Lab Course Manual TI2725-C

M. Dufour, A.J.C. van Gemund
Embedded Software Lab
Faculty of Electrical Engineering, Mathematics and Computer Science

University of Technology, Delft

January 2014

Preface

This course has survived several curriculum changes since its conception in 2006. Currently it's known as TI2725-C, but throughout this document you will find references to the original name IN2305 as a tribute to the developers (and nobody bothered to change all the links and documents mentioned in this manual :-).

Koen Langendoen
Delft, January 2014

The objective of the IN2305-II Embedded Programming lab course is to give hands-on experience with programming embedded programs which are time-critical, I/O heavy and concurrent. Because there are three IN2305 lab courses, this lab course is kept small intentionally. Spread over three blocks you will do a series of small assignments, which result in the design of a simple "cruise control" application.

This is the second edition of this manual, written for the second edition of the course IN2305-II. Most of the used hardware (motor-setups) and software (uC/OS/X32 port, debugger) are developed by the writers for this lab course. The X32 is the masters project of Sijmen Woutersen. To support the software mentioned above, he has expanded and improved it several times.

The writers hope that the result of their work will contribute to a lab course which is fun to do and will also be a good learning experience. We thank the people who have contributed to this result: Sijmen Woutersen, for developing the X32 soft core and tool set, which are used in other projects as well, and for helping debug the uC/OS/X32 port; and the teaching assistants Joost Heijkoop and Thomas Schaap for giving valuable feedback on this version of the manual.

M. Dufour and A.J.C. van Gemund
Delft, November 2006

Contents

1 Introduction

In this lab course you will develop a “cruise control” application step by step over the course of three days. A cruise control system enables the driver of a car to automatically maintain a certain speed (“cruising speed”) [1]. Depending on the wind, road friction and gravity (e.g. driving up a hill), the system will constantly adjust the engine (“controlling” the engine) such that the speed of the car will remain more or less constant (see Figure 1.1).

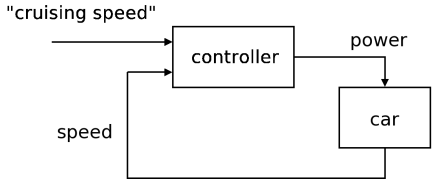

Figure 1.1: A cruise control system

To imitate the real life situation as close as possible, you will use “embedded hardware” at the lab course to drive a small motor, to which a small aluminum wheel is attached (see Figure 1.2). The effects of the wind, hills, etc. will have to be applied by yourself, by slowing down or speeding up the wheel manually.

The embedded hardware consists of an FPGA board with a few buttons, leds and a display on which numbers can be displayed. This FPGA board is connected to the motor and receives a signal from it which indicates the current speed of the motor (the “sensor path”); the FPGA sends back a signal which controls the power of the motor (the “actuator pad”). The buttons on the board are used to simulate the car control, including the cruise control function; the display is used to show the signals mentioned above.

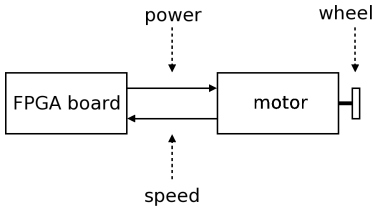

Figure 1.2: The hardware at the lab course

Embedded systems are traditionally programmed in C. In the cases where there are certain bounds on time (“real-time constraints”), real-time kernels are often used to make it easier to execute multiple tasks “concurrently” and to synchronize them, to meet these real-time requirements. Because programs are becoming simpler and are making explicit use of synchronization, they are also becoming more readable and easier to “port” to other platforms. At this lab course you will develop a C program, which makes use of a real-time kernel. The real-time kernel you will work with is uC/OS, the kernel which is discussed in the book used by the course.

2 Hardware

On the FPGA there is a 32-bits “soft core” (processor), called X32 [2]. Connected to the motor is a sensor (“encoder”) which transmits the current speed of the wheel to the FPGA board, and an actuator, with which the FPGA board can control the power of the motor (see Figure 1.2).

2.1 FPGA Board

The FPGA board which you will use is a Nexys-2 (Figure 2.1). The FPGA chip on this board consists of 500,000 configurable gates and runs on a clock speed of 50 MHz. Located on the board is a memory module with 16 MB RAM and a flash memory module which contains the current hardware description. Every time the FPGA board is reset the FPGA chip will be configured automatically according to this description.

The board contains several buttons, leds, switches and I/O ports (serial port, VGA out, PS/2 and some general ports), all of which can be "used" by the FPGA chip. Also, there is a "seven segment display" (SSD) on which you can display (a maximum of) 16-bits numbers. Then there is a USB connector to flash the memory with a new hardware description from a host PC.

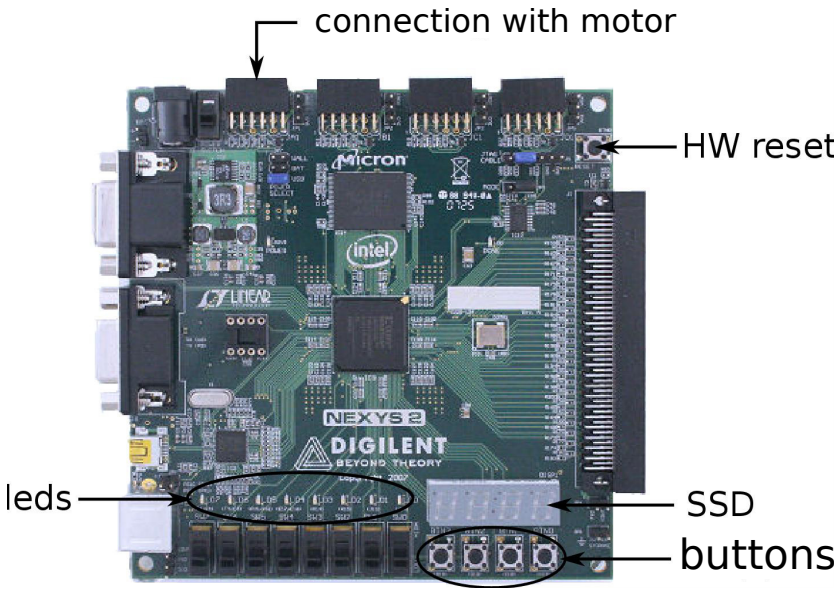

Figure 2.1: Nexys-2

2.2 X32 Soft Core

The X32 soft core is an experimental 32-bit processor, developed by a graduate student at the Embedded Software Lab [2]. It has a simple instruction set based on the "intermediate" bytecode format of the LCC C-compiler. This bytecode format is stack-based (cmp. Java and Python bytecode), which result in most operations being performed through a stack in memory, instead of registers. For example, to add two numbers, these numbers are first put on the stack, then an addition instruction replaces them by their sum.

The stack-based character of the instruction set results in requiring more instruction for simple operations, like addition, and constant communication with memory (because of the stack). The X32 is therefore not developed with performance in mind, but to simplify the software chain by directly executing bytecode. The speed at which instructions are executed on the used FPGA board, which runs at 50 MHz as mentioned, is about 4 MIPS (millions of instructions per second).

Because in real-time systems it is often important to react quickly to certain "events" (think of a "meltdown"), the X32 offers the possibility to assign a certain priority in software to each interrupt. Interrupts with a high priority can interrupt (pre-empt) low priority interrupts. The

ISR of the low priority interrupt is halted and will resume again if this interrupt has the highest priority of all "waiting" interrupts (interrupts can be "nested").

Appendix C offers an explanation on how to use the X32. For detailed information on programming for the X32 we refer to the X32 Programmers Manual [2].

2.3 Signals

The encoder sends two signals to the FPGA board, with which one can determine the speed and the direction of the rotating wheel (the "sensor pad"). The FPGA board send back a signal, which indicates the power of the motor (the "actuator pad"; Figure 1.2). If you look at the cable between the FPGA and the motor, you will see four pins (one is used as ground). The signals which are generated by the encoder are called `a` and `b`. Both are in fact the same signal, but shifted in phase (Figure 2.2). A phase difference of 90 or -90 degrees indicates the direction of the wheel. The encoder sends a pulse for each certain number of revolutions the wheel is turning. The speed of the wheel can be calculated by counting all pulses per time unit.

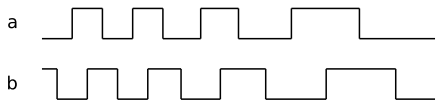

Figure 2.2: Example of `a` en `b` signals (during slowdown)

The X32 is set up to generate an interrupt on every "edge" of `a` and `b` (that is, every time `a` or `b` changes). During the lab you will make use of this fact to in a software "decoder". The decoder counts the number of (combined) pulse "cycles", and makes a light go on when an error occurs (for example, when an edge is missed).

At maximum speed there are only a few microseconds between consecutive interrupts. Because the X32 runs at about 4 MIPS, this means that the software decoder will not be able to keep up with the pulses at a certain speed and therefore turns on the led. That's why further in the lab, you will replace the software decoder by an existing hardware (VHDL) decoder, which is delivered with the X32 and which can easily keep up with the signal rate.

The signal the FPGA board sends back to the motor-setup is called `m`. This is a so called "pulse width modulated" (PWM) signal. A PWM signal consists of a block wave with a fixed period, but with a variable pulse width (Figure 2.3). Because it is too "expensive" to realize a (high frequency) PWM signal in software (like decoding signals `a` and `b`), you will use a ready made PWM generator, which is also delivered with the X32.

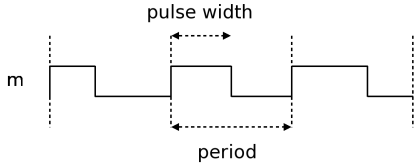

Figure 2.3: Example of m signal (during speed up)

Appendix C contains an explanation on how to use both signals and how to use the mentioned hardware components.

3 Software

During the lab you will use the uC/OS real-time kernel, which is adjusted ("ported") to run on the X32 [4]. You will be given a ready-made "monitor" task, with which you can monitor and even influence your uC/OS applications. Also, a more advanced monitor is given in the form of a debugger.

3.1 uC/OS

uC/OS [4] is the real-time kernel as is discussed in the book which is used in the course. It offers the typical functionality one expects of a simple real-time kernel, like multitasking, a "delay" function, semaphores, queues and mailboxes.

uC/OS is limited on two important points. Firstly, you cannot give the same priority to two tasks. This allows the kernel to easily determine which task should currently be executed. Secondly, uC/OS has virtually no facilities to handle interrupts. Because of this you are forced to do interrupt handling with the standard (low-level) X32 library, and indicate in each interrupt service routine (ISR) that uC/OS can temporarily not perform context switching (context switching is undesirable in a ISR [6]).

Appendix D provides details on how to use ISRs in uC/OS applications and gives an overview of the uC/OS functions which you can use during the lab course.

3.2 Monitor Task

A monitor enables you to look into the memory of a program and perhaps changing it at certain locations while the program is running. It also contains a "sense" function, with which you can see how a program variable changes over time.

A monitor operates on (unsigned) byte level in principle. To view a variable in memory, it would therefore be convenient to know how many bytes a certain variable occupies and how (signed) values are stored. Next to that, a monitor has no knowledge of the original C code. This means you will have to find out the addresses of the variables yourself, for instance by adding "debug output" to your program. This is an overview you can generate during compilation which, among other things, tell you where each C variable is located in memory.

A monitor can be combined with programs in different ways. However, in the case of a real-time kernel it is more obvious to use a low-priority task for this. The RT-kernel will make sure that the monitor will only be executed when there are no other tasks to perform.

In the lab course you will be given a ready-made monitor task. You will use this to optimize a certain number of variables (parameters of the controller), without having to recompile and upload the program each time you change a value.

3.3 X32 Debugger

Although a monitor can be useful in simple situations, it will turn out to be awkward and slow to use it. Some of the problems are that it is very low-level, you must relate to the C source code yourself and a monitor can use quite some memory itself (because it also runs on the embedded hardware). Also, although you can view and change the memory, you would rather like to view the program itself and influence it (more directly).

A "debugger" is a more advanced tool, which enables you to "freeze" the execution of a program when it reaches a certain instruction (a "break point"). This enables you to execute a program in a controlled way, for instance by "jumping" from one C line to another. Each time the program is paused, you can relax and look at what has changed in memory (cmp. a monitor).

A good debugger also has some other important properties. Firstly, one can use this with the original C code. This means that the "current location" of the program is given and that you can

just use the name of the variable without having to know its address. Secondly, a good debugger can evaluate C expressions (including assignments). This means that you can simply change the value of a variable with a command like `a = *b+2`.

During the lab course you could use a graphical debugger, which is developed to debug C programs on the X32 [5]. This debugger runs "remotely" on the host PC and communicates with the X32 bootloader to set break points and to read/write in memory. It is able to evaluate C expressions (including assignments) and it can display assembly. Next to that, you can graphically enable/disable break points by clicking in front of the line of C code.

Appendix A contains an explanation on how to start and use the X32 debugger.

4 Assignment

We describe the lab assignment with the following diagram, which outlines what the final system should look like:

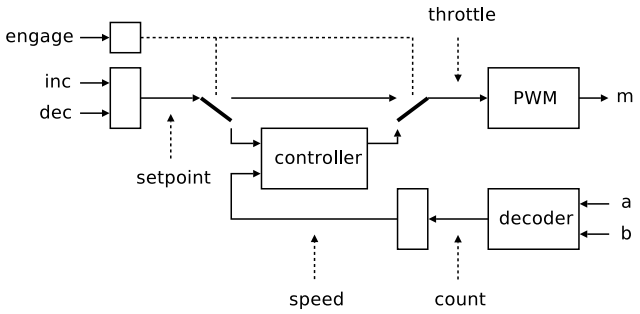

Figure 4.1: A diagram of the final system

The assignment consists roughly of two parts: implementing a controller (center of the diagram) and handling the interface with the driver (the three controls on the left in the diagram).

The controller controls the signal which goes to the motor setup (through `throttle`), based on a user given cruising speed (`setpoint`) and the current speed of the wheel (`speed`). To determine the current speed, you will have to write a software"decoder", which keeps track of the number of `a` en `b` cycles (`count`). The `speed` is equal to the change of `count` per uC/OS time slice. Because the software decoder will prove not to be quick enough, you will eventually replace this by a hardware (VHDL) decoder.

Handling the interface forms an important part of the assignment, especially because the X32 version we use at the lab course does not have a built in functionality for "debouncing" buttons. Because of this, you will have to implement this yourself in software in a "correct" way. An incorrect way of implementing this is by "polling" (reading periodically), because this results in an unnecessary CPU load. Also, it shouldn't go wrong when the driver pushes multiple buttons at the same time or quickly after another. You will solve this last problem by handling different buttons "in parallel".

A second reason why the interface is an important part of the assignment, is that you will also have to build an "autorepeat" function. To keep it simple to adjust the speed, this function automatically increases or decreases the `throttle` every uC/OS time slice. The function will be activated when the driver presses a certain button for more than 500 ms and deactivates when the button is released.

To make it easier for the teaching-assistants to review your work, you will have to name the variables in the same way they are named in above diagram; then there are some precise demands on the interface to the driver. From left to right the four buttons on the FPGA board (Figure 2.1) should have the following functionality:

- A “reset” function; this shuts down the motor, resets the SSD and quits the program.
- A “(dis)engage” function; this will enable or disable the cruise control mode. When it is enabled, `setpoint` will equal `speed`.
- A “decrement” function; this will lower `throttle`, or `setpoint`, when cruise control mode is enabled.
- An “increment” function; this will increase `throttle`, or `setpoint`, when cruise control mode is enabled.

The left most led indicates an error while processing signals `a` en `b`. In this case the led will be on for a duration of 100ms, after which it will automatically go off again. The led next to it indicates whether the program is in cruise control mode or not.

On the left and right hand side of the SSD respectively `speed` en `throttle` are displayed, both shifted to the right to fit into 8 bits.

5 Day 1

During the first day you will develop the basic framework for the application, to which you will add debouncing and a cruise control mode on the second and third day respectively. This means that after this day the application will respond to pressing buttons, the sensor and actuator pads work (signals can be sent and received to and from the motor setup), that these signals are shown on the SSD and that the “error” led temporarily goes on when an error occurs on the sensor pad.

Before you can start with the lab, you need to log in to your GNU/Linux account, download the software and install it. Appendix A contains information on how to install the lab software and an overview of programs you can use after installation. Appendices B, C and D contain tutorials on how to use UNIX, the X32 soft core and uC/OS respectively and an overview of the most important functions/commands. From now on, we assume that you have thoroughly read these appendices.

Assignment 1a After finishing this assignment you can control the `throttle` by means of the buttons; this is displayed on the SSD (by a periodical uC/OS task), and sent in the form of a PWM signal to the motor.

- Write a program which exits to the bootloader the moment the “reset” button is pressed. This means you should define a ISR `isr_buttons`, in which you call the standard C `exit` function.
- >From the ISR, keep an (integer) `throttle`, which can be decreased or increased by pressing the “decrement” en “increment” buttons and display this on the SSD. (You could use the standard C `printf` function as an intermediate step.) The problem of bouncing should become clear now!
- Create a periodical task which writes `throttle` to the SSD every uC/OS time slice (20 ms). Delete the displaying code from the ISR to keep its size as small as possible.
- Update the PWM pulse width in the periodical task with the current value of `throttle`. Set the period of the PWM signal to 1024 at the start of the program (this corresponds to 1024 clock-cycles of the FPGA chip; see Appendix C).

Assignment 1b At the end of this assignment the application will determine `speed` by using a software decoder (Section 2.3), and displays this on the SSD. The error led will temporarily switch on when a problem occurs on the sensor path.

- Define an ISR `isr_decode` which is called when `a` or `b` changes (Appendix C). Use the code from Appendix E. The ISR reads the current stat of `a` and `b` and decreases or increases `count`, depending on the direction of the wheel. Give a high priority to these interrupts. (Because these signals could have a high frequency it is important to react fast.)
- Let the periodical task evaluate `speed` and display it on the left part of the SSD. (Both the actuator pad and the sensor pad should be visualized now.)
- Create an "error" task, which lights the error led for 100 ms, when there is a problem with `a` and `b`. (This could lead to wrong counting, which can result in wrong commands being sent to the motor.) A good indication for this is the ISR detecting a state transition where both signals change *simultaneously.* Use a semaphore for the communication between the ISR and the task and make sure this is reset when the led is turned off again. (During an error it is likely for the ISR to be called a number of times.) For this, use the code as follows (we will see a better solution later on):

  ```
  while(OSSemAccept(sem_err) > 0); // reset semaphore
  ```

  Verify that the error led goes on (and off again!) when you switch the motor setup on or off (when doing this, illegal state transitions always occur).
- Increase `throttle` and notice that the error led will turn on at a certain moment. At this moment the ISR (written in C, on a slow processor!) is too "slow" to process all events `a` en `b`. This leads to the program only processing these (high-priority) signals and the buttons and the SSD will be stuck.
- Stop the wheel (so that the frequency of `a` en `b` decreases) and decrease `throttle` in the mean time, until the program runs normally again.

6 Day 2

This second day you will replace the software decoder with the hardware decoder and after this you will implement the debouncing of the buttons.

Assignment 2a First of all, we still need to make use of the hardware component for decoding `a` and `b`. After this assignment the error led should only go on while switching on/off the motor setup.

- Replace the decoder written in C by the delivered VHDL decoder (Appendix C). Use the "error" interrupt it generates to control the error led.

Assignment 2b After this assignment the buttons will be debounced in parallel, but will have the same (ugly) while loop like the error task.

- Create a separate task for each button (if you have previous C experience, or if you like to give yourself a hard time, then you can combine the code for these tasks into one "task body"); define as many semaphores for the communication between the ISR and the tasks.
- let `isr_buttons` wake the task for a certain button when this button is **pressed** *or* **released**.

- Use the uC/OS delay function to wait for *at least* 20 ms (the length of one uC/OS time slice), before a task reads the state of the button. The bouncing time for the used FPGA board is always shorter, making sure the correct value is always read.
- Use the solution (while loop) which we used before for the error task, to counter the effect of bouncing.

Assignment 2c While the while loops solve the bouncing problem, especially in real-time systems you don't want to create unnecessary overhead. Repeatedly increasing a semaphore only to reset it right after is therefore not a nice solution.

Another way to counter bouncing is to temporarily disable the (hardware) interrupt for the buttons. Having a correct solution using this is however difficult, since all buttons share the same interrupt, so it can go wrong when multiple buttons are pressed (almost) simultaneously.

A correct, nice solution "simulates" in software a separate interrupt for each button. The ISR keeps an "interrupt enable flag" for each button and disables it the moment the button is pressed; the associated task will re-enable it when it has finished the debouncing process.

- Define an (integer) "interrupt enable register" and reserve inside it a separate "interrupt enable flag" (bit) for each button.
- Use the C logical operators (^, &, | and ~) to determine which buttons are pressed or released (compared to the previous situation). Make sure the corresponding interrupts are then disabled.
- Use the corresponding semaphores to awaken the task which belong to the buttons which are "newly" pressed or released. Make sure these tasks re-enable the proper interrupt flag, when the debouncing process is done.
- You can also replace the while loop in the error task. The overhead of this is however less problematic, because errors should not occur anyhow.

7 Day 3

During this last day you will add an "autorepeat" feature for the "increment" and "decrement" buttons, to make it easier adjusting `throttle`. After this you will implement the controller. You will do this using a general control algorithm, the so called PID-algorithm. You will tune the three parameters (P, I and D) using a monitor task (section 3.2). Finally, using `gnuplot` you will make a figure of the behavior of the controller to show it is working correctly.

Assignment 3a By the end of this assignment you can, by pushing the "decrement" or "increment" button for more than 500 ms, automatically increase or decrease `throttle` each time slice ("autorepeat").

- Define two global variables for the state of the "decrement" and "increment" buttons after debouncing and update these from the corresponding tasks.
- Define two tasks with corresponding semaphores. Make sure that the right task is awoken at the moment that the 500 ms starts.
- Make sure the task waits 500 ms (as long as the button remains pressed) and that `throttle` is automatically adjusted each time slice after that.

Assignment 3b After this assignment you will have a working (though simple) controller with the associated functionality of the buttons and a led which indicates that the application is in cruise control mode or not.

- Add a cruise control mode. Let the cruise control led indicate the application being in cruise control mode. Store the current speed when cruise control mode is switched on (`setpoint = speed`).
- Define a cruise control function and call this from the periodic task. Add a simple controller to this function, for example as follows:

  ```
  eps = setpoint - speed;
  throttle += eps/4;
  ```

  Verify you have a working controller by playing (gently) with the wheel.
- Make sure that in cruise control mode you can adjust `setpoint` with the "decrement" and "increment" buttons. For safety reasons, the autorepeat function should be disabled in cruise control mode.

Assignment 3c After this assignment you have replaced the simple controller by a more advanced controller. You have tuned the three (PID) parameters of this controller using a monitor.

- Replace the simple controller by the following, more generally applicable controller. Use integers for the parameters (note that the X32 does not support floating point numbers).

  ```
  eps = setpoint - speed;
  throttle += (P * ((eps - last_eps) + I * eps +
  D * (eps - 2 * last_eps + lastlast_eps)))/64;
  if (throttle < 0) throttle = 0;
  else if (throttle > 1024) throttle = 1024;
  lastlast_eps = last_eps;
  last_eps = eps;
  ```
- Add the monitor (section 3.2) to the application: "include" the header file `monitor.h` and add a low priority "monitor task" which calls the function `run_monitor` (see `monitor.c`). Restart the application and familiarize yourself with the available commands. (Note: the X32 debugger can not operate simultaneously with the monitor!)
- Search for the physical address of the `throttle` variable in the `cc.dbg` file and "sense" the 4 bytes at this address while playing with the "decrement" and "increment" buttons.
- Adjust the PID variables using the monitor task. Start with P, then look at the effect of I and eventually D.

Assignment 3d After finishing this assignment you will be able to compare the behaviors of the simple controller and the advanced controller using the well known UNIX `gnuplot` program.

- Use `printf` to print four numbers every 20 ms from the periodic task: an indication of time, `setpoint`, `throttle` and `speed`. The output of the application will be as follows:

  ```
  0 18 120 100
  1 18 67 80
  ..
  ```

- Test the (simple and advanced) controllers for several seconds and use "standard output redirection" (Appendix B) to store the generated output in two files. Delete any "unwanted" lines from these files.
- Use a `gnuplot` script (Appendix B) to graphically display the data in each file using the following commands:

```
plot "filename" using 1:2 title "setpoint" with lines, \
"filename" using 1:3 title "throttle" with lines, \
"filename" using 1:4 title "speed" with lines
pause -1
```

References

[1] K. Nice, *How Cruise Control Systems Work*
http://auto.howstuffworks.com/cruise-control.htm

[2] S. Woutersen; *X32 Programmers Manual (Draft)*
http://x32.ewi.tudelft.nl/x32progman.pdf

[3] A. van Gemund, *IN2305 Resource Page*
http://www.st.ewi.tudelft.nl/~koen/ti2725-c/

[4] M. Dufour, *Running uC/OS on the X32 Soft Core*
http://www.st.ewi.tudelft.nl/~koen/ti2725-c/ucosx32.pdf

[5] M. Dufour, *X32-debug: A Remote Source-Level Debugger for the X32 Soft Core (draft)*
http://www.st.ewi.tudelft.nl/~koen/ti2725-c/x32db.pdf

[6] D.E. Simon, *An Embedded Software Primer*, ISBN 020161569, Addison-Wesley

[7] M. Stonebank, *UNIX Tutorial for Beginners*
http://www.ee.surrey.ac.uk/Teaching/Unix

[8] G. van Rossum et al., *The Python Programming Language Website*
http://python.org

[9] T. Williams et al., *The Gnuplot Homepage*
http://www.gnuplot.info

A Lab Course Software

This appendix describes how to install and use the lab course software. See Appendix B for information on the used UNIX commands.

A.1 Installation

To install the lab course software, you'll first need to log in to your GNU/Linux account. Download the in2305_StudPackage_1.0.0.tgz "tarball" from the IN2305 Resource Page [3] and unpack this using the following command:

```
tar -zxf in2305_StudPackage_1.0.0.tgz
```

The lab course software is now installed into a newly created in2305 directory.

A.2 Usage

To be able to use the software, enter the `in2305` directory and "source" a script containing lab course settings:

```
cd in2305
source setup_in2305
```

Repeat this every time you want to use the software using another shell or when you log in on your account!

When you'll use the `ls` command, you will see the following directories, among others:

`x32`: the X32 soft core, including bootloader program (VHDL)
`x32-tools`: X32 compiler tools and instruction-set simulator (C)
`ucos`: X32 version of the uC/OS real-time kernel (C)
x32-debug: the graphical, C-level X32 debugger (Python)
`doc`: X32, uC/OS and X32 debugger documentation (PDF)

In the `cruise` directory several preparations are made for the cruise control application. These assume that you will use a file named `cc.c`. Use your favorite UNIX editor to create this file (if you haven't got a favorite editor you can use `kate`):

```
cd cruise
kate cc.c
```

When you want to test your program during the lab course, you can compile it, upload it to the FPGA, start a terminal and start the program, all with a single "make" command:

```
make cc
```

When something goes wrong during this process (a program freezes for example), it is often a good idea to manually reset the FPGA board (always turn off the motor setup first). In addition, the X32 is made in such a way that you can issue a software reset by pressing all four buttons at once (figure 2.1).

A.3 X32 Tools

To finish the lab you don't need any specific X32 commands. But to be complete, here you'll find an overview of implicitly used (through `make`) and other specific X32 commands.

- `x32-term`: A simple terminal program, which serves as an alternative of minicom (Appendix B). It contains little functionality, but enough to be used in the lab course. If you give an "`s`" argument, it will automatically issue the "start program" command to the X32 bootloader (Appendix C).
- `x32-upload`: This program communicates with the X32 bootloader to upload an executable (`*.ce`) file (by default from address 0). You can let the program start automatically or you can use a terminal program to issue the "start program" command manually.
- `x32-debug`: This command will start the X32 debugger (section 3.3). Argument to the program is the name of an executable (`*.ce`) file. The debugger runs `make` (to be sure) for the file, uploads it to the FPGA, sets an initial break point at the `main` function and starts the program.

(To use the debugger at home, you will need to install the following "dependencies" next to the lab course software: `python`, `pyqt`, `qscintilla` and possibly `pyqt-qscintilla`.)

- `x32-sim`: This is an instruction-set simulator for the X32. An instruction-set simulator is very useful for developing a processor itself (for applications you're better off with a monitor or debugger). A simulator can also be of use when you don't have an FPGA board at home. You will not be able to communicate with the hardware (like buttons, the SSD and the motor setup), but you're still able to test certain aspects of your program.
- `lcc-x32`: This is a version of the LCC compiler, modified to generate X32 bytecode (normally, this is only used internally) and debugging output for the X32 debugger. The LCC compiler has similar command-line options as GCC. A few of these are present in the `cruise/Makefile`; a complete overview can be seen when it is executed without any arguments.

B UNIX Tutorial

GNU/Linux is a modern variant of the old UNIX operating system. In this appendix you will find an overview of the UNIX commands used at the lab course and a few other useful commands. Also, an introduction on "redirecting" output of these commands is given.

B.1 UNIX Commands

The average UNIX computer contains an abundance of useful tools. A great number of these tools is standardized, including the way arguments/input is given; many other tools (like `gnuplot`) have a single source, making it able to run on most UNIX systems in the same way.

The following commands are needed during installation of the lab course software, during the lab course itself or are just useful. See [7] for an extensive UNIX tutorial.

- `cd`: With this you can change your current directory (cmp. `cd` in DOS).
- `ls`: Print a list of files in the current directory (cmp. `dir` in DOS); by using the "-l" option you will see several other properties of these files, like size.
- `cat`: Prints the contents of one or more text files after another to the screen (cmp. `type` in DOS).
- `less`: Prints the contents of a text file, but at every page you're asked to press space, for example, to go on and you can scroll backwards.
- `grep`: With this tool you can search the contents of one or more text files for a certain string (or regular expression). By default, these lines are printed, preceded by the name of the associated file, but the output can be adjusted in a variation of ways. The following options are quite useful:

  `-R`: recursively search directories (for example: `grep -R .`).
  `-i`: ignore differences between capitalized and non-capitalized letters
  `-l`: print only names of files containing the string/expression, not the matching lines
  `-v`: show only files/lines which do *not* contain the string/expression
  `-n:` print the line number in front of each line
- `diff`: Prints the "difference" between two text files. This is useful for comparing large (program) files with each other. The difference can be stored in a file (section B.2) and can later on be "applied" again, although the original file is changed on other locations in the mean time. This makes it very useful for (online) cooperation. Often the option "-u" is used to print out the differences in a standardized, readable format.

- `tar`: Pack and unpack files, using compression or not. The "-z" option indicates `gzip` (another UNIX tool) should be used to (de)compress. The "-c" and "-x" options are used to create or to extract respectively. The "-f" option indicates a file name will follow. For example, to pack everything from the current directory and to unpack it again you can use the following commands:

  ```
  tar -zcf file.tgz *
  tar -zxf file.tgz
  ```

- `source`: With this tool you can execute a script in the current shell (normally, a separate shell is started before executing a script). This means that "environment variables" are kept after executing the script.

- `man`: Will display a manual of a UNIX command. For example: `man diff` will give a description of the `diff` command and gives an overview of all available options.

- `make`: A very useful tool, with which you can specify for, for example, C/C++ projects how some parts depend on others and how it has to recompile whenever these parts have changed. The usage of `make` can save a lot of time (and key strokes) because programs are now compiled "incrementally" and you'll need less (low-level) commands. To be able to use `make`, you have to create a file named `Makefile` in the current directory with the following contents:

  ```
  cc.ce:  cc.c monitor.c
  lcc-x32 -g cc.c
  ```

  These two lines indicate that the command `lcc-x32 -g cc.c` has to be executed whenever `cc.c` or `monitor.c` has been changed and `make` is requested to update `cc.ce`. This request can be either explicit (`make cc.ce`) or implicit, because another part depends on `cc.ce`.

- `minicom`: This is a widely used serial communication program with far more features than `x32-term`. The lab course setup script makes sure you'll only have to start it up to communicate with the FPGA board.

- `gnuplot`: With this program you can visualize 2d and 3d functions and datasets in various ways. It forms a excellent combination with a programming language like Python because it can easily cope with ad hoc datasets. The resulting pictures can be exported to PostScript format so that they can easily be used in papers and such. You can use it interactively by simply starting the program:

  ```
  gnuplot
  ```

  You can also store commands into a file and execute these by giving this file as parameter to the program:

  ```
  gnuplot file
  ```

  With the `plot` command you can generate 2d figures. The next command displays a simple function:

  ```
  plot x**3
  ```

  You can also `plot` multiple functions or datasets, which are separated by commas:

  ```
  plot x**3, x, "file"
  ```

This command plots two functions and data from a text file in the same plot. It assumes the text file consists of several columns. (by default the first two columns are used to plot)

With every function/dataset several options can be given, like:

```
plot sqrt(x) title "square root", "file" using 1:3 with lines
```

The "title" option adds a title for the associated function/dataset. The "using" option denotes which two columns to use in the given file. The "with lines" option makes sure the plotted points are connected with lines.

To view a specific area, you can use the following commands:

```
set grid; set xtics 1; set ytics 1
set xrange [-2:2]; set yrange [-2:2]
```

With the `splot` command one can generate 3d plots. The following example illustrates some interesting possibilities:

```
set xrange [-2:2]; set yrange [-2:2]
set hidden3d; set contour
splot x**2+y**2, 2*sin(x+y)
```

If you select the output window at this point, you can rotate the plot in 3d using the left mouse button and zoom in and out using the middle mouse button.

B.2 Output Redirection

It's an interesting fact that UNIX and C are developed almost simultaneously at Bell Labs and that ARPANET came to be in the same year (1969). Nonetheless, there is something about UNIX and C why the are still used by IT professionals after the relative "eternity" (in computer terms) since their inception, and even gaining popularity through the GNU Project and Linux. Apparently the developers of UNIX and C have intuitively knew how to write software which keeps standing even in a quickly changing environment.

An important part of the philosophy behind UNIX is about how programs (interfaces) are subdivided by smaller programs (interfaces). The idea behind this is that every part is specialized in executing a single simple task. Complicated options or options that are not absolutely necessary are not wanted. Larger programs are made by combining such "simple" parts by means of mostly simple text based protocols. Simple interfaces and reuse of mostly generic specialized programs make the global complexity manageable.

The described philosophy is clearly shown with standard UNIX tools. Tools like `grep`, `make` and `gnuplot` are extremely specialized and can simply be combined from the command line to perform randomly difficult tasks.

Most UNIX tools accept input through so called "standard input" and print their output on the so called "standard output". The first corresponds in principle with the key board and the last with the screen. However, we can also use the output of one program as the input of another, using a so called "pipe". For example, the following command uses two pipes (indicated by "|") to scroll through all file names in the current directory (and subdirectories) which contain a certain (case-insensitive) string:

```
ls -l | grep -iR cpp | less
```

To use a file's input as standard input you can mostly use an argument as well, but the next example is perhaps nicer (`uniq` and `sort` are standard UNIX tools):

```
cat filename | uniq | sort
```

obviously, you can also save the output of a program to a file. To do this you'll use the ">" character. In the next examples three files are combined into a new one and the difference between two files are stored in a third file:

```
cat a b c > d
diff -u e f > g
```

C X32 Tutorial

This appendix describes how to use the X32 with its available hardware (VHDL) components and peripherals (like buttons and the SSD). For a more extensive description on using the X32 we refer to the X32 Programmers Manual [2].

First of all, you need to "include" the X32 "header file" in your C program. In this file some important memory addresses and interrupt numbers are defined. Next to that, it contains some useful macros and functions to handle interrupts. You can use this all by putting the following at the top of your program code:

```
#include <x32.h>
```

If you would like to view the header file yourself, you'll find this in the `x32-tools/lib-x32` directory.

For memory mapped I/O you'll use a predefined part of the memory. The start of this memory area is indicated by an integer pointer called peripherals. To use a certain (memory mapped) address, you'll add a predefined offset to this. The following line of code will write a value to the SSD:

```
peripherals[PERIPHERAL_DISPLAY] = 0x1234;
```

During the lab course you'll use the following offsets to read a (peripheral) value:

`PERIPHERAL_BUTTONS`: the state of the buttons (one bit per button)
`PERIPHERAL_ENGINE_DECODED`: the current value of the counter of the VHDL decoder

The following offsets are used to write a value:

`PERIPHERAL_LEDS`: the value of the leds (one bit per led)
`PERIPHERAL_DISPLAY`: the number on the SSD
`PERIPHERAL_DPC1_PERIOD`: the PWM signal's period
`PERIPHERAL_DPC1_WIDTH`: the pulse width of the PWM signal

To catch a certain interrupt, you have to set the ISR, set a priority and enable the interrupt. To do this, you'll use three different macros:

```
SET_INTERRUPT_VECTOR(INTERRUPT_BUTTONS, isr_buttons);
SET_INTERRUPT_PRIORITY(INTERRUPT_BUTTONS, 10);
```

```
ENABLE_INTERRUPT(INTERRUPT_BUTTONS);
```

In this example a function `isr_buttons` needs to be defined, without arguments or return type. To be sure the X32 really handles the interrupts, the interrupt should also be enabled "globally":

```
ENABLE_INTERRUPT(INTERRUPT_GLOBAL);
```

Note: when using uC/OS, enabling the "global" interrupt should be left to uC/OS itself (see Appendix D).

During the lab course you can use the following interrupt (IRQ) numbers:

`INTERRUPT_BUTTONS`: a button has been pressed or released
`INTERRUPT_OVERFLOW`: the result of an arithmetic operation does not fit into 32 bits (note: this can interrupt with some standard functions like `printf`!)
`INTERRUPT_DIVISION_BY_ZERO`: there was a division by 0
`INTERRUPT_ENGINE_A`: signal `a` has been changed ("edge")
`INTERRUPT_ENGINE_B`: signal `b` has been changed ("edge")
`INTERRUPT_ENGINE_ERROR`: the VHDL decoder has detected an error

For the priority one can choose a number ranging from 1 to 15. In fact, you can choose a number far greater than 15, but to keep things portable, this is not a wise thing to do. A larger number means a higher priority. You can give multiple interrupts the same priority; however, it's undefined which interrupt will get precedence.

D uC/OS Tutorial

In this appendix you'll find an explanation on how to use uC/OS and an overview of the uC/OX functions used at the lab course.

D.1 Usage

The use of uC/OS is illustrated best by an elaborate example. The following program defines a simple task, which prints "hello, world" once a second:

```
#include <ucos.h>
#define PRIORITY 10

#define STACK_SIZE 1024
int stack[1024];

void task(void *data) {
    while(TRUE) {
        printf("hello, world\n");
        OSTimeDly(50);
    }
}

void main() {
    OSInit();
```

```
    OSTaskCreate(task, (void *)0, (void *)stack, PRIORITY);
    OSStart();
}
```

In the following section you'll find an explanation on the uC/OS function used at the lab. Here are some *important* remarks about the using uC/OS.

- Priorities are "inverted" compared to X32's priorities; a lower number means a higher priority in this case.
- Different tasks may not have the same priorities.
- Because interrupts are mostly occur outside uC/OS, you have to start each ISR with a call to `OSIntEnter()` and end it with `OSIntExit()`; this causes uC/OS to temporarily stop context switching.
- Tasks, semaphores and other uC/OS datastructures should be initialized after calling `OSInit`.
- Enabling `INTERRUPT_GLOBAL` (Appendix C) should be left to uC/OS (more specifically, to `OSStart`; see next section).

D.2 uC/OS Functions

Here follows an overview (non-alphabetic) of the uC/OS functions used at this lab course. See [4] and/or the `ucos` directory for a complete overview.

- `OSTaskCreate`: Creates a new task. Arguments are respectively the address of a function, which functions as "task body", an initial argument for this function, a pointer to a "working stack" and a priority. For the priority you may choose a number ranging from 1 to 62. Make sure to "cast" the initial argument and the stack pointer by prefixing it with `(void *)`.

  The next line of code creates a task with `7` as initial argument.

  ```
  OSTaskCreate(some_func, (void *)7, (void *)stack, PRIORITY)
  ```

- `OSTimeDly`: Temporarily puts a task to a non-active state. As argument you put in the number of time ticks after which the task is supposed to become active again. At the lab uC/OS is setup so that time ticks occur with 20 ms intervals.

  The following will let the task "wait" until the next time tick:

  ```
  OSTimeDly(1);
  ```

- `OSSemCreate`: Creates a semaphore. The argument is the initial value of the "semaphore counter". The return type is a pointer to a generic type for uC/OS data structures:

  ```
  OS_EVENT *sem;
  sem = OSSemCreate(0);
  ```

- `OSSemPost`: "Posts" to a semaphore. If there are tasks "pending" on the semaphore, a switch will be made to the task with the highest priority. If there are no such tasks, the semaphore is incremented by 1:

  ```
  OSSemPost(sem);
  ```

- `OSSemPend`: "Pend" on a semaphore. Arguments are the semaphore, a timeout value and the address of a variable to which a possible timeout error can be given. At the lab you will use `WAIT_FOREVER` as timeout value, with which a timeout error can not occur:

  ```
  UBYTE err;
  OSSemPend(sem, WAIT_FOREVER, &err);
  ```

- `OSSemAccept`: decrements a semaphore counter if it's greater than 0 and returns the original counter value:

  ```
  OSSemAccept(sem);
  ```

- `OSInit`: Initializes uC/OS and creates an "idle task" with the lowest possible priority (63). This task runs when all other tasks are non-active, where it runs an infinite loop ("busy loop").

- `OSStart`: Starts multitasking by switching to the highest priority task and enabling the interrupts using `INTERRUPT_GLOBAL` (Appendix C).

E

In this appendix an example implementation of a software decoder is given.

```
/* software decoder for signals a and b,
 * isr for channel 'a' and 'b' int new_a, new_b
 */
void isr_decoder() {
    OSIntEnter();
    new_a = peripherals[PERIPHERAL_ENGINE_A];
    new_b = peripherals[PERIPHERAL_ENGINE_B];

    // both signals have changed: error
    if(new_a != state_a && new_b != state_b)
        OSSemPost(sem_error);
    // both signals are high: update up/down counter
    else if(new_a == 1 && new_b == 1) {
        // b signal comes later: positive direction
        if(state_b == 0)  dec_count++;
        // a signal comes later: negative direction
        else dec_count--;
    }

    //update state
    state_a = new_a;
    state_b = new_b;

OSIntExit();
}
```