X32 Programmers Manual

S. Woutersen

Embedded Software Lab
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

November, 2006

Contents

Chapter 1

Introduction

The X32 is a 32-bit softcore designed for the Spartan 3 Starter Board. The design of the X32 is based on a top-down design approach, which starts at the programming language and ends at the processor. The LCC Compiler is used to convert ANSI C programs in the intermediate language of the LCC Compiler: LCC Bytecode. LCC Bytecode is a complete hardware-independent assembly language, and consists of nothing more than a simplified version of the C programming language. The LCC Bytecode instruction set is used as instruction set architecture for the X32 softcore. The top-down approach caused the X32 to be designed in a relative small amount of time, which allows it to be available free of charge. The compatibility with the C programming language is also very good due to the top-down approach (although no floating point unit is available), which makes programming the X32 very easy.

The performance of the X32 is somewhat limited due to the top-down design, as the design of the softcore did not take any hardware specifications into account, however, the compatibility with the programming language causes the X32 to outperform most conventional microcontrollers such as the 6052 and the 8051, and is only a factor 20 slower than the highly optimized Pentium 4 processor on equal clockspeeds. More information on the design of the X32 can be found in The X32 Softcore: A top-down approach on processor design [10].

The featureset of the X32 softcore includes:

- 32-bit processor core

- Works out of the box on a 400K Gates Spartan 3 FPGA

- Support for the buttons, switches, leds, display, RS232 connection and 1MB SRAM memory located at the Spartan 3 Starter Board

- Complete toolchain including an ANSI C Compiler

- Most of the C standard library available

- Interrupt system with configurable priorities and interrupt vectors

- Out of memory, division by zero and overflow protection

- X32 Configuration system using C header files, allows unique peripheral sets for any situation

This manual describes how to install the X32 and it's toolchain, and use all of the supported features.

Chapter 2

Installation

This chapter handles the installation and configuration of the X32 and its tools. The X32 and its toolset comes in three different packages.

The X32 tools package contains several programs to compile and upload C programs to the X32. This package must be installed by anyone wanting to develop software for the X32. The installation of the tools is described in Section 2.1. The second package contains the X32 source code. This package is required by anyone who wants to make changes to the X32 design. Section 2.2 handles the installation and compilation of the X32 source code. Finally, the X32 binary package contains the bit files of all standard X32 configurations, and can be used by anyone who wants to use, but not alter the X32 design. How to use the X32 binaries is described in Section 2.3.

2.1   X32 Tools Installation

The X32 tools contain all programs required to compile a C program into an X32 executable, as well as an upload utility to upload programs to the X32, and a bytecode interpreter (instruction simulator), which can run X32 executables without the need of any additional hardware.

The X32 tools are required to develop and test software for the X32. They must also be installed when working on the X32 itself, or any X32 extensions.

2.1.1   Requirements

The following applications must be installed on the system before installing the X32 tools:

- GCC Compiler

- GNU Make

For Windows, the GCC Compiler, and GNU Make can be downloaded as part of the MinGW Project which can be downloaded from sourceforge [9]. On most Linux systems, these programs are already installed. Check the manuals of your distribution for more information.

2.1.2   X32 Tools Installation

1. Download `x32-tools.tgz` from the download site [3].

2. Extract the archive to your hard drive (the package does not contain an installer, this will be the directory the tools will run from).

3. Compile the tools using the supplied makefiles, Makefile.linux for Linux, Makefile.windows for Windows. Execute `make -f Makefile.linux` for Linux, or `mingw32-make -f Makefile.windows` for Windows.

4. Set the `X32TOOLDIR` environment variable to the full path name of the `x32-tools/bin` directory created after extracting the package. Check the help files of your operation system on how to set environment variables.

5. Set the `X32LIBDIR` environment variable to the full path name of the `x32-tools/lib-x32` directory created after extracting the package.

6. Add the `x32-tools/bin` directory to your `PATH` environment variable (optional).

7. To use X32 configuration specific code, the X32 library (`x32.h`, `x32.cl`) must be copied or linked into the `x32-tools/lib-x32`. These files depend on the X32 configuration, and are created after building the X32. The precompiled libraries for the standard configurations can also be extracted from `x32-binaries.tgz`, which can be downloaded from the download site [3].

The X32-tools contain two batch files (`setup_X32` for Linux, `setup_X32.cmd` for Windows) which setup all environment variables automatically. These scripts can be added to the startup scripts, to set the environment variables. On Linux, it might be required to execute `source setup_X32` instead of `./setup_X32`.

2.2 X32 Source Code Installation

The X32 source code must be installed to make modifications to the X32, or build a new custom configuration of peripheral devices and X32 settings. The X32 tools must be installed prior to installing the X32 source code. When a default configuration of the X32 is to be used, it is also possible to download the standard X32 bitfiles directly. This is discussed in the next section.

In the first subsection, all requirements which must be met prior to installing and synthesizing the X32 are listed. In the second subsection, the steps to install, synthesize and program the X32 on a FPGA are given, and finally, in the last section, it is described how the X32 can be programmed in extra effort mode, which is necessary for some configurations.

2.2.1 Requirements

The following applications must be present on a system to compile the X32 source code.

- Compiled X32-Tools (See previous section)
- GNU Make, or similar
- A Perl interpreter
- Xilinx Webpack ISE 8.3 (or later) [13]
- xc3sprog [8] (optional)

2.2.2 X32 Source Code Installation

1. Download `x32.tgz` from the download site [3].

2. Extract the archive to your hard drive (it is recommended to extract the package into the same directory the tools where extracted, such that the `x32` directory from `x32.tgz` shares the same parent directory as the `x32-tools` directory from `x32-tools.tgz`).

3. If the archive is not extracted to the same folder as the toolset, change the `TOOLDIR` variable in the makefile (Makefile.linux for Linux, Makefile.windows for Windows).

4. The X32 can now be compiled using the supplied makefiles. The different configurations can be compiled by executing
   `make -f Makefile.linux <configuration_directory>`, e.g.
   `make -f Makefile.linux X32-minimal-interrupts`. For Windows replace
   `make -f Makefile.linux` with `mingw32-make -f Makefile.windows`. See Appendix C on the available standard configurations.

5. After compilation, the files `x32.bit`, `x32.cl` and `x32.h` are created for the compiled configuration. `x32.bit` can be uploaded to any Spartan 3 FPGA using either Impact (part of the Xilinx Webpack ISE) or xc3sprog [8]. The files `x32.h` and `x32.cl` should be copied, or linked into the `lib-x32` directory in the `x32-tools` to access X32 specific library functions and configuration specific peripheral devices. Make sure to distribute these files amongst all computers building software for this X32 configuration.

2.2.3 Compiling the X32 in extra effort mode

Some configurations might be too complicated to be placed and routed on the first try. In this case, after building the X32 configuration, an error message will appear telling the timing constraints could not be met. For example, when the X32-example configuration is synthesized for the first time, it will have paths up to 26ns long. On a 50MHz clock, the X32 is thus overclocked by more than 25%. To get better routing results, the place and route utility can be set in "extra effort mode". In extra effort mode, the utility keeps looking for better results until one is found which meets the timing constraints. Note that no guarantee is given the utility will find such a result, and may keep running forever.

To run the place and route utility in extra effort mode, use
`make -f Makefile.linux X32-example/X32.dir`. Off course,
`Makefile.linux` should be changed to `Makefile.windows` on Windows, and `X32-example` should be changed to the configuration used.

The place and route utility will now start building designs in the
`x32-example/x32.dir` directory. The different results will be named `H_H_#.ncd`, where `#` is a unique number. When a valid result is found, this result can be copied, or symlinked into the `x32-example` directory with the name `x32.ncd`. When this is done, the X32 can be completed by typing `make x32.bit`.

2.3 X32 Binary Installation

When using one of the standard X32 configurations, it is possible to directly download the X32 binaries. This avoids installing the Xilinx Webpack, which costs several gigabytes of

hard disk space, and rebuilding the X32, which might take up to 30 minutes. Off course, this comes at the cost of not being able to make any changes to the X32 configuration, and a standard configuration must be chosen.

2.3.1 Requirements

- Xilinx Webpack ISE 8.3 (or later) [13], or xc3sprog [8]

2.3.2 X32 Binary Installation

1. Download `x32-binaries.tgz` from the download site [3].

2. Extract the archive to your hard drive (it is recommended to extract the package into the same directory the tools where extracted, such that the `x32` directory from `x32.tgz` shares the same parent directory as the `x32-tools` directory from `x32-tools.tgz`).

3. After extraction, the files `x32.bit`, `x32.cl` and `x32.h` are created for each configuration. `x32.bit` can be uploaded to any Spartan 3 FPGA using either Impact (part of the Xilinx Webpack ISE) or xc3sprog [8]. The files `x32.h` and `x32.cl` should be copied, or linked into the `lib-x32` directory in the `x32-tools` to access X32 specific library functions and configuration specific peripheral devices. Make sure to distribute these files amongst all computers building software for this X32 configuration.

Chapter 3

Running software on the X32

A processor alone is useless without any software to execute. This chapter describes how software (programmed in the C programming language) can be executed by the X32 processor.

The first section briefly gives the support of the X32 for the C programming language. The second section describes how software can be compiled into a binary format which can be understood by the X32. In the third section, it is explained how software can be uploaded to the X32. Finally, in the fourth section, it is described how software can be executed after uploading.

3.1   X32 C support

The X32 and its tools support the almost the entire ANSI C programming language as described in [5]. The language syntax is completely supported, and all but the floating point variable types (`float` and `double`) are supported. The `long long`, `long` and `int` variable types, however, are all 32 bit.

A great amount of functions from the standard library included in the ANSI C specifications have also been ported to the X32 programming platform. An overview of the available library functions can be found in Appendix A.

3.2   Compiling software

The first step in executing software is compiling it into a binary format the X32 can understand. From here, it is assumed, the software is available in C source files, and the X32-tools are installed correctly. To compile C source files, the files must be processed by the compiler (`rcc`), the assembler (`x32-asm`) and the linker (`x32-link`). Conveniently, the lcc driver (`lcc-X32`) is able to do all that, such that programs can be compiled using only one command. Suppose there are two source files, called `source1.c` and `source2.c`, and they must be compiled into a single executable, called `program.ce`, the following command can be used:

```
lcc-X32 source1.c source2.c -o program.ce
```

The lcc driver can also be used to compile a program in steps:

```
# compile
lcc-X32 -S source1.c -o source1.bc
```

```
lcc-X32 -S source2.c -o source2.bc
# assemble
lcc-X32 -c source1.bc -o source1.co
lcc-X32 -c source2.bc -o source2.co
# link
lcc-X32 source1.co source2.co -o program.ce
```

To make sure that the compiler always identifies your files correctly, always make sure the source files have the `.c` extension, the bytecode (compiled) files the `.bc` extension, the object (binary bytecode) files the `.co` extension, the executables the `.ce` extension, and any libraries the `.cl` extension.

3.2.1   Compiling and using custom libraries

To manage larger programs, it is sometimes easier to compile often used pieces of source code into a library. All functions compiled into a library are available to any source file. To compile `source2.c` and `source3.c` into a library called `library.cl`, use the following command:

```
lcc-X32 source2.c source3.c -Wl-buildlib -o library.cl
```

To use the new library, the compiler know where to find the library. The library can either be placed in one of the specified library directories (see Section 2.1), or it can be specified during the compilation of the program:

```
lcc-X32 source1.c -Wl-lib -Wllibrary.cl \
    -o program.ce
```

In addition to library a library file, the `-lib` parameter also supports directories. Assuming several library (`.cl`) files are placed in a directory called `libraries`, the following command would load all libraries from `libraries`.

```
lcc-X32 source1.c -Wl-lib -Wllibraries \
    -o program.ce
```

The available library functions included with X32 tools can be found in Appendix A.

It is important to understand that the source files compiled into a library are included completely, or completely not in an executable file. For example, assume the files `source1.c`, `source2.c` and `source3.c` from the previous example have the following layout:

```
source1.c
    function main()
source2.c
    function add()
    function sub()
source3.c
    function mul()
    function div()
```

If `main` in `source1.C` uses the functions `add` and `mul`, the linker will automatically include all code from `source2.c` and `source3.c`, thus the functions `sub` and `div` are included as well.

However, if `main`, only uses function `add`, the function `sub` is included, but the functions `mul` and `div` are not. Also, when function `mul` from `source3` uses function `add` from `source2`, and only function `div` is called by `main`, all functions are included in the final executable, since by including function `div`, `mul` is included, which requires `add`, and by including `add`, `sub` is included as well. It is therefore recommended to spread the library functions over as much source files as possible, giving the linker total control over which functions to include in the final executable. This will only slightly increase the overhead of library files, but might greatly reduce the size of the final executable file.

3.3   Uploading software

To execute binary executables on the X32, they must first always be loaded into the RAM of the X32. There are two possible ways to get software loaded into the X32 RAM; by including the executable in the processors ROM, or by loading the executable directly into the RAM by using a loader program. Both have their advantages and disadvantages which are explained in the next two sections.

3.3.1   Uploading to ROM

When the X32 boots up, it automatically copies the entire contents from its ROM to its RAM and starts executing from the RAM. To include software in the ROM of the X32, the binary executable must be synthesized with the X32 design (which contains the ROM entity). The major disadvantage of this scenario is that each time the software is changed, the entire X32 must be resynthesized, which may take up to 30 minutes. The advantage however is that X32 will always start executing the software placed in the ROM after a reset automatically, and the software remains on the X32 when the power is taken off. This is therefore the recommended scenario for final releases of a product.

   To store programs in the ROM of the X32, the executable file must first be converted into a VHDL ROM description, and then be added to the list of X32 VHDL source files. To convert a binary file to a VHDL ROM description, a small program `bin2vhd` is included in the X32-tools. To convert the program `program.ce` into the ROM description `rom.vhd`, use the following command:

```
bin2vhd program.ce -o rom.vhd
```

The Makefile of the X32 must then be modified to include the custom `rom.vhd`. The easiest way to do this is by overriding the standard `rom.vhd` creation rule by adding a rule like:

```
X32-minimal/rom.vhd: program.ce
    bin2vhd program.ce -o X32-minimal/rom.vhd
```

When this rule is added, the X32-minimal configuration will include program.ce in its ROM. Before synthesizing, make sure the ROM software is compiled correctly.

   All standard configurations of the X32, have the ROM copied to an address other than zero (but 0xC0000). The reason for this is that the bootloader, which allows custom programs to be uploaded directly into the X32 RAM, does not occupy any RAM in the range of 0-768K. When custom software is compiled into the X32 ROM, either the software must be compiled to run from 0xC0000 by adding `-base=C0000` to the compiler command line, or the X32 must

be configured to copy the ROM to RAM address zero. See Section 5.3.2 for more info about configuring the X32.

3.3.2 Uploading to RAM

As described in the previous section, running software from the X32 ROM requires a resynthesis of the X32, which takes a lot of time. When software is being developed, it must often be tested, and having to wait half an hour for each test is often a waste of time. Therefore, a bootloader program has been developed which is placed in the ROM of the X32 for all standard configurations. The bootloader contains a simple shell which runs on stdin and stdout, which are by default connected to the primary RS232 connection [12]. Using a terminal program, it is possible to communicate to the bootloader, and access the X32 memory directly. To upload programs, a small upload utility is created which communicates with the bootloader. To upload a program executable to the X32, use the following command:

```
X32-upload program.ce -c /dev/ttyS0
```

Assuming `/dev/ttyS0` is the tty the X32 is connected to. On windows, `com#` can be used, when replacing `#` with the serial port number the X32 is connected to.

3.4 Running software

Software installed in the ROM of the X32 is automatically started. However, when uploading software to the RAM of the X32, the program must be started manually[1], as the X32 has started executing the resident bootloader. To do this connect to the bootloader using a terminal program (e.g. minicom for Linux, or HyperTerminal for Windows). If the menu doesn't show immediately, type `m [enter]` to show the menu with the accepted commands. If no menu appears, there is a problem with the connection. The command to start programs is the `s` command.

There are two ways to start a program, using simply `s` causes the bootloader to make a function call to `main()`, and when using the `s j` command, the bootloader jumps to the first byte of the executable. The difference between these starting methods are explained in the next two sections. The memory layout for both types is shown in the third section.

3.4.1 Calling a program

Calling a program is the preferred way to quickly test small programs. The program is simply executed as a function call (to main()) from the bootloader program. When the program exits, the X32 will automatically return to the bootloader and show some timing statistics. The stack of the program runs on top the stack of the bootloader. Also the bootloader must remain intact to be able to return to the bootloader. See Section 3.4.3 on the memory requirements to use the call method.

When returning from a program to the bootloader, and restarting the program without re-uploading, care should be taken with assigned global variables, as they are not reset, as shown in the following example:

---

[1]It is possible to start the software automatically by supplying the -s command line parameter to the upload tool

```
int n = 0;
int main() {
    n++;
    return n;
}
```

On any system, this program will always return 1. However, when running the program on the x32 for a second time, without re-uploading the program, it will return 2. This problem can be avoided by dynamically assigning all global variables:

```
int n;
int main() {
    n = 0;
    n++;
    return n;
}
```

3.4.2   Jumping into a program

Jumping into a program gives complete control to this program. To program generates its own stack, and the full memory range is available to the program, however, it is not possible to return to the loader. This method should in general only be used when the full memory range is required, or to test the behavior of the X32 when the software is stored in the ROM.

3.4.3   Memory layout

When testing programs using the bootloader, it is important to know that two programs are located in the RAM of the X32, and it is therefore required to understand the memory layout of the X32 running the bootloader, to make sure the bootloader and the program do not stand in each others way.

The bootloader is by default placed at address 0xC0000 (or 768KB) and requires about 30KB of memory. No program uploaded using the bootloader may thus ever exceed address 0xC0000, and thus must not be larger than 768KB. The stack of a program is placed directly beyond the memory required by the code and variables of a program, and grows upward in memory. In case of the bootloader, the stack of the bootloader thus starts at about 0xC8000 (or 800KB), allowing 200KB of stack when 1MB of RAM is available. The stack space available to a program executed using the call method is thus slightly less than 200KB. When executing a program using the jump method, the bootloader may be overwritten by dynamic allocated data and the stack, allowing the use of the full RAM. This is indicated in Figure 3.1.

In short, for programs being able to be executed by the bootloader, they must follow the following rules:
To use the call method:

- The size of the executable plus the size of the uninitialized variables must not exceed 768KB

- The size of the stack must not exceed 200KB

To use the jump method:

```
0x00000000  ┌─────────────────┐    0x00000000  ┌─────────────────┐
            │     Room for    │                │     Room for    │
            │     programs    │                │     programs    │
            │    and global   │                │    and global   │
            │ variables (768K)│                │ variables (768K)│
            │                 │    0000xxxxxx   ├─────────────────┤
            │                 │                │                 │
            │                 │                │  Program stack  │
            │                 │                │  (may overwrite │
            │                 │                │     loader)     │
0x000C0000  ├─────────────────┤    0x000C0000  ├─────────────────┤
            │   Loader code   │                │   Loader code   │
            │      image      │                │      image      │
0x000Cxxxx  ├─────────────────┤    0x000Cxxxx  ├─────────────────┤
            │   Loader stack  │                │   Loader stack  │
0x000xxxxx  ├─────────────────┤                │                 │
            │  Program stack  │                │                 │
            │  (max +/-200K)  │                │                 │
            └─────────────────┘                └─────────────────┘
```
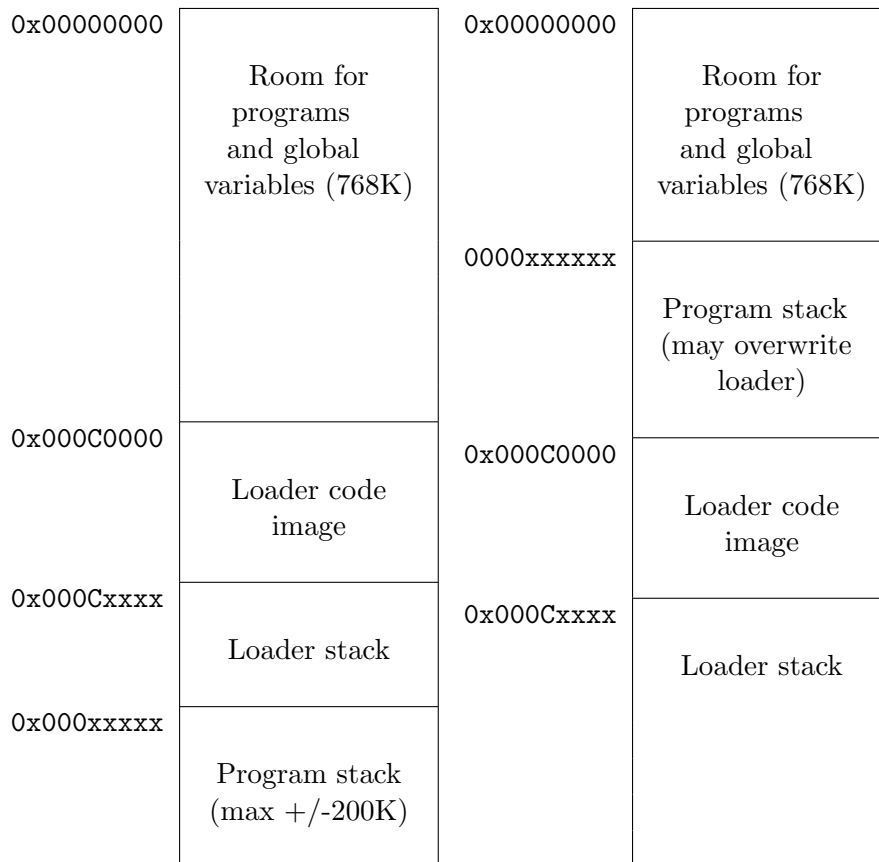
Figure 3.1: Memory layout when running the loader
On the left, the memory is shown when executing programs using the call method, on the right, the memory is shown when executing programs using the jump method.

- The size of the executable must not exceed 768KB

- The size of the executable plus the size of the uninitialized variables plus the size of the stack must not exceed 1024KB

To get an indication of the memory a program uses, the bytecode interpreter can be used. See Section 3.6 for how to use the interpreter.

3.5   Running multiple programs on the X32

It is possible to upload a program to any location in the memory, and thus store multiple programs into the memory of the X32, and have them interact. To upload a program to a location other than zero, use the -l command on the uploader. For example; to upload program.ce to location 0xA000, use the following command:

```
X32-upload program.ce -c /dev/ttyS0 -l A000
```

The program can be started using using either s A000 or s A000 j on the bootloader command prompt.
    To execute programs from a different location however, the linker must know the location in advance, by supplying the -base parameter when compiling. To compile source1.c and source2.c into a program.ce which can be executed from address 0xA000, use the following command:

```
lcc-X32 source1.c source2.c -o program.ce -Wo-base=A000
```

Note that the program can now *only* be executed from 0xA000, and no longer from address zero. When using multiple programs at the same time, it is even more important to understand the memory layout of the X32, which can be found in the previous section.

3.6   Running programs on the bytecode interpreter

In addition to the hardware X32, a software interpreter has been written in C, which can execute X32 executables. The X32 interpreter is a part of the X32-tools and gets automatically compiled when compiling the X32 tools. The X32 interpreter should be used whenever the X32 hardware is not available, or to get a better insight of the memory usage of a program.
    To interpret the program program.ce, type the following on the command line (assuming the bin directory of the X32 tools are added to the PATH environment variable):

```
x32-sim program.ce
```

The program will now be executed. When it finishes, some information about the memory usage is given. To load a program at a different address than zero (e.g. to test the loader), the -base parameter can be used. The following command interprets the loader from address 0xC0000:

```
x32-sim program.ce -base C0000
```

The interpreter can also be used to analyze C programs using the -a parameter:

```
x32-sim program.ce -a program.txt
```

Creates the `program.txt` file containing some information about the executed instructions, and memory actions taken by the interpreter.

The complete list of peripheral devices supported by the bytecode interpreter can be found in Section C.9.

Chapter 4

X32 specific code

This chapter describes how to write X32 specific code, how to access X32 peripherals, interrupts and some advanced coding examples for the X32.

In the first two sections, Section 4.1 and Section 4.2, it is described how peripheral devices attached to the X32 can be accessed from C code, and how the interrupt controller can be used. In the following sections, some more advanced features of the interrupt system are described. Sections 4.3, 4.4 and 4.5 respectively describe software interrupts, resource locking and context switching on the X32. The last section, Section 4.6 describes how the current time can be retrieved from the X32.

4.1   Accessing peripheral devices

Peripheral devices, in short peripherals, are all connected to the memory bus of the X32. Writing data to, and reading data from a peripheral, can thus be done by simple memory reads and writes. In C, a peripheral can thus be seen as a simple variable at a specific location. The memory layout of the X32 is shown in Figure 4.1

By default, the address at which the peripherals are located is `0x80000000`, and the peripheral bus is 32 bit wide. The first peripheral is therefore located at `0x80000000`, the second, 32 bit (or 4 bytes) higher, at `0x80000004`, the third at `0x80000008`, and so on. All peripherals can be accessed as integers, although some devices might not use the full integer range. If the data is signed or not depends on the peripheral device.

For easy peripheral access, an integer array called `peripherals` is declared in `x32.h`[1], and is located at `0x80000000`. The first peripheral is therefore located at peripherals[0], the second at peripherals[1], and so on. Writing to and reading from a peripheral device in C is thus nothing other than writing to and reading from the peripheral array.

The available peripheral devices are dependent of the X32 configuration used. Each configuration has its own version of `x32.h`, which contains macros starting with `PERIPHERAL_` which hold the peripheral locations. For example, `x32.h` might contain `PERIPHERAL_BUTTONS`. If so, the state of the buttons can be read from `peripheral[PERIPHERAL_BUTTONS]`. In Appendix C the peripherals for the standard configurations are listed. In the following three subsections, three special registers are handled which can be connected to the peripheral bus.

---

[1]The peripherals array is declared as an extern variable in x32.h, the variable itself is located in x32.c in the library directory of the x32 sourcecode.
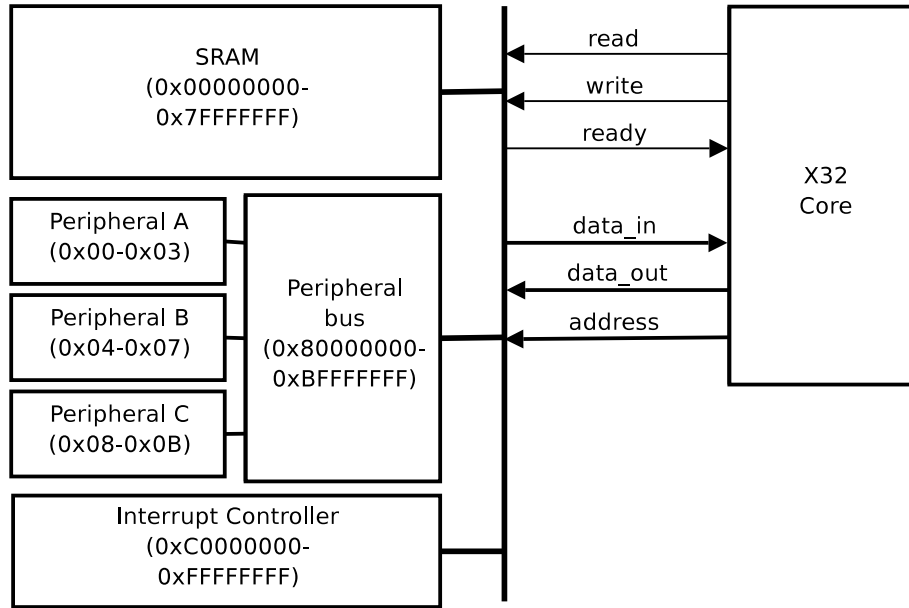
Figure 4.1: X32 Memory bus layout
The X32 connected to the main memory, the interrupt controller and three peripheral devices.

These registers are not considered devices since they are unique; only one of each may be connected to the peripheral bus.

4.1.1   Peripheral bus id register

The peripheral bus id register can be found on `peripherals[PERIPHERAL_UID]`. The 32-bit register is read-only and contains a number which is unique for each X32 configuration. It can be used to identify X32 configurations. The ids used by various configurations can be found in Appendix C.

4.1.2   Instruction counter register

The instruction counter register can be found on
`peripherals[PERIPHERAL_INSTRCNTR]`. It is a 32-bit read-only register containing the amount of instructions executed since the last X32 reset. Note that the register overflow approximately once per half hour.

4.1.3   Processor state register

The processor state register is a special register on which each bit is set on a special processor event. The register is reset when it is read by software. The register can be found at `peripherals[PERIPHERAL_PROCSTATE]`. The explanation of each bit in the register can be found in Table 4.1.

Table 4.1: Processor state register

Bit	Meaning
0	Booting, this bit is set while booting (resetting) the X32. It is thus only high the first time the processor state register is read
1	Simulating, this bit is always high on the bytecode interpreter, and always low on the X32.
2	Not connected
3	Division by zero, this bit is set when division by zero occurs. It can be used to detect division by zero on cores without interrupt support
4	Overflow, this bit is set when overflow occurs. It can be used to detect overflow on cores without interrupt support
5	Not connected
6	Out of memory, this bit is set when the X32 runs out of memory, this bit can currently not be used
7	Trapped, this bit is set when the X32 executes a trap instruction

4.2 Using interrupts

The X32 can be configured to support a basic interrupt system. An interrupt system allows hardware to notify the software it has some information to share. Without it, the software must continuously keep asking the hardware for it, which is known as polling. Although most software can be written using the polling technique, using interrupts often improves the performance and simplicity of a system, and is necessary for, e.g., real-time operating system support.

The interrupt system of the X32 makes use of interrupt request lines (interrupt request, or IRQ lines) which can be asserted by peripheral devices. When this happens, the interrupt controller interrupts the processor, and forces it to jump to a predefined address. This address, should be the address of an interrupt service routine, also known as ISR. When the interrupt service routine finishes, the processor returns to normal execution.

The interrupt controller also has support for interrupt priorities, which allows prioritizing device interrupts. A timer controlling a time-sensitive control algorithm may for example require a higher priority than a button controlling a user interface. Finally, each interrupt can be individually enabled and disabled, and a master interrupt is available which allows completely disabling all interrupts.

4.2.1 Programming the interrupt controller

The interrupt controller is equipped with a few bytes of RAM which allows programming the interrupt controller. The RAM is connected to the X32 through the standard memory bus, just like the peripherals are, and can therefore be accessed using memory pointers, or even the `peripheral` array. By writing the interrupt controller memory, for each interrupt,

an interrupt vector (pointer to ISR) and an interrupt priority can be defined. The ISR is a function which is called when the corresponding interrupt occurs. The function may not take any parameters, and it must return `void`. In addition, a register is available which holds the enable flags for each individual interrupt, and a global interrupt. To enable an interrupt, both the bit for the individual interrupt, and the bit for the global interrupt must be set.

The locations of the interrupt controller RAM, and the interrupt enable register are defined in `x32.h`, as indices of the `peripheral` array. The RAM starts at `peripherals[PERIPHERAL_INT_VECT_BASE]`, with the vector for the first interrupt. The next item, `peripherals[PERIPHERAL_INT_VECT_BASE+1]`, contains the priority for the first interrupt. The third and fourth items, `peripherals[PERIPHERAL_INT_VECT_BASE+2]` and `peripherals[PERIPHERAL_INT_VECT_BASE+3]`, contain the vector and priority for the second interrupt, and so forth. The interrupt enable register is located at `peripherals[PERIPHERAL_INT_ENABLE]`. Bit 0 of this value holds the enable flag for the first interrupt, and bit `n-1` the enable flag for interrupt `n`. The global interrupt flag is located at bit `n`, where `n` is the amount of interrupts supported by the interrupt controller.

To make programming the interrupt controller a little easier, several macros are defined in `x32.h`. Each device which uses one or more IRQs, generates a macro prefixed with `INTERRUPT` defining its IRQ index. For example, a peripheral connecting buttons to the X32, may define the `INTERRUPT_BUTTON` interrupt. The available interrupts can be found in Appendix C for premade configurations. More information on the enabling and disable mechanism used by the X32 can be found in Section 4.2.3. In addition, the macro `INTERRUPT_GLOBAL` is defined and holds the bit index of the global enable flag for the interrupt enable register.

The following macros are available to program the interrupt controller: `INTERRUPT_VECTOR`, `INTERRUPT_PRIORITY`, `ENABLE_INTERRUPT` and `DISABLE_INTERRUPT`. All four macros take one parameter: the IRQ index of the interrupt. The `ENABLE_INTERRUPT` and `DISABLE_INTERRUPT` enable and disable individual interrupts respectively. Using the `INTERRUPT_VECTOR` macro, an ISR can be specified. Using the `INTERRUPT_PRIORITY` macro, a priority can be given to an interrupt. More information about priorities can be found in Section 4.2.2. The `INTERRUPT_VECTOR` and `INTERRUPT_PRIORITY` may not take `INTERRUPT_GLOBAL` as an argument. It is important to always define the ISR and priority before enabling the interrupt.

It is recommended to always program the interrupt controller using the supplied macros for three reasons. At first, some of the addresses and locations may be different on different configurations of the X32. A recompilation of the code will in this case be enough to automatically change all addresses and indices, and even detect if the used interrupts are supported. The second reason is that changing a single bit in the interrupt enable register, requires accessing the register twice; once to read, and once to write the register. An interrupt occurring between these two actions might cause problems. The `INTERRUPT_ENABLE` and `INTERRUPT_DISABLE` macros, temporarily disable all interrupts, making the action atomic. The third reason, is off-course, that it makes the code a lot more readable and maintainable.

The following example runs initializes the button interrupt, prints dots until a button is pressed (or released) and then exits.

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

int quit = 0;
```

```
    void button_isr() {
        quit = 1;
    }

    int main() {
        /* set button interrupt address & priority */
        INTERRUPT_VECTOR(INTERRUPT_BUTTONS) = &button_isr;
        INTERRUPT_PRIORITY(INTERRUPT_BUTTONS) = 10;

        /* enable interrupt */
        ENABLE_INTERRUPT(INTERRUPT_BUTTONS);
        ENABLE_INTERRUPT(INTERRUPT_GLOBAL);

        /* run until quit becomes nonzero */
        while(!quit) putchar('.');

        /* disable interrupts */
        DISABLE_INTERRUPT(INTERRUPT_GLOBAL);
        DISABLE_INTERRUPT(INTERRUPT_BUTTON_STATE_CHANGED);

        return 0;
    }
```

4.2.2   Interrupt priorities

As explained above, interrupt priorities can be used to give one interrupt priority over another. An interrupt with a higher priority may interrupt an ISR called with a lower priority, but an interrupt will never interrupt an ISR called with a higher or equal priority.

The interrupt controller does not sort pending interrupts on priority. When two interrupts with different priorities are pending, either one of them is serviced first. However, the lower priority interrupt will be interrupted immediately by the higher priority interrupt.

The interrupt priority system depends on a special processor register called the execution level register. The execution level is saved on the stack on each function call, and restored on each function return. When an interrupt causes a call to an ISR, the execution level is (after being saved) overwritten by the priority of the interrupt. The interrupt controller uses the value of the execution level to determinate whether it should generate an interrupt; the execution level must always be lower than the priority of the pending interrupt, to have the controller service the interrupt.

When the processor resets, the execution level is reset to zero. Normal code does not change the execution level, and thus runs at an execution level of zero. Each interrupt must thus have a priority greater than zero to have it serviced.

The total amount of priority levels is in theory only bounded by the size of the execution level, which is 32-bit (and allows values up to 4.2 billion). In practice, it is recommended to only use the range from 1 to 999 (which is more than any system will ever need). Since the execution level can also be set manually (see Section 4.2.4), execution level 1000 can be used to temporarily disable all interrupts. Execution level 0xF000, 0xF001 and 0xF002 are used by the bootloader application to catch division by zero (which is discussed in Section 4.2.5),
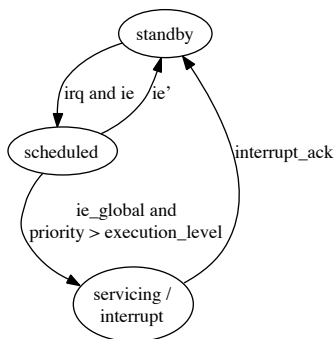
Figure 4.2: Finite state machine for a single non-critical interrupt
Each interrupt starts in standby. Whenever a device asserts the irq signal associated with that interrupt, the interrupt gets scheduled. Disabling an interrupt will force it back in the standby state. When an interrupt is scheduled, it is serviced as soon as the current processor priority drops below the priority of the interrupt, and the global interrupt is enabled.

the TRAP instruction (which is discussed in Section 4.3.2 and the out of memory interrupt (discussed in Section 4.2.6). It is therefore recommended to always keep the execution level below these values.

4.2.3 Enabling and disabling interrupts

Before entering an ISR, the interrupt must first be scheduled and serviced. Servicing an interrupt means forcing the processor to make a function call to the ISR. Before servicing an interrupt, the interrupt must be scheduled for service. These two steps are automatically taken by the interrupt controller, if the interrupt meets the predefined criteria: An interrupt gets *scheduled* if:

- The IRQ line is asserted.

- The individual interrupt is enabled.

An interrupt gets *serviced* if:

- The interrupt is scheduled.

- The individual interrupt is enabled.

- The global interrupt is enabled, or the interrupt is critical.

- The interrupt priority is higher than the current execution level.

This is also depicted in Figure 4.2 It can thus be seen that when individual interrupts are disabled, they will not be scheduled. This prevents interrupts to schedule long before the program starts to accept them, causing to trigger immediately after the interrupt is enabled. Interrupts may however still be scheduled when they don't have a high enough priority, or the global interrupt is disabled. They will then be serviced immediately whenever the execution level is lowered, and the global interrupt is enabled. The state machine on which the interrupt controller decides whether an interrupt gets serviced is shown in Figure 4.2.

As can be seen, some interrupts are critical, and will not be stopped by a disabled global interrupt. Which interrupts are critical depends on the configuration, and can be found in Appendix C. In general, all processor exceptions, such as division by zero, overflow, the trap instructions and the out of memory interrupt are marked as critical interrupts.

4.2.4 Programming the execution level

The X32 library contains three functions to control the execution level manually. This enables the programmer to raise the priority of code running on the processor, and disabling several low priority interrupts at once, without having to touch the interrupt enable flags, which are often controlled by other parts of a program/operating system. The functions are defined in `x32.h` and are called `get_execution_level()`, `set_execution_level()` and `restore_execution_level()`. They return the current execution level, set the execution level to any value, and reset the execution level to the level active when entering this function, respectively. The following example shows how to get, set and restore the execution level.

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

/* the execution level is preserved on function calls */
void print_execution_level() {
    printf("The execution level is now: %d\r\n",
        get_execution_level());
}

int main() {
    print_execution_level();

    /* raise the current code priority to 10, and
            consequently, disable all interrupts with a
            priority of 10 and lower */
    set_execution_level(10);
    print_execution_level();

    /* restore to original level */
    restore_execution_level();
    print_execution_level();

    return 0;
}
```

Note that the restoration of the execution level at the end of the main function in the previous example is not required. The execution level is always automatically restored on a function return, which is required to reset the execution level when returning from an ISR. It is therefore also not (easily) possible to set the execution level of caller functions.

When writing an real-time operation system for the X32, raising the execution level is the preferred way to implement critical OS sections.

4.2.5 Arithmetic error interrupts

The processor core itself is able to generate two interrupts: one when overflow occurs, and one when division by zero occurs. The IRQs for these interrupts are defined as `INTERRUPT_OVERFLOW` and `INTERRUPT_DIVISION_BY_ZERO`.

The division by zero is by default catched by the loader, after which an error message is printed and the program is terminated. Off-course, it is also possible to use a different ISR to catch division by zero interrupts, or even to disable the interrupt. The overflow interrupt is by default disabled, and must be catched by the program itself. It is recommended to only enable the overflow only at the code which should be checked for overflow, and to immediately disable it in the overflow ISR. This is because sometimes, including in several library functions, overflow is intended, and does not cause any problems with the program results.

The following example shows the use of the division and overflow interrupts:

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

int overflow;

void div0_isr() {
    printf("ERROR: Division by zero occurred!\r\n");
    exit(EXIT_FAILURE);
}

void overflow_isr() {
    DISABLE_INTERRUPT(INTERRUPT_OVERFLOW);
    overflow = 1;
}

int main() {
    int i, j;
    INTERRUPT_VECTOR(INTERRUPT_OVERFLOW) = &overflow_isr;
    INTERRUPT_PRIORITY(INTERRUPT_OVERFLOW) = 10;
    INTERRUPT_VECTOR(INTERRUPT_DIVISION_BY_ZERO) = &div0_isr;
    INTERRUPT_PRIORITY(INTERRUPT_DIVISION_BY_ZERO) = 10;
    ENABLE_INTERRUPT(INTERRUPT_DIVISION_BY_ZERO);

    overflow = i = 0;
    j = 1;
    while(!overflow) {
        ENABLE_INTERRUPT(INTERRUPT_OVERFLOW);
        j = j * 10;
        DISABLE_INTERRUPT(INTERRUPT_OVERFLOW);
        i++;
        printf("10^%d = %d\r\n", i, j);
    }
    printf("Overflow at 10^%d\r\n", i);
```

```
        while(1) {
            printf("%d / %d = ", 10, i);
            j = 10/i;
            printf("%d\r\n", j);
            i--;
        }

        return 0;
    }
```

4.2.6   Out of memory interrupt

The x32 has support for a forbidden memory space. Any read or write actions to this memory space cause the processor to raise the out of memory interrupt. This interrupt can be used to prevent the processor from crashing whenever a program consumes too much memory. The index of the out of memory interrupt is defined in `INTERRUPT_OUT_OF_MEMORY`, and is by default catched by the loader.

The blocked memory range on the standard X32 configurations lie between 0x000FF000 and 0x7FFFFFFF. The lower bound of the blocked memory range is located slightly below the upper memory bound of the Spartan 3 Starter Board [12] (0x00100000). This is done such that the interrupt triggers some time before the processor completely runs out of memory, leaving the address range from 0x000FF000 to 0x00100000 (4KB) for stack space required by the out of memory ISR. The upper bound of the blocked memory range is located just below the first peripheral device. Reading from and writing to the range from 0x80000000 to 0xFFFFFFFF will thus never cause an out of memory interrupt. The following code shows how to protect a program from running out of memory:

```
    #include <x32.h>

    void run_out_of_memory() {
        /* recursively call the current function, which stack will
            eventually run into the forbidden memory space */
        run_out_of_memory();
    }

    void out_of_memory_isr() {
        printf("Out of memory, exiting!\r\n");
        /* disable the out-of-memory interrupt, since this ISR may
            use the forbidden memory space */
        DISABLE_INTERRUPT(INTERRUPT_OUT_OF_MEMORY);
        exit(1);
    }

    int main() {
        INTERRUPT_VECTOR(INTERRUPT_OUT_OF_MEMORY) = &out_of_memory_isr;
        INTERRUPT_PRIORITY(INTERRUPT_OUT_OF_MEMORY) = 1000;
```

```
        ENABLE_INTERRUPT(INTERRUPT_OUT_OF_MEMORY);

        printf("Running out of memory...\r\n");
        run_out_of_memory();
        return 0;
    }
```

Without using the out of memory interrupt, the example shown above will crash the processor. Note that the out of memory interrupt is disabled in its ISR. This is required, since the stack of `out_of_memory_isr` runs in the forbidden memory space. When the interrupt is not disabled, it will trigger again immediately after returning from the ISR.

4.3 Software Interrupts

Software interrupts are interrupts generated by program code. They are useful for communication between programs which do not know each others exact position in memory. The X32 does not directly support software interrupts, but a simple peripheral device is available, which causes an interrupt when any data is written to it. By writing to this device, a software interrupt can thus be simulated. A server program can set the ISR for the software interrupt, after which a client program can do service requests on the master program by raising this interrupt. This method is often used in modern operating systems.

4.3.1 Passing data trough software interrupts

When using software interrupt, it is often required to pass data between the interrupt raising function and the interrupt service routine. This can be accomplished by combining the stackframes for these two functions, which allows sharing parameters given to a function, and sharing a return value and address. To combine two stackframes, the `combine_stackframe` function is defined in `x32.h`. The following example shows how to use this function, and how parameters can be passed through a software interrupt:

```
    #include <x32.h>

    int main() {
        int a = 1;
        int b = 2;
        /* set software ISR to softint_isr, and enable the
               interrupt (the ISR should normally be a
               void(void) function, this is however not
               required by the compiler) */
        INTERRUPT_VECTOR(INTERRUPT_SOFTINT) = &softint_isr;
        /* the priority must be >1000, see below for details */
        INTERRUPT_PRIORITY(INTERRUPT_SOFTINT) = 1001;
        ENABLE_INTERRUPT(INTERRUPT_SOFTINT)


        /* call raise_interrupt, which will compute the sum of
```

```
            a and b */
    printf("%d + %d = %d\r\n", a, b, raise_interrupt(a, b));
}

int raise_interrupt(int a, int b) {
    /* raise the execution level to 1000, make sure all
            hardware interrupt priorities are below 1000,
            and the software interrupt priority is above
            1000 */
    set_execution_level(1000);

    /* raise the software interrupt by writing to the
            software interrupt device (the value to write
            doesn't matter) */
    peripherals[PERIPHERAL_INT_SOFTINT] = 1;

    /* this code will never be executed! */
    return -1;
}


/* interrupt service routine: this function MUST have the
        EXACT same prototype as the function calling this
        interrupt */
int softint_isr(int a, int b) {
    /* this function is called by the interrupt controller.
            on execution, the processor expects the function
            to be a void(void) function, returning, or using
            any of the parameters now will result in "random"
            behavior */
    /* call combine_stack frame to merge the stack frames of
            softint_isr and raise_interrupt: */
    combine_stackframe();

    /* reset the execution level to level 10, alternatively,
            the restore_execution_level() function can be used
            to restore the execution level to the value active
            when calling raise_interrupt */
    set_execution_level(10);

    /* the a and b parameter are now available, and a return
            from this function will result in a return from
            raise_interrupt, return the sum of a and b: */
    return a + b;
}
```

The code above uses two tricks to accomplish passing data through software interrupts. The first one is declaring the ISR as an `int(int, int)` function, instead of a standard `void(void)`

function. This is only allowed when using the `combine_stackframe` function, and the function prototype must be the same as the calling function. The second trick is using the execution level to temporarily disable all interrupts. This is required to make sure no hardware interrupt triggers between the interrupt raising function and interrupt service routine of the software interrupt. If this would happen, the `softint_isr` would combine with the hardware ISR, and since this is a `void(void)` function, the stack will get corrupted.

4.3.2 The trap instruction

The trap instruction is a special case of software interrupt. It is not triggered by a write action to the peripheral bus, but by a special instruction called the trap instruction. This trap instruction can be used by debugging software to set breakpoints within programs. The trap instruction has a binary opcode in which only the most significant bit matters: it must be one. All other instructions have this bit set to zero, therefore, when a debug program sets a breakpoint, only the most significant bit of the instruction has to be set, and the original instruction can easily be restored by unsetting the most significant bit, eliminating the requirement of a data structure holding all original instructions.

The following code sets a `TRAP` instruction at the function `breakpoint`. The `INSTRUCTION` variable type is defined in `x32.h`, as a 16 bit unsigned integer.

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

void breakpoint() {
    printf("In breakpoint()\r\n");
}

void trap_isr() {
    /* pointer to an instruction */
    INSTRUCTION *code;

    printf("In trap()\r\n");

    /* 'fix' the instruction, so normal code execution can
        continue */

    /* set code to point to the first instruction in the
        breakpoint() function */
    code = (INSTRUCTION*)&breakpoint;
    /* reset msb of the instruction */
    *address &= ~0x8000;
    /* return to breakpoint */
}

int main() {
    /* pointer to an instruction */
    INSTRUCTION *code;
```

```
        /* have the trap function listen for the trap interrupt */
        INTERRUPT_VECTOR(INTERRUPT_TRAP) = &trap_isr;
        INTERRUPT_PRIORITY(INTERRUPT_TRAP) = 10;
        ENABLE_INTERRUPT(INTERRUPT_TRAP);

        /* set code to point to the first instruction in the
            breakpoint() function */
        code = (INSTRUCTION*)&breakpoint;
        /* set msb of the instruction */
        *code |= 0x8000;

        /* call breakpoint() */
        breakpoint();

        DISABLE_INTERRUPT(INTERRUPT_TRAP);
        return 0;
    }
```

When building a debugger, it is important to realize that the trap instruction suffers from the same interrupt controller latency as normal software interrupts, however a different solution must be used, since it is easy to replace one instruction in the program to be debugged, but very hard to replace several. The solution does thus not lie in preventing the problem from occurring, but handling the problem after it occurred.

This means scanning previous stack frames until the stack frame of the function which contains the trap instruction is found, which can be done by checking the instruction at the return address of each stack frame, this must be the trap instruction. If it is not, go up one stack frame and check its return instruction, repeat until the correct stack frame is found.

4.4 Locking resources

When building a concurrent system, resource locking is sometimes required. The X32 library has two functions which allows resource locking: `lock` and `unlock`. In addition, the variable type `LOCK` is defined in `x32.h` which specifies whether a device is locked or not. The `lock` function tries to lock a `LOCK` variable. It returns non-zero when it succeeds, or zero when the lock is already locked. The function `unlock` unlocks a lock, after which it can be locked again. Both functions are atomic. A `LOCK` variable must always be initialized prior to using it, which can be done by simply unlocking the lock. The following code demonstrates the use of the `lock` and `unlock` functions. It is assumed `task_a` and `task_b` are running concurrently, and they both try to increase a counter variable:

```
    #include <x32.h>

    int counter;
    LOCK counterlock;

    void task_a(void) {
```

```
        while(1) {
            if (lock(counterlock)) {
                /* Task A now has exclusive access to counter */
                counter = counter + 1;
                unlock(counterlock);
            } else {
                /* Task B has exclusive access, can't access
                    counter */
            }
        }
    }

    void task_b(void) {
        while(1) {
            if (lock(counterlock)) {
                /* Task A now has exclusive access to counter */
                counter = counter + 1;
                unlock(counterlock);
            } else {
                /* Task B has exclusive access, can't access
                    counter */
            }
        }
    }

    int main() {
        counter = 0;

        /* initialize the lock */
        unlock(counterlock);

        /* start tasks A & B */
        ...

        return 0;
    }
```

4.5 Context switching

Context switching is one of the most important building blocks of any a concurrent (multi-threading) system. This paragraph explains how task switching can be accomplished by the X32.

The X32 library contains two functions which makes task switching very easy: `init_stack` and `context_switch`. The first function initializes a stack for a new task, and returns a task pointer. The second function switches between two task pointers. The following example shows the switching between two tasks.

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

/* create a 4KB stack for a second task */
void* stack[1024];
/* context pointer for task running main() */
void** task0_context;
/* context pointer for task running task1() */
void** task1_context;

/* task function, must be of type void(void*), the argument
    is filled with the third parameter of init_stack, in
    this case; 0. */
void task1(void* argument) {
    int i = 0;
    while(i < 10) {
        putchar(i + '0');
        i++;
    }
    /* switch to task0_context, save current task in
        task1_context */
    context_switch(task0_context, &task1_context);
}

int main() {
    /* create a new task, using stack, and executing
        function task1 */
    task1_context = init_stack(stack, task1, (void*)0);

    /* switch to task1, store current task in task0_pointer,
        note that task0_context is assigned with the current
        task context. */
    task_context(task1_context, &task0_context);

    /* any code here only gets executed when task1 switches
        back to task0 */
    return 0;
}
```

In the previous example, the main task (task0) creates a new task (task1) and switches to the new task. After 10 seconds, the new task returns to the main task, which exits the program. A simple operation system would call the context_switch function from a task scheduling routine, which might be called by a timer interrupt to generate periodic context switches (known as time slicing).

The previous code could also be created using the setjmp and longjmp library functions, however, this would require a little more work.

4.6   Time on the X32

Being a microcontroller which can be powered down at any moment, there is no way for the X32 to keep track of time. It is therefore impossible to know the time and/or date within an X32 program. Most X32 configurations are, however, equipped with clocks which count fixed time periods elapsed since the last reset. The values from these clocks can be used to get the amount of time elapsed between to points.

The X32 configuration files automatically supply two macros to `x32.h` which return some information about time: `X32_MS_CLOCK` and `X32_US_CLOCK`. These macros return the amount of time elapsed since a specific point in milliseconds and microseconds respectively. Both macros however, are not guaranteed to have a millisecond and microsecond accuracy. An X32 configuration which, for example, is only equipped with a millisecond clock, will simply return this value, times 1000, to `X32_US_CLOCK`. Both values are also 32 bit values, and will overflow in time. The maximum amount of time which can be measured with the clocks is therefore approximately 49 days ($2^{32}/1000/60/60/24$), and 71 minutes ($2^{32}/1000/1000/60$) for the `X32_MS_CLOCK` and `X32_US_CLOCK` respectively.

These macros are also used in the `sleep` and `usleep` functions declared in `x32.h`. These functions both take one parameter containing the amount of milliseconds and microseconds respectively. The functions run a while loop, until the specified amount of time has elapsed, and then return. Note that the same overflow restrictions apply to these functions. The following program prints the amount of seconds elapsed since the program started:

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

int main() {
    int seconds = 0;

    while(1) {
        printf("%d seconds elapsed\r\n", seconds++);
        sleep(1000);
    }
}
```

Chapter 5

Configurability

This chapter describes the configurability system of the X32. Using the configurability system, it is possible to create different setups of the X32 specific for different situations, such as an X32 setup with multiple DPC/PDC components to control a robot, or an X32 setup with multiple high precision timers for real-time applications. Several standard configurations exists, and are listed in Appendix C. When neither of these configurations are sufficient for a particular task, a new configuration can be created as explained in this chapter.

In Section 5.1, it is explained how a new configuration can be started. The second section, Section 5.2 describes how the X32 peripheral set can be configured by modifying the VHDL sourcecode of the X32. Section 5.3 contains a guide for configuring a new X32 configuration through C-style header files, without needing to write or change any VHDL code. In the final section of this chapter it is described how new peripherals can be added to the automatic configuration system of the X32.

5.1 Setting up a new configuration

A X32 configuration is built in its own subdirectory, which is by default a subdirectory of the x32 directory located in a standard X32 download. This directory should contain one file: `config.vh`. This file contains all settings of the new X32 configuration. The file can be build up from scratch, but it is recommended to copy the file from an existing configuration. The `x32-example` configuration file contains all available settings, and comment on how to use them. This configuration can easily be used as the base of a new configuration.

From the configuration file, the configuration specific library (`x32.h` and `x32.cl`) is created, which is used to compile the loader, and finally build the new X32 bitfile (`x32.bit`). The entire compilation process is executed by using the supplied Makefile in the root of the X32 directory. By typing `make <name_of_subdirectory>/x32.bit` the new X32 configuration is compiled into `<name_of_subdirectory>` using `<name_of_subdirectory>/config.vh`.

5.2 Manual configuration

The X32 can be configured manually by editing the VHDL source files of the X32. Manually editing the X32 source files gives great configuration power, as everything can be changed, however does require some knowledge about the VHDL programming language. This section will only cover changing the set of peripheral devices connected to the X32. All changes

to the X32 with respect to peripheral devices are made in two files: `toplevel.vhd` and `peripherals.vhd`. The first file contains the top level VHDL entity, and holds all I/O definitions. When adding, removing or changing peripherals connected to an I/O FPGA pin, changes must be made in this file. The `peripherals.vhd` holds the connection between the X32 and all peripherals. New peripherals must thus be instantiated and connected to the X32 core in this file.

As the X32 supports automatic configuration of these two files, they do not exist as VHDL source files, but as the unprocessed sourcefiles `toplevel._vhd` and `peripherals._vhd`. It is not recommended to make any changes in these files when manually configuring the X32. Instead, these files should first be preprocessed into standard VHDL source files. To create `toplevel.vhd` and `peripherals.vhd`, a new configuration should be set up, as described in the previous section. It is recommended to base the design on a simple configuration such as the `x32-minimal` or `x32-minimal-with-interrupts` configurations, no changes have to be made to the configuration file. When the new configuration directory is created, the two files can be created by executing `make <name_of_subdirectory>/toplevel.vhd` and `make <name_of_subdirectory>/peripherals.vhd` (which is basically the first step of building an automatic configuration). The files can now be edited, after which the configuration build can be finished by executing `make <name_of_subdirectory>/x32.bit`.

As described earlier, the `toplevel.vhd` file contains the top level entity, which contains all FPGA I/O port declarations. New I/O pins can be declared for a new peripheral device by adding the port to the toplevel entity declaration, adding a `LOC` statement to the declaration zone of this file connecting the I/O port to a specific FPGA pin, and routing the pin to the peripherals instance which is located in `peripherals.vhd`.

To actually add, remove or change a peripheral, the `peripherals.vhd` file should be modified. This file consists of all peripheral device instances, and (de)multiplexers to connect these devices to the X32 memory bus. To connect a new peripheral device to the X32, the peripheral VHDL code should first be included or instantiated within this file. The connection with the X32 consists of six signals: `data_in`, `data_out`, `address`, `read`, `write` and `ready`. The `data_in` and `data_out` signals are the data lines from and to the processor respectively. These values must always contain a valid 32-bit value, if a device generates or requires less bits, the value should be sign-extended. The `address` signal contains the memory address the X32 is reading from or writing to, and should be compared to the address of the new peripheral, before acting on the `read` and `write` signals. The `read` and `write` signals are driven by the processor, and are raised whenever the processor wants to read or write a value from a peripheral. After the `read` or `write` signals are asserted, the peripheral address specified by the `address` input should perform the read or write action. When done, the `ready` signal must be asserted, after which the processor will de-assert the `read` or `write` signal. A read and write cycle are shown in Figures 5.1 and 5.2.

As can be read in Section 4.1, peripherals are located at `0x80000000`, and are 32-bit apart. The peripheral index is obtained by removing the two most significant, and the two least significant bits from the address, such that peripheral index 0, corresponds to memory address `0x80000000`, peripheral index 1 corresponds to memory address `0x80000004`, etc. In `peripherals.vhd`, the address is already stripped from the two most significant bits, the two least significant bits should be ignored in this file.

To add a peripheral which allows to be read to, the main multiplexer in the `peripheral` file should be extended with the address the new peripheral listens to. The main multiplexer already strips the two least significant bits, such that the peripheral index can be used. Note

Figure 5.1: Peripheral read cycle

A read action starts with raising the read *signal. At the same time, the address is placed on the* address *signal. The peripheral should then set the data to the* data_out *signal and raise the* ready *signal for one clock cycle. The time* tMEM_READ_CYCLE *is not limited, but should be kept to a minimum.*
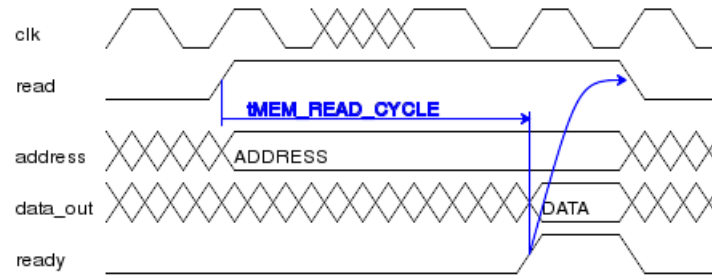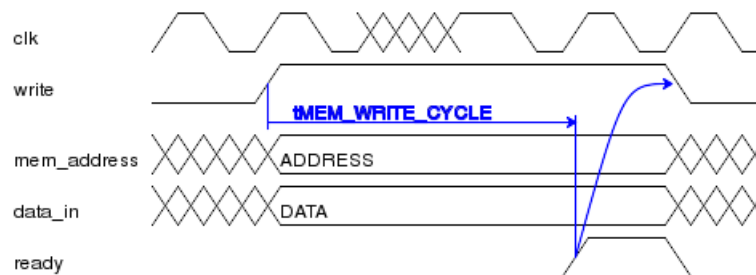

Figure 5.2: Peripheral write cycle

A write action starts with raising the write *signal. At the same time, the address is placed on the* address, *and the data on the* data_in *signal. The peripheral should then save the* data_in *signal, and respond with raising the* ready *signal for one clock cycle. The time* tMEM_WRITE_CYCLE *is not limited, but should be kept to a minimum.*

that the demultiplexer is only addressed by 8 address bits. This is done to decrease the size of the multiplexer, as most designs will not have more than 256 peripherals. To add support for reading from a peripheral located at peripheral index `0x04`, the following code is added to the demultiplexer:

```
-- check only 8 address bits
case address(9 downto 2) is
    ...
    when x"04" =>
        ready <= '1';
        data_out <= peripheral_value;
    ...
end case
```

The `ready` signal is constantly set to logic high whenever reading (or writing) to this peripheral. In most cases this will work fine, as most peripherals are able to perform a read or write signal in a single clock cycle. If this is not the case, the statement should be replaced by `ready <= peripheral_ready`, and the peripheral device should generate its own ready signal. In some cases, the peripheral needs to know whenever the X32 reads from the peripheral. When using buffers for example, the value must be popped from the buffer when the X32 performs a read operation on the buffer. This can be achieved by demultiplexing the `read` signal as follows:

```
peripheral_read <= '1' when address(9 downto 2) = x"04" and read = '1'
    else '0';
```

The `peripheral_read` signal will be high whenever the X32 is reading from the new peripheral device. The length `peripheral_read` will be high depends on the `ready` signal, but will be one clock cycle whenever `ready` is fixed to logic high as shown in the example above.

To add a peripheral which allows to be written to, the peripheral should listen to assertions on the `write` signal, when the `address` signal is equal to the address of the peripherals. A peripheral specific write address can be obtained as follows:

```
peripheral_write <= '1' when address(9 downto 2) = x"04" and write = '1'
    else '0';
```

In this case, the peripheral listens to peripheral index `0x04`. Note that again only 8 bits of the address are checked. When the `peripheral_write` signal is asserted, the peripheral should copy the value on the `data_in` signal to an internal register, to use it for further processing. It is important to remember that the `data_in` signal only contains valid data for the peripheral at the moment `peripheral_write` is high.

Devices can generate IRQs by generating a pulse on one of the IRQ signals. In `peripherals.vhd`, the std_logic_vector `irqs` is available, which is connected to the interrupt controller. When a specific peripheral device has an output line `device_irq`, on which pulses are generated whenever an interrupt is requested, the device can be connected to IRQ index 4 as follows:

```
irqs(4) <= device_irq;
```

Although it is technically possible to connect different devices to the same IRQ using OR-gates, this is not recommended as it will not be possible to detect which device generated the IRQ.

5.3 Automatic configuration

Automatic configuration allows configuring the X32 by modifying a C-style header file, without requiring any knowledge about the VHDL programming language, such that each C programmer can create his own X32 configuration containing all peripheral devices he needs for a specific task. Configuring the X32 is done through modifying the configuration file (`config.vh`), which comes with each configuration.

The following sections describe the different configuration options available for configuring the X32.

5.3.1 X32 core configuration

The X32 core configuration consists of a set of `#define` statements which configure the core of the X32 system. Most of these values should always be set to there default values, since the X32 system is only compatible with the default X32 core. These settings are merely included for future versions of the X32 system which might, for example, support a 64 bit core.

```
/*
 * The target platform, currently not used by the preprocessor
 *  (REQUIRED)
 */
#define TARGET  xs3e400
/*
 * The clock speed, used to compute several timing constants
 *  (REQUIRED)
 */
#define CLOCKSPEED  50000000
/*
 * The amount of bits of a (long) long integer, currently
 *  only 32 is supported (REQUIRED)
 */
#define CORE_SIZE_LONG  32
/*
 * The amount of bits of an integer, currently only 32 is
 *  supported (REQUIRED)
 */
#define CORE_SIZE_INT  32
/*
 * The amount of bits of a short integer, currently only 16
 *  is supported (REQUIRED)
 */
#define CORE_SIZE_SHORT  16
/*
 * The amount of bits of a pointer, currently only 32 is
 *  supported (REQUIRED)
 */
#define CORE_SIZE_POINTER  32
/*
```

```
 * The amount of bits used to address peripheral devices,
 *  8 bits results in a maximum of 256 different devices,
 *  16 in 65536 and so on. The maximum value is
 *  2^CORE_SIZE_POINTER-2. (REQUIRED)
 */
#define PERIPHERAL_ADDRESS_BITS  8
/*
 * The amount of bits used by the peripheral data bus, this
 *  should be the same as CORE_SIZE_LONG, such that each
 *  peripheral can return long integers. Most peripheral
 *  devices require a 32 bit data bus (REQUIRED)
 */
#define PERIPHERAL_DATA_BITS  32
```

5.3.2   ROM Location

The ROM is copied to the RAM at a processor reset. Two variables define where to the ROM is copied: `ROM_LOCATION` and `ROM_ADDRESS_BITS`.

The following rules apply to both variables: The least significant `ROM_ADDRESS_BITS` bits of `ROM_LOCATION` must be 0, and the size of the ROM may not be larger than 2^`ROM_ADDRESS_BITS`.

Default is a `ROM_ADDRESS_BITS` of 16, which means a maximum ROM of 64KB, and a `ROM_LOCATION` of 0x000C0000, or 768K (last 16 bits zero). These settings are default for the bootloader. When the bootloader is running at 768K, the first 768K in RAM are available for programs. When compiling custom software into the ROM of the X32, these values should be set to `ROM_LOCATION` 0x00000000, and `ROM_ADDRESS_BITS` 32, which places the custom ROM at address zero, and allows the entire RAM to be written by the ROM.

```
 /*
  * ROM Target location (REQUIRED)
  */
 #define ROM_LOCATION 0x000C0000
 /*
  * Number of address bits used by the ROM (REQUIRED)
  */
 #define ROM_ADDRESS_BITS 16
```

5.3.3   Reset signal configuration

The X32 reset signal is generally controlled by a peripheral device, such as a button. Two different reset signals are generated, the initial reset and the normal reset. The normal reset always resets the X32, the initial reset may only reset the X32 for the first time. The normal reset is required, the initial reset is optional.

For both reset signals, the macros should contain a valid VHDL condition. The VHDL code generated is as follows:

```
reset <= '1' when NORMAL_RESET else '0').
```

The double reset system is created to allow the use of all 4 buttons on the Spartan 3 Starter Board, instead of allocating 1 as reset button. Instead, the normal reset is initiated when all 4 buttons are pressed. The board however sends an initial pulse on button(3) when it boots up, button(3) must thus reset the X32 on the first time it triggers.

Note: buttons is a signal name generated by the 1:1 input device buttons.

```
/*  initial reset signal (OPTIONAL) */
#define INITIAL_RESET buttons(3) = '1'
/*  normal reset signal (REQUIRED) */
#define NORMAL_RESET_SIGNAL buttons = "1111"
```

5.3.4   Interrupt configuration

The X32 can be compiled with, or without interrupts. Interrupts support is automatically added when the `INTERRUPTS_ENABLE` macro is defined. Furthermore, the amount of IRQ lines, and some standard IRQ indices generated by the processor core itself can be configured in this section.

```
/*
 * define INTERRUPTS_ENABLE to compile the X32 with interrupt
 *  support (OPTIONAL)
 */
#define INTERRUPTS_ENABLE
/*
 * nr of devices which can generate an interrupt (number of
 *  IRQ lines). IRQ's may not be shared between devices.
 *  (REQUIRED if INTERRUPTS_ENABLE)
 */
#define INTERRUPT_COUNT 16
/*
 * device index for the interrupt enable register (the
 *  peripheral will be located at peripherals[index] or
 *  0x80000000 + index * 4) (must be unique) must be defined
 *  to use interrupts (otherwise all interrupts will remain
 *  disables all the time). It may safely remain defined when
 *  interrupts are disabled, the register won't be created)
 *  (REQUIRED if INTERRUPTS_ENABLE)
 */
#define INTERRUPT_ENABLE_INDEX 0x20
/*
 * trap instruction IRQ (mostly used by debugging software)
 *  (OPTIONAL)
 */
#define TRAP_IRQ 0x09
/*
 * overflow IRQ (OPTIONAL)
 */
```

```
#define OVERFLOW_IRQ 0x0E
/*
 * division by zero IRQ (OPTIONAL)
 */
#define DIV0_IRQ 0x0F
/*
 * out of memory irq
 */
#define OOM_IRQ 0x03
/*
 * The following settings define the behavior of the out of
 *  memory irq. OOM_LOWER_BOUND should be set to the maximum
 *  allowed memory address, and OOM_UPPER_BOUND should be set
 *  to the minimum allowed memory address, and must be larger
 *  than OOM_LOWER_BOUND. The area between OOM_LOWER_BOUND and
 * OOM_UPPER_BOUND is therefore forbidden. OOM_LOWER_BOUND
 * should be set  slightly less then the memory size, since
 * some memory space is needed to handle the out of memory
 * interrupt.
 *
 * Note: both values must be smaller than 0x80000000 for the
 *  Xilinx compiler
 *
 */
/* everything above 0x80000000 is ok (peripherals) */
#define OOM_UPPER_BOUND 0x7FFFFFFF
/* everything below 1020K is ok (leaves 4K for interrupt) */
#define OOM_LOWER_BOUND 0x000FF000
```

5.3.5   X32 library configuration

Some often used library functions depend on the X32 configuration used. These functions are not included in the standard library, but in the X32 library, which depends on the X32 configuration. The functions `putchar` and `getchar`, as well as the macro `X32_MS_CLOCK` can be configured using five macros explained below. All macros will appear in `x32.h`, only the `LIBCODE` prefix is replaced by the `X32` prefix.

```
/*
 * Code to get a number which represents time, both in
 *  milliseconds and microseconds. All X32 configurations
 *  should support a millisecond clock, the microsecond
 *  clock may be left out, in which case LIBCODE_US_CLOCK
 *  should return LIBCODE_MS_CLOCK * 1000 (REQUIRED)
 *
 * Usage in C:
 *   ms = X32_MS_CLOCK;
 *   us = X32_US_CLOCK;
```

```
 */
#define LIBCODE_MS_CLOCK (peripherals[0x04])
#define LIBCODE_US_CLOCK (peripherals[0x04]*1000)
/*
 * Code to write a character to stdout (REQUIRED)
 */
#define LIBCODE_X32_STDOUT (peripherals[0x01])
/*
 * Code to get the status of stdout. This should return
 *  nonzero when it is possible to write to stdout (REQUIRED)
 *
 * Usage in C:
 *    while(!X32_STDOUT_STATUS);
 *    X32_STDOUT = 'A';
 */
#define LIBCODE_X32_STDOUT_STATUS (peripherals[0x02] & 0x01)
/*
 * Code to read a character from stdin (REQUIRED)
 */
#define LIBCODE_X32_STDIN (peripherals[0x01])
/*
 * Code to get the status of stdin. This should return
 *  nonzero when a byte is available to read (REQUIRED)
 *
 *  Usage in C:
 *     while(!X32_STDIN_STATUS);
 *     c = X32_STDIN;
 */
#define LIBCODE_X32_STDIN_STATUS (peripherals[0x02] & 0x02)
```

5.3.6 Peripheral configuration

The peripheral configuration section allows configuring several special peripherals which can either be included once, or not in a specific X32 configuration.

```
/*
 * unique id for an unique peripheral bus setup (this number
 *  is returned when reading peripheral device 0), use this
 *  value to identify a design when working with multiple
 *  configurations (REQUIRED)
 */
#define PERIPHERAL_ID 1
/*
 * device index for the instruction counter (the peripheral
 *  will be located at peripherals[index] or 0x80000000 +
 *  index * 4) (must be unique) may be undefined to suppress
 *  an instruction counter device (OPTIONAL)
```

```
 */
#define INSTRUCTION_COUNTER_INDEX 0x03
/*
 * the processor state register, this is a special register
 *  which is connected to the error output lines of the
 *  processor. The error bits are set when the lines go high
 *  (the errors occur), they automatically reset when they
 *  are read. This register can be used to detect processor
 *  exceptions when no interrupt controller is available
 *
 *  The complete register looks like this:
 *  bit 7: trapped
 *  bit 6: out of memory
 *  bit 5: <not used>
 *  bit 4: overflow
 *  bit 3: division by zero
 *  bit 2: <not used>
 *  bit 1: running in simulator
 *  bit 0: booting
 *
 * These bits can also be read using one of the following macros
 *  defined in x32.h, however, as these read the entire state
 *  register, all flags will be reset when reading one of them.
 *
 *  STATE_BOOTING returns true whenever the processor booted since
 *     last time the state was read
 *  STATE_DIVISION_BY_ZERO returns true whenever the processor
 *     encountered a division by zero since last time the state was
 *     read
 *  STATE_OVERFLOW returns true whenever the processor
 *     encountered overflow since last time the state was read
 *  STATE_OUT_OF_MEMORY returns true whenever the processor ran
 *     out of memory since the last time the state was read
 *  STATE_TRAPPED returns true whenever the processor encountered
 *     a trap instruction since last time the state was read
 *  STATE_SIMULATOR returns always false on the X32, and always
 *     true on the bytecode interpreter
 */
#define PROCSTATE_REGISTER_INDEX 0x09
```

5.3.7 Configuring the peripheral bus

Peripheral devices can be added to the peripheral bus using macros. These macros are device type depended, but all look like
ADD_SOME_DEVICE(parameter1, parameter2, parameter3, ...). The devices available in the standard X32 release can be found in Appendix B, along with the exact macro definitions, and some sample code to access them.

5.4 Creating new peripherals

New types of peripheral devices can be added to the configuration system of the X32. Some knowledge about the VHDL programming language, and the use of a preprocessor (such as the C preprocessor) is required to include new peripheral devices.

Each peripheral device included in the X32 configuration must have a header file containing some code which links the peripheral to the X32. It is recommended to put as much of the VHDL code required for the peripheral in normal VHDL files, as they are much easier to write and debug. This section describes how the X32 configuration system works, and how the peripheral header files can be created.

In Section 5.4.1, the configuration system is roughly described, which is important to understand before writing peripheral header files. Sections 5.4.2 to 5.4.4 describe how new header files can be created, and Section 5.5.1 to 5.5.3 contain some examples of peripherals already included in the X32.

5.4.1 The configurability system

Before being able to attach custom components to the X32, some basic understanding of the way the X32 configuration is build is required. Each X32 configuration differs in only three files: `peripherals.vhd`, which contains the link between the X32 and all peripheral devices, `toplevel.vhd`, which contains all in-/output signals, and `x32.h`, which is automatically generated to contain macro names containing all peripheral and interrupt addresses of the included devices.

These three files are generated by preprocessing `peripherals._vhd`, `toplevel._vhd` and `x32._h`. These all include the configuration file `config.vh`, and their contents greatly depend on the contents of the configuration file. The generated VHDL source files `peripherals.vhd` and `toplevel.vhd` are then combined with the other VHDL source files, and synthesized into the X32, while the generated C header file `x32.h` is combined with several C sources and is compiled into the X32 (configuration specific) library. This process is visualized in Figure 5.3. The preprocessor used is called FilePP [7], which behaves much like the standard C preprocessor, but is a bit more powerful. The most important differences with the C preprocessor are listed here:

- FilePP is modular, different modules with different tasks can be imported into FilePP to include extra functionality. These modules are used to add some required extra functionality to the preprocessor.

- All C comment is automatically removed from all files, which allows the use of C comment within VHDL files. A side effect is that its no longer possible to include comment in `x32.h`.

- Multi-line macros can be created using the `#bigfunc` and `#endbigfunc` keywords (see the FilePP Manual [7] for more info).

- The function `UnHex()` is available, which converts C style constants (`0x[0-9A-F]+` for hexadecimal and `0[0-7]+` for octal) to decimal. The decimal values can be converted to the std_logic_vector type using the `conv_std_logic_vector` VHDL function.
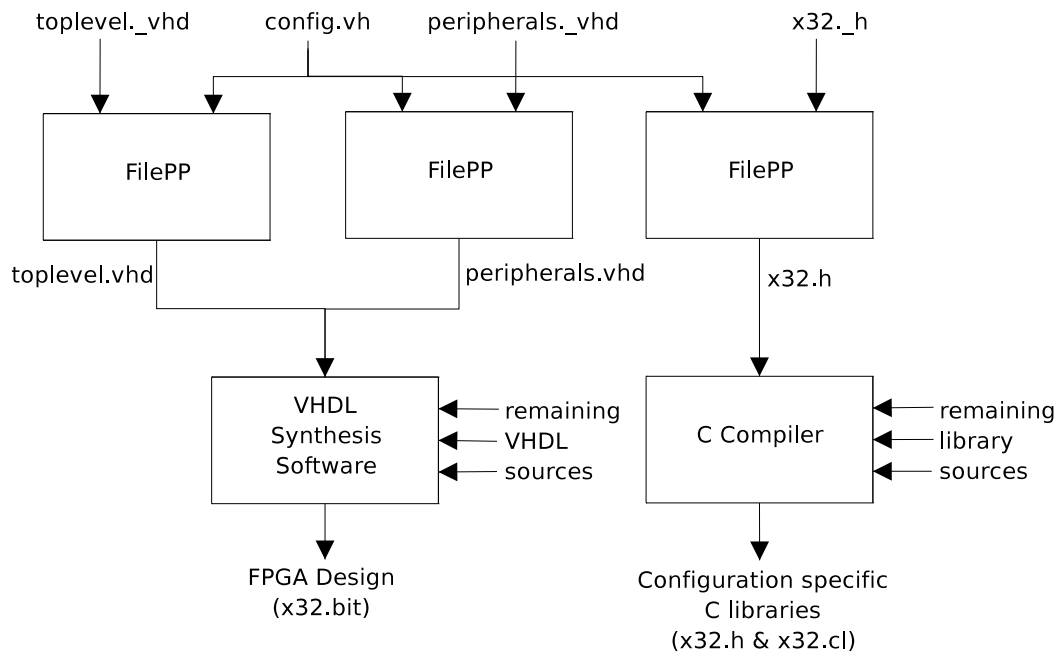
Figure 5.3: Configuration Path

The VHDL source files `peripherals._vhd` *and* `toplevel._vhd`, *and the C header file* `x32._h` *are preprocessed using FilePP according the the configuration stored in* `config.vh`. *The preprocessed VHDL files are then combined with the remaining VHDL source files, and synthesized into an FPGA design. The X32 C library (containing configuration specific functions) is compiled using* `x32.h` *and creates* `x32.cl`. `x32.h` *is also available to C programmers, and will contain the locations of all peripheral devices, all interrupt indices, and the prototypes of all functions available in the X32 C library.*

- The function `pp_to_std_logic_vector` is available, which converts C style constants (`0x[0-9A-F]+` for hexadecimal, `0[0-7]+` for octal and `[0-9]+` for decimal) to `std_logic_vector` format. The second parameter to this function denotes the number of bits. e.g.: `pp_to_std_logic_vector(0x10, 8)` preprocesses into `"00010000"`.

The standard X32 settings are simply stored in macros. For example, the number of interrupts available, is stored in the `INTERRUPT_COUNT` macro. All occurrences of `INTERRUPT_COUNT` in the VHDL source files are thus automatically replaced by the value of the macro by the preprocessor. When adding a peripheral device, code at different locations within both VHDL source files must be added. To achieve this, the configuration file is included several times, at all the locations VHDL code might need to be added. Each time, one of seven macros is defined. By checking which of these macros is defined, code for the correct section can be added. The available sections are described in Section 5.4.3. One of the sections, for example, is the peripheral VHDL section. When generating code for this section, the `__IN_VHDL` macro is defined. To include code in the this section, the following code must be used:

```
#ifdef __IN_VHDL
    -- This line is added to the peripherals.vhd architecture section!
#endif
```

It is imperative that not a single line is generated when preprocessing the configuration file, and none of the section macros is defined. All code statements must thus be surrounded by `#ifdef` and `#endif` sections.

When including a peripheral device, multiple code statements, at different locations must be added to the VHDL source files. Therefore, the preprocessor has support for multi-line macros. Each peripheral device has its own macro (which can be stored in any file, as long as the file is included in the main configuration file). Within this macro, code is generated for the different sections. To make it a little easier, `connection_builder.vh` can be included, which contains several predefined macros to attach a device to the X32 peripheral bus and interrupt system. These macros are listed and explained in Section 5.4.4.

5.4.2 Creating a new peripheral header file

Peripheral header files contain the `ADD_*_DEVICE` macro, and can be included by the configuration file. The file should be named `device.vh` where device is the name of the peripheral device. For easy access, it is recommended to place the file in the `/x32/vhdl/peripherals` directory. The macro is created using the `#bigfunc` statement, which is similar to the standard `#define` statement, except that the end of the macro is not defined by the end of the line, but by the occurrence of the `#endbigfunc` statement. The macro should have at least two parameters: then `NAME` and `INDEX` parameter, which respectively hold a unique name for the device, and the index on which the device is connected to the peripheral bus. All signals used by the peripheral should be prefixed by `NAME`, such that it is possible to include multiple instances of the same peripheral device. The following code shows a very basic macro declaration:

```
#bigfunc ADD_SIMPLE_DEVICE(NAME, INDEX, SOME_OTHER_DEVICE_PARAMETER)
    -- VHDL code connecting simple device named NAME to the X32 at
    --    index INDEX
#endbigfunc
```

VHDL code can be generated by modifying the code the macro generates, which is explained in the next sections. Any extra VHDL files used by a peripheral device should be added to the `PERIPHERAL_SOURCES` variable in the X32 Makefiles `Makefile.linux` and `Makefile.windows`.

5.4.3 The code sections

Using the configurability system, VHDL and C code can be generated at seven different sections within the X32. To identify which section is currently created, each section has its own unique macro, which is defined when the preprocessor is generating code in this section. The list of macros is given here:

- __IN_VHDL: The architecture of the peripherals bus component, any VHDL code can be written here. This section is mostly used to instantiate other VHDL components used by the peripheral device.

- __IN_DECL: The declaration zone of `__IN_VHDL`. Signals used in the `__IN_VHDL` section can be declared here.

- __IN_TL_DECL: The declaration zone of the top level VHDL component. Port maps are made here using the `LOC` attribute. Nothing should have to be written to this zone, the `REGISTER_PIN` macro automatically generates the `LOC` attribute line.

- __IN_SIGLIST: The signal list of the peripheral bus component, used to route signals from the top level component to the peripheral bus component such they are available in the `__IN_VHDL` section. Like the `__IN_TL_DECL` section, this section does not need to be used, the `REGISTER_PIN` macro takes care of routing signals.

- __IN_SIGMAPLIST: The signal mapping section of the top level and the peripheral bus components. Again, this section is completely maintained by the `REGISTER_PIN` macro.

- __IN_MUX: The mux which connects all peripheral devices to the peripheral bus (its not actually a real bus). The code for this section is automatically generated by using the `CREATE_MUX_ENTRY` macro.

- __IN_HEADER: All code in this section is automatically included in `x32.h`. It is thus the only section which should contain C code instead of VHDL code. The `PERIPHERAL_*` macros are automatically generated when using `CREATE_MUX_ENTRY`. The `INTERRUPT_*` macros are automatically created when using `REGISTER_IRQ` of any of its derivatives; `CREATE_INTERRUPT_ON_CHANGE` and `CREATE_INTERRUPT_ON_RISING_EDGE`. Any extra macros can be defined in this section. Note that by adding `#define SOMEMACRO` to the `__IN_HEADER` section, this will result in the creation of the `SOMEMACRO` macro, not in the generation of the `#define SOMEMACRO` statement. Therefore, code which might be interpreted by the preprocessor during configuration, must be written using the `printf` statement, e.g. `printf("#define SOMEMACRO")`. It is not possible to include comments in `x32.h`.

In general, only the `__IN_VHDL` and the `__IN_DECL` sections are required, the code for the other sections can be automatically generated by some predefined macros, which are located in `connection_builder.vh`. The available macros are discussed in Section 5.4.4.

5.4.4 Available macros

The following macros are available when including `connection_builder.vh` in a peripheral device header file. These macros all generate VHDL code for tasks shared amongst many peripheral devices, such as connecting a signal to a specific entry on the multiplexer connecting all peripherals to the X32, and routing an FPGA pin to a signal available in `peripherals.vhd`. Some macros expect signal names as input, all signals should have a unique name and can best be prefixed with the unique device name. All signals should also be of type `std_logic_vector`, even if a signal is only one bit wide.

REGISTER_INPUT_PIN

Format:

```
REGISTER_INPUT_PIN(NAME, WIDTH, PINS)
```

This macro adds an input pin to the toplevel component of the X32, and routes it to the peripheral bus component. The signal is automatically synchronized with the clock using a flipflop. After using this macro, a signal named `NAME` of type `std_logic_vector(WIDTH-1 downto 0)` is available in the `__IN_VHDL` section. The FPGA pins must be specified by the `PINS` parameter. Since the signal is an input signal, it can only be read. Note that one bit signals are not created as `std_logic`, but as `std_logic_vector(0 downto 0)`.

Example:

```
/* Connect input pins T3 and N11 to signal sig_input */
REGISTER_INPUT_PIN(sig_input, 2, "T3 N11")
```

REGISTER_OUTPUT_PIN

Format:

```
REGISTER_OUTPUT_PIN(NAME, WIDTH, PINS)
```

This macro is the counterpart of `REGISTER_INPUT_PIN`. The format is exactly the same, the only difference is the fact that this macro creates an output pin instead of an input pin. The signal can therefore only be assigned to.

Example:

```
/* Connect output pins T3 and N11 to signal sig_output */
REGISTER_OUTPUT_PIN(sig_output, 2, "T3 N11")
```

REGISTER_IRQ

Format:

```
REGISTER_IRQ(INDEX, SIGNAME, INTNAME)
```

This macro registers connects an existing signal to an IRQ line. The first parameter is the IRQ index, the second the name of the signal the IRQ line must be connected to. The final parameter should contain a name identifying the IRQ. The name will be used to create the `INTERRUPT_<NAME>` macro in `x32.h`.

Example:

```
    /* Connect IRQ 4 to signal sig_input(0). The macro
        INTERRUPT_INPUT is automatically created, and will have
        the value 4. */
    REGISTER_IRQ(4, sig_input(0), input)
```

CREATE_MUX_ENTRY

Format:

```
    CREATE_MUX_ENTRY(INDEX, SIGNAME, WIDTH, DEVNAME, SIGREADY)
```

This macro makes a connection from the peripheral device to the X32. It is required for allowing C code to read from the peripheral. The `INDEX` parameter is the index on the peripheral bus. The `SIGNAME` should contain the name of the signal to connect to the peripheral bus, and `WIDTH` the size of the signal (which must not be greater than the `PERIPHERAL_DATA_BITS` setting). The third parameter is used to create the `PERIPHERAL_<DEVNAME>` macro in `x32.h`. The last parameter is connected to the `ready` signal. If the peripheral is able to execute read and write operations in a single clock cycle, this value can be set to `'1'`, otherwise, this value should be set to the name of the ready signal generated by the peripheral device.

Example:

```
    /* Connect the sig_input signal to the peripheral bus. The two
        bit signal can be read from the two least significant bits
        of peripherals[4]. The macro PERIPHERALS_INPUT is
        automatically created and will hold the value 4. */
    CREATE_MUX_ENTRY(4, sig_input, 2, input, '1')
```

CREATE_WRITE_SIGNAL

Format:

```
    CREATE_WRITE_SIGNAL(INDEX, NAME)
```

This macro creates a new signal which is asserted during a write operation of the X32 to a specific peripheral address. This signal can be used by devices the X32 is allowed to write to. At the moment the generated signal is high, the `data_in` signal contains data specific for the current device.

Example:

```
    /* Create a signal named 'write_to_address_8' which is high
        during the time the X32 writes to peripheral device 8. */
    CREATE_WRITE_SIGNAL(8, write_to_address_8)
```

CREATE_READ_SIGNAL

Format:

```
    CREATE_READ_SIGNAL(INDEX, NAME)
```

This macro is the counterpart of the CREATE_WRITE_SIGNAL, and creates a signal which is asserted during a write operation of the X32 from a specific peripheral address. This signal can be used to detect when the X32 reads from a specific device. Some devices need to react on read operations, for example, the RS232 UART pops a byte from the input buffer when the X32 reads it.

Example:

```
/* Create a signal named 'read_from_address_8' which is high
    during the time the X32 reads from peripheral device 8. */
CREATE_READ_SIGNAL(INDEX, NAME)
```

CREATE_REGISTER

Format:

```
CREATE_REGISTER(INDEX, SIGNAME, WIDTH, DEVNAME)
```

The CREATE_REGISTER macro creates a register and attaches it to the peripheral mux. This macro can be used by output devices which expect a constant valid input. The register can be read and written by the X32 Core, and the output value of the register is constantly available for the peripheral device. The INDEX parameter contains the index on the peripheral mux. The SIGNAME contains the name of the signal connected to the output of the register, and can be used as input for the peripheral device. The WIDTH parameter contains the size in bits of the register, and the DEVNAME parameter contains the name of the device, which will appear in x32.h.

Example:

```
/* the following macro creates a register at peripheral index
    0x40. The X32 can read from and write to this 32-bit register,
    and the output is available in sig_to_peripheral, which can
    be used by the peripheral device. The macro
    PERIPHERAL_DPC_PERIOD is created in x32.h and will have the
    value 0x40. */
CREATE_REGISTER(0x40, sig_to_peripheral, 32, DPC_PERIOD)
```

CREATE_INTERRUPT_ON_CHANGE

Format:

```
CREATE_INTERRUPT_ON_CHANGE(INT_INDX, NAME, WIDTH)
```

The CREATE_INTERRUPT_ON_CHANGE macro uses the REGISTER_IRQ macro to generate an irq when a source signal changes. The INT_INDX parameter holds the IRQ index, the NAME parameter the signal name to check for changes, and the WIDTH parameter the size of the NAME parameter.

Example:

```
/* raise an interrupt on irq 6 when any of the 4 signals in inputs
    (inputs must be of type std_logic_vector(3 downto 0) change */
CREATE_INTERRUPT_ON_CHANGE(6, inputs, 4)
```

CREATE_INTERRUPT_ON_RISING_EDGE

Format:

```
CREATE_INTERRUPT_ON_RISING_EDGE(INT_INDX, NAME)
```

The `CREATE_INTERRUPT_ON_RISING_EDGE` macro is similar to the
`CREATE_INTERRUPT_ON_CHANGE` macro, only it creates interrupts on the rising edge of a signal,
rather than on signal change. This macro can also only be used on single signals, of type
`std_logic_vector(0 downto 0)`.

Example:

```
/* generate an interrupt on irq 4 on the rising edge of the signal
    'input' */
CREATE_INTERRUPT_ON_RISING_EDGE(4, input)
```

CREATE_INTERRUPT_ON_FALLING_EDGE

The `CREATE_INTERRUPT_ON_FALLING_EDGE` is similar to the
`CREATE_INTERRUPT_ON_RISING_EDGE` macro, except it generates interrupts on the falling edge
of the source signal.

5.5 Device examples

This section contains some examples on attaching new devices to the X32. The first to
subsections handle a simple output, and a simple input device. The third subsection describes
the Maxon decoder, used by the in2305 configuration of the X32.

5.5.1 A simple output device

This section describes how a simple output device can be made. The device consist of a simple
register connected to the peripheral bus, such that the X32 can read and write the register.
The output of the register is also directly connected to FPGA pins. This device is equivalent
to the 1 to 1 output device.

At first, a new header file must be created for the device. In this case, `1to1.vh` is used.
Then, the macro definition must be created. A macro definition is usually done using the
`#define` statement. However, this only allows macros consisting of one line only. FilePP has
support for a `#bigfunc` statement, which acts like `#define`, except it does not terminate at
the end of the line, but at an `#endbigfunc` statement. All device settings which must be
configurable, must be parameters of the macro. The following macro definition is used:

```
#include "connection_builder.vh"

#bigfunc ADD_1TO1_OUTPUT_DEVICE(NAME, INDEX, WIDTH, PINS)
#endbigfunc
```

The `NAME` parameter here is used as an unique name. All signals created by this device
are prefixed by this name, allowing multiple instances of this device. The second parame-
ter, `INDEX`, is the index on which the device is accessible at the peripheral bus. The `WIDTH`

parameter contains the amount of bits, and thus output pins, the device controls, and finally, the `PINS` parameter should contain the pin names of the output pins. The include of `connection_builder.vh` is required to use any of the macros described in the previous section.

The first step is creating a register, and connecting it to the peripheral bus. The macro `CREATE_REGISTER`, can be used to achieve this in one line, as shown in the following code:

```
#include "connection_builder.vh"

#bigfunc ADD_1TO1_OUTPUT_DEVICE(NAME, INDEX, WIDTH, PINS)
    /* create a WIDTH-bit register, and connect it to the
        peripheral bus at address INDEX. The output of the
        register is connected to the signal NAME_reg, and the
        name of the device, used as macro name in x32.h, is
        NAME. */
    CREATE_REGISTER(INDEX, NAME_reg, WIDTH, NAME)
#endbigfunc
```

The output signal of the register created above must now be connected to an FPGA output pin, which is done by the following code:

```
#include "connection_builder.vh"

#bigfunc ADD_1TO1_OUTPUT_DEVICE(NAME, INDEX, WIDTH, PINS)
    CREATE_REGISTER(INDEX, NAME_reg, WIDTH, NAME)
    /* connect output pins PINS to the signal named NAME */
    REGISTER_OUTPUT_PIN(NAME, WIDTH, PINS)
    /* some vhdl code to connect the output of the register
        (NAME_reg) to the pin signal (NAME) */
    #ifdef __IN_VHDL
        NAME <= NAME_reg;
    #endif
#endbigfunc
```

5.5.2 A simple input device

The simple input device is the counterpart of the simple output device described in the previous section. It connects one or more input pins to the peripheral bus. In addition, it creates an IRQ signal which is raised when the input signal (or one of the input signals) changes. Just like the simple output device, first a macro must be defined in a vhdl header file. The macro used is as follows:

```
#include "connection_builder.vh"

#bigfunc ADD_1TO1_INPUT_DEVICE(NAME, INDEX, INT_INDX, WIDTH, PINS)
#endbigfunc
```

The configuration parameters `NAME`, `INDEX`, `WIDTH`, and `PINS` are the same as the ones used on the output device. The respectively contain the unique name of the device, the index on

the peripheral bus, the amount of bits for the device and the pin names. The new parameter, `INT_INDX`, is used to select an IRQ number. Note the missing `E` in `INT_INDX`. This is done because one macro name must never be a part of another macro name. If `INT_INDEX` would have been used, all instances of `INT_INDEX` would have been replaced by `INT_###`, where `###` would be the contents of the `INDEX` parameter. Changing the name of `INDEX` to `BUS_INDEX` would also solve this problem.

The next step is connecting the input pin of the FPGA to the X32 peripheral bus, using the `REGISTER_INPUT_NAME` and `CREATE_MUX_ENTRY` macros:

```
#include "connection_builder.vh"

#bigfunc ADD_1TO1_INPUT_DEVICE(NAME, INDEX, INT_INDX, WIDTH, PINS)
    /* create the NAME signal, which is connected to input
        pin(s) PINS */
    REGISTER_INPUT_PIN(NAME, WIDTH, PINS)
    /* connect signal NAME with width WIDTH to peripheral bus
        index INDEX. The fourth parameter is the NAME used for
        the PERIPHERAL macro in x32.h */
    CREATE_MUX_ENTRY(INDEX, NAME, WIDTH, NAME)
#endbigfunc
```

Finally, the IRQ signal must be created. The `CREATE_INTERRUPT_ON_CHANGE` macro generates a signal which is asserted whenever the input signal changes. The new signal is then connected to the IRQ signal.

```
#include "connection_builder.vh"

#bigfunc ADD_1TO1_INPUT_DEVICE(NAME, INDEX, INT_INDX, WIDTH, PINS)
    REGISTER_INPUT_PIN(NAME, WIDTH, PINS)
    CREATE_MUX_ENTRY(INDEX, NAME, WIDTH, NAME)
    /* connect IRQ INT_INDX to signal NAME */
    CREATE_INTERRUPT_ON_CHANGE(INT_INDX, NAME, WIDTH)
#endbigfunc
```

The code shown above is identical to the 1 to 1 input device included in all X32 releases.

5.5.3 The IN2305 Maxon decoder device

The maxon decoder is a very specific device used in the TU Delft IN2305 course. The decoder is used to decode the speed and direction of a Maxon motor. The vhdl components for the decoder, which will not be discussed in this section, take the two lines coming from the motor as an input, and generates a 32 bit signed integer representing the speed and direction. The component also has an error output, which is raised for one clock cycle on protocol errors, and must generate an interrupt.

One part of the lab exercise is to see the difference between a hardware (vhdl) decoder, and a decoder written in C. Both motor lines, must therefore also be available as raw inputs, and must generate an interrupt on the rising and falling edges of the lines.

The first step in creating a new component is creating the macro definition. As seen in the previous sections, this must be done using the `#bigfunc` and `#endbigfunc` keywords. To maintain as much configurability as possible, The following settings must be configurable:

- The device name, allowing multiple instances of the device

- The peripheral addresses for both raw signal values, and the decoded value

- The IRQ numbers for both signals, and decoder error

- The pin names for both inputs

This leads to the following macro header:

```
/* the device creation macro */
#bigfunc ADD_MAXON_DEVICE(NAME, INDEX_32, INDEX_A, INDEX_B,
    INT_ERROR, INT_A, INT_B, PIN_A, PIN_B)
#endbigfunc
```

As explained above, both input lines must be connected directly to the peripheral bus, and must generate interrupts on change. The easiest way to accomplish this is simply by including two direct input devices, which do exactly that. Two 1-bit devices are created, to distinguish the two interrupts (if a 2-bit device would have been used, only one interrupt signal would be available):

```
/* include 1to1.vh to gain access to ADD_1TO1_INPUT_DEVICE */
#include "peripherals/1to1.vh"

/* the device creation macro */
#bigfunc ADD_MAXON_DEVICE(NAME, INDEX_32, INDEX_A, INDEX_B,
    INT_ERROR, INT_A, INT_B, PIN_A, PIN_B)

    /* create the 1 to 1 input devices. connect PIN_A to
        INDEX_A and PIN_B to INDEX_B, IRQ INT_A and INT_B are
        automatically created. the signals NAME_A and NAME_B
        will become available containing the synchronized
        inputs of PIN_A and PIN_B (note: these are of type
        std_logic_vector(0 downto 0), not std_logic. the
        macros PERIPHERAL_NAME_A, PERIPHERAL_NAME_B,
        INTERRUPT_NAME_A and INTERRUPT_NAME_B are also
        automatically created in x32.h */
    ADD_1TO1_INPUT_DEVICE(NAME_A, INDEX_A, INT_A, 1, PIN_A)
    ADD_1TO1_INPUT_DEVICE(NAME_B, INDEX_B, INT_B, 1, PIN_B)
#endbigfunc
```

Next, the maxon decoder VHDL component is included in the device. VHDL components must be included in the `__IN_VHDL` section, and must be valid VHDL code. The code requires two additional signals, the error signal and the 32-bit decoder output signal. Since the error signal can directly be connected to the IRQ line, this signal is called `NAME_int`. The 32-bit value signal is called `NAME_dig`. They must both be declared in the `__IN_DECL` section.

```
#include "peripherals/1to1.vh"
```

```
#bigfunc ADD_MAXON_DEVICE(NAME, INDEX_32, INDEX_A, INDEX_B,
    INT_ERROR, INT_A, INT_B, PIN_A, PIN_B)

    ADD_1TO1_INPUT_DEVICE(NAME_A, INDEX_A, INT_A, 1, PIN_A)
    ADD_1TO1_INPUT_DEVICE(NAME_B, INDEX_B, INT_B, 1, PIN_B)

    #ifdef __IN_DECL
        /* declare NAME_dig and NAME_int */
        signal NAME_dig : std_logic_vector(31 downto 0);
        signal NAME_int : std_logic;
    #endif
    #ifdef __IN_VHDL
        /* the port map with the maxon decoder */
        l_NAME: entity work.maxon(behaviour)
            port map (
                /* clk and reset are the available clock and
                    reset signals */
                clk => clk,
                reset => reset,
                /* NAME_A and NAME_B are
                    std_logic_vector(0 downto 0), the inputs
                    a and b are of type std_logic */
                a => NAME_A(0),
                b => NAME_B(0),
                err => NAME_int,
                value => NAME_dig
            );
    #endif
#endbigfunc
```

The final step is connecting the `NAME_dig` signal with the peripheral bus, so it can be read, and connecting `NAME_int` to the required IRQ line.

```
#include "peripherals/1to1.vh"

#bigfunc ADD_MAXON_DEVICE(NAME, INDEX_32, INDEX_A, INDEX_B,
    INT_ERROR, INT_A, INT_B, PIN_A, PIN_B)

    ADD_1TO1_INPUT_DEVICE(NAME_A, INDEX_A, INT_A, 1, PIN_A)
    ADD_1TO1_INPUT_DEVICE(NAME_B, INDEX_B, INT_B, 1, PIN_B)

    /* connect NAME_dig to the peripheral bus at address
        INDEX_32, the macro PERIPHERAL_NAME_DECODED is added
        to x32.h */
    CREATE_MUX_ENTRY(INDEX_32, NAME_dig, 32, NAME_DECODED)

    /* connect NAME_int to IRQ INT_ERROR */
```

```
      REGISTER_IRQ(INT_ERROR, NAME_int, NAME_ERROR)

  #ifdef __IN_DECL
      signal NAME_dig : std_logic_vector(31 downto 0);
      signal NAME_int : std_logic;
  #endif
  #ifdef __IN_VHDL
      l_NAME: entity work.maxon(behaviour)
          port map (
              clk => clk,
              reset => reset,
              a => NAME_A(0),
              b => NAME_B(0),
              err => NAME_int,
              value => NAME_dig
          );
  #endif
#endbigfunc
```

Bibliography

[1] Arjan van Gemund: *IN2305 Course Website*,
http://www.st.ewi.tudelft.nl/ gemund/Courses/In4073/index.html (2006)

[2] Arjan van Gemund: *IN4073 Course Website*,
http://www.st.ewi.tudelft.nl/ gemund/Courses/In2305/index.html (2006)

[3] Arjan van Gemund: *X32 Download Site*, http://x32.ewi.tudelft.nl/

[4] John Hauser: *Softfloat Library*, http://www.jhauser.us/arithmetic/SoftFloat.html
(2006)

[5] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language*, Prentice
Hall P T R (2004)

[6] Jean J. Labrosse: *MicroC OS II: The Real Time Kernel*, Newnes (2002)

[7] Darren Miller: *FilePP Website*, http://www.cabaret.demon.co.uk/filepp/ (2003)

[8] Andrew Rogers: *Spartan3 JTAG programming tools for GNU/Linux*,
http://www.rogerstech.force9.co.uk/xc3sprog/

[9] SourceForce.net: http://www.sourceforge.net

[10] Sijmen Woutersen: *The X32 Softcore: A top-down approach on processor design*, Soft-
ware Technology Department, TU Delft (2006)

[11] Tics Realtime: *Tics Realtime Multi-tasking Kernel*,
http://concentric.net/ tics/ticsinfo.htm (2000)

[12] Xilinx Inc.: *Spartan-3 Starter Kit Board User Guide*, Xilinx Inc. (2004)

[13] Xilinx Inc. *Xilinx Webpack ISE*,
http://www.xilinx.com/ise/logic_design_prod/webpack.htm

Appendix A

Supported library functions

Allthough the library is not completely ported yet, a great number of standard C library functions are allready available for the X32. This appendix lists all available functions, macro's and variables available in the X32 library. None of these functions are documented here.

A.1   Standard C library

The following sections list the standard C library functions. Documentation for these functions can be found in most C manuals.

A.1.1   assert.h

assert

A.1.2   ctype.h

isalnum	islower	isupper
isalpha	isprint	isxdigit
iscntrl	ispunct	tolower
isdigit	isspace	toupper
isgraph		

A.1.3   limits.h

CHAR_BIT	LONG_MAX	SHRT_MIN
CHAR_MAX	LONG_MIN	UCHAR_MAX
CHAR_MIN	SCHAR_MAX	UINT_MAX
INT_MAX	SCHAR_MIN	ULONG_MAX
INT_MIN	SHRT_MAX	USHRT_MAX

A.1.4   setjmp.h

jmp_buf	setjmp	longjmp

A.1.5   stdarg.h

va_list	va_start	va_arg
va_end		

A.1.6   stddef.h

NULL	size_t

A.1.7   stdio.h

puts	printf	vprintf
sprintf	vsprintf	puts

Note: `putchar` and `getchar` are configuration dependent, and are therefore included in the `X32` library.

A.1.8   stdlib.h

atoi	malloc	div
atol	free	ldiv
strtol	abort	div_t
strtoul	exit	ldiv_t
rand	atexit	EXIT_SUCCESS
srand	abs	EXIT_FAILURE
calloc	labs	

Note: malloc can only allocate blocks of memory within a user defined memory space. To use malloc, statically allocate a memory block with the name `malloc_memory`. In addition, an integer named `malloc_memory_size` is required which should hold the size of the memory block. The following code allocates 1KB for malloc to use:

```
char malloc_memory[1024];
malloc_memory_size = 1024;
```

A.1.9   string.h

memchr	strcmp	strlwr
memcmp	strcpy	strncmp
memcpy	strcoll	strncpy
memmove	strcspn	strspn
memset	strerror	strtok
strchr	strlen	strupr
strcat		

A.2   Other libraries

The following subsections contain all non-standard libraries available for the X32. Documentation for the softfloat library can be found at the softfloat website [4]. The X32 library is discussed in Chapter 4.

A.2.1 softfloat.h

float32	float32_round_to_int	float32_eq
float_detect_tininess	float32_add	float32_le
float_rounding_mode	float32_sub	float32_lt
float_exception_flags	float32_mul	float32_eq_signaling
float_raise	float32_div	float32_le_quiet
int32_to_float32	float32_rem	float32_lt_quiet
float32_to_int32	float32_sqrt	float32_is_signaling_nan
float32_to_int32_round_to_zero		

A.2.2 x32.h

peripherals	restore_execution_level	lock
get_execution_level	combine_stackframe	unlock
set_execution_level	LOCK	putchar
getchar	clock	CLOCKS_PER_SEC

Note: all configuration specific macro's are not listed here.

Appendix B

Supported peripheral devices

This chapter lists all peripheral devices contained in the standard release of the X32, including information on how to include them in a X32 configuration, and how to access them from C.

B.1 RS232 UART

The RS232 UART device is the main communication device of the X32. It allows sending and receiving bytes over two lines according to the RS232 specifications, perfect for communicating with other computer systems. The settings for the RS232 device can not be changed by software, and only the bautrate can be changed in the configuration file. The RS232 settings for the device are listed in Table B.1.

The device is connected to the X32 with two registers; a data and a status register. Writing to the 8-bit data register causes the byte to be placed in the output buffer. The RS232 unit constantly polls the buffer for new bytes, and when one is found, it codes the byte onto the RS232 TX line. At the same time, the RS232 is continuously monitoring the RS232 RX line. When a byte is received and decoded, it is placed into the input buffer. When reading from the data register, the first byte in the buffer is returned, and removed from the buffer. Any bytes received when the hardware buffer is full are lost. The status register is a two-bit register, which should be monitored before writing to or reading from the data register. When bit 0 is 1, the data register can be written to, when bit 0 is 0, the buffer is full and the device is busy sending. When writing to the data register when bit 0 of the status register is 0, the byte is lost. When bit 1 of the status register is 1, a byte is waiting in the input buffer. When this bit is 0, the input buffer is empty. When reading the data register when no byte is waiting, the result is undefined.

Table B.1: RS232 device settings

Setting name	Value
Bautrate	Configuration specific
Stop bits	1
Parity	None
Handshaking	None

Two interrupts are generated when a byte is send and received. Note that a successful send does not mean the byte is also successfully received by the other endpoint.

The following code adds an RS232 to a configuration file:

```
ADD_RS232_DEVICE(primary, 0x01, 0x02, 8, 8, 115200,
    "R13", "T13", 0x03, 0x04)
```

The first parameter (`primary`) is an identifier, which must be unique in the configuration file. The second and third parameters (`0x01`, `0x02`) are the addresses on which respectively the data and status register are connected to the peripheral bus. The fourth and fifth parameters (both `8`) contain the sizes of the input buffer and output buffer in bytes respectively. The sixth parameter (`115200`) contains the bautrate of the RS232 connection. This should be the same as the bautrate of the other endpoint, and can not be changed by software. The seventh and eighth parameters (`"R13"` and `T13`) contain the pin names of the TX and RX line respectively. Finally, the ninth and tenth parameters (`0x03`, `0x04`) contain the IRQ numbers for the RX and TX interrupts.

Note that the bautrate is generated using complete clock cycles, and may be at most half the clock speed. On a 50MHz clock, the maximum bautrate is thus 25000000, and the time for each bit must be dividable by 20ns. A bautrate of 20000000 is thus not possible, since this would require a period of 50ns, which is not dividable by the 20ns clock period.

B.2 4x7 Segment display

The 4x7 display segment device is used to control 4x7 display segment devices such as the display of the Spartan 3 Starter Board. The display is controller by 12 lines, 8 controlling each led for one of the four 7-segment display, and 4 lines to select the 7-segment display (see the Spartan 3 Starter Board Manual [12] for more information.

The device creates a 16 bit register at a specified peripheral address. The contents of this register is written in hexadecimal to the display. Using this device, it is not possible to individually control the display segments.

The following code adds control for the display device to a configuration file:

```
ADD_4x7SEGDISP_DEVICE(display, 0x05,
    "E14 G13 N15 P15 R16 F13 N16 P16", "E13 F14 G14 D14")
```

The first parameter (`display`) is an identifier, which must be unique in the configuration file. The second parameter (`0x05`) is the peripheral bus address on which the register is created. The third parameter are the eight data pins, and the last parameter the four control pins of the display.

B.3 One-to-one input and outputs

One-to-one in-/output devices are direct connections between the X32 peripheral bus and FPGA pins. Up to 32 pins can be controlled per device (after which each bit of the peripheral bus connection is used). When only one pin is connected, only the least significant bit is used.

Input pins are automatically buffered using a flip-flop, to synchronize the incoming value with the X32 clock. Also, an interrupt is generated whenever the input line switches from low to high, or from high to low. When multiple pins are connected to the same port, an
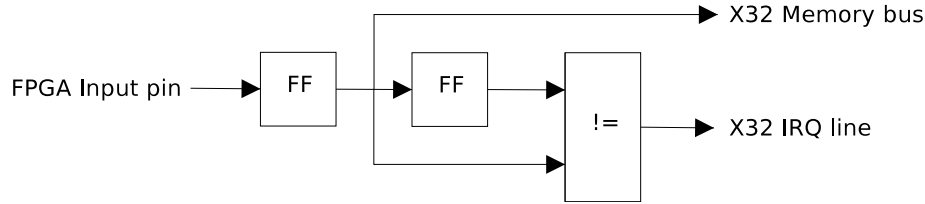
Figure B.1: One-to-one input device

interrupt indicates a change on any of the input lines. The block scheme of a 1-bit input device is shown in Figure B.3.

The following code adds direct control eight input, and eight output pins, which are connected to the Spartan 3 Starter Board switches and led respectively:

```
ADD_1TO1_INPUT_DEVICE(switches, 0x06, 0x08, 8,
    "K13 K14 J13 J14 H13 H14 G12 F12")
ADD_1TO1_OUTPUT_DEVICE(leds, 0x07, 8,
    "P11 P12 N12 P13 N14 L12 P14 K12")
```

For both devices, the first parameter (`switches`, `leds`) are identifiers, which must be unique in the configuration file. The second parameter (`0x06`, `0x07`) are the peripheral bus addresses on which the devices can be accessed. The third parameter for the input device is the IRQ to use. The fourth parameter on the input device, and third parameter on the output device (8), represents the size of the input data in bits. All 8 leds, and all 8 switches are controlled using the same peripheral device. The last parameter for both devices contains the FPGA pins names for the in-/output lines.

The following example links the switches to the leds. The leds are automatically updated when the state of on of the switches is changed:

```
int main() {
    INTERRUPT_ADDRESS(0x08) = &switch_handler();
    INTERRUPT_PRIORITY(0x08) = 10;
    ENABLE_INTERRUPT(0x08);
    ENABLE_INTERRUPT(GLOBAL_INTERRUPT);
    while(1);
}

void switch_handler() {
    peripherals[0x07] = peripherals[0x06];
}
```

B.4   Clock

A clock device is a simple register, which is incremented each n clock cycles, and can therefore be used to time pieces of code. Each clock is reset to zero on a processor reset, but will eventually overflow, and should thus never be used to measure absolute time.

The following code adds a millisecond clock to a configuration file:

```
ADD_CLOCK_DEVICE(ms_clock, 0x04, 50000)
```

The first parameter (`ms_clock`) is an identifier, which must be unique in the configuration file. The second parameter (`0x04`) is the peripheral bus address on which the counter value can be read. The last parameter (`50000`) is the number of clock cycles to wait between counter increments. On a 50MHz clock, the example above will thus increment each millisecond.

The following example shows how to use the clock to time the C function `function()`:

```
unsigned start, stop;
start = ((unsigned*)peripherals)[0x04];
function();
stop = ((unsigned*)peripherals)[0x04];
printf("function() took %d ms\r\n", stop - start);
```

Note that a single overflow still gives a correct result, since the overflow is canceled out by a second overflow caused by the subtraction. Double overflow, which is caused when `function()` takes longer than the maximum period of time the clock register can hold, can not be corrected.

B.5 Timer

A timer device counts to a certain amount of clock cycles. Each time it reaches this amount, is fires an interrupt, and it restarts counting. It can therefore be used to execute code at a specific interval. The timer contains a register which holds the amount of clock cycles to count. The register can both be read, and be written through the peripheral bus.

The following code adds a timer device to a configuration file:

```
ADD_TIMER_DEVICE(timer1, 0x26, 0x01)
```

The first parameter (`timer1`) is an identifier, which must be unique in the configuration file. The second parameter (`0x26`) is the peripheral bus address on which the counter value can be read. The last parameter (`0x01`) is the IRQ for the timer interrupt.

The following example shows how the timer can be used to print a dot each 500 milliseconds:

```
int main() {
    /* assume 50MHz clock */
    ((unsigned*)peripherals)[0x26] = 500*50000;
    INTERRUPT_ADDRESS(0x01) = &timer_handler();
    INTERRUPT_PRIORITY(0x01) = 10;
    ENABLE_INTERRUPT(0x01);
    ENABLE_INTERRUPT(GLOBAL_INTERRUPT);
    while(1);
}

void timer_handler() {
    printf(".");
}
```

B.6 Watchdog timer

The timer handled in the previous section can also be used to create a watchdog timer by using its interrupt signal as a reset signal. This can be done by including a new timer without interrupt support (the timer may be connected to an interrupt, but it will never trigger, since it also causes a hardware reset). The following code will create a timer without interrupt (the `#undef` and `#define` statements are not required when compiling without interrupt support):

```
#undef INTERRUPTS_ENABLE
ADD_TIMER_DEVICE(watchdog, 0x27, 0x00)
#define INTERRUPTS_ENABLE
```

The timer device will automatically create a signal named `watchdog_int`. This signal is normally connected to the IRQ line, but in this case, it must be connected to the reset signal:

```
#define NORMAL_RESET_SIGNAL buttons = "1111" or watchdog_int = '1'
```

The watchdog timer is controlled just like a normal timer. It can be disabled by setting its period to zero.

B.7 Software interrupt

A software interrupt device is a very simple device which generates an interrupt when it is written to. The following code adds a software interrupt device to a configuration file:

```
ADD_SOFTINT_DEVICE(softint1, 0x21, 0x00)
```

The first parameter (`softint1`) is an identifier, which must be unique in the configuration file. The second parameter (`0x21`) is the peripheral bus address on which a write will result in an interrupt. The last parameter (`0x00`) is the IRQ for the software interrupt. See Section 4.3 on how to use software interrupts.

B.8 Pulse width modulator

A pulse width modulated signal is a signal consisting of a variable length pulse which repeats itself at a fixed interval. Two standard devices are available to use pulse width modulation: a digital to pulse converter (DPC), which converts a digital value into a pulse width modulated signal, and a pulse to digital converter (PDC), which does the opposite. The dpc can be used to create a pulse width modulated signal, while the PDC can be used to sense and decode pulse width modulated signals.

B.8.1 Digital to pulse converter

The digital to pulse converter converts two 32-bit inputs, period and pulse width, to a pulse width modulated signal. Both inputs are directly exposed to the peripheral bus, and should be accessed as unsigned integers. The period register contains the number of clock cycles after which the pulse repeats itself, and the pulse width register the number of clock cycles the pulse is high. When the pulse width register is zero, the line is continuously low, when the

pulse width register is greater than, or equal to the period register, the line is continuously high.

The syntax to add a DPC device to the configuration file is as follows:

```
ADD_DPC_DEVICE(dpc1, 0x40, 0x41, "T3")
```

The first parameter (`dpc1`) is an identifier, which must be unique in the configuration file. The second and third parameter (`0x40`, `0x41`) are the peripheral addresses of the period and pulse width registers respectively. Finally, the fourth parameter (`"T3"`) contains the FPGA output pin name for the pulse width modulated signal.

The following C code uses the dpc created above:

```
/* initialize at a period of 40 ms (assume 50MHz clock) */
((unsigned int*)peripherals)[0x40] = 40*50000;
/* duty cycle of 20 ms (assume 50MHz clock) */
((unsigned int*)peripherals)[0x41] = 20*50000;
```

Both the period, and the pulse width registers can be read back by reading the peripherals array. Using the period register, it is very easy to set the dpc to a 50% duty cycle:

```
/* set dpc 1 to a duty cycle of 50% */
((unsigned int*)peripherals)[0x40] =
    (((unsigned int*)peripherals)[0x41]*50)/100;
```

B.8.2 Pulse to digital converter

The pulse to digital converter is the counterpart of the digital to pulse converter, and is used to sense and decode pulse width modulated signals. The device has one 32-bit output exposed to the peripheral bus. This output contains the number of clock cycles the previous pulse was high, and is updated each time a pulse ends. In addition, it generates an interrupt each time the output is updated.

The syntax to add a PDC device to the configuration file is as follows:

```
ADD_PDC_DEVICE(pdc1, 0x30, 0x0A, "C15")
```

The first parameter (`pdc1`) is an identifier, which must be unique in the configuration file. The second parameter (`0x30`) is the address of the pdc output on the peripheral bus. The third parameter (`0x0A`) contains the IRQ number for the device, and the last parameter (`"C15"`) the FPGA input pin name for the pulse width modulated signal.

The following C code prints the value of the pdc in milliseconds, each time the pdc receives a new pulse:

```
int main() {
    INTERRUPT_ADDRESS(0x0A) = &pdc_handler();
    INTERRUPT_PRIORITY(0x0A) = 10;
    ENABLE_INTERRUPT(0x0A);
    ENABLE_INTERRUPT(GLOBAL_INTERRUPT);
    while(1);
}
```

```
void pdc_handler() {
    /* assume 50MHz clock */
    int ms = peripherals[0x30] / 50000;
    printf("pulse width: %d ms\r\n", ms);
}
```

B.9   PS/2 Reader

The PS/2 reader device is able to decode signals from a PS/2 port, which might be connected to a keyboard or mouse. Currently, the PS/2 port can only be read, writing is not (yet) supported.

The syntax to add a PS/2 reader device to the configuration file is as follows:

```
ADD_PS2_DEVICE(ps2, 0x14, 0x15, 0x0C, 8, "M16", "M15")
```

The first parameter (`ps2`) is an identifier, which must be unique in the configuration file. The second and third parameters (`0x14`, `0x15`) are the addresses of the PS/2 input and PS/2 status on the peripheral bus respectively. The last received byte can be read from the first register, the second will return 1 when a byte is available, 0 otherwise. The fourth parameter (`0x0C`) contains the IRQ number for the device, and the fifth parameter contains the size of the input buffer (in bytes). The sixth and seventh parameters (`"M16"`, `"M15"`) contain the FPGA pins for the PS/2 clock and data signals respectively.

Appendix C

Default configurations

This chapter contains information about the default configurations which ship with the X32 download. Each of the following sections contain a small description about this configuration, and the peripherals and interrupts available when using that configuration.

C.1 Core

The X32 clean configuration (located in /X32-clean) contains no peripherals whatsoever. It is merely used to compile the core on its own to gather statistics about the X32 core. Some configuration details are listed in Table C.1.

Table C.1: X32 Core Configuration

Name	X32 Core
Location	/x32-core
Peripheral bus ID	1
Size (400K Spartan 3)	47%
Interrupts enabled	no
Reset signal	reset button

C.2 Minimal

The X32 minimal configuration (located in /x32-minimal) contains a minimal set of peripherals. Some configuration details are listed in Table C.2, and the supported peripherals are listed in Table C.3.

C.3 Minimal with interrupts

The X32 minimal with interrupts configuration is similar to the X32 minimal configuration, with the exception of interrupts being enabled in this configuration. This configuration is able to run most software, including real time operating systems such as uCos [6] and tics [11].

Table C.2: X32 Minimal Configuration

Name	X32 Minimal
Location	/x32-minimal
Peripheral bus ID	2
Size (400K Spartan 3)	49%
Interrupts enabled	no
Reset signal	All 4 buttons

Table C.3: X32 Minimal Peripheral Devices

Type	Location macro	Size	Access
Unique ID	`PERIPHERAL_UID`	32	r
RS232 Channel	`PERIPHERAL_PRIMARY_DATA`	8	rw
RS232 Status Register	`PERIPHERAL_PRIMARY_STATUS`	2	r
Instruction Counter	`PERIPHERAL_INSTRCNTR`	32	r
MS Counter	`PERIPHERAL_MS_COUNTER`	32	r
4x7 Segment Display	`PERIPHERAL_DISPLAY`	16	rw
Switches	`PERIPHERAL_SWITCHES`	8	r
Leds	`PERIPHERAL_LEDS`	8	rw
Buttons	`PERIPHERAL_BUTTONS`	4	r
Processor State Register	`PERIPHERAL_PROCSTATE`	4	r

Interrupts marked with an asterisk are critical, and are not blocked by the global interrupt

Some configuration details are listed in Table C.4, the supported peripherals are listed in Table C.5, and the interrupts in Table C.6. All critical interrupts are marked with an asterisk.

Table C.4: X32 Minimal with Interrupts Configuration

Name	X32 Minimal With Interrupts
Location	/x32-minimal-interrupts
Peripheral bus ID	3
Size (400K Spartan 3)	59%
Interrupts enabled	yes
Reset signal	All 4 buttons

Table C.5: X32 Minimal with Interrupts Peripheral Devices

Type	Location macro	Size	Access
Unique ID	`PERIPHERAL_UID`	32	r
RS232 Channel	`PERIPHERAL_PRIMARY_DATA`	8	rw
RS232 Status Register	`PERIPHERAL_PRIMARY_STATUS`	2	r
Instruction Counter	`PERIPHERAL_INSTRCNTR`	32	r
MS Counter	`PERIPHERAL_MS_COUNTER`	32	r
4x7 Segment Display	`PERIPHERAL_DISPLAY`	16	rw
Switches	`PERIPHERAL_SWITCHES`	8	r
Leds	`PERIPHERAL_LEDS`	8	rw
Buttons	`PERIPHERAL_BUTTONS`	4	r
Processor State Register	`PERIPHERAL_PROCSTATE`	4	r
Interrupt Enable Register	`PERIPHERAL_INT_ENABLE`	32	rw
Software Interrupt	`PERIPHERAL_SOFTINT1`	n/a	w
Timer 1 Period	`PERIPHERAL_TIMER1_PERIOD`	32	rw
Timer 2 Period	`PERIPHERAL_TIMER2_PERIOD`	32	rw

C.4   IN2305

The IN2305 configuration is a configuration specifically made for the TU Delft IN2305 labcourse [1]. It is based on the x32-minimal configuration, with support for interrupts. In addition, a DPC and Maxon decoder are added to control a Maxon motor.

Some configuration details are listed in Table C.7, the supported peripherals are listed in Table C.8, and the interrupts in Table C.9. All critical interrupts are marked with an asterisk.

C.5   IN4073

The IN4073 configuration is a configuration specifically made for the original TU Delft IN4073 labcourse [2] to control the Picolo model helicopter. It is based on the x32-minimal configu-

Table C.6: X32 Minimal with Interrupts Interrupts

Type	Index macro
Software Interrupt*	`INTERRUPT_SOFTINT1`
Timer 1 Interrupt	`INTERRUPT_TIMER1`
Timer 2 Interrupt	`INTERRUPT_TIMER2`
RS232 Rx	`INTERRUPT_PRIMARY_RX`
RS232 Tx	`INTERRUPT_PRIMARY_TX`
Buttons	`INTERRUPT_BUTTONS`
Switches	`INTERRUPT_SWITCHES`
TRAP Instruction*	`INTERRUPT_TRAP`
Overflow*	`INTERRUPT_OVERFLOW`
Division By Zero*	`INTERRUPT_DIVISION_BY_ZERO`
Out Of Memory*	`INTERRUPT_OUT_OF_MEMORY`

Interrupts marked with an asterisk are critical, and are not blocked by the global interrupt

Table C.7: X32 IN2305 Configuration

Name	X32 IN2305
Location	/x32-in2305
Peripheral bus ID	2305
Size (400K Spartan 3)	66%
Interrupts enabled	yes
Reset signal	All 4 buttons

Table C.8: X32 IN2305 Peripheral Devices

Type	Location macro	Size	Access
Unique ID	`PERIPHERAL_UID`	32	r
RS232 Channel	`PERIPHERAL_PRIMARY_DATA`	8	rw
RS232 Status Register	`PERIPHERAL_PRIMARY_STATUS`	2	r
Instruction Counter	`PERIPHERAL_INSTRCNTR`	32	r
MS Counter	`PERIPHERAL_MS_COUNTER`	32	r
US Counter	`PERIPHERAL_US_COUNTER`	32	r
4x7 Segment Display	`PERIPHERAL_DISPLAY`	16	rw
Switches	`PERIPHERAL_SWITCHES`	8	r
Leds	`PERIPHERAL_LEDS`	8	rw
Buttons	`PERIPHERAL_BUTTONS`	4	r
Processor State Register	`PERIPHERAL_PROCSTATE`	4	r
Interrupt Enable Register	`PERIPHERAL_INT_ENABLE`	32	rw
Software Interrupt	`PERIPHERAL_SOFTINT1`	n/a	w
Timer 1 Period	`PERIPHERAL_TIMER1_PERIOD`	32	rw
Timer 2 Period	`PERIPHERAL_TIMER2_PERIOD`	32	rw
DPC Period	`PERIPHERAL_DPC_PERIOD`	32	rw
DPC Width	`PERIPHERAL_DPC_WIDTH`	32	rw
Maxon decoder input 1	`PERIPHERAL_ENGINE_A`	1	r
Maxon decoder input 2	`PERIPHERAL_ENGINE_B`	1	r
Maxon decoder output	`PERIPHERAL_ENGINE_DECODED`	32	r
Custom output	`PERIPHERAL_CUSTOM`	4	rw

Table C.9: X32 IN2305 Interrupts

Type	Index macro
Software Interrupt*	`INTERRUPT_SOFTINT1`
Timer 1 Interrupt	`INTERRUPT_TIMER1`
Timer 2 Interrupt	`INTERRUPT_TIMER2`
RS232 Rx	`INTERRUPT_PRIMARY_RX`
RS232 Tx	`INTERRUPT_PRIMARY_TX`
Buttons	`INTERRUPT_BUTTONS`
Switches	`INTERRUPT_SWITCHES`
TRAP Instruction*	`INTERRUPT_TRAP`
Overflow*	`INTERRUPT_OVERFLOW`
Division By Zero*	`INTERRUPT_DIVISION_BY_ZERO`
Maxon input 1 change	`INTERRUPT_ENGINE_A`
Maxon input 2 change	`INTERRUPT_ENGINE_B`
Maxon decoder error	`INTERRUPT_ENGINE_ERROR`
Out Of Memory*	`INTERRUPT_OUT_OF_MEMORY`

Interrupts marked with an asterisk are critical, and are not blocked by the global interrupt

ration, with support for interrupts. In addition, five PDCs and four DPCs where added.

Some configuration details are listed in Table C.10, the supported peripherals are listed in Table C.11, and the interrupts in Table C.12. All critical interrupts are marked with an asterisk.

Table C.10: X32 IN4073 Configuration

Name	X32 IN4073
Location	/x32-in4073
Peripheral bus ID	4703
Size (400K Spartan 3)	75%
Interrupts enabled	yes
Reset signal	All 4 buttons

C.6 IN4073-TREX

The IN4073-TREX configuration is a configuration specifically made for the new TU Delft IN4703 labcourse [2], controlling the T-Rex model helicopter. It is based on the x32-minimal configuration, with support for interrupts. In addition, a special component was included which communicates to the microcontroller available at the T-Rex helicopter.

Some configuration details are listed in Table C.13, the supported peripherals are listed in Table C.14, and the interrupts in Table C.15. All critical interrupts are marked with an asterisk.

C.7 Example

The X32 example configuration is a large configuration containing all supported peripheral devices. The configuration file is also filled with comments on how to modify this file, and it can be used as the base of new configurations.

Due to the large amount of peripherals available in the X32 example configuration, the longest path is slightly longer than the allowed 20ns. The X32 example configuration is therefore overclocked by the 50MHz clock. Although the longest path estimates are based on extreme conditions, and several tests show that the X32 example configuration works correctly at several devices, it is not recommended to use this configuration.

Some configuration details are listed in Table C.16, the supported peripherals are listed in Table C.17, and the interrupts in Table C.18. All critical interrupts are marked with an asterisk.

C.8 RS232 Debug

The RS232 Debug configuration is a special configuration which does not used the standard SRAM located at the Spartan 3 Starter Board [12]. Instead, the memory runs on a PC connected to the X32 using the primary RS232 connection. This is a very useful setup to

Table C.11: X32 IN4073 Peripheral Devices

Type	Location macro	Size	Access
Unique ID	`PERIPHERAL_UID`	32	r
RS232 Channel	`PERIPHERAL_PRIMARY_DATA`	8	rw
RS232 Status Register	`PERIPHERAL_PRIMARY_STATUS`	2	r
Instruction Counter	`PERIPHERAL_INSTRCNTR`	32	r
MS Counter	`PERIPHERAL_MS_COUNTER`	32	r
US Counter	`PERIPHERAL_US_COUNTER`	32	r
4x7 Segment Display	`PERIPHERAL_DISPLAY`	16	rw
Switches	`PERIPHERAL_SWITCHES`	8	r
Leds	`PERIPHERAL_LEDS`	8	rw
Buttons	`PERIPHERAL_BUTTONS`	4	r
Processor State Register	`PERIPHERAL_PROCSTATE`	4	r
Interrupt Enable Register	`PERIPHERAL_INT_ENABLE`	32	rw
Software Interrupt	`PERIPHERAL_SOFTINT1`	n/a	w
Timer 1 Period	`PERIPHERAL_TIMER1_PERIOD`	32	rw
Timer 2 Period	`PERIPHERAL_TIMER2_PERIOD`	32	rw
PDC 1	`PERIPHERAL_PDC1`	32	r
PDC 2	`PERIPHERAL_PDC2`	32	r
PDC 3	`PERIPHERAL_PDC3`	32	r
PDC 4	`PERIPHERAL_PDC4`	32	r
PDC 5	`PERIPHERAL_PDC5`	32	r
DPC 1	`PERIPHERAL_PDC1_PERIOD`	32	r
DPC 1	`PERIPHERAL_PDC1_WIDTH`	32	r
DPC 1	`PERIPHERAL_PDC2_PERIOD`	32	r
DPC 1	`PERIPHERAL_PDC2_WIDTH`	32	r
DPC 1	`PERIPHERAL_PDC3_PERIOD`	32	r
DPC 1	`PERIPHERAL_PDC3_WIDTH`	32	r
DPC 1	`PERIPHERAL_PDC4_PERIOD`	32	r
DPC 1	`PERIPHERAL_PDC4_WIDTH`	32	r

Table C.12: X32 IN4073 Interrupts

Type	Index macro
Software Interrupt*	`INTERRUPT_SOFTINT1`
Timer 1 Interrupt	`INTERRUPT_TIMER1`
Timer 2 Interrupt	`INTERRUPT_TIMER2`
RS232 Rx	`INTERRUPT_PRIMARY_RX`
RS232 Tx	`INTERRUPT_PRIMARY_TX`
Buttons	`INTERRUPT_BUTTONS`
Switches	`INTERRUPT_SWITCHES`
TRAP Instruction*	`INTERRUPT_TRAP`
Overflow*	`INTERRUPT_OVERFLOW`
Division By Zero*	`INTERRUPT_DIVISION_BY_ZERO`
PDC 1	`INTERRUPT_PDC1`
PDC 2	`INTERRUPT_PDC2`
PDC 3	`INTERRUPT_PDC3`
PDC 4	`INTERRUPT_PDC4`
PDC 5	`INTERRUPT_PDC5`
Out Of Memory*	`INTERRUPT_OUT_OF_MEMORY`

Interrupts marked with an asterisk are critical, and are not blocked by the global interrupt

Table C.13: X32 IN4073-TREX Configuration

Name	X32 IN4073
Location	/x32-in4073
Peripheral bus ID	4703
Size (400K Spartan 3)	65%
Interrupts enabled	yes
Reset signal	All 4 buttons

Table C.14: X32 IN4073-TREX Peripheral Devices

Type	Location macro	Size	Access
Unique ID	`PERIPHERAL_UID`	32	r
RS232 Channel	`PERIPHERAL_PRIMARY_DATA`	8	rw
RS232 Status Register	`PERIPHERAL_PRIMARY_STATUS`	2	r
Instruction Counter	`PERIPHERAL_INSTRCNTR`	32	r
MS Counter	`PERIPHERAL_MS_COUNTER`	32	r
US Counter	`PERIPHERAL_US_COUNTER`	32	r
4x7 Segment Display	`PERIPHERAL_DISPLAY`	16	rw
Switches	`PERIPHERAL_SWITCHES`	8	r
Leds	`PERIPHERAL_LEDS`	8	rw
Buttons	`PERIPHERAL_BUTTONS`	4	r
Processor State Register	`PERIPHERAL_PROCSTATE`	4	r
Interrupt Enable Register	`PERIPHERAL_INT_ENABLE`	32	rw
Software Interrupt	`PERIPHERAL_SOFTINT1`	n/a	w
Timer 1 Period	`PERIPHERAL_TIMER1_PERIOD`	32	rw
Timer 2 Period	`PERIPHERAL_TIMER2_PERIOD`	32	rw
T-Rex Count	`PERIPHERAL_TREX_COUNT`	32	r
T-Rex Timestamp	`PERIPHERAL_TREX_TIMESTAMP`	32	r
T-Rex Sensor 0	`PERIPHERAL_TREX_S0`	32	r
T-Rex Sensor 1	`PERIPHERAL_TREX_S1`	32	r
T-Rex Sensor 2	`PERIPHERAL_TREX_S2`	32	r
T-Rex Sensor 3	`PERIPHERAL_TREX_S3`	32	r
T-Rex Sensor 4	`PERIPHERAL_TREX_S4`	32	r
T-Rex Sensor Actuator	`PERIPHERAL_TREX_A`	32	w

Table C.15: X32 IN4073-TREX Interrupts

Type	Index macro
Software Interrupt*	`INTERRUPT_SOFTINT1`
Timer 1 Interrupt	`INTERRUPT_TIMER1`
Timer 2 Interrupt	`INTERRUPT_TIMER2`
RS232 Rx	`INTERRUPT_PRIMARY_RX`
RS232 Tx	`INTERRUPT_PRIMARY_TX`
Buttons	`INTERRUPT_BUTTONS`
Switches	`INTERRUPT_SWITCHES`
TRAP Instruction*	`INTERRUPT_TRAP`
Overflow*	`INTERRUPT_OVERFLOW`
Division By Zero*	`INTERRUPT_DIVISION_BY_ZERO`
T-Rex Inbound communication	`INTERRUPT_TREX`
Out Of Memory*	`INTERRUPT_OUT_OF_MEMORY`

Interrupts marked with an asterisk are critical, and are not blocked by the global interrupt

Table C.16: X32 Example Configuration

Name	X32 Example
Location	/x32-example
Peripheral bus ID	5
Size (400K Spartan 3)	71%
Interrupts enabled	yes
Reset signal	All 4 buttons

Table C.17: X32 Example Peripheral Devices

Type	Location macro	Size	Access
Unique ID	PERIPHERAL_UID	32	r
RS232 Channel	PERIPHERAL_PRIMARY_DATA	8	rw
RS232 Status Register	PERIPHERAL_PRIMARY_STATUS	2	r
Instruction Counter	PERIPHERAL_INSTRCNTR	32	r
MS Counter	PERIPHERAL_MS_COUNTER	32	r
4x7 Segment Display	PERIPHERAL_DISPLAY	16	rw
Switches	PERIPHERAL_SWITCHES	8	r
Leds	PERIPHERAL_LEDS	8	rw
Buttons	PERIPHERAL_BUTTONS	4	r
Processor State Register	PERIPHERAL_PROCSTATE	4	r
Interrupt Enable Register	PERIPHERAL_INT_ENABLE	32	rw
Software Interrupt	PERIPHERAL_SOFTINT1	n/a	w
Timer 1 Period	PERIPHERAL_TIMER1_PERIOD	32	rw
Timer 2 Period	PERIPHERAL_TIMER2_PERIOD	32	rw
Watchdog Timer	PERIPHERAL_WATCHDOG_PERIOD	32	rw

Table C.18: X32 Example Interrupts

Type	Index macro
Software Interrupt*	INTERRUPT_SOFTINT1
Timer 1 Interrupt	INTERRUPT_TIMER1
Timer 2 Interrupt	INTERRUPT_TIMER2
RS232 Rx	INTERRUPT_PRIMARY_RX
RS232 Tx	INTERRUPT_PRIMARY_TX
Buttons	INTERRUPT_BUTTONS
Switches	INTERRUPT_SWITCHES
TRAP Instruction*	INTERRUPT_TRAP
Overflow*	INTERRUPT_OVERFLOW
Division By Zero*	INTERRUPT_DIVISION_BY_ZERO
Out Of Memory*	INTERRUPT_OUT_OF_MEMORY

Interrupts marked with an asterisk are critical, and are not blocked by the global interrupt

detect errors in the controller statemachine of the X32, since the X32 can be stepped through each memory action (at least one per instruction). For more information on using the X32 RS232 debugger, see the readme file in the configuration directory. Some configuration details are listed in Table C.19.

Table C.19: X32 RS232 Debug Configuration

Name	X32 RS232 Debug
Location	/x32-rs232debug
Peripheral bus ID	6
Size (400K Spartan 3)	55%
Interrupts enabled	no
Reset signal	all 4 buttons

C.9   Bytecode interpreter

The interpreter acts like an X32 with interrupt support. The supported peripherals are listed in Table C.21, and the supported interrupts in Table C.22. Note, that the buttons, switches, leds and display are not connected. The console peripheral acts just like an RS232 peripheral, except it is connected to `stdin` and `stdout` of the interpreter, rather than an RS232 port.

Some configuration details are listed in Table C.20, the supported peripherals are listed in Table C.21, and the interrupts in Table C.22. All critical interrupts are marked with an asterisk.

Table C.20: Bytecode Interpreter Configuration

Name	Bytecode Interpreter
Location	n/a
Peripheral bus ID	11
Size	n/a
Interrupts enabled	yes
Reset signal	n/a

When more, less or different peripherals need to be tested, they must manually be added to the `interpreter_peripherals.c` file in the `x32-tools/src` directory. See the comments in this file for more information on changing the peripherals of the interpreter.

Table C.21: Bytecode Interpreter Peripheral Devices

Type	Location macro	Size	Access
Unique ID	`PERIPHERAL_UID`	32	r
Console	`PERIPHERAL_PRIMARY_DATA`	8	rw
Console Register	`PERIPHERAL_PRIMARY_STATUS`	2	r
Instruction Counter	`PERIPHERAL_INSTRCNTR`	32	r
MS Counter	`PERIPHERAL_MS_COUNTER`	32	r
4x7 Segment Display	`PERIPHERAL_DISPLAY`	16	rw
Switches	`PERIPHERAL_SWITCHES`	8	r
Leds	`PERIPHERAL_LEDS`	8	rw
Buttons	`PERIPHERAL_BUTTONS`	4	r
Processor State Register	`PERIPHERAL_PROCSTATE`	4	r
Interrupt Enable Register	`PERIPHERAL_INT_ENABLE`	32	rw
Software Interrupt	`PERIPHERAL_SOFTINT1`	n/a	w
Timer 1 Period	`PERIPHERAL_TIMER1_PERIOD`	32	rw
Timer 2 Period	`PERIPHERAL_TIMER2_PERIOD`	32	rw

Table C.22: Bytecode Interpreter Interrupts

Type	Index macro
Software Interrupt*	`INTERRUPT_SOFTINT1`
Timer 1 Interrupt	`INTERRUPT_TIMER1`
Timer 2 Interrupt	`INTERRUPT_TIMER2`
Console Rx	`INTERRUPT_PRIMARY_RX`
Console Tx	`INTERRUPT_PRIMARY_TX`
Buttons	`INTERRUPT_BUTTONS`
Switches	`INTERRUPT_SWITCHES`
TRAP Instruction*	`INTERRUPT_TRAP`
Overflow*	`INTERRUPT_OVERFLOW`
Division By Zero*	`INTERRUPT_DIVISION_BY_ZERO`
Out Of Memory*	`INTERRUPT_OUT_OF_MEMORY`

Interrupts marked with an asterisk are critical, and are not blocked by the global interrupt

Appendix D

Frequently Asked Questions

- *The compiler always creates .dbg files, even if I don't use the -g command line parameter*
  The linker always creates .dbg files if any debug symbols are found, regardless of the -g parameter. If .dbg files are created without using the -g parameter this is probably caused by importing object or library files which are compiled with debugging symbols. Make sure the libraries are not compiled with debugging symbols.

- *The X32 is showing strange behavior when executing my software*
  Make sure all function prototypes are correct (and not missing), LCC generates only warnings for incompatible function declarations, but the X32 can't handle them.

- *The upload tool hangs when trying to upload software*
  Try resetting the X32. Also, make sure no other applications are using the serial device, and if required, restart the computer to free the serial device.

- *When I try to program the ROM of the Digilent Inc. Spartan 3 Starter Board, I get the message 'The ROM is write protected' (or similar)*
  This is probably caused by a cable disconnect during programming. Try clearing the ROM several times using the authentic Xilinx programming tool (Impact), this will reset the write protection flag.

- *The compiler is not able to find the putchar and/or getchar library functions.*
  These functions are X32 configuration specific, and are not part of the standard library. Instead they are located in the X32 library (`x32.cl`, `x32.h`, which ship with the X32 configuration used). Move or link these files into the `lib-X32` folder in `X32-tools`.

- *I installed a custom software program into the ROM of the X32, but it does not show any sign of life. When the software is loaded directly into the RAM using the loader it works fine*

  - Make your software is compiled to run from the same location as the X32 is configured to store its RAM. See 3.3.1 for details.
  - The software might be too big for the ROM. Try and see if a smaller program has the same problems

- *When compiling my X32 configuration I get a "timing constraints not met" message*
  This means the longest path in the generated design is longer than 20 nanoseconds. Since the Spartan 3 Starter Board runs at 50MHz, the X32 is running overclocked. There are three solutions to this problem:

  - Run the place and route utility in extra effort mode. See Section 2.2.3 on how this can be achieved.

  - Remove one or more unused peripheral devices from the design. The smaller the design, the easier it can be routed. Easy routings are more likely to fit the timing constraints.

  - Accept the overclocked X32. The timing estimates are worst case scenarios. When the X32 is little overclocked (20-22ns longest path), it will most likely work fine. The design must off-course always used with care, and different chips may produce different results. 26ns longest paths have still been reported to work correctly on several FPGAs.

- *LCC often gives the "Expression with no effect elided" Warning*
  This warning is often generated when converting constants, such as `(void*)0`. In most cases, this warning is harmless, but care should be taken when encountering these warnings.