Real-time excutive for 8051-type single-chip microcomputers.

Author : Øyvind Teig.
Date : 26.09.88
Address : Autronica A/S
: P.O.Box 3010, N 7001
: Trondheim, Norway
: Telephone: 47 7 918080

Abstract.

The paper describes a real-time executive (rx or scheduler) designed for the 8051- or 8052- type microcomputers. Up to 8 concurrent processes (or tasks) may be defined.

A synchronizing primitive consisting of a protected mailbox is defined.

Shared resources may be protected, this is done with an implicit concept. The concept is the scheduling algorithm that does not reschedule any process when a subroutine is running. If the shared resource is accessed at subroutine-level only, it is by definition protected. This scheme removes the necessesity of semaphores. Buffers or IO or subrotines may be protected just be moving to subroutine-level.

Without making reentrant code, all subroutines still will behave as if they were reentrant because of the scheduling algorithm. Reentrant in this context means that the programmer may call the same procedures from more than one process, not that the procedure will be able to call itself. Passing of parameters to such routines may be done from subrotine level, so that the parameters are properly protected.

A timeout- or wait- mechanism is implemented. The wait-mechanism may be of type "reschedule after nnn mS", or "reschedule if not mail or interrupt received within nnn mS".

The scheduler may run on a single-chip 8051 with 128 bytes internal ram or an 8052- compatible microcomputer with 256 bytes internal ram. It does not use external ram. The scheduler and its associated routines have a code size of approximately 1400 bytes.

The scheduler is written in the Intel PL/M-51 language, and thus may be recompiled at any time by the user. The PL/M-51 syntax is used to some extent in the paper. The figures given in the text is from a system realized with an 8052- compatible single-chip microcomputer running at 12 MHz with 4 processes. The system neither has external data storage nor external program storage. With 4 processes defined, the rx uses approximately 75 bytes internal ram. The system easily handles one serial channel at 9600 baud and a system timer interrupt every 2 mS. The system still is idle for 50% of the time. The scheduler switching time is approximately 250 microseconds with a 12 MHz crystal.

The basic philosophy of the scheduler has been tested for several years with a similar scheduler designed for the Intel 8035 microcomputer, now running in some 1300 delivered systems.

Scheduler procedures and functions.

The user has to learn how to use a small number of procedure- and function- calls. The procedure calls are:

```
call wait      returns to calling process.
call rx        returns to calling process after being rescheduled.
```

The function calls are the following:

```
Mailbox_reserved      returns to calling process.
Mailbox_filled        returns to calling process.
Mailbox_emptied       returns to calling process.
```

Prior to all the 5 calls above, the global interrupt enable flag must be turned off. This is done with the "disable" statement.

The global interrupt enable flag is tested within all the functions described above and also within the "wait" procedure. A crash exception is done if the flag is enabled at those points.

When a process is rescheduled at the line following the "call rx" statement, the global interrupt enable flag is always true, turning the global interrupt enable on. The rx scheduler is the only procedure or function that modifies this flag.

Initializing phase.

The real-time executive starts all the defined processes using the wait-mechanism. The wait-mechanism uses one 8-bit software counter for each process. In the initializing phase each wait-counter is set to any non-zero value, but at least one must be 1 to count to zero on the first timer interrupt.

The (starting) address of each process is referenced as EXTERNAL to the scheduler, which uses the starting address when scheduling a process. Later on, when a process has been running at least once, it will be rescheduled (when necessary) at the statement after the last running instruction.

Before any process is allowed to run, the system timer is initialized to count to overflow in 2.00 mS. This is done just after power-up and the interrupts are not yet enabled. Then the interrupt priority register is set to give all interrupts "low" priority, meaning that the interrupts are prioritized in descending order with external interrupt 0 to be the highest and timer 0 the second highest priority. When the scheduler initializing phase is over, the timer interrupt and the global interrupt enable is turned on.

After the scheduler initializing phase it enters a loop where it waits for a timer 0 interrupt. If the interrupt is not received within the specified time, the loop will terminate when the loop-counter has reached a predefined limit, and a call to the exception handler is done.

Exception handler.

An exception handler must be defined by the user. It has a one byte input parameter. The scheduler calls the exception handler whenever an error or inconsistency is found. All exceptions from the scheduler are crash-exceptions, meaning that a new restart will be done from a watch-dog circuit. This is very dramatic, and is allowed to be so because those errors will show up "at once" in a real system. Besides: to make the system live through such an exception would require a much more complicated scheduler exception handler. It should not only be able to recover, but do it correctly!

The error messages are listed at the end of this paper.

Some specific error situations.

The scheduler does not allow a single process to run for more than 512 mS. This will cause a crash-exception. This is a useful feature to track eternal loops in a process.

Likewise, a crash-exception will be the result if the processor does not enter the idle state at least once within a 512 mS period. This defines an overloaded system that obviously needs a rethinking of the design of one or more processes.

The first system timer interrupt.

When06the first timer 0 interrupt is received after 2 mS, the timer 0 interrupt procedure decrements all non-zero wait counters. As already mentioned, at this first timer 0 interrupt at least one counter must hit zero. This will cause a call to the scheduler with a boolean parameter telling that it is called because of an interrupt. Inside the scheduler the return address to the interrupt procedure is removed, and the stack pointer is decremented by two, to modify the "call" to a "goto".

The scheduler now will store (on the stack) the code address of the first statement of the process to be run. The stack pointer is increased by two to make it possible to "return to" the address just stored on the stack. Since the scheduler was called from an interrupt procedure this time, the two last instructions within the scheduler is 1.) to enable the global interrupt flag and 2.) the "reti" (return from interrupt) instruction. After this, the first running process is running its first statement.

The "reti" instruction is in fact run from an assembly code procedure which is called by the scheduler. It is called "interrupt_return" and it decrements the stack pointer by two, enables the global interrupt flag and then the "reti" instruction is run. As within the scheduler, this technique is used to modify a "call" to a "goto". PL/M-51 does not allow a goto to an external label when the goto is not on the lowest stack-level within the compiled file.

The anatomy of a process.

A process often consists of an initializing part and an eternal loop.

```
prc0__:
DO;
  process_0: PROCEDURE PUBLIC;
  DECLARE <process_0 variables>;

    subroutine: PROCEDURE;
    DECLARE <subroutine variables>;
      <subroutine (stack) level>
    END subroutine;

    <process initializing part>
    <process (stack) level>
    DO WHILE 1=1;
      <process eternal loop part>
      <process (stack) level>
    END;
  END process_0;
  <system level, no code in processes>
END prc0__;
```

Later, "FOREVER" is used for "WHILE 1=1". The only file in the system which has system level code, is the scheduler itself. Thus the linked system will start at that code part at power-up

Call to the scheduler (rx).

Usually the initializing part is run without calls to the scheduler. However, because of the scheduling philosophy all processes must contain at least one call to the scheduler. A call to the scheduler will look like this:

```
disable;
call rx (next);
```

The parameter to the rx is a boolean having the value "next" or "interrupt_received". "Next" is used when called from process level, "interrupt_received" is used when called from an interrupt procedure.

In the example above the process suspends itself. The process will stay in the suspended state until an interrupt or mail in a mailbox causes it to be rescheduled.

If the process scans a keyboard every 10 mS, it needs to be rescheduled accordingly. The construct will look like this:

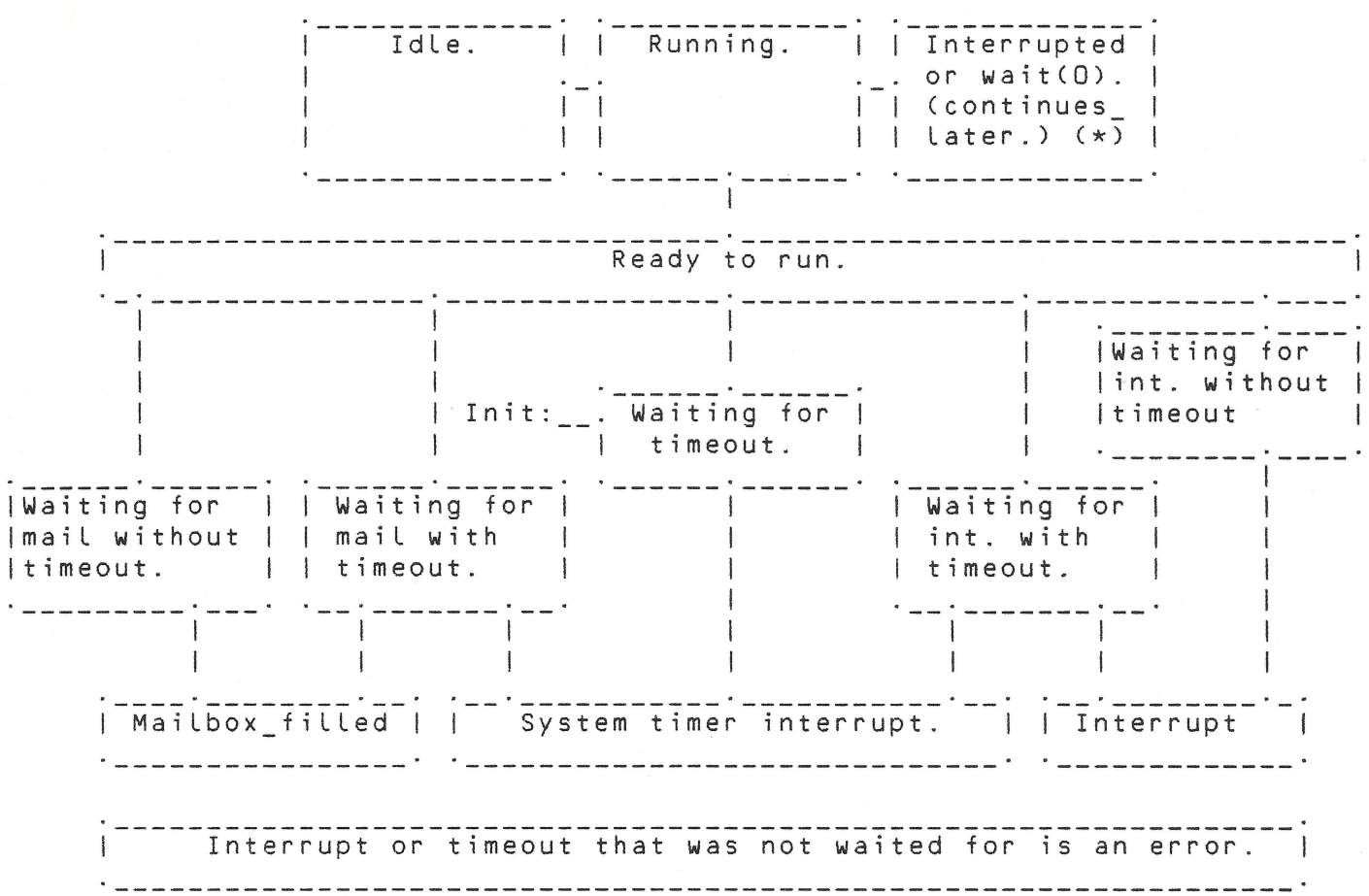
```
disable;
call wait (10);
call rx (next);
```

The process will be rescheduled some time after 10 mS, or it may be rescheduled earlier if an interrupt or mail in a mailbox made it necessary. If the process needs to know if it was rescheduled because of a time-out, it must check a boolean variable called "time_bit". This variable is actually located within the process "psw" (program status word). Thus every process has its own "time_bit" to check.

With the scheduling scheme described later, the rx can be called from process level or interrupt level only. If the RX is called from a subroutine that has been called from interrupt level or process level, then an exception-crash is encountered.

Process states.

The diagram below shows the different states a process may posess.



(*) in this case "interrupted" means that the process has been running at process level (not subroutine level), and another process has been rescheduled, leaving the "interrupted" process unaware of the fact that it was suspended. It must, of course, continue later.

Accuracy of system timer and "call wait".

If a system timer interrupt is lost, a crash-exception is encountered. This is done by making the system timer overflow cause the interrupt flag to be set, and letting the system timer continue counting. When the interrupt is served, the present value of the system timer is checked. A too far counted value indicates that a system timer interrupt has been lost. Actually, the interrupt has not been lost **yet**, but it is already too late not to lose it if the interrupt is enabled again. This is considered a crash-exception since it means that the correct absolute time measure has been lost. If, however, no timer interrupts has been lost, the timer is preset to a value that will make the next overflow appear **exactly** 2 mS after the previous timer overflow.

Since the system timer interrupt is run exactly 500 times every second, it may be used to count external variables that do not belong to the scheduler. Thus, an "absolute time" measure is available. However, the scheduler does not use an "absolute time" reference.

The "call wait (nnn mS)" will cause a process to be rescheduled not at nnn mS sharp, but at some probabilistic time after the nnn mS has passed. Therefore the "call wait (nnn mS)" can be used only as a coarse time measure. (nnn maximum is 255 mS).

If an "immediate" rescheduling is wanted, the "call wait (0)" construct may be used. This will cause all the other processes to be run (if necessary) before the process that contains the "call wait (0)" is again rescheduled. The construct may also be considered as a "signal to itself".

Scheduling policy.

Some details about the scheduling scheme have already been disclosed. Since the scheduler can handle up to 8 concurrent processes, the scheduler uses some 8-bit variables to determine which process to run. The scheduler checks and modifies these variables:

1. process_int_received.
2. process_timeout.
3. process_mail_present.
4. process_continues_later.

If no process needs to be rescheduled, the "idle" state is entered.

If an interrupt other than the timeout is signalled to a process, (process_int_received), it will be rescheduled if its process number is less than the process number of the running process. With 4 active processes they are numbered from 0 to 3, with process 0 given the highest priority. In all other scheduling cases, a round-robin scheduling policy is used.

However, a very important assumption has been made. A running process always continues to run if it has called a subroutine, no other process will be rescheduled. (The called procedure is named a subroutine here to emphasize that it is a user program called from process stack level, or from another subroutine stack level).

The assumption, then, is that the running process is making a call to the scheduler within a time frame that will give acceptable system response time. With the system described this has been no problem. The processor is running at 12 MHz (maximum is 16 MHz), and even with the code written in high-level language, the speed penalty involved is quite acceptable.

This scheduling scheme prohibits the implementation of a process time-slice mechanism. This limitation has not proved to be significant to the system behaviour.

With the scheme described, the process scheduler did not turn out to be excessively complex.

Pseudo reentrant subroutines.

The consequence of the scheduling philosophy is that all subroutines called from process level will look reentrant, even if they actually are not. In other words, a subroutine that is called from one process may be called from any other process as well. The only drawback to this is the fact that input and output parameters to the subroutine must be protected. This is easily done by calling the subroutine from another subroutine that handles the parameters. Observe that interrupts need not be disabled during any phase.

Shared ram for pseudo reentrant subroutines.

An advantage with the pseudo reentrant subroutines is that their local variables may be overlapped. When subroutines within the same file are linked (by the Intel RL linker) and their variables are assigned physical addresses, the variables will be overlapped provided the subroutines do not call each other. With the pseudo reentrant subroutines this may be done also with variables within subroutines that are separately compiled. The disadvantages are that this must be done by hand, and that it decreases some of the high-levelness experienced without it. However, it may save ram, and may actually also increase the high-levelness by making the user able to build the system out of more and thus smaller program files. The same restrictions apply to this as the compiler/linker must obey: special care must be taken if the subroutines call each other.

In the system described an array of 10 bytes is used for these local variables. The array is declared public in one file, external in the subroutine file. Then, the local variables are declared as <variable name> at (.share_ram (n)), where share_ram is the public shared ram.

Stack need.

PL/M-51 uses the stack strictly to push and pop return addresses during a call to and return from a subroutine. The architecture of the 8051 processors defines the stack to be on-chip. It is addressed relative to the stack pointer, but it resides within the 128- or 256- bytes on-chip ram. (The same ram may be accessed relative to the R0- or R1- registers as well). Within a real-time system a scheduler usually takes care of the administration of the stack, it takes care that each process has its own stack. This administration is not necessary with the scheme described in this paper. A process is rescheduled only when the stack pointer is "at the bottom". It is not the sum of the processes stack area that is the total amount of stack needed. Thus the necessary ram area used for stack is equal to the need of only one of the processes, namely the process with the maximum need.

Semaphores not necessary to protect buffers.

In a real-time system, it is necessary that only one process at a time is granted control of a shared resource. This is usually done with a semaphore or a specific freepool-semaphore. The processes have to reserve the semaphore, then use the buffer, then free the semaphore again. With the scheduling scheme described in this paper, the protection mechanism is implicit.

If the buffer that needs to be protected is **modified at subroutine level**, then it is protected, since the scheduler in this case will not reschedule any other process, but handle interrupts only.

Process synchronization: the mailbox primitive.

The scheduler administrates two arrays of bytes:

```
mailbox (no_of_mailboxes) byte idata,
mailbox_owner (no_of_mailboxes) byte idata;
```

A mailbox may be owned by a process, then the mailbox_owner is equal to the process number of the owner. Or the mailbox may be empty, then the mailbox_owner is the successor of the last process number (= 4 when 4 processes, since the processes are numbered 0-3).

The process that is waiting for a mailbox must use these two function calls:

```
Mailbox_reserved
Mailbox_emptied
```

The process that sends information to the waiting process must use this function call:

```
Mailbox_filled
```

Actually, the sender does not send the mail to a known process, but to a mailbox. The sender does not know which process is waiting for mail in the mailbox.

The scheduler does administrate the mailbox synchronization, but not when a process may get a mailbox reserved. It uses a "self-administration" scheme where a process spins around waiting for the resource it needs until the resource is received.

This spinning around may be realized with a loop with no "call wait (nnn); call rx (next)" inside the loop. This will lock out lower prioritized processes. Therefore it is not a good solution. (The scheduler will issue a crash exception after 512 mS anyhow: error_singe_process_running or error_idle_never_running).

The process should instead do a sampling of the resource within the context of the scheduler: that is, insert a "call wait .." within the loop. This is shown in all the examples in this paper. Remember that the time interval is rather fast: 2 mS in the realized system. It is not the usual 50 mS counter realized in other real-time schedulers, in such a case the solution would have been prohibitive.

```
Process n: this is the process that is being signalled to:
DO FOREVER
   disable;
   Performs a wait for the mailbox:
   DO WHILE not Mailbox_reserved (box_0);
      call wait (2);
      call rx (next);
      disable;
   END;
   call wait (timeout_counter) (optional)
   call rx (next);
   Sleep here until wakeup.
   .
   wakeup done.
   .
   disable;
   IF Mailbox_emptied (box_0, .mail) THEN
      mail received because another process has sent mail via the
      "Mailbox filled" function.
   ELSE
   DO;
      IF time_bit = timeout THEN
         timeout because no other process has sent mail within
         the legal time interval (=timeout counter millisecond)
      ELSE
         interrupt received. It is actually not necessary to wait
         for an interrupt via a mailbox.
   END;
   time_bit = false;

   Process the mail.
   During this phase the mail named box 0 is empty and any other
   process may signal to it.
END;
```

The scheduler checks the previously mentioned process_mail_received 8 bit value, and reschedules the waiting process whenever needed.

This may look "messy", but if used uncorrectly, usually some lock occurs that causes a crash exception. So it has not proven to be a "low-quality" construct.

The sending process may look like this:

```
Process m (any process except process n). The process is signalling.
DO FOREVER

   Process data and make it ready to be mail.

   disable;
   Performs a filling of the mailbox:
   DO WHILE not Mailbox_filled (box_0, mail);
      call wait (2);
      call rx (next);
      disable;
   END;
   enable;

   The mailbox has been filled.
   Continue any other processing.
END;
```

The "disable" statement turns off the interrupts during the "critical region" phase. This will prevent dead-lock from occuring if two processes enter the critical region. The first process to run the "disable" will be the one to get any unused resource.

The "mail" is declared as a byte that is public within the scheduler. Any other mail type may be defined. However, in that case the 3 "Mailbox.."- functions must be slightly modified.

Use of registers and register banks.

All the processes and subroutines will use the register bank 0, in addresses 0 to 7. No other register banks are used. The registers within a register bank are denoted R0 - R7. The scheduler stores R0, R1, R6 and R7 for every process. Use of R2 - R5 at process level is considered a critical region that must be protected by a "disable - enable" section. This is necessary since these registers are not stored when changing running process. However, since processes are not rescheduled at subroutine level, all the registers R0 - R7 may be used at that level.

The reason for not storing R2 - R5 is that the PL/M-51 compiled code seldom uses them. To make sure, the "$ code" compiler directive may be used. Then text-editor searches for "R4" etc. may be done. This should eventually be done on all the files in the system, to acquire as high a quality as possible.

Exclusive process environment.

Each running process must have an environment that is exclusive to that particular process every time it is running. The following environment is exclusive to each process:

1. b-register, 1 byte
2. accumulator, 1 byte
3. program status word or psw, 1 byte
4. R0, R1, R6, R7, 4 bytes
5. data pointer high and low, 2 bytes
6. program counter high and low (pch, pcl) 2 bytes

This totals to 11 bytes. This may be changed if another definition of a critical region is done or another scheduling philosophy is chosen. The scheme shown is chosen as something between the ideal which is to give each process all the above plus R2 - R5, and the existentially minimum which is the program counter high and low.

Interrupts.

The PL/M- facility "interrupt n using register bank number" may be used, since interrupts are not routed directly to the scheduler. However, if one wishes to use the register bank 0, a scheme is provided within the scheduler file to show how it may be done. Here is an example of how an interrupt serving the serial channel is done:

```
interrupt_serial_channel: PROCEDURE PUBLIC;

   copy_of_R0_in_process        = process_and_rx_R0;
   copy_of_acc_in_process       = acc;
   copy_of_psw_in_process       = psw;

   IF Interrupt_function = signal_to_process THEN
   DO;
      int_to_process = int_to_process0;
      int_mask = int_to_process0_mask;
      es = 0; /* process0 may not immediately run */
      call_no_return rx (input_rx_int_received);
   END;
   ELSE
   DO;
      process_and_rx_R0 = copy_of_R0_in_process;
      acc = copy_of_acc_in_process;
      psw = copy_of_psw_in_process;
      call_no_return interrupt_return;
   END;

END interrupt_serial_channel;
```

The Interrupt_function is a function call to the actual interrupt procedure. If the function value returned is equal to signal_to_process then the serial channel process (process_0) is "signalled". If not, a return from interrupt is done immediately. "Call_no_return" simply means "call".

Whenever the scheduler reschedules a process waiting for an interrupt it manipulates the processor interrupt enable register **ie**. Each process has its own mask that is used to zero the necessary bits in the **ie**-register. In other words: the interrupt level that causes the process to be rescheduled will have its interrupt enable flag disabled while the process is running. In the example above the "es" flag is set to zero, so the scheduler does not need to reset it to zero a second time. However, if the interrupt was waited for with a timeout, it is quite necessary to disable the corresponding interrupt, since interrupt to running is considered a crash exception.

The interrupt priority register **ip** is set to all zeros. This implies the following scheme: whenever two interrupts are pending at the same time, the external interrupt 0 has the highest priority, and no interrupt routine can interrupt another.

The interrupt priority register **ip** may also have non-zero bits. See below in the "Assembly file and interrupts" chapter.

Assembly file.

With this scheme, the interrupt vectors are provided within a small assembly-code file. This file contains the interrupt vectors with jumps to the defined interrupt procedures. In the example above, at the address 23H is a jump to "interrupt_serial_channel" .

The assembly code file also contains the earlier mentioned "interrupt_return" procedure.

Assembly file and interrupts.

When one interrupt may interrupt another interrupt, it is called "multilevel interrupt". Within the context of the RX, multilevel interrupts are not allowed.

However, the RX allows different bits in the ip-register to be set to 1. In the file assembly-file, the first instruction of every interrupt routine is "disable" or clr 0A8H.7. This allows for different priority levels, but still multilevel interrupts are avoided.

The idle program and the watchdog.

Whenever all processes are emptied, the scheduler returns to a procedure called "idle". Here, the processor is in fact put into the "idle" state. This will cause the "ALE" pin to go continually high. A filter and differentiation circuit triggers whenever the idle state is left. If idle is not entered within 1 second, or if idle lasts for 1 second, the RESET input is pulled high. The functioning of the scheduler does not depend on this implementation of the watchdog, even if the idle state **must** be visited regularly.

Debugging processes.

If a high-level debugger is available, the line numbers and variables of the scheduler are directly available. Almost all that is needed is the "running" byte, telling which process is running, and the restart addresses of the different processes. Then it is possible to concentrate on the different processes and "forget" the scheduler.

Error messages.

Some fingerprints of the system often show up in the error messages. These values are input parameters to the exception handler. All messages cause crash-exceptions:

error_call_to_rx_level	call rx not from process or interrupt
error_interrupt_to_running	enable/disable used incorrectly
error_no_timebase	did not start at power-up
error_process_reg_bank	not register bank 0
error_single_process_running	stuck in loop
error_stack_overflow	less than power-up value
error_timeout_overflow	wait timeout already done
error_timeout_to_running	wait used incorrectly
error_timer_interrupt_lost	disable too long
error_idle_never_running	processes too busy
error_disable_missing	forgot disable
error_no_mailbox	not declared

Availability of the scheduler.

This paper describes a real-time scheduler and its basic ideas. Autronica is using the scheduler in a new hardware product. The scheduler itself is not a product that Autronica will market commercially.

Acknowledgments.

I must thank professor Odd Pettersen of NTH, Trondheim for reading and commenting on this paper. Also, Skogstad and Solli at Autronica have read this paper and the source code cautiously. Ellingsen at Autronica has implemented the scheduler in another soon to be released product, and given many valuable comments.