

Embedded Systems

Laboratory 5 – Operating System¹

1 Introduction

1.1 Goals

After this laboratory exercise, you should have deepened your understanding in how to use C to perform low-level programming, understand how context switch in a simple multithreading kernels can be done and understand how multiple threads can simplify programming.

1.2 Literature

Study the following literature before starting the lab homework or exercises:

- Chapter 6.1-6.5 in: Wolf, Computers as Components.

2 Preparations

Read the literature and this laboratory exercise in detail, and solve the home assignments. Note that you must solve the home assignments, or you will not be allowed to start the laboratory exercise.

You must also have done lab 2 and 4 before being allowed to perform this lab.

All programs described in this lab manual can be found at the course homepage.

2.1 Home Assignment 1 – Polling in C

Study the program `dancepolling.c`, which can be seen below.

```
1
2 #include <stdio.h>
3
4 short *outAddress = (short*) 0x70001004;
5 volatile short *controlAddress = (short*) 0x70001008;
6
7 #define MAX 64
8 #define LOOP_TIME 50
9
10 int values[MAX] = {
11 0x00, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01,
12 0x00, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x81, 0x02, 0x04, 0x10, 0x20, 0x41, 0x82, 0x04,
13 0x08, 0x10, 0x21, 0x42, 0x84, 0x08, 0x10, 0x21, 0x42, 0x84, 0x08, 0x11, 0x22, 0x44, 0x89, 0x12,
14 0x25, 0x4B, 0x97, 0x2F, 0x5F, 0xBF, 0x7F, 0xFF, 0x00, 0xFF, 0x00, 0xFF, 0x00, 0xFF, 0x00, 0xFF
15 };
```

¹ Edited by Rene Krenz and Ingo Sander based on the original version of Mats Brorsson and Nguyen Phan Nguyen Thai

```

16
17 int index = 0;
18 int loop = 0;
19
20 void dance() {
21     *outAddress = values[index];
22     index = (index+1) % MAX;
23     if (index == 0) loop++;
24 }
25
26 int main () {
27     while (loop < LOOP_TIME) {
28
29         /* Add your code here */
30
31         dance();
32     }
33     return 0;
34 }
35

```

As written, the program plays with dancing LEDs on the I/O-module. It cycles a number of times through the whole dancing procedure.

Add polling code on line 29 where it says, “Add your code here” so that the dance routine is called only when one of the buttons is pressed or when the timer signals. You should first wait for the button (timer) to be pressed and then wait for it to go back to low. You can check both buttons and the timer with one single statement. *Tip:* Look at lab 2 for help on how the polling can be done and for information about the I/O module.

2.2 Home Assignment 2 – Interrupt programming in C

Study the following C-code (danceint.c), which is supposed to be a variant of the program in home assignment 1. There are some important parts missing, though.

```

1
2 #include <stdio.h>
3 short *outAddress = (short*) 0x70001004;
4 short *controlAddress = (short*) 0x70001008;
5
6 #define MAX 64
7 #define LOOP_TIME 5
8
9 int values[MAX] = {
10 0x00,0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01,
11 0x00,0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x81,0x02,0x04,0x10,0x20,0x41,0x82,0x04,
12 0x08,0x10,0x21,0x42,0x84,0x08,0x10,0x21,0x42,0x84,0x08,0x11,0x22,0x44,0x89,0x12,
13 0x25,0x4B,0x97,0x2F,0x5F,0xBF,0x7F,0xFF,0x00,0xFF,0x00,0xFF,0x00,0xFF,0x00,0xFF
14 };
15
16 int index = 0;

```

```

17 int loop = 0;
18
19 __inline void enable_IF(void)
20 {
21 int tmp;
22 __asm
23 {
24     MRS tmp, CPSR
25     BIC tmp, tmp, #0xC0
26     MSR CPSR_c, tmp
27 }
28 }
29
30 __inline void disable_IF(void)
31 {
32 int tmp;
33 __asm
34 {
35
36     /* Add your code here */
37
38 }
39 }
40
41 void dance() {
42     *outAddress = values[index];
43     index = (index+1) % MAX;
44     if (index == 0) loop++;
45 }
46
47 __irq void IRQHandler(void)
48 {
49
50 /* Add your code here */
51
52 }
53
54 extern int exit(int);
55
56 /* This function constructs a branch instruction to
57 be put in the place of the interrupt vector */
58 unsigned int Install_Handler( unsigned routine, unsigned *vector )
59 {
60     unsigned vec, oldvec;
61
62     vec = ((routine - (unsigned)vector - 0x8) >> 2 );
63     if( vec & 0xff000000 ) {
64         printf( "Installation of exception handler failed\n" );
65         exit(1);
66     }

```

```

67
68  vec = 0xea000000 | vec;
69  oldvec = *vector;
70  *vector = vec;
71  return (oldvec);
72 }
73
74 int main () {
75     unsigned int old_int_handler;
76     unsigned *irqvec = (unsigned*) 0x18;
77
78     old_int_handler = Install_Handler((unsigned)IRQHandler, irqvec);
79     enable_IF();
80
81     while (loop < LOOP_TIME);
82
83     disable_IF();
84
85     *irqvec = old_int_handler;
86     return 0;
87 }
88

```

There are some important additions to a normal C program, which we discuss here. Study the functions `enable_IF` and `disable_IF` at lines 19 and 30. They use two modifications to normal C behaviour. At the function definition, the use `__inline`. This means that the compiler should not use normal function calls to invoke these functions but rather take the body of the function and put it at the place where it is called, i.e. *in line* where it is called. The program is then generally more efficient as we remove the function call overhead and we also get longer basic blocks.

The second addition can also be seen in these functions. The key word `__asm` denotes that the code in the structured block following the key word is assembler file and not C. This can be used to perform low-level programming without using separate assembler programs.

Finally there is an addition on line 47. The key word `__irq` is used to denote that the function `IRQHandler` is an interrupt handling routine for the interrupt `IRQ`. This is needed as normal functions do not perform the bookkeeping needed by interrupt routines. The `__irq` keyword:

- preserves all APCS corruptible registers.
- preserves all other registers (excluding the floating-point registers) used by the function.
- exits the function by setting the program counter to $(lr - 4)$ and restoring the CPSR to its original value.

The function `Install_Handler` on line 58 installs a function given as argument as interrupt handler for the interrupt vector given as the second argument. `IRQ` has vector `0x18` so this function writes a branch-instruction at address `0x18` with a branch to the interrupt handler routine.

The task of this homework is:

- What does the function `enable_IF` do?
- Add the code needed to make `disable_IF` complete.
- Add the code needed in `IRQ_handler` so that the program has the same functionality as the program in home assignment 1 except that interrupts are used instead of polling.

2.3 Home Assignment 3 – Prime number generation

Study the following program so that you understand what it does.

```

1  #include <stdio.h>
2
3  #define NUMPRIME 1000
4
5  int NextPrime(int start_value) {
6      int i, res, val, max, found;
7      found = 0;
8      val = start_value+1;
9      while (!found) {
10         found = 1;
11         max = val / 2;
12         for (i = 2; i<=max; i++) {
13             res = val % i;
14             if (res==0) {
15                 found = 0;
16                 val = val+1;
17                 break;
18             }
19         }
20     }
21
22     return val;
23 }
24
25 int main() {
26     int i;
27     int prime = 1;
28     printf("Searching first %d prime numbers\n", NUMPRIME);
29     for (i=0; i < NUMPRIME; i++) {
30         prime = NextPrime(prime);
31         printf("%d(th) prime number is %d\n", i, prime);
32     }
33 }

```

2.4 Home Assignment 4 – A simple multithreaded kernel

Browse through the source code of the small multithreaded kernel *ITS Lite* below. It is written in both C and assembler, and has the following functions: Make sure that you have a good overview of the kernel-functionality.

- *ITS_init()*; Create the internal data structures of the kernel. This call must be made before the other kernel functions can be used.
- *ITS_create-thread(void (*entry), void *arg, unsigned int *stack)*; This call creates a thread. The first argument is the name of the function that contains the code for the thread. The second argument is a pointer to a data structure, which will be passed as an argument to the subroutine, and the third argument is a pointer to an array, which will be used as the stack space for the thread.
- *ITS_enter_critical()*; Call this function when the thread needs to enter a critical region.
- *ITS_leave_critical()*; Call this function when the thread needs to leave a critical region.
- *ITS_yield()*; Call this function when the thread wants to wait and leave the execution to another thread.

The kernel source code can be found in the location given on page 2 and also through links below (on the web version of this document). You can copy the file [lab5.zip](#) and unpack it somewhere in your own directory. This file also contains APM projects.

- [its.c](#): most of the kernel routines
- [its.h](#): declaration of kernel routines, to be included files where ITS Lite is used
- [itsasm.s](#): those kernel routines that are written in assembler, for example, saving registers and changing processor state
- [interrupt.s](#): routines for enabling and disabling interrupts
- [utilasm.s](#): routines to perform some low-level functions
- [util.h](#): declarations for the routines above
- [itsliteexempel.c](#): an example of how to use the ITS Lite kernel

3 A Simple Real-Time Kernel

Note that the “thaimulator”-module is needed for all assignments. Please check lab 2 and activate the module.

3.1 Assignment 1

Execute the program you completed in home assignment 1 and verify that it works as expected.

3.2 Assignment 2

Execute the program you completed in home assignment 2 and verify that it works as expected. Using the disassembly view, study how the function `Install_Handler` works and the code that the

compiler have generated for the interrupt handler. Verify that the inlined functions really are inlined. If not, try to compile with a higher level of optimization.

3.3 Assignment 3

Executed the program in home assignment 3 and verify that it works as expected.

3.4 Assignment 4

Build the program of home assignment 4. Report the output behaviour of the program when the different threads are executing. Remember to enable interrupts and timer in the simulated I/O module.

Why appear sometimes two or more A's and sometimes only one between the two longer printouts?

What is the purpose of the call to functions `its_enter_critical` and `its_leave_critical`?

3.5 Assignment 5

Take the polling LED-dancing program from home assignment 1 and the prime computing function from home assignment 3 and let them be one thread each in the multithreaded program example.

Remove the original threads (and change the main thread to not do any printouts).

Execute the program and verify that it works as expected.

4 Conclusions

Before you pass the laboratory exercise, think about the questions below and explain to your supervisor:

- Why are threads needed?
- What is a context switch?
- What is the actual state of the processor that must be saved for each thread?
- Why must the registers r0-r3 be saved at a context switch, since they don't need to be saved at a function call?
- Why do you need the `__irq` key word to write an interrupt handler in C?