Realtime Operating Systems

Concepts and Implementation of Microkernels

for Embedded Systems

Dr. Jürgen Sauermann, Melanie Thelen

Preface

Every year, millions of microprocessor and microcontroller chips are sold as

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.

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.

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.

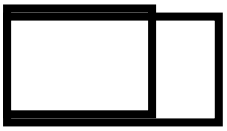
2.1.1 Compiling and Linking

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

```
#include <stdio.h>
```

preprocessing. The purpose of **stdio.h** is to tell the compiler that **printf** is not a

FIGURE 2.2 libc.a Structure



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

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

There is an important invariant: **Whenever a task examines the variable State, it will find this variable set to RUN**. **State**

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). **P()** requires O(N) if	

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

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 on1.1667 TD (PutIndex)Tj 742 TD (GetIndex)Tj /F4 1 Tf 4.4

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**

For each

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

- **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
```

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**

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:

3.4 Semaphores

Semaphores are declared in file **Semaphore.hh**

...

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

3.7 Serial Input and Output

The basic model for serial input and output has already been discussed in Section

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

```
            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

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

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

```
  1 .86 Prin (SerialOu.cc */c)]TJ 0 -1.2222 TD (...)TjT*m [1321 .86 Prin (SerialOu::(PrintCch
```

```
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

46T_CTLfT*     f  init_level== Interrupt_I;
47T_CTLfT*     }R
138
```

that all interrupts are accepted. The **main()**

```
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 variable **_consider_ts** may or may not have been set during the interrupt

3.9 Memory Management

As we will see in Section 6.4, a library **libgcc2**

```
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 667 TD igrruc*)-1(3to.3,)-241(our)]TJ-14/F40f

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

Thus after RESET, processing continues at label

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()**

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 actuallys-2l,48code,-2l,48buo ory task
Ffunction

task. Finally, use of the **delete** operator requires use of the **malloc** package, in

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.

5.3 A Monitor Session

The commands of the monitor are best understood by looking at a commented

m4237T 41.4227m

```
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.

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;`

-

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 are to become

CROSS

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

```
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)
```

```
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

thatseg

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**

7 Miscellaneous

7.1 General

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

```
...
 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.

irouingeoa0(rs]3762Mcheck10(o)d]3762Mwheherj-2761ia]3762Mchange]3762Mback12761io)-2762Mseou

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

```
183           MOVE.W  (SP)+, D1             | next output char (valid if D0 = 0)
184           TST.L   D0                    | char valid ?
185           BEQ     Ld1i21                | yes
```

```
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 D0     | get wa get4 resources
```

A.2 Task.hh

```
1   #ifdef ASSEMBLER
2
3   #define TaskNext
```

A.3 Task.cc

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;
}

//------------------------------------------------------------------------
```

```
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.8 Queue.cc

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;
```

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)
```

```
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
```

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

```
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.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      )
```

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
```

```
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 '?':
14-1349(                unsigned lon signed lon si 'h': case 'H': casd(ed lon signe]TJ T* (1
139   4             }
139   4       case 't':  T* [(127)-1349( 4           {)]TJ T* []TJ T* (117)Tj T* [5038   5
6            unsigned lif swnnel))
```

```
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);


  so.Print%3d  ,t = so.oritytut( );
so.Print%16s ,t = askNtut( );394     foritat0;s 1< TASKID_COUNT;s ++fix)
```

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

```
107             fprintf(stderr, "%s: FW Revision49(598.5> %u.%u\n",)]TJ (1-1.2222 TD07)-1
```

```
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
```