while (cc = *f++)
153         if (cc != '%')    { putc(cc); len++; }
154         else              len += print_form(putc, ap, f);
155
156      return len;
157   }
158   //----------------------------------------------------------------
159   int SerialOut::Print(const char * format, ...)
160   {
161   void (*putc)(int);
162   const unsigned char ** ap = (const unsigned char **)&format;
163   const unsigned char * f = *ap++;
164   int len = 0;
165   int cc;
```

```
166
167     switch(channel)
168        {
169          case SERIAL_0:         putc = Putc_0;           break;
170          case SERIAL_1:         putc = Putc_1;           break;
171          case SERIAL_0_POLLED:  putc = Putc_0_polled;    break;
172          case SERIAL_1_POLLED:  putc = Putc_1_polled;    break;
173          case DUMMY_SERIAL:     putc = Putc_dummy;       break;
174          default:               return 0;
175        }
176
177     while (cc = *f++)
178        if (cc != '%')    { putc(cc);   len++; }
179        else              len += print_form(putc, ap, f);
180
181     return len;
182  }
183  //=================================================================
184  int
185  SerialOut::print_form(void (*putc)(int),
186                        const unsigned char **& ap,
187                        const unsigned char * & f)
188  {
189  int len = 0;
190  int min_len = 0;
191  int  buf_idx = 0;
192  union { const unsigned char * cp;
193          const char * scp;
194          long lo;
195          unsigned long ul; } data;
196  int cc;
197  unsigned char buf[10];
198
199     for (;;)
200        {
201          switch(cc = *f++)
202             {
203               case '0':   min_len *= 10;                    continue;
204               case '1':   min_len *= 10;   min_len += 1;   continue;
205               case '2':   min_len *= 10;   min_len += 2;   continue;
206               case '3':   min_len *= 10;   min_len += 3;   continue;
207               case '4':   min_len *= 10;   min_len += 4;   continue;
208               case '5':   min_len *= 10;   min_len += 5;   continue;
209               case '6':   min_len *= 10;   min_len += 6;   continue;
210               case '7':   min_len *= 10;   min_len += 7;   continue;
211               case '8':   min_len *= 10;   min_len += 8;   continue;
212               case '9':   min_len *= 10;   min_len += 9;   continue;
213
214               case '%':
215                     putc('%');
216                     return 1;
217
218               case 'c':
219                     data.cp = *ap++;
220                     putc(data.lo);
221                     return 1;
```

```
222
223               case 'd':
224                    data.cp = *ap++;
225                    if (data.lo < 0)
226                       {
227                         data.lo = -data.lo;
228                         putc('-');   len++;
229                       }
230
231                    do { buf[buf_idx++] = '0' + data.ul%10;
232                         data.ul = data.ul/10;
233                       } while (data.lo);
234
235                  while (min_len-- > buf_idx)   { putc(' ');   len++;
}
236
237                    do { cc = buf[--buf_idx];   putc(cc);   len++; }
238                         while (buf_idx);
239                    return len;
240
241               case 's':
242                    data.cp = *ap++;
243                    if (data.scp == 0)   data.scp = "(null)";
244                    while (cc = *data.cp++)
245                          { putc(cc);   len++;   min_len--; }
246
247                    while (min_len-- > 0)
248                          { putc(' ');   len++; }
249                    return len;
250
251               case 'x':
252                    data.cp = *ap++;
253                    do { buf[buf_idx++] = hex[0x0F & data.ul];
254                         data.ul >>= 4;
255                       } while (data.ul);
256
257                  while (min_len-- > buf_idx)   { putc('0');   len++;
}
258
259                    do { cc = buf[--buf_idx];   putc(cc);   len++; }
260                          while (buf_idx);
261                    return len;
262
263               case 'X':
264                    data.cp = *ap++;
265                    do { buf[buf_idx++] = HEX[0x0F & data.ul];
266                         data.ul >>= 4;
267                       } while (data.ul);
268
269                  while (min_len-- > buf_idx)   { putc('0');   len++;
}
270
271                    do { cc = buf[--buf_idx];   putc(cc);   len++; }
272                         while (buf_idx);
273                    return len;
274             }
```

```
275         }
276     }
277     //=================================================================
```

A.13 SerialIn.hh

```
 1  /* SerialIn.hh */
 2
 3  #ifndef __SERIALIN_HH_DEFINED__
 4  #define __SERIALIN_HH_DEFINED__
 5
 6  #include "Channels.hh"
 7
 8  // forward declarations...
 9  class Semaphore;
10  class SerialOut;
11  template <class Type> class Queue_Gsem;
12
13  class SerialIn
14  {
15  public:
16     SerialIn(Channel);
17     ~SerialIn();
18
19     static unsigned int getOverflowCounter(Channel);
20
21     int Getc();
22     int Pollc();
23     int Peekc();
24     int Gethex(SerialOut &);
25     int Getdec(SerialOut &);
26
27     enum SerialError
28        {
29          OVERRUN_ERROR = 1,
30          PARITY_ERROR  = 2,
31          FRAME_ERROR   = 3,
32          BREAK_DETECT  = 4
33        };
34  private:
35     Channel channel;
36
37     static Semaphore Channel_0;
38     static Semaphore Channel_1;
39
40     static Queue_Gsem<unsigned char> inbuf_0;
41     static Queue_Gsem<unsigned char> inbuf_1;
42  };
43
44  #endif  __SERIALIN_HH_DEFINED__
```

A.14 SerialIn.cc

```
 1  /* SerialIn.cc */
 2
 3  #include "System.config"
 4  #include "SerialIn.hh"
 5  #include "SerialOut.hh"
 6  #include "Task.hh"
 7  #include "Queue.hh"
 8
 9  Queue_Gsem<unsigned char> SerialIn::inbuf_0 (INBUF_0_SIZE);
10  Queue_Gsem<unsigned char> SerialIn::inbuf_1 (INBUF_1_SIZE);
11
12  Semaphore SerialIn::Channel_0;
13  Semaphore SerialIn::Channel_1;
14
15  //=====================================================================
16  SerialIn::SerialIn(Channel ch) : channel(ch)
17  {
18     switch(channel)
19        {
20          case SERIAL_0:  Channel_0.P();    break;
21          case SERIAL_1:  Channel_1.P();    break;
22        }
23  }
24  //=====================================================================
25  SerialIn::~SerialIn()
26  {
27     switch(channel)
28        {
29          case SERIAL_0:  Channel_0.V();    break;
30          case SERIAL_1:  Channel_1.V();    break;
31        }
32  }
33  //=====================================================================
34  int SerialIn::Getc()
35  {
36  unsigned char cc;
37
38     switch(channel)
39        {
40          case SERIAL_0:  inbuf_0.Get(cc);   return cc;
41          case SERIAL_1:  inbuf_1.Get(cc);   return cc;
42          default:        return -1;
43        }
44  }
45  //=====================================================================
46  int SerialIn::Pollc()
47  {
48  unsigned char cc;
49
50     switch(channel)
51        {
52        case SERIAL_0:  return inbuf_0.PolledGet(cc)  ? -1 : cc;
53          case SERIAL_1:  return inbuf_1.PolledGet(cc)  ? -1 : cc;
54          default:        return -1;
```

```
 55        }
 56  }
 57  //=================================================================
 58  int SerialIn::Peekc()
 59  {
 60  unsigned char cc;
 61
 62     switch(channel)
 63        {
 64          case SERIAL_0:   return inbuf_0.Peek(cc)  ? -1 : cc;
 65          case SERIAL_1:   return inbuf_1.Peek(cc)  ? -1 : cc;
 66          default:         return -1;
 67        }
 68  }
 69  //=================================================================
 70  int SerialIn::Gethex(SerialOut &so)
 71  {
 72  int ret = 0;
 73  int cc;
 74
 75     for (;;)   switch(cc = Peekc())
 76        {
 77          case -1:   // no char arrived yet
 78               Task::Sleep(1);
 79               continue;
 80
 81          case '0': case '1': case '2': case '3': case '4':
 82          case '5': case '6': case '7': case '8': case '9':
 83               ret <<= 4;
 84               ret += cc-'0';
 85               so.Print("%c", Pollc());   // echo char
 86               continue;
 87
 88          case 'A': case 'B': case 'C':
 89          case 'D': case 'E': case 'F':
 90               ret <<= 4;
 91               ret += cc+10-'A';
 92               so.Print("%c", Pollc());   // echo char
 93               continue;
 94
 95          case 'a': case 'b': case 'c':
 96          case 'd': case 'e': case 'f':
 97               ret <<= 4;
 98               ret += cc+10-'a';
 99               so.Print("%c", Pollc());   // echo char
100               continue;
101
102         default:
103               return ret;
104        }
105  }
106  //=================================================================
107  int SerialIn::Getdec(SerialOut &so)
108  {
109  int ret = 0;
110  int cc;
```

```
111
112     for (;;)   switch(cc = Peekc())
113        {
114          case -1:   // no char arrived yet
115               Task::Sleep(1);
116               continue;
117
118          case '0': case '1': case '2': case '3': case '4':
119          case '5': case '6': case '7': case '8': case '9':
120               ret *= 10;
121               ret += cc-'0';
122               so.Print("%c", Pollc());   // echo char
123               continue;
124
125         default:
126               return ret;
127        }
128  }
129  //=================================================================
130  unsigned int SerialIn::getOverflowCounter(Channel channel)
131  {
132     switch(channel)
133        {
134          case SERIAL_0:   return inbuf_0.getOverflowCount();
135          case SERIAL_1:   return inbuf_1.getOverflowCount();
136          default:         return 0;
137        }
138  }
139  //=================================================================
```

A.15 TaskId.hh

```
1  // TaskId.hh
2
3  enum { TASKID_IDLE = 0,
4         TASKID_MONITOR,
5         TASKID_COUNT                  // number of Task IDs
6       };
7
8  #define IdleTask            (Task::TaskIDs[TASKID_IDLE])
9  #define MonitorTask         (Task::TaskIDs[TASKID_MONITOR])
```

A.16 duart.hh

```
 1  #ifndef  __DUART_HH_DEFINED__
 2  #define  __DUART_HH_DEFINED__
 3
 4  /* DUART base address */
 5  #define DUART           0xA0000000
 6
 7  /* DUART channel offsets */
 8  #define _A              0x00
 9  #define _B              0x20
10
11  /* DUART register offsets */
12  #define x_MR            0x00
13  #define r_SR            0x04
14  #define w_CSR           0x04
15  #define w_CR            0x08
16  #define r_RHR           0x0C
17  #define w_THR           0x0C
18  #define r_IPCR          0x10
19  #define w_ACR           0x10
20  #define r_ISR           0x14
21  #define w_IMR           0x14
22  #define x_CTUR          0x18
23  #define x_CTLR          0x1C
24  #define x_IVR           0x30
25  #define r_IPU           0x34
26  #define w_OPCR          0x34
27  #define r_START         0x38
28  #define w_BSET          0x38
29  #define r_STOP          0x3C
30  #define w_BCLR          0x3C
31
32  /* DUART read/write registers */
33  #define xDUART_MR_A     (DUART + x_MR + _A)
34  #define xDUART_MR_B     (DUART + x_MR + _B)
35  #define xDUART_IVR      (DUART + x_IVR)
36  #define xDUART_CTUR     (DUART + x_CTUR)
37  #define xDUART_CTLR     (DUART + x_CTLR)
38
39  /* DUART read only registers */
40  #define rDUART_SR_A     (DUART + r_SR   + _A)
41  #define rDUART_RHR_A    (DUART + r_RHR  + _A)
42  #define rDUART_IPCR     (DUART + r_IPCR     )
43  #define rDUART_ISR      (DUART + r_ISR      )
44  #define rDUART_SR_B     (DUART + r_SR   + _B)
45  #define rDUART_RHR_B    (DUART + r_RHR  + _B)
46  #define rDUART_IPU      (DUART + r_IPU      )
47  #define rDUART_START    (DUART + r_START    )
48  #define rDUART_STOP     (DUART + r_STOP     )
49
50  /* DUART write only registers */
51  #define wDUART_CSR_A    (DUART + w_CSR  + _A)
52  #define wDUART_CR_A     (DUART + w_CR   + _A)
53  #define wDUART_THR_A    (DUART + w_THR  + _A)
54  #define wDUART_ACR      (DUART + w_ACR      )
```

```
 55  #define wDUART_IMR      (DUART + w_IMR      )
 56  #define wDUART_CSR_B    (DUART + w_CSR  + _B)
 57  #define wDUART_CR_B     (DUART + w_CR   + _B)
 58  #define wDUART_THR_B    (DUART + w_THR  + _B)
 59  #define wDUART_OPCR     (DUART + w_OPCR     )
 60  #define wDUART_BSET     (DUART + w_BSET     )
 61  #define wDUART_BCLR     (DUART + w_BCLR     )
 62
 63  /* DUART MR1 bit definitions */
 64  #define MR1_RxRTS       (1<<7)
 65  #define MR1_FFUL        (1<<6)
 66  #define MR1_EBLOCK      (1<<5)
 67
 68  #define MR1_P_EVEN      (0<<2)
 69  #define MR1_P_ODD       (1<<2)
 70  #define MR1_P_LOW       (2<<2)
 71  #define MR1_P_HIGH      (3<<2)
 72  #define MR1_P_NONE      (4<<2)
 73  #define MR1_P_void      (5<<2)
 74  #define MR1_M_DATA      (6<<2)
 75  #define MR1_M_ADDR      (7<<2)
 76  #define MR1_P_MASK      (7<<2)
 77
 78  #define MR1_BITS_5      (0<<0)
 79  #define MR1_BITS_6      (1<<0)
 80  #define MR1_BITS_7      (2<<0)
 81  #define MR1_BITS_8      (3<<0)
 82  #define MR1_BITS_mask   (3<<0)
 83
 84  #define MR1_DEFAULT     (MR1_P_NONE | MR1_BITS_8)
 85
 86  /* DUART MR2 bit definitions */
 87  #define MR2_NORM        (0<<6)
 88  #define MR2_ECHO        (1<<6)
 89  #define MR2_LOLO        (2<<6)
 90  #define MR2_RELO        (3<<6)
 91
 92  #define MR2_TxRTS       (1<<5)
 93  #define MR2_TxCTS       (1<<4)
 94  #define MR2_STOP_2      (15<<0)
 95  #define MR2_STOP_1      (7<<0)
 96
 97  #define MR2_DEFAULT     MR2_STOP_2
 98
 99  /* DUART SR bit definitions */
100  #define SR_BREAK        (1<<7)
101  #define SR_FRAME        (1<<6)
102  #define SR_PARITY       (1<<5)
103  #define SR_OVERRUN      (1<<4)
104  #define SR_TxEMPTY      (1<<3)
105  #define SR_TxRDY        (1<<2)
106  #define SR_RxFULL       (1<<1)
107  #define SR_RxRDY        (1<<0)
108
109  /* DUART CSR bit definitions */
110  #define BD_600          5
```

```
111  #define BD_1200         6
112  #define BD_2400         8
113  #define BD_4800         9
114  #define BD_9600         11
115  #define BD_19200        12
116  #define BD_38400        BD_19200
117  #define BD_TIMER        13
118
119  #define CSR_600         (BD_600   | BD_600  <<4)
120  #define CSR_1200        (BD_4800  | BD_4800 <<4)
121  #define CSR_2400        (BD_2400  | BD_2400 <<4)
122  #define CSR_4800        (BD_4800  | BD_4800 <<4)
123  #define CSR_9600        (BD_9600  | BD_9600 <<4)
124  #define CSR_19200       (BD_19200 | BD_19200<<4)
125  #define CSR_38400       (BD_38400 | BD_38400<<4)
126  #define CSR_TIMER       (BD_TIMER | BD_TIMER<<4)
127
128  /* DUART CR bit definitions */
129  #define CR_NOP          (0<<4)
130  #define CR_MR1          (1<<4)
131  #define CR_RxRESET      (2<<4)
132  #define CR_TxRESET      (3<<4)
133  #define CR_ExRESET      (4<<4)
134  #define CR_BxRESET      (5<<4)
135  #define CR_B_START      (6<<4)
136  #define CR_B_STOP       (7<<4)
137
138  #define CR_TxENA        (1<<2)
139  #define CR_TxDIS        (2<<2)
140
141  #define CR_RxENA        (1<<0)
142  #define CR_RxDIS        (2<<0)
143
144  /* DUART ACR bit definitions */
145  #define ACR_BRG_0       (0<<7)
146  #define ACR_BRG_1       (1<<7)
147
148  #define ACR_CNT_IP2     (0<<4)
149  #define ACR_CNT_TxCA    (1<<4)
150  #define ACR_CNT_TxCB    (2<<4)
151  #define ACR_CNT_XTAL    (3<<4)
152  #define ACR_TIM_IP2     (4<<4)
153  #define ACR_TIM_IP2_16  (5<<4)
154  #define ACR_TIM_XTAL    (6<<4)
155  #define ACR_TIM_XTAL_16 (7<<4)
156
157  #define ACR_INT_IP3     (1<<3)
158  #define ACR_INT_IP2     (1<<2)
159  #define ACR_INT_IP1     (1<<1)
160  #define ACR_INT_IP0     (1<<0)
161
162  #define ACR_DEFAULT     (ACR_TIM_XTAL_16 | ACR_BRG_0)
163  #define XTAL_FREQ       (3686400/2)
164  #define XTAL_FREQ_16    (XTAL_FREQ/16)
165  #define TS_RATE         100
166  #define CT_DEFAULT      (XTAL_FREQ_16/TS_RATE)
```

```
167  #define CTUR_DEFAULT    (CT_DEFAULT / 256)
168  #define CTLR_DEFAULT    (CT_DEFAULT & 255)
169
170  /* DUART IMR/ISR bit definitions */
171  #define INT_IPC         (1<<7)
172  #define INT_BxB         (1<<6)
173  #define INT_RxB         (1<<5)
174  #define INT_TxB         (1<<4)
175  #define INT_CT          (1<<3)
176  #define INT_BxA         (1<<2)
177  #define INT_RxA         (1<<1)
178  #define INT_TxA         (1<<0)
179
180  #define INT_DEFAULT     (INT_RxB | INT_TxB | INT_RxA | INT_TxA |
INT_CT)
181
182  /* DUART OPCR bit definitions */
183  #define OPCR_7_TxRDY_B  (1<<7)
184  #define OPCR_6_TxRDY_A  (1<<6)
185  #define OPCR_5_RxRDY_B  (1<<5)
186  #define OPCR_4_RxRDY_A  (1<<4)
187
188  #define OPCR_3_OPR_3    (0<<2)
189  #define OPCR_3_CT       (1<<2)
190  #define OPCR_3_TxC_B    (2<<2)
191  #define OPCR_3_RxC_B    (3<<2)
192
193  #define OPCR_2_OPR_2    (0<<0)
194  #define OPCR_2_TxC_A16  (1<<0)
195  #define OPCR_2_TxC_A    (2<<0)
196  #define OPCR_2_RxC_A    (3<<0)
197
198  #define OPCR_DEFAULT    0
199
200  #endif  __DUART_HH_DEFINED__
201
```

A.17 System.config

```
 1  #define ROMbase 0x00000000
 2  #define ROMsize 0x00040000
 3  #define RAMbase 0x20000000
 4  #define RAMsize 0x00040000
 5  #define RAMend  (RAMbase+RAMsize)
 6
 7  #define OUTBUF_0_SIZE  80
 8  #define OUTBUF_1_SIZE  80
 9  #define INBUF_0_SIZE   80
10  #define INBUF_1_SIZE   80
```

A.18 ApplicationStart.cc

```
 1  // ApplicationStart.cc
 2
 3  #include "os.hh"
 4  #include "Channels.hh"
 5  #include "SerialIn.hh"
 6  #include "SerialOut.hh"
 7  #include "Task.hh"
 8  #include "TaskId.hh"
 9  #include "Monitor.hh"
10
11  Channel MonitorIn  = DUMMY_SERIAL;
12  Channel MonitorOut = DUMMY_SERIAL;
13  Channel ErrorOut   = DUMMY_SERIAL;
14  Channel GeneralOut = DUMMY_SERIAL;
15
16  //---------------------------------------------------------------
17  //
18  // Note: do not Print() here !
19  // Multitasking and interrupt IO is not yet up and running
20  //
21  //
22  void setupApplicationTasks()
23  {
24     MonitorIn  = SERIAL_1;
25     MonitorOut = SERIAL_1;
26     ErrorOut   = SERIAL_1;
27     GeneralOut = SERIAL_1;
28
29     Monitor::setupMonitorTask();
30  }
```

A.19 Monitor.hh

```
 1  // Monitor.hh
 2
 3  #ifndef MONITOR_HH_DEFINED
 4  #define MONITOR_HH_DEFINED
 5
 6  #include "Channels.hh"
 7
 8  class SerialIn;
 9  class SerialOut;
10
11  class Monitor
12  {
13  public:
14     Monitor(Channel In, Channel Out)
15        :  si(In), channel(Out), currentChannel(0), last_addr(0) {};
16
17     static void setupMonitorTask();
18
19  private:
20     static void monitor_main();
21
22     // menus...
23     void MonitorMainMenu();
24     void InfoMenu();
25     void DuartMenu();
26     void TaskMenu();
27     void MemoryMenu();
28
29     int getCommand(const char * prompt);
30     int getCommand(const char * prompt, char arg);
31     int echoResponse();
32     // complex functions...
33     void setTaskPriority();
34     void showTasks();
35     void showTask();
36     void showTask(SerialOut &, const Task *, const char *);
37     const char * const  showTaskStatus(const Task * t);
38     void displayMemory(int cont);
39
40     SerialIn si;
41     const Channel channel;
42
43     int currentChannel;          // used in DuartMenu()
44     int currentChar;             // used in DuartMenu()
45     unsigned long last_addr;     // used in MemoryMenu()
46
47     enum { ESC = 0x1B };
48  };
49
50  #endif  MONITOR_HH_DEFINED
```

A.20 Monitor.cc

```
 1  // Monitor.cc
 2
 3  #include "System.config"
 4  #include "os.hh"
 5  #include "SerialIn.hh"
 6  #include "SerialOut.hh"
 7  #include "Channels.hh"
 8  #include "Task.hh"
 9  #include "TaskId.hh"
10  #include "Monitor.hh"
11
12  //-----------------------------------------------------------------
13  void Monitor::setupMonitorTask()
14  {
15     MonitorTask = new Task        (
16                     monitor_main,    // function
17                     2048,            // user stack size
18                     16,              // message queue size
19                     240,             // priority
20                     "Monitor Task");
21  }
22  //-----------------------------------------------------------------
23  void Monitor::monitor_main()
24  {
25     SerialOut::Print(GeneralOut,
26                      "\nMonitor started on channel %d.",
27                      MonitorOut);
28
29     Monitor Mon(MonitorIn, MonitorOut);
30     Mon.MonitorMainMenu();
31  }
32  //-----------------------------------------------------------------
33  int Monitor::getCommand(const char * prompt)
34  {
35     SerialOut::Print(channel, "\n%s > ", prompt);
36     return echoResponse();
37  }
38  //-----------------------------------------------------------------
39  int Monitor::getCommand(const char * prompt, char arg)
40  {
41     SerialOut::Print(channel, "\n%s_%c > ", prompt, arg);
42     return echoResponse();
43  }
44  //-----------------------------------------------------------------
45  int Monitor::echoResponse()
46  {
47  int cc = si.Getc() & 0x7F;
48
49    switch(cc)
50       {
51         case ESC:  SerialOut::Print(channel, "ESC ");   break;
52         case '\n':                                      break;
53         case '\r':                                      break;
54         default:   if (cc < ' ')                        break;
```

```
55                    SerialOut::Print(channel, "%c ", cc);
56       }
57    return cc;
58  }
59  //---------------------------------------------------------------
60  void Monitor::MonitorMainMenu()
61  {
62     SerialOut::Print(channel, "\nType H or ? for help.");
63     SerialOut::Print(channel, "\nMain Menu [D I M T H]\n");
64
65     for (;;)    switch(getCommand("Main"))
66         {
67           case 'h': case 'H': case '?':
68                {
69                  SerialOut so(channel);
70                  so.Print("\nD - Duart Menu");
71                  so.Print("\nI - Info Menu");
72                  so.Print("\nM - Memory Menu");
73                  so.Print("\nT - Task Menu");
74                }
75                continue;
76
77           case 'd': case 'D': DuartMenu();    continue;
78           case 'i': case 'I': InfoMenu();     continue;
79           case 'm': case 'M': MemoryMenu();   continue;
80           case 't': case 'T': TaskMenu();     continue;
81         }
82  }
83  //---------------------------------------------------------------
84  void Monitor::InfoMenu()
85  {
86     SerialOut::Print(channel, "\nInfo Menu [O S T H Q]");
87     for (;;)   switch(getCommand("Info"))
88         {
89           case 'h': case 'H': case '?':
90                {
91                  SerialOut so(channel);
92                  so.Print("\nO - Overflows");
93                  so.Print("\nS - System Memory");
94                  so.Print("\nT - System Time");
95                }
96                continue;
97
98           case ESC: case 'Q': case 'q':
99                return;
100
101          case 'o': case 'O':
102               {
103                 SerialOut so(channel);
104                 so.Print("\nCh 0 in  : %d",
105                          SerialIn::getOverflowCounter(SERIAL_0));
106                 so.Print("\nCh 1 in  : %d",
107                          SerialIn::getOverflowCounter(SERIAL_1));
108               }
109               continue;
110
```

```
111           case 's': case 'S':
112                {
113                  SerialOut::Print(channel, "\nTop of System Memory:
%8X",
114                                   os::top_of_RAM());
115                }
116                continue;
117
118           case 't': case 'T':
119                {
120                  unsigned long long time = os::getSystemTime();
121                  unsigned long t_low  = time;
122                  unsigned long t_high = time>>32;
123
124                  SerialOut::Print(channel, "\nSystem Time: %d:%d",
125                                   t_high, t_low);
126                }
127                continue;
128     }
129   }
130   //-----------------------------------------------------------------
131   void Monitor::DuartMenu()
132   {
133   int currentChar;
134   int databits;
135   int parity;
136   int baud;
137
138      SerialOut::Print(channel, "\nDuart Menu [B C M T H Q]");
139      for (;;)    switch(getCommand("Duart", 'A' + currentChannel))
140          {
141            case 'h': case 'H': case '?':
142                 {
143                   SerialOut so(channel);
144                   so.Print("\nB - Set Baud Rate");
145                   so.Print("\nC - Change Channel");
146                   so.Print("\nM - Change Mode");
147                   so.Print("\nT - Transmit Character");
148                 }
149                 continue;
150
151           case ESC: case 'Q': case 'q':
152                return;
153
154           case 'b': case 'B':
155                {
156                  SerialOut so(channel);
157                  so.Print("\nBaud Rate ? ");
158                  baud = si.Getdec(so);
159                  Channel bc;
160
161                  if (currentChannel)   bc = SERIAL_1;
162                  else                  bc = SERIAL_0;
163
164                  if (os::setBaudRate(bc, baud))
165                     so.Print("\nIllegal Baud Rate %d", baud);
```

```
166               }
167               continue;
168
169          case 'c': case 'C':
170               currentChannel = 1 & ++currentChannel;
171               continue;
172
173          case 'm': case 'M':
174               SerialOut::Print(channel, "\nData Bits (5-8) ? ");
175               databits = echoResponse() - '0';
176               if (databits < 5 || databits > 8)
177                  {
178                    SerialOut::Print(channel,
179                                     "\nIllegal Data bit count %d",
180                                     databits);
181                    continue;
182                  }
183
184
185               SerialOut::Print(channel, "\nParity (N O E M S) ? ");
186               parity = echoResponse();
187
188               {
189                 SerialOut so(channel);
190                 Channel bc;
191
192                 if (currentChannel)   bc = SERIAL_1;
193                 else                  bc = SERIAL_0;
194
195                 switch(parity)
196                    {
197                      case 'E': case 'e':
198                           os::setSerialMode(bc, databits, 0);
199                           break;
200
201                      case 'O': case 'o':
202                           os::setSerialMode(bc, databits, 1);
203                           break;
204
205                      case 'M': case 'm':
206                           os::setSerialMode(bc, databits, 2);
207                           break;
208
209                      case 'S': case 's':
210                           os::setSerialMode(bc, databits, 3);
211                           break;
212
213                      case 'N': case 'n':
214                           os::setSerialMode(bc, databits, 4);
215                           break;
216
217                      default:
218                           so.Print("\nIllegal Parity %c", parity);
219                           continue;
220                    }
221                  so.Print("\nDatabits = %d / Parity = %c set.",
```

```
222                             databits, parity);
223               }
224               continue;
225
226           case 't': case 'T':
227               {
228                 SerialOut so(channel);
229                 currentChar = si.Gethex(so);
230
231                 so.Print("\nSending 0x%2X", currentChar & 0xFF);
232               }
233               {
234                 Channel bc;
235
236                 if (currentChannel)   bc = SERIAL_1;
237                 else                  bc = SERIAL_0;
238
239                 SerialOut::Print(bc, "%c", currentChar);
240               }
241               continue;
242         }
243   }
244   //------------------------------------------------------------------
245   void Monitor::TaskMenu()
246   {
247      SerialOut::Print(channel, "\nTask Menu [P S T H Q]");
248      for (;;)   switch(getCommand("Task"))
249          {
250            case 'h': case 'H': case '?':
251                 {
252                   SerialOut so(channel);
253                   so.Print("\nP - Set Task Priority");
254                   so.Print("\nS - Show Tasks");
255                   so.Print("\nT - Show Task");
256                 }
257                 continue;
258
259            case ESC: case 'Q': case 'q':
260                 return;
261
262            case 'p': case 'P':
263                 SerialOut::Print(channel, "Set Task Priority:");
264                 setTaskPriority();
265                 continue;
266
267            case 's': case 'S':
268                 SerialOut::Print(channel, "Show Tasks:");
269                 showTasks();
270                 continue;
271
272            case 't': case 'T':
273                 SerialOut::Print(channel, "Show Task:");
274                 showTask();
275                 continue;
276         }
277   }
```

```
278  //---------------------------------------------------------------
279  void Monitor::MemoryMenu()
280  {
281  int gotD = 0;
282
283     SerialOut::Print(channel, "\nMemory Menu [D H Q]");
284     for (;;)   switch(getCommand("Memory"))
285         {
286           case 'h': case 'H': case '?':
287                {
288                  SerialOut so(channel);
289                  so.Print("\nD - Dump Memory");
290                  gotD = 0;
291                }
292                continue;
293
294           case ESC: case 'Q': case 'q':
295                return;
296
297           case 'd': case 'D':
298               SerialOut::Print(channel, "Dump Mamory at address 0x");
299                displayMemory(0);
300                gotD = 1;
301                continue;
302
303           case '\n':
304                if (gotD)   displayMemory(1);
305                continue;
306         }
307  }
308  //---------------------------------------------------------------
309  void Monitor::displayMemory(int cont)
310  {
311  unsigned int addr = last_addr;
312
313     if (cont == 0)  // dont continue
314        {
315          SerialOut so(channel);
316          addr = si.Gethex(so);
317          si.Pollc();   // discard terminating char for Gethex()
318        }
319
320     for (int line = 0; line < 16; line++)
321         if (  ROMbase <= addr && addr < ROMbase+ROMsize-16
322            || RAMbase <= addr && addr < RAMbase+RAMsize-16
323            )
324            {
325              SerialOut so(channel);
326              int j;
327              char cc;
328              so.Print("\n%8X: ", addr);
329
330              for (j = 0; j < 8; j++)
331                 so.Print("%4X ", 0xFFFF & (int)(((short *)addr)[j]));
332
333              for (j = 0; j < 16; j++)
```

```
334                 {
335                   cc = ((char *)addr)[j];
336                   if (cc < ' ' || cc > 0x7E)   cc = '.';
337                   so.Print("%c", cc);
338                 }
339
340             addr += 16;
341           }
342     last_addr = addr;
343   }
344   //-----------------------------------------------------------------
345   void Monitor::setTaskPriority()
346   {
347   Task * t = Task::Current();
348   unsigned short priority;
349      {
350        SerialOut so(channel);
351        while (si.Pollc() != -1) /* empty */ ;
352        so.Print("\nTask number = ");
353
354        for (int tindex = si.Getdec(so); tindex; tindex--)
355            t = t->Next();
356
357        while (si.Pollc() != -1) /* empty */ ;
358        so.Print("\nTask priority = ");
359        priority = si.Getdec(so);
360
361        if (priority == 0)   priority++;
362        so.Print("\nSet %s Priority to %d", t->Name(), priority);
363      }
364      t->setPriority(priority);
365   }
366   //-----------------------------------------------------------------
367   void Monitor::showTask()
368   {
369   const Task * t = Task::Current();
370   SerialOut so(channel);
371
372      so.Print("\nTask number = ");
373      for (int tindex = si.Getdec(so); tindex; tindex--)
374          t = t->Next();
375
376   const char * const stat = showTaskStatus(t);
377   unsigned int stackUsed  = t->userStackUsed();
378
379      so.Print("\nTask Name:   %s", t->Name());
380      so.Print("\nPriority:    %d", t->Priority());
381      so.Print("\nTCB Address: %8X", t);
382      if (stat)   so.Print("\nStatus:      %s",  stat);
383      else        so.Print("\nStatus:     %2X",  t->Status());
384      so.Print("\nUS Base:     %8X", t->userStackBase());
385      so.Print("\nUS Size:     %8X", t->userStackSize());
386      so.Print("\nUS Usage:    %8X (%d%%)",
387               stackUsed, (stackUsed*100)/t->userStackSize());
388   }
389   //-----------------------------------------------------------------
```

```
390  void Monitor::showTasks()
391  {
392  const Task * t = Task::Current();
393  SerialOut so(channel);
394
395     so.Print(
396        "\n--------------------------------------------------");
397     so.Print(
398        "\n    TCB      Status Pri TaskName        ID  US Usage");
399     so.Print(
400        "\n--------------------------------------------------");
401     for (;;)
402         {
403           if (t == Task::Current())   showTask(so, t, "-->");
404           else                        showTask(so, t, "   ");
405
406           t = t->Next();
407           if (t == Task::Current())   break;
408         }
409     so.Print(
410        "\n==================================================\n");
411  }
412  //-----------------------------------------------------------------
413  void Monitor::showTask(SerialOut & so, const Task * t,
414                         const char * prefix)
415  {
416  const char * const stat = showTaskStatus(t);
417  int i;
418
419     so.Print("\n%s %8X ", prefix, t);
420     if (stat)   so.Print("%s", stat);
421     else        so.Print("%4X     ", t->Status());
422     so.Print("%3d ", t->Priority());
423     so.Print("%16s", t->Name());
424
425     for (i = 0; i < TASKID_COUNT; i++)
426         if (t == Task::TaskIDs[i])  break;
427
428     if (i < TASKID_COUNT)   so.Print("%2d   ", i);
429     else                    so.Print("---  ");
430
431     so.Print("%8X ", t->userStackUsed());
432  }
433  //-----------------------------------------------------------------
434  const char * const  Monitor::showTaskStatus(const Task * t)
435  {
436     switch(t->Status())
437        {
438          case Task::RUN:        return "RUN    ";
439          case Task::BLKD:       return "BLKD   ";
440          case Task::STARTED:    return "START  ";
441          case Task::TERMINATED: return "TERM   ";
442          case Task::SLEEP:      return "SLEEP  ";
443          case Task::FAILED:     return "FAILED ";
444          default:               return 0;
445        }
```

```
446  }
447  //---------------------------------------------------------------
```

A.21 Makefile

```
# Makefile for gmake
#

# Development environment.
# Replace /CROSS by where you installed the cross-environment
#
CROSS-PREFIX:= /CROSS
AR         := $(CROSS-PREFIX)/bin/m68k-sun-sunos4.1-ar
AS         := $(CROSS-PREFIX)/bin/m68k-sun-sunos4.1-as
LD         := $(CROSS-PREFIX)/bin/m68k-sun-sunos4.1-ld
NM         := $(CROSS-PREFIX)/bin/m68k-sun-sunos4.1-nm
OBJCOPY    := $(CROSS-PREFIX)/bin/m68k-sun-sunos4.1-objcopy
CC         := $(CROSS-PREFIX)/bin/m68k-sun-sunos4.1-gcc
MAKE       := gmake

# Target memory mapping.
#
ROM_BASE:= 0
RAM_BASE:= 20000000

# compiler and linker flags.
#
ASFLAGS    := -mc68020
CCFLAGS    := -mc68020 -O2 -fomit-frame-pointer -fno-exceptions

LDFLAGS    := -i -nostdlib \
               -Ttext $(ROM_BASE) -Tdata $(RAM_BASE) \
              -Xlinker -Map -Xlinker Target.map

# Source files
#
SRC_S      := $(wildcard *.S)
SRC_CC     := $(wildcard *.cc)
SRC        := $(SRC_S) $(SRC_CC)

# Dependency files
#
DEP_CC     := $(SRC_CC:.cc=.d)
DEP_S      := $(SRC_S:.S=.d)
DEP        := $(DEP_CC) $(DEP_S)

# Object files
#
OBJ_S      := $(SRC_S:.S=.o)
OBJ_CC     := $(SRC_CC:.cc=.o)
OBJ        := $(OBJ_S) $(OBJ_CC)

CLEAN      := $(OBJ) $(DEP) libos.a \
              Target Target.bin \
              Target.td Target.text Target.data \
              Target.map Target.sym

# Targets
#
```

```
 55  .PHONY:    all
 56  .PHONY:    clean
 57  .PHONY:    tar
 58
 59  all:       Target Target.sym
 60
 61  clean:
 62             /bin/rm -f $(CLEAN)
 63
 64  tar:       clean
 65  tar:
 66             tar -cvzf ../src.tar *
 67
 68  include    $(DEP)
 69
 70  # Standard Pattern rules...
 71  #
 72  %.o:       %.cc
 73             $(CC) -c $(CCFLAGS) $< -o $@
 74
 75  %.o:       %.S
 76             $(CC) -c $(ASFLAGS) $< -o $@
 77
 78  %.d:       %.cc
 79             $(SHELL) -ec '$(CC) -MM $(CCFLAGS) $< \
 80                  | sed '\''s/$*\.o/$*\.o $@/'\'' > $@'
 81
 82  %.d:       %.S
 83             $(SHELL) -ec '$(CC) -MM $(ASFLAGS) $< \
 84                  | sed '\''s/$*\.o/$*\.o $@/'\'' > $@'
 85
 86  libos.a:$(OBJ)
 87             $(AR) -sr libos.a $?
 88
 89  Target:    Target.bin
 90             $(OBJCOPY)  -I binary -O srec $< $@
 91
 92  Target.text:Target.td
 93             $(OBJCOPY)  -R .data -O binary $< $@
 94
 95  Target.data:Target.td
 96             $(OBJCOPY)  -R .text -O binary $< $@
 97
 98  Target.bin:Target.text Target.data
 99             cat Target.text | skip_aout | cat - Target.data > $@
100
101  Target.sym:Target.td
102             $(NM) -n --demangle $< \
103             | awk '{printf("%s %s\n", $$1, $$3)}' \
104             | grep -v compiled | grep -v "\.o" \
105             | grep -v "_DYNAMIC" | grep -v "^U" > $@
106
107
108  Target.td:crt0.o libos.a libgcc.a
109             $(CC) -o $@ crt0.o  -L. -los -lgcc $(LDFLAGS)
```

A.22 SRcat.cc

```
 1  // SRcat.cc
 2
 3  #include <stdio.h>
 4  #include <stdlib.h>
 5  #include <string.h>
 6  #include <assert.h>
 7
 8  FILE * infile;
 9
10  enum { MAX_REC_SIZE = 256 };
11  enum { AOUT = 0x20 };
12
13  class SRecord
14  {
15  public:
16     SRecord() {};
17
18     int readRecord();
19     void writeRecord(int rtype);
20     enum { ERR_EOF = -1,
21            ERR_BAD_CHAR = -2,
22            ERR_CHECKSUM = -3
23          };
24
25     unsigned int address;
26     unsigned int size;
27     char data[MAX_REC_SIZE];
28  private:
29     int type;
30     int getHeader();
31     int getWord();
32     int getByte();
33     int getNibble();
34     void putByte(unsigned int);
35
36     unsigned char checksum;
37  };
38
39  int load_file(const char * filename);
40  void store_file(unsigned int address, unsigned char * data,
unsigned int size);
41  void store_odd_even(unsigned int odd, unsigned char * data,
unsigned int size);
42  unsigned long compute_crc(unsigned char * data, unsigned int size);
43
44  unsigned char * ROM = 0;
45  const char * prog = 0;
46  int rom_index = 0;
47  int skip = AOUT;
48  int crlf = 0;
49
50  enum { ROMSIZE = 0x00020000 };
51
52  // ---------------------------------------------------------------
```

```
 53  int main(int argc, char * argv[])
 54  {
 55  int exit_code = 0;
 56  const char * argv1 = 0;
 57
 58     prog = argv[0];
 59
 60     if (argc < 2)   exit(-8);
 61     else            argv1 = argv[1];
 62     if (!strcmp(argv1, "aout"))          skip = AOUT;
 63     else if (!strcmp(argv1, "noaout"))   skip = 0;
 64     else                                exit(-9);
 65
 66     ROM = new unsigned char[ROMSIZE];
 67     if (ROM == 0)   exit(-1);
 68
 69     for (int i = 0; i < ROMSIZE; i++)   ROM[i] = 0;
 70
 71     for (int arg = 2; arg < argc; arg++)
 72        {
 73          const char * av = argv[arg];
 74          int address = 0;
 75
 76          if (!strcmp(av, "-dsp_code"))
 77             {
 78               printf("// This file is automatically generated, don't
edit !\n");
 79               if (rom_index == (3*(rom_index/3)))
 80                 printf("enum { dsp_code_bytes = %d, dsp_code_words =
%d };\n",
 81                          rom_index, rom_index/3);
 82               else
 83                  printf("#error \"Byte Count not multiple of 3\"\n");
 84               printf("const char dsp_code[dsp_code_bytes] = {");
 85
 86               for (int i = 0; i < rom_index; i++)
 87                   {
 88                     if (!(i & 15))   printf("\n");
 89                     printf("0x%2.2X,", ROM[i] & 0xFF);
 90                   }
 91               printf("\n                                 };\n\n");
 92             }
 93          else if (!strcmp(av, "-crlf"))
 94             {
 95               crlf = 1;
 96             }
 97          else if (!strcmp(av, "-version"))
 98             {
 99               unsigned long Release  = (ROM[0x100] << 24)
100                                      | (ROM[0x101] << 16)
101                                      | (ROM[0x102] << 8 )
102                                      | (ROM[0x103]      );
103               unsigned long Revision = (ROM[0x104] << 24)
104                                      | (ROM[0x105] << 16)
105                                      | (ROM[0x106] << 8 )
106                                      | (ROM[0x107]      );
```

```
107               fprintf(stderr, "%s: FW Revision   -> %u.%u\n",
108                                prog, Release, Revision);
109            }
110          else if (!strcmp(av, "-crc"))
111            {
112              unsigned long crc = compute_crc(ROM, ROMSIZE-4);
113              fprintf(stderr, "%s: CRC          -> 0x%8.8X\n", prog,
crc);
114              ROM[ROMSIZE-4] = crc>>24;
115              ROM[ROMSIZE-3] = crc>>16;
116              ROM[ROMSIZE-2] = crc>> 8;
117              ROM[ROMSIZE-1] = crc;
118              rom_index = ROMSIZE;
119            }
120          else if (!strcmp(av, "-even"))
121            {
122              store_odd_even(0, ROM, rom_index);
123            }
124          else if (!strcmp(av, "-odd"))
125            {
126              store_odd_even(1, ROM, rom_index);
127            }
128          else if (!strncmp(av, "0x", 2))
129            {
130              if (sscanf(av, "%X", &address) == 1)
131                 {
132                   fprintf(stderr, "%s: Storing       -> 0x%8.8X\n",
133                           prog, address);
134                   store_file(address, ROM, rom_index);
135                 }
136              else
137                 exit_code = -2;
138              if (exit_code)   break;
139            }
140          else   // file name
141            {
142              fprintf(stderr, "%s: Loading         %s:\n", prog, av);
143              exit_code = load_file(av);
144              if (exit_code)   break;
145            }
146        }
147
148     delete ROM;   ROM = 0;
149     exit(exit_code);
150  }
151
152  int load_file(const char * filename)
153  {
154  SRecord srec;
155  int mini = -1;
156  int maxi = -1;
157  int record = 0;
158  int exit_code = 0;
159  int initial_skip = skip;
160
161     infile = fopen(filename, "r");
```

```
162     if (infile == 0)   return exit_code = -3;
163
164     for (;;)
165         {
166           int res = srec.readRecord();
167           record++;
168
169           switch(res)
170              {
171                case 0:
172                     fprintf(stderr, "%s: S0 %s\n", prog, srec.data);
173                     continue;
174
175                case 1:
176                case 2:
177                case 3:
178                     {
179                       if (mini == -1)     // first data record
180                          {
181                            mini = srec.address;
182                            fprintf(stderr, "%s: S%d 0x%8.8X ->
0x%8.8X\n",
183                                    prog, res, mini, rom_index);
184                          }
185                       else if (res != 1 && srec.address != maxi)
186                          {
187                            fprintf(stderr,
188                                    "%s: Record %d: Gap/Overlap at
0x%8.8X\n",
189                                    prog, record, srec.address);
190                           exit_code =  -7;
191                           break;
192                          }
193
194                       maxi = srec.address + srec.size;
195
196                       for (int i = 0; i < srec.size; i++)
197                           {
198                             if (skip)
199                                skip--;
200                             else if (rom_index <= ROMSIZE)
201                                ROM[rom_index++] = srec.data[i];
202                             else
203                                {
204                                fprintf(stderr, "%s: S%d above ROM\n",
205                                          prog, res);
206                                  exit_code = -5;
207                                  break;
208                                }
209                           }
210                     }
211                     continue;
212
213                case 7:
214                case 8:
215                case 9:
```

```
216                       fprintf(stderr, "%s: S%d 0x%8.8X -> 0x%8.8X\n",
217                                       prog, res, maxi, rom_index);
218                       break;
219
220                 default:
221                       fprintf(stderr, "%s: Bad Record S%d\n", prog,
res);
222                       exit_code = -5;
223                       break;
224                }
225             break;
226           }
227
228       fclose(infile);
229       fprintf(stderr, "%s: Size             0x%8.8X\n",
230               prog, maxi-mini-initial_skip);
231       return exit_code;
232    }
233    // --------------------------------------------------------------
234    void store_file(unsigned int addr, unsigned char * data, unsigned
int size)
235    {
236    SRecord srec;
237    char name[20];
238    int i, sl, dr, er;
239
240       sprintf(name, "Image_0x%8.8X", addr);
241       sl = strlen(name);
242
243       // write S0 record
244       srec.address = 0;
245       for (i = 0; i < sl; i++)   srec.data[i] = name[i];
246       srec.size = sl;
247       srec.writeRecord(0);
248
249       if ((addr+size) <= 0x01000000)   { dr = 2;   er = 8; }   // S2/S8
250       else                             { dr = 3;   er = 7; }   // S3/S7
251
252       // write S2/S3 records
253       for (int idx = 0; idx < size; idx += 32)
254           {
255             srec.address = addr+idx;
256             srec.size = 0;
257             for (i = 0; i < 32; i++)
258                 {
259                   if ((idx+i) >= size)   break;
260                   srec.data[i] = data[idx+i];
261                   srec.size++;
262                 }
263             srec.writeRecord(dr);
264           }
265
266       // write S8/S7 records
267       srec.address = 0;
268       srec.size = 0;
269       srec.writeRecord(er);
```

```
270  }
271  // -------------------------------------------------------------
272  void store_odd_even(unsigned int odd, unsigned char * data,
unsigned int size)
273  {
274  unsigned int addr;
275  SRecord srec;
276  char * name;
277  int i, sl;
278
279     if (odd)
280        {
281          name = "EEPROM.ODD";
282          addr = 1;
283        }
284     else
285        {
286          name = "EEPROM.EVE";
287          addr = 0;
288        }
289
290     sl = strlen(name);
291
292     // write S0 record
293     srec.address = 0;
294     for (i = 0; i < sl; i++)   srec.data[i] = name[i];
295     srec.size = sl;
296     srec.writeRecord(0);
297
298     // write S2/S3 records
299     for (int idx = 0; idx < size; idx += 32)
300         {
301           srec.address = idx>>1;
302           srec.size = 0;
303           for (i = addr; i < 32; i+=2)
304               {
305                 if ((idx+i) >= size)   break;
306                 srec.data[i>>1] = data[idx+i];
307                 srec.size++;
308               }
309           srec.writeRecord(1);
310         }
311
312     // write S9 records
313     srec.address = 0;
314     srec.size = 0;
315     srec.writeRecord(9);
316  }
317  // -------------------------------------------------------------
318  void SRecord::writeRecord(int rtype)
319  {
320  int i;
321  const char * CRLF = "\n";
322
323     if (crlf)   CRLF = "\r\n";
324
```

```
325     checksum = 0;
326     switch(type = rtype)
327        {
328          case 0:  printf("S0");
329                   putByte(size+3);
330                   putByte(address>>8);
331                   putByte(address);
332                   for (i = 0; i < size; i++)
333                       putByte(data[i]);
334                   checksum = ~checksum;
335                   putByte(checksum);
336                   printf(CRLF);
337                   return;
338
339          case 1:  printf("S1");
340                   putByte(size+3);
341                   putByte(address>>8);
342                   putByte(address);
343                   for (i = 0; i < size; i++)
344                       putByte(data[i]);
345                   checksum = ~checksum;
346                   putByte(checksum);
347                   printf(CRLF);
348                   return;
349
350          case 2:  printf("S2");
351                   putByte(size+4);
352                   putByte(address>>16);
353                   putByte(address>>8);
354                   putByte(address);
355                   for (i = 0; i < size; i++)
356                       putByte(data[i]);
357                   checksum = ~checksum;
358                   putByte(checksum);
359                   printf(CRLF);
360                   return;
361
362          case 3:  printf("S3");
363                   putByte(size+5);
364                   putByte(address>>24);
365                   putByte(address>>16);
366                   putByte(address>>8);
367                   putByte(address);
368                   for (i = 0; i < size; i++)
369                       putByte(data[i]);
370                   checksum = ~checksum;
371                   putByte(checksum);
372                   printf(CRLF);
373                   return;
374
375          case 7:
376                   printf("S7");
377                   putByte(size+5);
378                   putByte(address>>24);
379                   putByte(address>>16);
380                   putByte(address>>8);
```

```
381                     putByte(address);
382                     for (i = 0; i < size; i++)
383                         putByte(data[i]);
384                     checksum = ~checksum;
385                     putByte(checksum);
386                     printf(CRLF);
387                     return;
388             case 8:
389                     printf("S8");
390                     putByte(size+4);
391                     putByte(address>>16);
392                     putByte(address>>8);
393                     putByte(address);
394                     for (i = 0; i < size; i++)
395                         putByte(data[i]);
396                     checksum = ~checksum;
397                     putByte(checksum);
398                     printf(CRLF);
399                     return;
400             case 9:
401                     printf("S9");
402                     putByte(size+3);
403                     putByte(address>>8);
404                     putByte(address);
405                     for (i = 0; i < size; i++)
406                         putByte(data[i]);
407                     checksum = ~checksum;
408                     putByte(checksum);
409                     printf(CRLF);
410                     return;
411           }
412   }
413   // ------------------------------------------------------------------
414   void SRecord::putByte(unsigned int val)
415   {
416      printf("%2.2X", val & 0xFF);
417      checksum += val;
418   }
419   // ------------------------------------------------------------------
420   int SRecord::readRecord()
421   {
422   int dat, w, total;
423
424      getHeader();
425      checksum = 1;
426      total = getByte();   if (total < 0)   return total;
427      switch(type)
428         {
429           case 0:  address = getWord();   if (address < 0)   return
address;
430                    total -= 2;
431                    break;
432
433           case 1:
434           case 9:  address = getWord();   if (address < 0)   return
address;
```

```
435                   total -= 2;
436                   break;
437
438           case 2:
439           case 8:  w = getByte();         if (w < 0)         return w;
440                   address = getWord();   if (address < 0)   return
address;
441                   address += w << 16;
442                   total -= 3;
443                   break;
444
445           case 3:
446           case 7:  w = getWord();         if (w < 0)         return w;
447                   address = getWord();   if (address < 0)   return
address;
448                   address += w << 16;
449                   total -= 4;
450                   break;
451
452           default: return ERR_BAD_CHAR;   // error
453         }
454
455     size = total-1;  // 1 checksum
456
457     for (int i = 0; i < total; i++)
458         { data[i] = dat = getByte(); if (dat < 0)   return dat; }
459     data[size] = 0;   // terminator if used as string, e.g. for S0
records
460
461     if (checksum)   return ERR_CHECKSUM;
462
463     return type;
464  }
465  // ------------------------------------------------------------
466  int SRecord::getHeader()
467  {
468  int c;
469
470     for (;;)
471         {
472           c = fgetc(infile);
473           if (c == 'S')   break;
474           if (c == EOF)   return type = ERR_EOF;
475           if (c <= ' ')   continue;   // whitespace
476           return type = ERR_BAD_CHAR;
477         }
478
479     // here we got an 'S'...
480     switch(c = fgetc(infile))
481         {
482           case '0':
483           case '1': case '2': case '3':
484           case '7': case '8': case '9':
485                     return type =  c - '0';
486
```

```
487           default:  fprintf(stderr, "\ngetHeader: not 0, 1-3 or 7-9
[%d]", c);
488                     return type = ERR_BAD_CHAR;
489         }
490  }
491  // ------------------------------------------------------------
492  int SRecord::getWord()
493  {
494  int b, w;
495
496     b = getByte();   if (b < 0)   return b;
497     w = getByte();   if (w < 0)   return w;
498     return (b<<8) + w;
499  }
500
501  // ------------------------------------------------------------
502  int SRecord::getByte()
503  {
504  int n, b;
505
506     n = getNibble();   if (n < 0)   return n;
507     b = getNibble();   if (b < 0)   return b;
508     b += n<<4;
509     checksum += b;
510     return b;
511  }
512
513  // ------------------------------------------------------------
514  int SRecord::getNibble()
515  {
516  int c;
517
518     for (;;)
519         {
520           c = fgetc(infile);
521           if (c == EOF)   return ERR_EOF;
522           if (c >  ' ')   break;
523         }
524
525     c &= 0x7F;   // strip parity
526     if (c <  '0')   return ERR_BAD_CHAR;
527     if (c <= '9')   return c - '0';
528     if (c <  'A')   return ERR_BAD_CHAR;
529     if (c <= 'F')   return c + 10 - 'A';
530     if (c <  'a')   return ERR_BAD_CHAR;
531     if (c <= 'f')   return c + 10 - 'a';
532     return ERR_BAD_CHAR;
533  }
534
535  // ------------------------------------------------------------
536  unsigned long compute_crc(unsigned char * ROM, unsigned int size)
537  {
538  unsigned long D5 = 0x00A00805;   // CRC-32 polynomial
539  unsigned long D1 = 0xFFFFFFFF;   // preset CRC value to all ones
540  unsigned long D2;                // data
541  unsigned long D3;                // temp data
```

```
542  unsigned long D4;               // bit counter
543
544     for (unsigned int D0 = 0; D0 < size; D0 += 4)   // long loop
545         {
546           D2 = (ROM[D0]   << 24) & 0xFF000000
547              | (ROM[D0+1] << 16) & 0x00FF0000
548              | (ROM[D0+2] <<  8) & 0x0000FF00
549              | (ROM[D0+3]      ) & 0x000000FF;
550
551           for (D4 = 0; D4 < 32; D4++)               // bit loop
552               {
553                 D3 = D1 ^ D2;
554                 D1 += D1;
555                 D2 += D2;
556                 if (D3 & 0x80000000)   D1 ^= D5;
557               }
558         }
559     return D1;
560  }
561  // ------------------------------------------------------------------
```

Index

W