

Implementing low-cost TTCS systems using assembly language

Simon Key, Michael J. Pont and Simon Edwards¹

*Embedded Systems Laboratory², Department of Engineering, University of Leicester,
University Road, LEICESTER LE1 7RH, UK.*

Introduction

We have previously described a “pattern language” consisting of nearly eighty components, which will be referred to here as the “PTTES Collection” (see Pont, 2001; Pont and Ong, in press). This language is intended to support the development of reliable embedded systems: the particular focus of the collection is on systems with a time triggered, co-operatively scheduled (TTCS) system architecture.

In this paper, we present a new pattern – ASSEMBLY-LANGUAGE SCHEDULER – which describes how to implement TTCS architectures using small, memory-limited, microcontrollers (such as 8051s, PICs, AVRs or similar devices).

Acknowledgements

We are grateful to Jorge Ortega-Arjona (our ‘shepherd’ at EuroPLoP 2003) for comments and suggestions on the first drafts of this paper.

Copyright

Copyright © 2003 by Simon Key, Michael J. Pont and Simon Edwards. Permission is granted to copy this paper for the purposes of EuroPLoP 2003. All other rights reserved.

¹ The first phase of the work described in this paper was carried out while Simon Edwards was a Final-Year student at the University of Leicester. Present address: MIRA Ltd, Watling Street, Nuneaton, Warwickshire CV10 0TU, UK

² <http://www.le.ac.uk/eg/embedded>

ASSEMBLY-LANGUAGE SCHEDULER

Context

- You are developing an embedded application using a microcontroller (or similar hardware).
- The microcontroller has very limited data memory and code memory.

Problem

How can you create a co-operative scheduler with minimal memory and CPU requirements?

Background

There are two ways of viewing a (co-operative) scheduler:

- At one level, a scheduler can be viewed as a simple operating system that allows tasks (functions) to be called periodically, or - less commonly - on a one-shot basis.
- At a lower level, a scheduler can be viewed as a single timer interrupt service routine that is shared between many different tasks. As a result, only one timer needs to be initialised, and any change to the timing generally requires only one function to be altered. Furthermore, we can generally use the same scheduler whether we need to execute 1, 10 or 100 different tasks.

Using such a scheduler, we can co-ordinate the execution and interaction of tasks in a highly deterministic manner. By making the scheduler co-operative (that is, only one task is active at any point in time), we simplify the design³ and greatly reduce the potential for task conflicts.

Solution

We have previously described, in detail, how to create a flexible scheduler using the C programming language (see CO-OPERATIVE SCHEDULER [Pont, 2001, p.255]). In the present pattern, we describe how to create a similar scheduler, using assembly language. By using assembly language, we can significantly reduce the system resource requirements.

In the examples, we will use code fragments taken from an 8051-based system. However, the techniques can be applied with virtually any microcontroller.

Key components

An assembly-language scheduler has the following key components:

1. The scheduler data structure.
2. An initialisation function.

³ Compared with an equivalent pre-emptive scheduler.

3. A single interrupt service routine (ISR), used to update the scheduler at regular time intervals.
4. A dispatcher function that causes tasks to be executed when they are due to run.
5. One or more tasks.

The links between these components during the program execution are shown schematically in Figure 1.

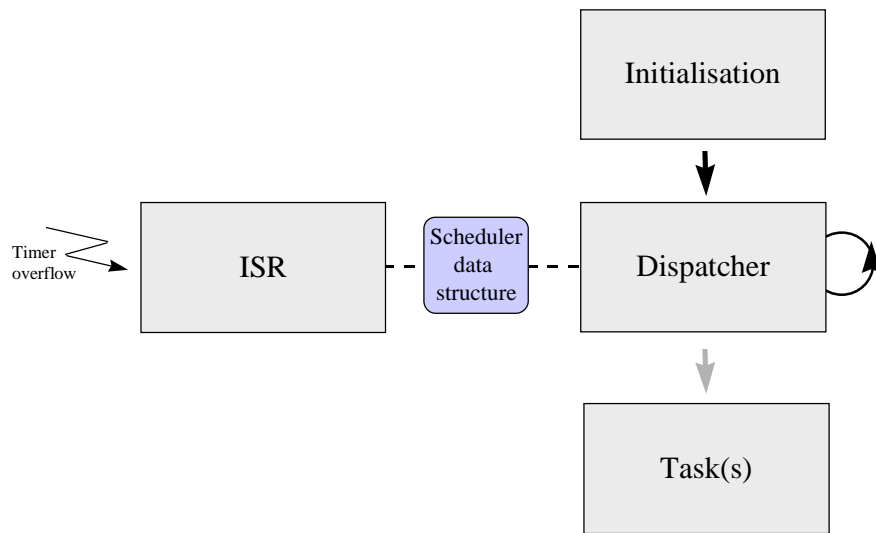


Figure 1: Links between the different scheduler components during the program run.

We consider each of these components in the sections that follow.

The scheduler data structure

To control a task we need to know three basic pieces of information:

1. The location of the task in program memory
2. The initial delay value (when should the task be run for the first time?)
3. The Reload value (if the task is periodic, what is the interval between runs?)

The location of the tasks is given in the task list. This is a 'jump table' (see Listing 1).

```

; *** This is where the scheduled tasks go list below in order (1 to 8)
TASK_LIST:
  a jmp  TASK1          ;goto task 1
  a jmp  TASK2          ;goto task 2
  a jmp  TASK3          ;goto task 3
  a jmp  TASK4          ;goto task 4
  a jmp  TASK5          ;goto task 5
  a jmp  TASK6          ;goto task 6
  a jmp  TASK7          ;goto task 7
  a jmp  TASK8          ;goto task 8
SCH_NO_TASK_HERE:      ;DO NOT MOVE OR RENAME THIS LINE
  
```

Listing 1 Task list (file: 8051SCH.ASM)

Each task has its own delay and reload values which are entered directly by the programmer into the appropriate task slot (Listing 2). These values are then copied at runtime into RAM memory locations.

```

;*****
; Scheduled Task Delay and Reload Values
;*****
cseg
SCH_Delay_Values:
db    0x00          ;delay value for task 1
db    0x00          ;delay value for task 2
db    0x00          ;delay value for task 3
db    0x00          ;delay value for task 4
db    0x00          ;delay value for task 5
db    0x00          ;delay value for task 6
db    0x00          ;delay value for task 7
db    0x00          ;delay value for task 8

SCH_Reload_Values:
db    0x63          ;reload value for task 1
db    0x00          ;reload value for task 2
db    0x00          ;reload value for task 3
db    0x00          ;reload value for task 4
db    0x00          ;reload value for task 5
db    0x00          ;reload value for task 6
db    0x00          ;reload value for task 7
db    0x00          ;reload value for task 8

```

Listing 2 Delay and reload Values (file: TASK_PERIODS.INC)

An initialisation function

As with most programs the system requires some initialisation. In the case of assembly language programs we are also required to initialise vectors and memory allocations.

Vectors

When an interrupt occurs the CPU finishes the instruction that it is currently processing and jumps to an area of memory known as a vector. This vector contains a pointer to the relevant ISR.

In this case we are considering only one vector, linked to the timer overflow. When the timer reaches its maximum count and ‘overflows’, the ISR “SCH_Update” is called.

```

;*****
; Vectors
;*****
CSEG  AT    0x0000          ; absolute Segment at Address 0
LJMP  MAIN          ; reset location (jump to start)
CSEG  AT    0x002b          ; interrupt vector
; *** There is only one interrupt in this program...
LJMP  SCH_Update          ;timer has overflowed, so update scheduler
CSEG  AT    0x0043          ; after all interrupt vectors
;start of main program.

```

Listing 3 Interrupt vectors (file: VECTORS.INC)

Memory Allocation

In assembly language the location of variables used within the program has to be specified manually. This is done by allocating a name to a section of memory; this name can then be referred to throughout the program (this is similar to variable initialisation in a high-level language).

```
*****
; Scheduler Memory Allocation
*****

START_OF_DATA_MEMORY      EQU    (00h)
START_OF_USER_MEMORY      EQU    (SCH_rloads + SCH_MAX_TASKS)
;scheduler status register
SCH_STATUS      DATA    (START_OF_DATA_MEMORY)
;temp register for Scheduler
SCH_temp      DATA    (START_OF_DATA_MEMORY + 01h)
;general temp register
temp      DATA    (START_OF_DATA_MEMORY + 02h)
; ***Scheduler variables
;Run flags for scheduled tasks
SCH_run_f      DATA    (START_OF_DATA_MEMORY + 04h)
;Run flag mask
SCH_run_mask DATA    (START_OF_DATA_MEMORY + 05h)
;Current task index
SCH_index      DATA    (START_OF_DATA_MEMORY + 06h)
;Task Dispatcher run flag mask
SCH_dis_mask DATA    (START_OF_DATA_MEMORY + 07h)
;Task Dispatcher task index
SCH_dis_index  DATA    (START_OF_DATA_MEMORY + 08h)
;Delay values for scheduled tasks
; put in ram comments to the size of the number of tasks -> SCH_MAX_TASKS

SCH_delays     DATA    (START_OF_DATA_MEMORY + 09h)
;Delay values for scheduled tasks
; put in ram comments to the size of the number of tasks -> SCH_MAX_TASKS

SCH_rloads     DATA    (SCH_delays + SCH_MAX_TASKS)

*****
*
```

Listing 4 Memory allocation (file: MEMORY_ALLOCATION.INC)

Device initialisation

To ensure correct operation of the scheduler we have to initialise run time variables, prepare the system timer and prepare any ports used for IO operations. The initialisation of runtime variables consists of copying the task delay and task reload values from non changeable ROM to RAM. Listing 5, shows general configuration settings for the device.

```
initialise:
; *** General Initialisation
mov     SP,#0x30          ; Set Stack Pointer
clr     SWDT              ; Clear Watchdog timer
```

Listing 5 Initialisation (file: 8051SCH.ASM)

Timer initialisation

The scheduler is driven by timer “ticks”. These are interrupts generated when a timer register overflows. Listing 6 shows a possible timer initialisation using Timer 2 in an 8051 device. With the timer reload values used here we have a 10 ms period between interrupts.

```

; *** Initialise timer for scheduler (using TMR2)
mov    T2CON,#0x10
mov    CCEN,#0x00

; *** The timer preload values
mov    TH2, #PRELOAD10H           ; predefined values giving 10ms tick
mov    CRCH, #PRELOAD10H
mov    TL2, #PRELOAD10L
mov    CRCL, #PRELOAD10L

; *** Enable Timer 2 interrupt, but not the external one.
setb   ET2
setb   EXEN2

```

Listing 6 Timer initialisation (file: 8051SCH.ASM)

In general, the initial task delays and reload values are stored directly in program memory using assembler directives, such as db. If these memory locations are referenced using a label a simple routine to move these fixed values from code memory to RAM can be used. Listing 7 shows the routine used to copy the data code section (scheduler data structure) from ROM to RAM.

```

; *** Initialise scheduler variables
mov    SCH_run_f,#0x00           ;clear all run flags
mov    SCH_index,#0             ;reset index counter
mov    R1,#(SCH_rloads)         ;get address of delay values
mov    R0,#(SCH_delays)         ;initialise indirect pointer

SCH_init_lp:
mov    DPTR,#SCH_Delay_Values   ;get delay value for task[index]
mov    A,SCH_index              ;get task index value
movc   A,@A+DPTR
mov    @R0,A                    ;store in array
inc    R0
mov    DPTR,#SCH_Reload_Values  ;get reload value for task[index]
mov    A,SCH_index              ;offset pointer by number of tasks
movc   A,@A+DPTR               ;get task index value
mov    @R1,A                    ;store in array
inc    R1
inc    SCH_index
mov    A,SCH_index

; decrease index value, if not zero, loop
cjne  A,#SCH_MAX_TASKS,SCH_init_lp

```

Listing 7 ROM to RAM copy (file: 8051SCH.ASM)

To conclude the initialisation, Timer 2 is started and the associated interrupt is enabled (Listing 8).

```

; *** Enable interrupts and reset TMR0
setb   T2I0                     ; Set the timer running
setb   EAL                      ; enable interrupts

```

Listing 8 Concluding the scheduler initialisation (file: 8051SCH.ASM)

An ISR

When the scheduler timer (discussed above) ‘overflows’, the scheduler interrupt service routine (ISR) is executed. The first action taken in the ISR is to clear the timer interrupt flag and save any registers that will be altered within this routine (Listing 9); these registers will be restored at the exit from the function.

```

SCH_Update:
  clr    TF2                ; clear TMR 2 interrupt flag
  push  ACC                ; save registers that might be needed
  push  PSW
  push  01h
  push  00h

```

Listing 9 Performing a context save (file: 8051SCH.ASM)

Each of the possible tasks is then checked. A non-zero delay time is decreased by one, and the update function then checks the next task. A zero delay time results in the delay time been re-initialised with the associated reload value, and the run flag for that task is set (Listing 10).

```

; *** Setup loop counter and run flag mask
mov     SCH_run_mask,#00000001b ;init value for run flag mask
mov     SCH_index,#SCH_MAX_TASKS ; reset index value to number of tasks
; in scheduler

mov     A,#SCH_delays
mov     R0,A                    ; get address of SCH_delays
mov     A,#SCH_rloads          ; get location of first reload ram
; location

mov     R1,A

SCH_Update_lp:
  mov     A,SCH_run_mask        ; get run flag mask
  cjne   @R0,#0,SCH_Task_notready ; test delay time if not zero task not
; ready
; *** Current task is ready to be run, so set run flag and reload delay
; value
SCH_Task_ready:
  orl    SCH_run_f,A           ; set appropriate run flag
  mov    A,@R1                 ; get value being pointed to
  mov    @R0,A                 ; store in delay value (task period)
  AJMP  SCH_Update_lp_end      ;goto end of loop

SCH_Task_notready:
;task isn't ready to be run
  dec    @R0                   ;decrease delay value by one

SCH_Update_lp_end:
  inc    R0                    ;increase pointer value by one
  inc    R1
  mov    A,SCH_run_mask        ;roll run mask one place to left
  rl     A
  mov    SCH_run_mask,A
  djnz  SCH_index,SCH_Update_lp ; decrease index value, not zero, so
; loop still tasks to look at

```

Listing 10 The core scheduler “update” code (file: 8051SCH.ASM)

Finally, the pre-ISR program context is restored, and system resumes “normal” operation (Listing 11).

```

pop     00h                    ; restore registers
pop     01h
pop     PSW
pop     ACC
reti                                ;return from interrupt

```

Listing 11 Restoring the pre-ISR context (file: 8051SCH.ASM)

A dispatch function

We have seen that a run flag is set for a given task to be performed in the SCH_Update function. In this section we see how the flag is used to activate the task that it relates to.

Until an interrupt occurs and a run flag is set, the dispatch function sits in an endless loop, ‘sleeping’ between timer ‘ticks’. When a run flag has been set, checks are performed and the associated task is called (Listing 12).

```

; *** This is the main task dispatcher process.  If a task is ready to be run
;     then the dispatcher will run it and clear its run flag.

    orl    PCON,#0x01          ; Enter idle mode (#1)
    orl    PCON,#0x20          ; Enter idle mode (#2) goto sleep
    mov    A,SCH_run_f         ; test run flags
    jz     SCH_Dispatch_Tasks ; no tick so loop

; initial value for mask, load dispatcher run flag mask
    mov    SCH_dis_mask,#0000001b
; get number of tasks in scheduler, load dispatcher task index
    mov    SCH_dis_index,#SCH_MAX_TASKS

SCH_Dispatch_Tasks_lp:
    mov    A,SCH_dis_mask      ; get run flag mask
    anl    A,SCH_run_f         ; logical AND with run flags
; if result was zero, so no run flag there, go back to loop
    jz     SCH_Dispatch_Tasks_end

; *** Task needs to be run, so clear run flag and jump to task
    mov    A,SCH_dis_mask      ;get inverse of run flag mask
    cpl    A
    anl    SCH_run_f,A         ;clear appropriate run flag
    mov    DPTR,#TASK_LIST     ; load start of task lists
    mov    A,#SCH_MAX_TASKS
    subb   A,SCH_dis_index     ; add current task no
    clr    C                   ; clear carry
    rlc    A                   ; multiply by two to get 'real' code location
    jmp    @A+DPTR             ; jmp to task

```

Listing 12 A possible dispatch function (file: 8051SCH.ASM)

The tasks have been stored in a jump table, and the relevant task number is added to the base address of this table (TASK_LIST). The resulting value is then stored in the program counter, causing a ‘jump’ to the task (Listing 13).

```

; *** This is where the scheduled tasks go.
;     the list below in order (1 to 8)
TASK_LIST:
    ajmp   TASK1               ;goto task 1
    ajmp   TASK2               ;goto task 2
    ajmp   TASK3               ;goto task 3
    ajmp   TASK4               ;goto task 4
    ajmp   TASK5               ;goto task 5
    ajmp   TASK6               ;goto task 6
    ajmp   TASK7               ;goto task 7
    ajmp   TASK8               ;goto task 8

SCH_NO_TASK_HERE:             ;DO NOT MOVE OR RENAME THIS LINE

```

Listing 13 A possible implementation of the task list jump table (file: 8051SCH.ASM)

Having checked (and, if necessary, run) all possible tasks, the whole process starts again (Listing 14).

```

SCH_Dispatch_Tasks_end:
  mov  A,SCH_dis_mask          ; roll run flag mask one place left
  clr  C
  rlc  A
  mov  SCH_dis_mask,A
; decrease task index , if not zero more tasks to do
  djnz SCH_dis_index,SCH_Dispatch_Tasks_lp
  ajmp SCH_Dispatch_Tasks      ; super loop!

```

Listing 14 A dispatch loop (file: 8051SCH.ASM)

Tasks

For demonstration purposes, we include a simple task that will flash an LED connected to an external pin (50% duty cycle, 0.5 Hz).

```

;*****
; Function:      LED_Flash
; Description:   Flashes LED on port0 pin0 on and off
; Pre:          None
; Post:         None
;*****
TASK1:
LED_Flash:
  mov  A,LED_Status          ; is led on?
  jz   LED_on                ; no got turn it on
  clr  LED_PORT              ; turn off
  mov  LED_Status,#00        ; set status
  ajmp SCH_Dispatch_Tasks_end

LED_on:
  setb LED_PORT              ; turn LED on
  mov  LED_Status,#01        ; set status
  ajmp SCH_Dispatch_Tasks_end ; Return from scheduled task

```

Listing 15 Flash LED Task (file: TASK1.ASM)

Reliability and safety implications

Many questions have been raised about the suitability of assembly language⁴ for use in applications which are safety-related or safety critical (see, for example, Cullyer et al., 1991; Cooling, 2003).

It is undoubtedly the case that implementing – say – a very large and complex air-traffic control system entirely in assembly language would probably not be sensible. However, in the present pattern, we focus on systems implemented using microcontrollers with very limited memory. A consequence of this is that the code size is (compared with many real-time and embedded systems) very small. What is not clear is whether – in such small systems, where the code can be carefully and completely checked – there are significant safety implications resulting from the use of assembly language vs. C. Further studies are required in order to clarify this issue.

Hardware resource implications

Embedded systems implemented using assembly language generally require significantly less memory than those implemented in high-level languages. To illustrate the likely savings in memory when using this pattern, please consider the results shown in Table 1.

⁴ It should also be noted that C is rarely considered to be an ideal language for safety-critical systems.

Microcontroller	Manufacturer	Size of assembly compared to C (ROM)	Size of assembly compared to C (RAM)	
			One task	Five tasks
16F877	Microchip	20%	39%	27%
AT90s2343	Atmel	26%	35%	31%
C515C	Infineon	32%	76%	46%

Table 1: A comparison of the memory requirements for schedulers implemented in assembly language and C, for a range of different microcontrollers (all figures are approximate).

In this table, the comparison is between an implementation of Co-operative scheduler (Pont, 2001, p.255), implemented in C, and the assembly-language scheduler described in the present pattern. Overall, these figures illustrate that the assembly-language implementations require around a third (or less) of the code memory than is required in the C-language implementation. Savings in code memory are more variable (in this table), but are still substantial.

These memory savings (particularly code-memory reductions) translate directly into reduced costs. This is illustrated in Figure 2, which shows the cost per device for a family of 8051 microcontrollers. The only difference between these microcontrollers is the (code) memory size.

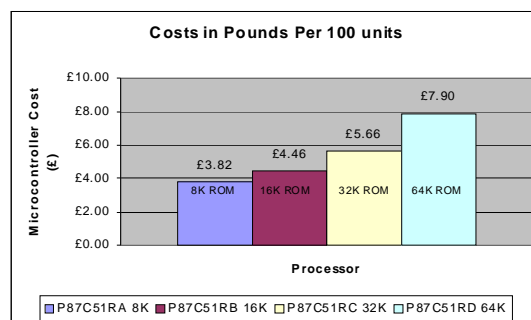


Figure 2 Relating memory size to device costs

When considering the resource implications of this approach to scheduling, please note that the scheduler presented in this paper is divided into multiple source files. This has been done because, when compared with a single-source version, we find this version much easier to use. Please bear in mind that this approach increases the code memory requirements (when using a single task) by 42 bytes (the increase is much less as subsequent tasks are added).

Portability

Assembly language is not a portable language, when compared to any high level language. For example porting the scheduler code from a Microchip 16F877 to an AVR 90S2323 generally requires a complete rewrite.

However within product families, manufacturers maintain compatibility of instruction sets. For example the 8051 or AVR allows quick porting of this scheduler between chips within the same range. Similar observations are noted in other families.

Related patterns and alternative solutions

CO-OPERATIVE SCHEDULER [Pont, 2001, p.255] describes a similar architecture, implemented using the C programming language.

Overall strengths and weaknesses

From previous work we have the following strengths for a co-operative scheduler:

- ☺ The scheduler behaviour is highly predictable.
- ☺ The scheduler is much simpler than pre-emptive alternatives.
- ☺ The scheduler is an integral part of the developed application.

Additionally for the assembly scheduler (compared with an implementation in “C”), we have one key strength:

- ☺ By using this scheduler we can reduce costs, through the use of a microcontroller with less code memory.

General weaknesses of co-operative scheduling:

- ☹ External events have to be polled, which may delay response times under some circumstances.
- ☹ Tasks that exceed the system tick interval can greatly disrupt the system performance.

Specifically for the assembly scheduler:

- ☹ Knowledge of the individual of microcontroller is required in order to write effective code.
- ☹ In the implementation presented here, a maximum of eight tasks that can be scheduled.
- ☹ The assembly-language scheduler is not as straightforward to use as the C-language scheduler and some “hand crafting” is required.

Example: An 8051 assembly scheduler

We present a complete listing of an assembly-language scheduler for the 8051 microcontroller in this section. It is targeted at an Infineon C515C microcontroller running at 10 MHz, and produces a 1ms tick. An example “flashing LED” task is included.

Please note that, as discussed in “Hardware resource implications”, this scheduler is split across multiple source files. This is intended to make the use of the program more straightforward. We can illustrate the use of these various files by outlining how we would

add a simple “flashing LED” task to the scheduler:

- Create the ‘flashing LED’ task (TASK1.ASM, Listing 22)
- Modify the task’s delay and reload values as required (TASK_PERIODS.INC, Listing 21)
- Specify the port pin to be used (PORTS.INC, Listing 20)
- Modify the number of tasks and - if required - the scheduler timing (TIMING.INC, Listing 17)
- Allocate memory for the task variables (file: MEMORY_ALLOCATION.INC, Listing 18)

```

;*****
;
; Description      8051 Assembler scheduler
;
;*****
;
; Filename:      8051SCH.ASM
; Company:
; Version:      Author:      Date:      Modification:
; 1.0          S Key        09/12/02   Created
; 1.1          S Key        30/05/03   separate files for easy of use
;
;*****
;
; Files required:
; REG515C.H, TIMING.INC, MEMORY_ALLOCATION.INC, vectors.inc
; ports.inc, task_periods.inc, TASK1.ASM, task2.asm, task3.asm
; task4.asm, task5.asm, task6.asm, task7.asm, task8.asm
;
;*****
;
; Notes:
; Base scheduler for 8051
; THIS FILE DOES NOT REQUIRE MODIFICATION
;
;*****/

$NOMOD51                ; disable predefined 8051 registers
$INCLUDE (REG515C.H)

;*****
; Sample prescale and reload values
;*****
$include (TIMING.INC)

;*****
; Memory Allocation
;*****
$include (MEMORY_ALLOCATION.INC)

;*****
; Vectors
;*****
$include (vectors.inc)

```

```

;*****
; Main Program
;*****
MAIN:

;*****
; Initialise Ports
;*****
#include (ports.inc)

; *** Initialise Ports
mov SP ,#0x30
clr SWDT
; *** Initialise timer for scheduler (using TMR2)
mov T2CON ,#0x10
mov CCEN ,#0x00
; *** The timer preload values.
mov TH2 , #PRELOAD10H
mov CRCH , #PRELOAD10H
mov TL2 , #PRELOAD10L
mov CRCL , #PRELOAD10L

; *** Enable Timer 2 interrupt, but not the external one.
setb ET2
setb EXEN2

; *** Initialise scheduler variables
mov SCH_run_f,#0x00 ;clear all run flags
mov SCH_index,#0 ;reset index counter

mov R1,#(SCH_rloads) ;get address of delay values
mov R0,#(SCH_delays) ;initialise indirect pointer

SCH_init_lp:
mov DPTR,#SCH_Delay_Values ;get delay value for task[index]
mov A,SCH_index ;get task index value
movc A,@A+DPTR
mov @R0,A ;store in array
inc R0
mov DPTR,#SCH_Reload_Values ;get reload value for task[index]
mov A,SCH_index ;offset pointer by number of tasks
movc A,@A+DPTR ;get task index value
mov @R1,A ;store in array
inc R1
inc SCH_index
mov A,SCH_index
cjne A,#SCH_MAX_TASKS,SCH_init_lp ;decrease index value, if not zero,
loop

; *** Enable interrupts and reset TMR0
setb T2I0 ; Set the timer running
setb EAL ; enable interrupts

SCH_Dispatch_Tasks:

; *** This is the main task dispatcher process. If a task is ready to be run
; then the dispatcher will run it and clear its run flag.
ORL PCON,#0x01 ; Enter idle mode (#1)
ORL PCON,#0x20 ; Enter idle mode (#2)
mov A,SCH_run_f ; test run flags
jz SCH_Dispatch_Tasks ; no tick so loop

mov SCH_dis_mask,#00000001b ;init value for mask, load
dispatcher run flag mask

mov SCH_dis_index,#SCH_MAX_TASKS ;get number of tasks in scheduler,
load dispatcher task index

SCH_Dispatch_Tasks_lp:
mov A,SCH_dis_mask ; get run flag mask
anl A,SCH_run_f ; logical AND with run flags

```

```

    jz     SCH_Dispatch_Tasks_end          ; if result was zero, so no run flag
there, go back to loop

; *** Task needs to be run, so clear run flag and jump to task
mov     A,SCH_dis_mask                    ;get inverse of run flag mask
cpl     A
anl     SCH_run_f,A                       ;clear appropriate run flag

mov     DPTR,#TASK_LIST                   ; load start of task lists
mov     A,#SCH_MAX_TASKS
subb   A,SCH_dis_index                    ; add current task no
clr     C                                 ; clear carry
rlc     A                                 ; multiply by two to get 'real' code location
jmp     @A+DPTR                            ; jmp to task
; *** This is where the scheduled tasks go list below in order (1 to 8)
TASK_LIST:
ajmp   TASK1                               ;goto task 1
ajmp   TASK2                               ;goto task 2
ajmp   TASK3                               ;goto task 3
ajmp   TASK4                               ;goto task 4
ajmp   TASK5                               ;goto task 5
ajmp   TASK6                               ;goto task 6
ajmp   TASK7                               ;goto task 7
ajmp   TASK8                               ;goto task 8

SCH_NO_TASK_HERE:                          ;DO NOT MOVE OR RENAME THIS LINE

SCH_Dispatch_Tasks_end:
mov     A,SCH_dis_mask                    ;roll run flag mask one place left
clr     C
rlc     A
mov     SCH_dis_mask,A
djnz   SCH_dis_index,SCH_Dispatch_Tasks_lp ; decrease task index , if not
zero more task to do
ajmp   SCH_Dispatch_Tasks ; super loop!

;*****
; Scheduler Functions
;*****

;*****
; Function:     SCH_Update
; Description:  Services TMR2 overflow interrupt and processes all tasks
; Pre:         None
; Post:        None
;*****

SCH_Update:
clr     TF2                                ; clear TMR 2 interrupt flag

push   ACC                                ; save registers that might be needed
push   PSW
push   01h
push   00h

; *** Setup loop counter and run flag mask
mov     SCH_run_mask,#00000001b          ;init value for run flag mask
mov     SCH_index,#SCH_MAX_TASKS        ; reset index value to number of
tasks in scheduler
mov     A,#SCH_delays
mov     R0,A                              ;get address of SCH_delays
mov     A,#SCH_rloads                     ; get location of first reload
ram location
mov     R1,A

SCH_Update_lp:
mov     A,SCH_run_mask                    ; get run flag mask
cjne   @R0,#0,SCH_Task_notready         ; test delay time if not zero task
not ready

```

```

; *** Current task is ready to be run, so set run flag and reload delay value
SCH_Task_ready:
    orl    SCH_run_f,A                ; set appropriate run flag
    mov    A,@R1                      ; get value being pointed to
    mov    @R0,A                      ; store in delay value (task period)
    AJMP   SCH_Update_lp_end          ; goto end of loop

SCH_Task_notready:                    ; task isn't ready to be run
    dec    @R0                        ; decrease delay value by one

SCH_Update_lp_end:
    inc    R0                          ; increase pointer value by one
    inc    R1
    mov    A,SCH_run_mask              ; roll run mask one place to left
    rl    A
    mov    SCH_run_mask,A
    djnz   SCH_index,SCH_Update_lp     ; decrease index value, not zero, so loop
still tasks to look at

    pop    00h                          ; restore registers
    pop    01h
    pop    PSW
    pop    ACC

    reti                                ; return from interrupt

;*****
*
; Task Name, Scheduled Task Delay and Reload Values
;*****
*

    $include (task_periods.inc)

;*****
*
; Scheduled Tasks
;*****
*

$include (TASK1.ASM)
$include (task2.asm)
$include (task3.asm)
$include (task4.asm)
$include (task5.asm)
$include (task6.asm)
$include (task7.asm)
$include (task8.asm)

end

;*****
*
; End of Program
;*****
*

```

Listing 16 Assembly-language scheduler for the 8051 microcontroller, main program (file: 8501SCH.ASM)

```

;*****
;
;   Description      8051 c515c Assembler scheduler
;
;*****
;
;   Filename:      TIMING.INC
;   Company:
;   Version:       Author:      Date:      Modification:
;   1.0            S Key        09/12/02  Created
;   1.1            S Key        30/05/03  separate files for easy of use
;
;*****
;
;   Files required:
;
;
;*****
;
;   Notes:
; This section deals with the timing of the scheduler, As set the
; tick is 1 ms vary these according to the settings below or work
; out your own. note some ACCURATE timings are not possible.
;
;*****/

;*****
; Sample prescale and reload values
;
; +-----+-----+-----+
; | PRELOAD10h | PRELOAD10l | time generated |
; +-----+-----+-----+
; | 0xBE       | 0xE5       | ~1ms @ 10 MHz  |
; +-----+-----+-----+
; | 0xB1       | 0xE0       | 1ms @ 12 MHz   |
; +-----+-----+-----+
;
;*****

SCH_MAX_TASKS equ (01h) ;Maximum number of tasks (MIN = 1)
                ;MIN = 1, MAX = 7 (depends on task variables)

; Oscillator / resonator frequency (in Hz) e.g. (11059200UL)
OSC_FREQ equ (10000000)
; Number of oscillations per instruction (6 or 12)
;OSC_PER_INST equ (6)
;PRELOAD10 equ (65536 - (OSC_FREQ / (OSC_PER_INST * 100)))
;PRELOAD10H equ (PRELOAD10 / 256) ;0xd7
PRELOAD10H equ (0xbe)
;PRELOAD10L equ (PRELOAD10 % 256) ; 0xd1
PRELOAD10L equ (0xe5)

;*****
; End of File
;*****

```

Listing 17 Scheduler 'tick' definition (file: TIMING.INC)

```

;*****
;
;   Description      8051 c515c Assembler scheduler
;
;*****
;
;   Filename:      MEMORY_ALLOCATION.INC
;   Company:
;   Version:      Author:      Date:      Modification:
;   1.0           S Key       09/12/02  Created
;   1.1           S Key       30/05/03  separate files for easy of use
;
;*****
;
;   Files required:
;
;
;*****
;
;   Notes:
; This section deals with the allocation of memory to scheduler
;and user variables. do not modify the scheduler variables,
;unless REALLY necessary.
;
;*****/

; DO NOT MODIFY THIS SECTION
;*****
; Scheduler Memory Allocation
;*****

START_OF_DATA_MEMORY EQU (00h)
START_OF_USER_MEMORY EQU (SCH_rloads + SCH_MAX_TASKS)
;miscellaneous variables

SCH_STATUS DATA (START_OF_DATA_MEMORY) ;scheduler status
register
SCH_temp DATA (START_OF_DATA_MEMORY + 01h) ;temp register for
Scheduler
temp DATA (START_OF_DATA_MEMORY + 02h) ;general temp register

;Scheduler variables

SCH_run_f DATA (START_OF_DATA_MEMORY + 04h) ;Run flags for
scheduled tasks
SCH_run_mask DATA (START_OF_DATA_MEMORY + 05h) ;Run flag mask
SCH_index DATA (START_OF_DATA_MEMORY + 06h) ;Current task index
SCH_dis_mask DATA (START_OF_DATA_MEMORY + 07h) ;Task Dispatcher run
flag mask
SCH_dis_index DATA (START_OF_DATA_MEMORY + 08h) ;Task Dispatcher
task index

;Delay values for scheduled tasks
; put in ram comments to the size of the number of tasks -> SCH_MAX_TASKS
SCH_delays DATA (START_OF_DATA_MEMORY + 09h)

;Delay values for scheduled tasks
; put in ram comments to the size of the number of tasks -> SCH_MAX_TASKS
SCH_rloads DATA (SCH_delays + SCH_MAX_TASKS)

;*****

; END OF DO NOT MODIFY

; user variables;
; define all user variables for start of user memory

;Scheduled Task One Variables
LED_Status DATA (START_OF_USER_MEMORY) ;led status register

;Scheduled Task Two Variables

```

```

; task_two_variable DATA (START_OF_USER_MEMORY + 01h) ;example
;Scheduled Task Three Variables

;Scheduled Task Four Variables

;Scheduled Task Five Variables

;Scheduled Task Six Variables

;Scheduled Task Seven Variables

;Scheduled Task Eight Variables

;*****
; End of File
;*****

```

Listing 18 Memory and register allocation (file: MEMORY_ALLOCATION.INC)

```

;*****
;
; Description      8051 c515c Assembler scheduler
;
;*****
;
; Filename:      vectors.inc
; Company:
; Version:      Author:      Date:      Modification:
; 1.0           S Key       09/12/02 Created
; 1.1           S Key       30/05/03 separate files for easy of use
;
;*****
;
; Files required:
;
;
;*****
;
; Notes:
;
;
;*****/
;*****
; Vectors
;*****

CSEG  AT      0x0000      ; reset
LJMP  MAIN      ; reset location (jump to start)

CSEG  AT      0x002b      ; timer 2 overflow
; *** There is only one interrupt in this
program...
ljmp  SCH_Update      ;timer has overflowed, so update scheduler
CSEG  AT      0x0043      ; after all interrupt vectors
;start of main program.

;*****
; End of File
;*****

```

Listing 19 8051 C515C Vector table (file: VECTORS.INC)

```

;*****
;
;   Description      8051 c515c Assembler scheduler
;
;*****
;
;   Filename:      ports.inc
;   Company:
;   Version:      Author:      Date:      Modification:
;   1.0           S Key       09/12/02  Created
;   1.1           S Key       30/05/03  separate files for easy of use
;
;*****
;
;   Files required:
;
;*****
;
;   Notes:
;   This section deals with the port settings. I find it easier to
;   give the port a name such as LED_PORT. Then if you port to a
;   different device you only need to change the port description
;   in this file
;
;*****/

; *** Initialise Ports

LED_PORT equ    (P0.0)

;*****
; End of File
;*****

```

Listing 20 Port definition file (file: PORTS.INC)

```

;*****
;
;   Description      8051 c515c Assembler scheduler
;
;*****
;
;   Filename:      task_periods.inc
;   Company:
;   Version:       Author:      Date:      Modification:
;   1.0            S Key        09/12/02  Created
;   1.1            S Key        30/05/03  separate files for easy of use
;
;*****
;
;   Files required:
;
;*****
;
;   Notes:
;   This section deals with the periods of all the task, enter the
;   start delay and interval between the repeat of the tasks below
;
;*****/

;*****
; Scheduled Task Delay and Reload Values
;*****
cseg
SCH_Delay_Values:

    db    0x00          ;delay value for task 1
    db    0x00          ;delay value for task 2
    db    0x00          ;delay value for task 3
    db    0x00          ;delay value for task 4
    db    0x00          ;delay value for task 5
    db    0x00          ;delay value for task 6
    db    0x00          ;delay value for task 7
    db    0x00          ;delay value for task 8

SCH_Reload_Values:

    db    0x31          ;reload value for task 1
    db    0x00          ;reload value for task 2
    db    0x00          ;reload value for task 3
    db    0x00          ;reload value for task 4
    db    0x00          ;reload value for task 5
    db    0x00          ;reload value for task 6
    db    0x00          ;reload value for task 7
    db    0x00          ;reload value for task 8

;*****
; End of File
;*****

```

Listing 21 Scheduler task period definitions (file: TASK_PERIODS.IN)

```

;*****
; Scheduled Tasks
;*****

;*****
; Function:      LED_Flash
; Description:   Flashes LED on port0 pin0 on and off
; Pre:          None
; Post:         None
;*****

TASK1:

LED_Flash:
    mov    A,LED_Status          ; is led on?
    jz     LED_on                ; no got turn it on
    clr    LED_PORT              ; turn off
    mov    LED_Status,#00        ; set status
    ajmp   SCH_Dispatch_Tasks_end

LED_on:
    setb   LED_PORT              ;turn LED on
    mov    LED_Status,#01        ;set status
    ajmp   SCH_Dispatch_Tasks_end ;Return from scheduled task

;*****
; End of File
;*****

```

Listing 22 Task1: Flashing LED (file: TASK1.ASM)

```

;*****
; Scheduled Tasks
;*****

;*****
; Function:
; Description:
; Pre:        None
; Post:       None
;*****

TASK2:
    ajmp   SCH_Dispatch_Tasks_end ;Return from scheduled task

;*****
; End of File
;*****

```

Listing 23 Task2: Empty task file (file: TASK2.ASM - TASK8.ASM)

References and further reading

- Allworth, S.T. (1981) “An Introduction to Real-Time Software Design”, Macmillan, London.
- Atmel (2002) “AVR Instruction Set”,
http://www.atmel.com/dyn/resources/prod_documents/DOC0856.PDF
- Bate, I. (2000) “Introduction to scheduling and timing analysis”, in “The Use of Ada in Real-Time System” (6 April, 2000). IEE Conference Publication 00/034.
- Cooling, J. (2003) “Software engineering for real-time systems”, Addison-Wesley, UK

- Cullyer, W J, Goodenough, S J and Wichmann, B A "The choice of computer language for use in safety-critical systems," Software Engineering Journal, Vol. 6, No. 2, March 1991.
- Infineon (1997) "C515C 8-Bit CMOS Microcontroller User's Manual 11.97",
http://www.infineon.com/cgi/ecrm.dll/ecrm/scripts/public_download.jsp?oid=8031&parent_oid=13734
- Infineon (2000) "C500 Architecture and Instruction Set",
http://www.infineon.com/cgi/ecrm.dll/ecrm/scripts/public_download.jsp?oid=27537&parent_oid=-8136
- Kopetz, H. (1995) "The Time-Triggered Approach to Real-Time System Design", in Predictably Dependable Computing Systems, Springer, Berlin, 1995
- Microchip (1997) "PICmicro® Mid-Range MCU Family Reference Manual",
<http://www.microchip.com/download/lit/suppdoc/refernce/midrange/33023a.pdf>
- Nissanke, N. (1997) "Realtime Systems", Prentice-Hall.
- Pont, M.J. (2001) "Patterns for time-triggered embedded systems: Building reliable applications with the 8051 family of microcontrollers", ACM Press / Addison-Wesley, UK
- Pont, M.J. (2002) "Embedded C", Addison-Wesley, UK.
- Storey, N. (1996) "Safety-critical computer systems", Prentice Hall, UK
- Styger, E. "The Usage of C++ for 8 Bit, 16 Bit and 32 Bit MCU's Compared with C and Assembly", Embedded Systems Conference Europe in 1999 (Spring)
- Ward, N. J. (1991) "The static analysis of a safety-critical avionics control system", in Corbyn, D.E. and Bray, N. P. (Eds.) "Air Transport Safety: Proceedings of the Safety and Reliability Society Spring Conference, 1991" Published by SaRS, Ltd.