Real Time Embedded Systems
Worksheet 4. The Time-Slicing Structure

This week we start work on the central components of an elementary real-time operating system - we will call it a 'runtime system' - that divides the processor's time between separate user tasks. These tasks will need to communicate with the system, and will request its services by means of a 'software interrupt'.

Implementation of Software Interrupts

A software interrupt is known as a 'trap'. It causes the processor to respond in a very similar way as it does to a hardware interrupt, and so allows system calls from the user program and interrupts from hardware devices to enter the operating system in a consistent way.

There are 16 trap instructions available, numbered 0 to 15, and written

```
trap  #0
...
trap  #15
```

Each of the 16 trap instructions may have its own interrupt service routine (ISR). After pushing the PC and SR, the processor then accesses a table in low memory, at address 80H. As for the hardware interrupts, the table contains a 4-byte value corresponding to the address of the ISR for each software interrupt. Vectors for the two types of interrupt will normally be combined into a single block of code.

```
                    ;interrupt vectors

        org     $64     ;origin 64H
hvec1   dc.l    hisr1   ;address of hardware ISR 1
hvec2   dc.l    hisr2   ; ... etc

        org     $80     ;origin 80H
svec0   dc.l    sisr0   ;address of software ISR 0
svec1   dc.l    sisr1   ; ... etc
```

Controlling Interrupts

There is, however, an important difference between hardware and software interrupts. Hardware interrupts are in order of priority, with 7 being the highest priority and 1 the lowest. If two hardware interrupts occur at the same time, then the one at the higher priority will be accepted and the other one will be kept waiting until the first ISR has completed. If a hardware interrupt occurs shortly after another one, but while the ISR for the first interrupt is still in execution, then the processor will again compare the priorities of the two interrupts. If the new interrupt is of a higher priority, then it will interrupt the lower priority ISR. If the new interrupt is at a lower priority than the currently executing ISR, it will be kept waiting until that ISR completes.

Software interrupts do not behave in an analogous way. Since the processor can only execute one instruction at a time, it would be impossible for two software interrupts to occur at the same time, and unless a programmer includes a trap instruction within an ISR, there will also be no occasions on which a trap takes place during the processing of another trap. There is therefore no point in prioritising the software interrupts, and all 16 are at the same priority. There is, however, the question of the relative priority of the hardware and software interrupts. What if a hardware interrupt is raised at the same time as the processor is executing a software interrupt instruction? This is

handled by assigning all the software interrupts to priority level 0. Processing of a software interrupt *is* therefore interruptible by a hardware interrupt at any of the priority levels 1 to 7.

Within your system, however, regardless of the type of interrupt being processed, you will want to prevent the acceptance of any other interrupt. Your system will therefore be completely uninterruptible. Once entered, it will always run to completion and then return to the user task that was running when the interrupt was raised. You will therefore need to disable interrupt acceptance, the procedure for which is explained now.

Using the simulator, examine the 16-bit status register. Bits 8, 9 and 10 (labelled 'INT') hold a 3-bit value that represents the interrupt priority mask. When an interrupt is accepted, the mask is set to the priority level of that interrupt. A hardware interrupt will only be accepted if its priority is greater than the current setting in the mask. Normally, the mask is set to 000 (decimal 0) thereby allowing the acceptance of any hardware interrupt. However, it will remain at zero during its response to a software interrupt, since that is the priority of these interrupts, and will thereby allow the hardware to interrupt the software ISR. If you wish to prevent this, then the following instruction, placed at the very start of a software ISR, sets the mask to binary 111 (decimal 7). Any hardware interrupts will now be disabled, and held pending until the mask is returned to zero.

```
or     #$0700,sr        ;disable hardware interrupts
```

The status register will have been automatically saved on the stack at the start of the interrupt servicing. On execution of the 'return from exception' instruction (RTE), it will be restored, and the mask reset to the zero value that it held previously, thereby allowing the acceptance of any hardware interrupt that might have been raised in the meantime and is currently pending.

If you want to enable hardware interrupts at any other time, the following instruction will set the mask to zero.

```
and   #$f8ff,sr        ;enable hardware interrupts*
```

Practical Work

Assessment question

Work in groups of three on this question. Your submission should include the following.

> *Your software, including the run-time system and the test programmes you used to demonstrate it. Submit the source code,* ***not*** *the assembler output listing.*
>
> *Documentation: a .PDF file is preferable, otherwise .DOC.*
>
> *An individual 5 - 10 minute video presentation from each member of the group, explaining parts the system and how they were tested. Each presentation should deal with specific aspects, and the three presentations together should cover the entire system.*

The names and student numbers of all three group members should be shown on the software heading and on the front page of the documentation. On the video presentations the name of the group member should be clearly announced or displayed as a caption.

There are therefore five items to be submitted: a single software file, the documentation, and three presentations. These items should be placed into a single zipped file, and uploaded to a Canvas submission point to be advised. The submission deadline is ***2pm on Friday 19th January, 2024****.*

Each item will now be described in detail.

The run-time system

The work consists of writing a basic time-slicing system, along the lines of the one discussed in the lecture. It should allow the execution of several concurrent user tasks, with support for task scheduling and inter-task communication. An outline programme is provided on Canvas, but you will write the service routines (including reset), the scheduler, and the user tasks for each application.

The user tasks are located in memory, above the system itself. Each task has its own area of memory, with the programme code at the lowest address, data above it, and top-of-stack at the next address above this task's memory area. For example, the following task occupies memory between 2000H and 2FFFH. It has its code at 2000H, data at 2C00H, and stack at the top of the data area.

Address	
2000H	Programme code
2C00H	Data
3000H	Top-of-stack

The system runs in the foreground, and is entered following either a timer interrupt, a software interrupt from one of the tasks requesting service, or another hardware interrupt.

The following system calls should be supported by means of software interrupts. They can either each be allocated to a separate trap number, or (as in the demonstration system) they can all be called on the same trap, with one of the registers used to hold a value identifying the requested function. Some of the calls also require additional parameters in other registers.

1. Create task

Function: A currently unused TCB is marked as in use and set up for a new task. It is placed on the ready list. The requesting task remains on the ready list. Two parameters indicate the start and end of the memory area occupied by the new task and its data.

Parameters: The start address of the new task, The address of its top-of-stack.

2. Delete task

Function: The requesting task is terminated, its TCB is removed from the list and marked as unused. Any memory allocated to it is returned to the system.

Parameters: None.

3. Wait mutex

Function: If the mutex variable is one, it is set to zero and the requesting task is placed back onto the ready list. If the mutex is zero, the task is placed onto the wait list, and subsequently transferred back to the ready list when another task executes a *signal mutex*.

Parameters: None.

4. Signal mutex

Function: If the mutex variable is zero, and a task is waiting on the mutex, then that task is transferred to the ready list and the mutex remains at zero. If the mutex is zero and no task is waiting, the mutex is set to one. In either case, the requesting task remains on the ready list.

Parameters: None.

5. Initialise mutex

Function: The mutex is set to the value 0 or 1, as specified in the parameter.

Parameters: 0 or 1.

6. Wait time

Function: The requesting task is placed onto the wait list until the passage of the number of timer interrupts specified in the parameter, when it is transferred back to the ready list.

Parameters: Number of timer intervals to wait.

The following functions are more difficult. Good results may be obtained without implementing these functions, but it would be expected that they will be included in the best submissions. Function 7 will require the use of an additional interrupt at level 2, and you will need to consider the implications of this for the correct working of the system. Function 8 will require some thought about the internal record-keeping relating to which areas of memory are in use. You will need to devise your own test programmes for these functions.

7. Wait I/O:

Function: The requesting task is placed onto the wait list, until an interrupt signifies completion of an I/O operation, at which time the task is transferred back to the ready list.

Parameters: None.

8. Allocate memory

Function: For tasks that require a large amount of memory, it is more efficient to allocate it as required when the task runs. A large area of memory is therefore kept free within the system, and a block from it, of say 16 kbytes, is returned to the requesting task. If the request is satisfied, the requesting task remains on the ready list. If there is insufficient free memory available, then the requesting task is put on the wait list until memory is returned when another task terminates.

Parameters: On return, the start address of the allocated memory is held in the parameter register.

An additional function is executed automatically at start-up, or if the user presses the reset button.

System reset

Function: The system is initialised: all internal variables are reset, and each TCB is marked as unused. A TCB for task T0 is then created, and T0 becomes the running task.

The system assumes that a default user task, T0, is present. The system runs this task immediately after a reset. It will need to be located at a predetermined address, which will be coded into the reset function.

Your system should be robust, and deal with errors in an intelligent way. For example, what if the user tries to create more tasks than there are available TCBs?

Test programmes

Test your system using the following programmes. These are modified versions of the questions in last week's worksheet, this time running under your RTS.

1. Testing create task and wait time functions.

A stopwatch counts in seconds. It starts when button 0 is pressed and stops when the button is pressed again. It is programmed as follows.

Timer interrupts are set to 100ms. Task 0 starts task 1, and both tasks run concurrently. A shared variable `running` is held in a memory location, and is set by T1 and read by T0. T0 displays a 2-digit value on the 7-segment display, initialised to zero. If `running` is set, T0 increments the display, then waits for 10 time intervals, and then repeats. If `running` is not set, T0 does nothing. T1 tests pushbutton 0. Each time the button is pressed, `running` changes state.

2. Testing initialise mutex, wait mutex, and signal mutex functions.

On a gaming device, two players each have a button that fires bullets. Internal counters `a` and `b` record the number of bullets fired by each player, and a third counter `c` records the total number of bullets fired. It would be difficult to programme this application on the simulator because it is not possible to press two buttons at once. This would often happen in reality, and dealing with it represents the most difficult technical challenge in the programming. However, we can focus on this specific problem by programming two tasks, one for each player, and having each task fire bullets continuously.

Timer interrupts are set to 100ms. Task 0 starts task1, and both tasks run concurrently. A shared variable `c` is held in a memory location (not a register) and is initialised to zero. Task 0 has a variable `a`, also in memory, and runs in a loop that continuously increments `a` and `c`. Task 1 has a variable `b` in memory, and runs in a loop that continuously increments `b` and `c`.

To check that the system is working, one of the tasks calculates `a` + `b` - `c`, which should be zero, and displays it on the 7-segment display.

3. A test of your own to check the delete task function. Also tests of the wait I/O and allocate memory functions, if you have implemented them.

Your test programmes should be included at the end of the RTS code. Place all the tests together. An individual one can then be selected by commenting out the others.

Documentation

This consists of a user manual. It will explain your system to a user, and will therefore focus on what it does and how to use it. It will include a brief overview of how the system works internally, but only to an extent that is required for the programmer to use the system correctly. It should be structured as follows.

1.
A general description, including, for example:
explanation of the principles of multitasking and time-slicing, and how they are implemented;
description of the memory layout of the user tasks as shown above;
explanation of the startup behaviour: a default task runs, which may then start other tasks.

2.
A description of each of the user functions and their parameters. Explain all aspects of the behaviour of each function that are of interest to the user. For example, when a new task is created, does it run immediately, or is it put at the end of the ready queue? What if two tasks are waiting for the same time, and so become ready together? Give some emphasis to the behaviour of any of the more advanced features that you may have implemented. For example, if you have implemented function 7, you should explain how the different interrupt priorities have been handled internally, and how this affects the user. Your descriptions should also state how long each function takes to execute. This time should be quoted in terms of instructions, and may be within a specified range, e.g., a particular function might execute in 30 - 50 instructions.

3.
A description of any other relevant aspects of system behaviour. For example, have you arranged for prioritised scheduling? If so, how does it behave? If not, then are all tasks that are ready likely to receive a similar amount of run time? Is there any possibility that a task may receive too little time, or none at all? State the timer interrupt period, and how long (in terms of instructions) the system takes to switch tasks. Using a reasonable average for the instruction execution time, calculate the proportion of time that is spent in the RTS, and how much is available for running the user tasks.

With normal typeface and spacing (such as used here) it would be reasonable to expect a length of no more than five or six pages. Submission in .PDF format is preferred, but .DOC(X) is also acceptable.

Individual presentation

Each of the three group members should include an individual video presentation explaining particular parts of the system, how they worked, and how they were tested. These will probably be the functions with which that member was most closely involved in writing. Between the three presentations, the entire system should be explained.

These presentations will complement the written material in your documentation. Instead of describing how to use the system, they will explain in more depth how it works internally. They do not need to be long and detailed, but should focus on the underlying working principles of the software. It is unlikely to be necessary to talk about individual machine instructions, unless you are discussing a relevant concept such as mutual exclusion. You should also explain how each part of the system was tested, and how the test results verify its correctness.

Each presentation should last 5 - 10 minutes.

The Demonstration System

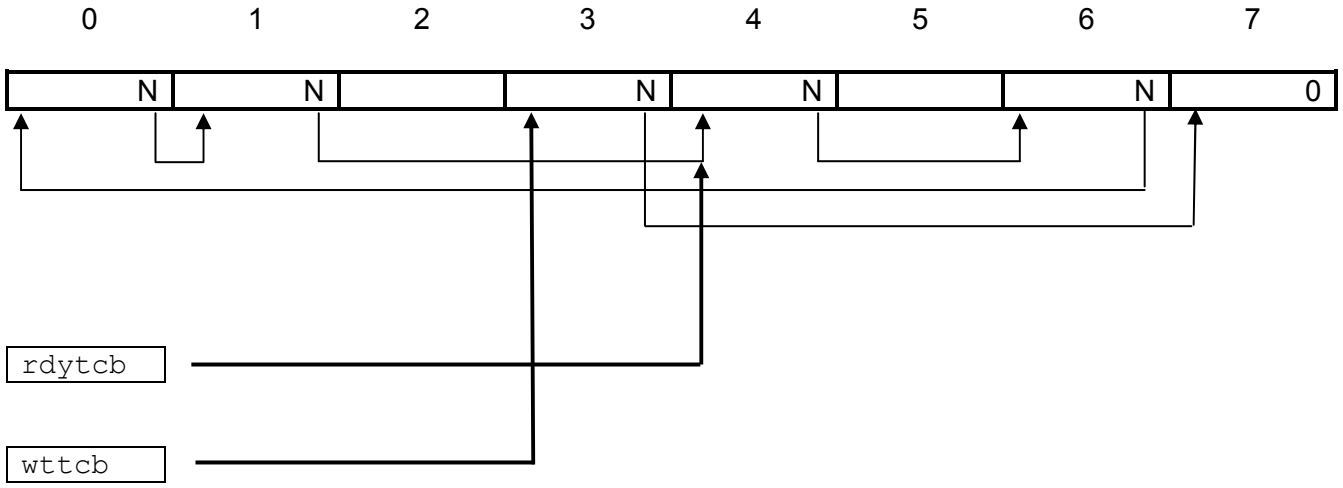
A system was demonstrated during the lecture. It recognises a hardware interrupt at level 1 from the timer. It also allows system calls by means of software interrupts, all of which have been allocated to trap 0. These system calls are programmed by placing a value that identifies the requested function into data register 0, and any other parameters as required by each of the individual functions in registers D1 onwards. For example, system call 1 is used to create a new task. Suppose that this new task is called T1, and that its top-of-stack is to be located at address 6000H. It would be programmed as follows.

```
move.l     #1,d0       ;set id in d0
move.l     #t1,d1      ;set address of new task in d1
move.l     #$6000,d2   ;set stack address in d2
trap       #0          ;call system
```

The main data structure used in this system is a list of task control blocks (TCBs). Each TCB represents the state of one of the tasks. It contains a copy of all that task's registers, together with some items of control information including a flag that indicates whether the TCB is in use.

At any time, each of the tasks will be in one of three states: the currently running task, ready to run when its turn comes up, or unable to run because it is waiting for the occurrence of some event, which could be a signal operation on a mutex, the expiry of a time interval, or an I/O interrupt.

At initialisation, all the TCBs in the list are marked as unused. As each new task is started, one of the unused TCBs is allocated for it and marked as used. These TCBs are organised into two linked lists, in which each element contains a pointer to the next element. These lists are called 'ready' and 'waiting'. Two more data items consist of pointers to the first element in each list. The pointer `rdytcb` holds the address of the first element in the list of ready TCBs. The first element in this list is the task that actually is running. The linkage in this list is circular, that is, the last entry points back to the first, so making it easy to access each TCB in rotation. The pointer `wttcb` holds the address of the first element in the list of TCBs that are waiting. Each element will contain an indication of the event the task is waiting for. There is no need to access elements of this list in rotation, so the last element has its pointer set to zero.



The example above shows a list of 8 elements, each of which has a pointer, labelled N, to the next element. Elements 4, 6, 0 and 1 are on the ready list, with element 4 being the TCB for the running task. Elements 3 and 7 are on the waiting list, and elements 2 and 5 are unused.

The system is organised into the following sections. Some of these sections, shown in italics, are provided for you in the outline code on Canvas. These can be used in your own systems: unchanged, modified, or rewritten as you wish.

Data definitions and equates
`org  0`
Interrupt vectors
Executable code
    System reset
    First-level interrupt handler entry
    FLIH Service routines
    Scheduler
    Dispatcher
Data storage
Default user task T0

Data definitions

Each TCB represents the state of one of the current tasks, and is defined as follows.

```
tcb     org     0               ;tcb record
tcbd0   ds.l    1               ; D register save
tcbd1   ds.l    1
tcbd2   ds.l    1
tcbd3   ds.l    1
tcbd4   ds.l    1
tcbd5   ds.l    1
tcbd6   ds.l    1
tcbd7   ds.l    1
tcba0   ds.l    1               ; A register save
tcba1   ds.l    1
tcba2   ds.l    1
tcba3   ds.l    1
tcba4   ds.l    1
tcba5   ds.l    1
tcba6   ds.l    1
tcba7   ds.l    1
tcbsr   ds.l    1               ; SR (status reg) save
tcbpc   ds.l    1               ; PC save
tcbnext ds.l    1               ; link to next record
tcbused ds.l    1               ; record in use flag
        ds.l    1               ; other fields as required
        ds.l    1               ;
tcblen  equ     *               ; length of tcb record in bytes
```

Data storage

Storage for a list of TCBs is defined as in the first line below. The constant `ntcb` represents the number of TCBs in the list, and should be set up as an equate. Other variables are described throughout these notes.

```
tcblst  ds.b    tcblen*ntcb             ;tcb list (length x no of tcbs)
rdytcb  ds.l    1                       ;^ ready tcb list
wttcb   ds.l    1                       ;^ waiting tcb list
a0sav   ds.l    1                       ;A0 temporary save
d0sav   ds.l    1                       ;D0 temporary save
id      ds.l    1                       ;function id
```

Interrupt vectors

The interrupt vectors are addresses of the code that will be executed as a result of an interrupt. The following three addresses are defined.

Address `res` is the location of the routine to which the processor branches when the it responds to a hardware reset. Address `fltint` is the location to which the processor branches following a timer interrupt at level 1, and flsint following a software interrupt. The address `stk` is the value that is loaded into the stack pointer following a hardware reset.

```
;*************************************************************************
                                        ;INTERRUPT VECTORS
;*************************************************************************

        org     0

        dc.l    stk                     ; initial SP
        dc.l    res                     ; reset
        org     $64
        dc.l    fltint                  ; interrupt 1 (timer)
        org     $80
        dc.l    flsint                  ; trap 0 (system call)
```

Executable Code

First-level interrupt handler

The first-level interrupt handler (FLIH) contains the code that services an interrupt. For convenience it is split into the common FLIH entry section that is executed immediately following an interrupt, and the FLIH service routines that carry out the processing specific to each type of interrupt.

FLIH entry

Hardware interrupts at level 1 are directed by the interrupt vector to enter the FLIH at `fltint`, while software interrupts arrive at `flsint`. The FLIH performs three main functions.

It takes the pointer to the TCB of the currently executing task, stored at `rdytcb`, and saves the values of the registers, including the PC and SR, within that TCB.

It also sets a value within a storage location, known as `id`, that identifies the source of the interrupt. If an interrupt has been raised by the hardware timer, then `id` is set to 0. For a software interrupt, `id` is set to the value, from 1 onwards, of the system call function number. The `id` will subsequently be used to select the corresponding service routine for processing this interrupt.

Programming the above two operations requires particular care, because saving the value of the user's registers as they were at the time of the interrupt requires the use of certain registers itself. Registers D0 and A0 are in use for this purpose. These registers are therefore saved in temporary locations, before being transferred to their long-term holding locations within the TCB.

The other function performed by the FLIH is to disable interrupts, if this has not already happened. A level-1 hardware interrupt from the timer will have set the interrupt priority mask to 1, thereby preventing any further interrupts. A software interrupt will have left the mask at 0, which would allow the timer device to interrupt the processing of the software interrupt. Therefore the first action taken at the software interrupt entry point is to disable hardware interrupts by setting the mask to 7.

```
;*******************************************************************************
flih                                    ;FIRST-LEVEL INTERRUPT HANDLER ENTRY
;*******************************************************************************

fltint                                  ;ENTRY FROM TIMER INTERRUPT
        move.l  d0,d0sav                ;save D0
        move.l  #$0,d0                  ;set id = 0
        move.l  d0,id
        move.l  d0sav,d0                ;restore D0
        bra     fl1

flsint                                  ;ENTRY FROM TRAP (SOFTWARE INTERRUPT)
        or      #%0000011100000000,sr   ;disable hardware interrupts
        move.l  d0,id                   ;store id
        bra     fl1

fl1     move.l  a0,a0sav                ;save working reg

        move.l  rdytcb,a0               ;A0 ^ 1st ready tcb (ie running tcb)

        move.l  d0,tcbd0(a0)            ;store registers
        move.l  d1,tcbd1(a0)
        move.l  d2,tcbd2(a0)
        move.l  d3,tcbd3(a0)
        move.l  d4,tcbd4(a0)
        move.l  d5,tcbd5(a0)
        move.l  d6,tcbd6(a0)
        move.l  d7,tcbd7(a0)
        move.l  a0sav,d0
        move.l  d0,tcba0(a0)
        move.l  a1,tcba1(a0)
        move.l  a2,tcba2(a0)
        move.l  a3,tcba3(a0)
        move.l  a4,tcba4(a0)
        move.l  a5,tcba5(a0)
        move.l  a6,tcba6(a0)

        move    (sp),d0                 ;pop and store SR
        add.l   #2,sp
        move.l  d0,tcbsr(a0)

        move.l  (sp),d0                 ;pop and store PC
        add.l   #4,sp
        move.l  d0,tcbpc(a0)

        move.l  a7,tcba7(a0)            ;store SP

                                        ;START OF SERVICE ROUTINES
```

FLIH service routines, including system reset

The service routines are arranged as a large switch statement, using `id` as the case variable. Each routine carries out one of the functions defined in the specification.

Scheduler

The scheduler examines the ready list, to which `rdytcb` points to the first element. This is the TCB of the task that was executing when the system was invoked, and which has just been interrupted. By following the links, the scheduler can locate each TCB that is currently ready to run. It selects

one of these tasks for running, and adjusts the value in `rdytcb` to point to the TCB for this task. This TCB will be then used by the dispatcher to resume execution of the task.

The scheduler may make the decision as to which task will run next by doing nothing more than following the link in the current TCB to the next one in the chain. This will result in each ready task running in rotation, receiving an approximately equal amount of run time each. Alternatively, it would be possible to assign a priority to each task as it is created, by adding another parameter to the 'create task' system call. Higher priority tasks would then receive a larger proportion of the available run time.

Dispatcher

The dispatcher reverses the action taken by the FLIH. Using the newly set value in `rdytcb`, it restores the registers of the selected task to the values that were stored when that task was interrupted. Careful housekeeping is again necessary, as this operation itself requires the use of registers D0 and A0. The dispatcher finishes by recreating the state of the stack as it was after the task was interrupted. The processor then uses a 'return from exception' instruction, as though it were returning from any normal interrupt, to transfer control back to the selected task.

```
;                                       ;END OF SCHEDULER

;*************************************************************************
disp                                    ;DISPATCHER
;*************************************************************************

        move.l  rdytcb,a0               ;A0 ^ new running tcb
        move.l  tcbd1(a0),d1            ;restore registers
        move.l  tcbd2(a0),d2
        move.l  tcbd3(a0),d3
        move.l  tcbd4(a0),d4
        move.l  tcbd5(a0),d5
        move.l  tcbd6(a0),d6
        move.l  tcbd7(a0),d7
        move.l  tcba1(a0),a1
        move.l  tcba2(a0),a2
        move.l  tcba3(a0),a3
        move.l  tcba4(a0),a4
        move.l  tcba5(a0),a5
        move.l  tcba6(a0),a6
        move.l  tcba7(a0),a7

        sub.l   #4,sp                   ;push PC
        move.l  tcbpc(a0),d0
        move.l  d0,(sp)

        sub.l   #2,sp
        move.l  tcbsr(a0),d0            ;push SR
        move    d0,(sp)

        move.l  tcbd0(a0),d0            ;restore remaining registers
        move.l  tcba0(a0),a0

        rte                             ;return
```

An example of a user programme running under this system is shown here. It consists of two concurrent tasks. Task T0 calls the system to start task T1, then switches on the RH LED. Task T1 calls the system to wait for 3 timer intervals, then switches on the LH LED. From then on, the two tasks run alternately. If the timer is set to interrupt at one-second intervals, the result is that the RH LED lights immediately, then after 3 seconds the two LEDs start alternating.

```
;*************************************************************************
                                        ;USER APPLICATION TASKS
;*************************************************************************

                                ;system call equates
sys     equ     0               ; system call trap (trap 0)
syscr   equ     1               ; create new task
sysdel  equ     2               ; delete task
syswttm equ     6               ; wait on timer

;*************************************************************************
                                        ;USER APPLICATION TASKS
;*************************************************************************

        org     usrcode

led     equ     $e00010         ;led
sw      equ     $e00014         ;switch

t0:                             ;TASK 0
        move.l  #sysscr,d0       ;start task 1
        move.l  #t1,d1          ;  address
        move.l  #$4000,d2       ;  top of stack
        trap    #sys
                                ;repeat
t00:    move.l  #$01,d1         ;  set led 0
        move.b  d1,led

        bra     t00

t1:                             ;TASK 1
        move.l  #syswttm,d0     ;wait for 3 clocks
        move.l  #3,d1
        trap    #sys
                                ;repeat
t10:    move.l  #$02,d0         ; set led 1
        move.b  d0,led

        bra     t10

        END    res
```