Vrije Universiteit Brussel

Faculty of Engineering

Implementing Concurrency Abstractions for Programming Multi-Core Embedded Systems in Scheme

Graduation thesis submitted in partial fulfillment of the
requirements for the degree of Master of Engineering: Applied Computer Science

Ruben Vandamme

Promotor: Prof. Dr. Wolfgang De Meuter
Advisors: Dr. Coen De Roover
Christophe Scholliers

2010

Vrije Universiteit Brussel

Faculteit Ingenieurswetenschappen

Implementing Concurrency Abstractions for Programming Multi-Core Embedded Systems in Scheme

Eindwerk ingediend voor het behalen van de
graad van Master in de Ingenieurswetenschappen: Toegepaste Computerwetenschappen

Ruben Vandamme

Promotor: Prof. Dr. Wolfgang De Meuter
Begeleiders: Dr. Coen De Roover
Christophe Scholliers

2010

Acknowledgements

This thesis would not have been possible without the support of various people.

First, I would like to thank Professor Wolfgang De Meuter for promoting this thesis. In particular I would like to thank my advisors Christophe and Coen for their extensive and essential support throughout the year. Without their effort, this thesis would not have been what it is today.

I thank my parents for making all this possible and for supporting me during my education at the Vrije Universiteit Brussel and during previous educations. And I cannot forget to thank Jennifer for her indispensable support during this undertaking.

Abstract

This dissertation presents a study of the limitations and problems related to the prevalent way embedded systems handle signals from the outside world. Such signals are frequently handled using either polling or interrupts. Polling software will continually check whether a signal needs handling. In interrupt-driven embedded systems, on the other hand, the CPU will generate an asynchronous signal when an event from the outside arrives. This signal will allow the software to react to this event. We show that both approaches have their disadvantages. The interrupt-driven approach can moreover introduce bugs that are subtle and difficult to fix in embedded software.

We study a new event-driven architecture and programming style developed by the XMOS company. The architecture's hardware support for multi-threading enables an event-driven style for programming embedded systems which does not suffer from the drawbacks associated with the use of polling and interrupts. To accomplish this, the thread support is implemented in hardware. Each thread has a dedicated set of registers and is assigned a guaranteed amount of CPU cycles.

Next we describe how we ported a Scheme interpreter to this new architecture. We exploit the multi-threaded nature of this architecture by running multiple interpreters in parallel, concretely one interpreter on each core. In addition, we extend each interpreter with abstractions to manage this concurrency and to exploit features specific of the XMOS hardware. Such abstractions include sending messages between interpreters over channels. Concretely, our effort enables an event-driven style for programming multi-core embedded systems in Scheme. We will illustrate the superiority of this approach over polling and interrupt-driven approaches through a realistic case study.

Contents

List of Figures

List of Tables

Listings

Glossary

8N1 RS232 data format (8 databits, No parity bit, 1 stopbit).

Baudrate communication speed of a serial port.

CSP Communicating Sequential Processes.

MIPS Million instructions per second.

PWM Pulse Width Modulation.

RS-232 serial port communication standard.

RX Receiver.

TX Transmitter.

XBee Zigbee compatible device.

XCore A computing unit inside an XMOS chip.

ZigBee low-power wireless communication standard.

Chapter 1

Introduction

Embedded software is increasingly becoming important. Digital watches, microwaves, cars, et cetera, all contain embedded systems. More than 98 % of the processors used today [25], are used in embedded systems. Software running on a PC or server differs significantly from embedded software. Typically, such a system consists of hardware and software that are tailored to one specific task. Next to application logic, embedded software also has to cater to interactions with the physical world. Such interactions include reading sensors, turning a motor or light on, communicating with other systems, et cetera. Certain protocols and peripheral hardware have strict timing constraints, where the system needs to respond within a fixed amount of time. When an embedded system has to handle various of those timing-sensitive interactions concurrently, programming and debugging gets even more complex.

Most microchips and accompanying embedded software is interrupt-driven. An interrupt is an asynchronous signals that indicates to the CPU that its attention is needed. In that case, the CPU will stop whatever it was doing and execute the appropriate interrupt handler. Afterwards the CPU continues executing the task it was executing before the interrupt. While this approach is widely used to ensure quick responses on various interactions, it has several drawbacks [23][22]. We will discuss these drawbacks before introducing event-driven architectures which comprise a completely different approach to embedded systems. We will illustrate each approach with a representative case study.

1.1 Interrupt-driven embedded systems

On chips, interrupts are often used to handle interactions from the outside. On interrupt-driven systems, these interrupts will stop the application code and save the current execution state onto the stack. After having executed the appropriate interrupt handler, the previous execution state is restored. Systems that use polling continuously check whether a condition is true. Compared to systems that use polling, interrupt-driven systems can achieve a reduction in latency and overhead. If this check isn't performed often enough, latency can become an issue. However, frequent polling might introduce an overhead. Reduced power consumption is another advantage of interrupt-driven systems. The CPU can sleep until it is interrupted. In less powerful chips, dedicated hardware is often used to do certain time and resource consuming IO tasks like serial communication. Interrupts are used to synchronize the dedicated hardware with the chip that runs the application code. The dedicated hardware can, for instance, signal that it received data from the serial port.

As outside signals can occur at any time, interrupts can be fired at any time too. This can introduce various problems which are hard to find because they occur only rarely under specific conditions. Among others, a stack overflow can occur when many interrupts arrive at the same time causing excessive stack usage by the various interrupt handlers [23][22]. In the case of an excessive number of interrupts, the main application can also be starved from CPU time [23]. Also each interrupt causes the application to halt, which complicates meeting real-time constraints. Existing approaches to prevent these problems usually try to empirically determine the system resources that are needed to handle all possible combinations of interrupts [23]. These approaches however, are not perfect and can complicate the development and debugging. In this dissertation, we will illustrate the above problems by means of a case study, showing the above-mentioned problems. Concretely we will create a case study illustrating the problems associated with polling and interrupt based software. In this case study we will implement an application performing pulse width modulation (PWM) on a LED. Two of these devices are connected via a wireless XBee module, synchronizing their PWM.

1.2 Event-driven embedded systems

Due to increasingly powerful chips, it is becoming possible to use software for tasks that used to be implemented in hardware, such as serial communication. This approach is advocated by the XMOS company[1]. Their chips comprise an event-driven architecture. A thread, implemented directly in the hardware, will subscribe to an event and perform the corresponding computations when that event occurs. Events can be triggered by changes in timers, communication or input and output operations. Threads have no shared memory, but communicate through message passing. When a thread wants to handle an event, it subscribes to the event and suspends itself until the event happens. By suspending itself, the thread allows other threads to run. Power consumption can also be reduced when fewer threads are running. In the XMOS architecture, threads each have their dedicated set of registers and each get a guaranteed amount of CPU cycles [16]. As each event is handled in an independent thread, timing constraints will always be met.

In traditional desktop software, spreading a program across multiple threads requires splitting up the application in parts that can run concurrently. However, most embedded software is already inherently concurrent as it has to handle application logic as well as various interactions with the outside world. Therefore, it is quite natural to map embedded software onto multiple threads, as enabled by the XMOS architecture.

On a single core chip, it is possible to mimic the parallel execution of multiple threads. For instance occam-π allows to write multi-threaded programs in a similar way as on the XMOS chips. Because occam-π runs on interrupt-driven platforms, this approach cannot give the same guarantees concerning timing and executing speed as the XMOS architecture. The scheduled threads can still be interrupted by outside events. This makes it impossible to determine with certainty when a calculation will be finished. As threads run interleaved, it is difficult to meet real-time constraints.

Another approach is the use of an operating system such as TinyOS [11]. This operating system allows the scheduling of tasks. In this dissertation, we will revisit the case study to illustrate the problems solved by the XMOS architecture and compare the event-based version with the interrupt-based version.

[1] http://www.xmos.com

1.3 High-level event-driven programming in Scheme

Currently the XMOS chips can only be programmed in low-level programming language derived from C. In this dissertation, we will therefore investigate whether the XMOS architecture can be programmed in the high-level programming language Scheme and whether using Scheme on this architecture simplifies the developer's task even more.

We ported the bytecode interpreter Bit Scheme to the XMOS platform. As this interpreter is very small, it fits in the available memory, while leaving enough space for the bytecode of the application and the application's runtime memory requirements. Bit Scheme comes with a compiler that translates the Scheme source code into bytecode. The bytecode interpreter features a real-time garbage collector, an important benefit in the embedded domain. In this context, real-time means that the garbage collector is guaranteed to complete its task within a fixed amount of time [5]. This is especially useful when timing constraints need to be met. In addition to porting the Bit Scheme interpreter to the XMOS platform, we also extended this interpreter to XMOS specific hardware features. To support the multi-core embedded system, we run four Scheme interpreters in parallel. We added new primitives to the Scheme interpreter in order to allow the interpreters to communicate via message passing. The Bit Scheme interpreter is also extended with XMOS specific IO abstractions. We will illustrate the advantages of this high-level approach in a case study.

Chapter 2

State of embedded software engineering

Embedded software is different from traditional software that runs on a PC or a server, in that it has to interact with the outside world. These interactions can be reading sensors, handling communication with other systems, handling user input, doing periodic tasks, et cetera. Currently almost all embedded systems either actively poll the outside world for changes or are notified of changes through interrupts.

Although prevalent, both approaches have significant disadvantages. When polling for events, the software constantly checks for changes in the outside world that need to be reacted to. If that check is not performed often, it will increase the latency between the event occurring and it being reacted to. This latency can be reduced by checking more frequently. However, this also means that the software will often check in vain whether an event has occurred. A computational overhead is therefore incurred.

In order to prevent having to poll constantly, so-called interrupts comprise a frequently used alternative to polling. The hardware interrupts the normal execution of the software to signal the occurrence of an event. This starts an interrupt handler which will handle the event accordingly. Interrupts allow reducing any latency in detecting the occurrence of an event without the computational overhead associated with polling for this event more frequently.

2.1 Using interrupts in embedded software

As mentioned before, being notified of outside events through interrupts reduces latency and overhead compared to software that uses polling. Apart from that, using interrupts can also significantly reduce power consumption. Especially battery powered devices can take advantage of this, for example sensor network nodes [23]. These would drain their batteries in a few days if the processor was constantly polling for changes in sensor readings. However by idling until a timer has fired, the lifespan of the batteries can be extended to several months. This will allow the processor to be in a power saving mode for an extended period of time. Only the timer that will signal the interrupt has to be powered.

In less powerful chips, dedicated hardware is often used to do certain time and resource consuming IO tasks, such as serial communication. This takes the task and thus the computing load away from the processor to a specialized piece of hardware. That way, the main application can continue executing. This concept introduces a limited amount of parallelism as the IO tasks runs in parallel to the main application. Interrupts are used by the dedicated hardware to, for example notify the main application that data was received from the serial port. The Atmel ATMEGA 168, for example, has three interrupts related to UART communication [3]. These interrupts signal that the receiver or the transmitter is ready or that the register to send data is empty.

2.2 Problems associated with interrupts

Interrupts are widely used in a variety of platforms and have multiple advantages over polling-based implementations. However, interrupts have some drawbacks of their own.

Processor-dependent problems

First of all, interrupts are more or less tied to a certain platform [23]. While the concept of interrupts is widely used, almost each platform or CPU features a different implementation. Porting software to a different interrupt-based architecture is therefore not trivial. Among others, the way an interrupt is entered and exited differs per platform. Certain chips save their entire

execution context before entering an interrupt (being the program counter and all registers). Others only save the program counter. In the latter case, the programmer needs to manually save the registers on the stack. Some compilers also relieve the programmer from this task, having the programmer indicate that a function is an interrupt handler through pragmas. In that case the compiler will add the necessary code to save the environment on the stack before executing the actual interrupt handler.

Secondly, most instructions cannot be interrupted. This means that the processor can only enter an interrupt "between" two instructions of the main application code. On reduced instruction set chips (RISC) this is usually no problem, because instructions are short. However, in complex instruction set chips (CISC), some instructions take a long time to execute. This can increase the interrupt latencies, which can be problematic for certain real-time applications. To alleviate this problem, certain embedded compilers try to keep these instructions out of the binaries to prevent this problem.

Stack overflow

A program's call stack grows and shrinks during program execution. The stack should never grow too large, because in that case adjacent memory may get corrupted causing unwanted behaviour and/or crashes. Therefore these stack overflows should be prevented at all cost.

However, in embedded systems memory comes at a premium. This is because more memory will obviously increase the economic cost of the device. From an economic standpoint the available memory should therefore be used as well as possible. However, to prevent a stack overflow from occurring, there needs to be enough memory to let the stack grow to handle every possible situation. Clearly, in the ideal case the memory should be just large enough to handle the biggest stack size, but no more.

One approach used to determine the needed stack size is based on empirical data [23]. This data is collected during simulated or actual tests of the system. On a simulator, the maximum stack size can be recorded directly. Determining the maximum stack size on a physical system can be accomplished by initializing the entire memory to a known value and after a program run checking how big the stack became. However, it is almost certain that during testing some code paths will be missed, resulting in an observed memory requirement that is smaller than the actual need of the system.

Another approach to determining the needed stack size is through analysis [23] [24]. During this analysis, instructions that affect the stack size (like push and pop operations, function calls, et cetera) are combined with the program flow. That way the maximum stack size can be determined. It is clear that this is a much more reliable approach than the testing-based one. Analysis takes much effort too, unless good tools are available to automate this. However, it is perfectly possible that after the analysis, the conclusion is that the memory needed to be safe is infinite. This can for example be the case with reentrant interrupts when interrupts are flowing in at a higher pace than the processor can execute the interrupt handlers. Reentrant interrupts handlers can be executed even when its previous call has not yet finished. This effectively means that the same interrupt handler can be executing multiple times at the same time.

$$\begin{aligned} &\textit{worst depth seen in testing} \leq \\ &\textit{true worst depth} \leq \\ &\textit{analytic worst depth} \end{aligned}$$

Figure 2.1: Stack depth [23]

The actual worst depth stack size will always be between the one measured during testing and the one computed through analysis, as depicted in Figure 2.1. The lower boundary (being testing) can be increased by doing more extensive testing. The upper boundary, on the other hand, can be lowered by checking for relationships between interrupts that for example cannot physically appear together. This is not without danger. Such relationships between interrupts are usually based on assumptions derived from the specifications of the system. These specifications can state that under normal operation two specified interrupts cannot happen together. However, it is possible that the system may get in a state outside its specifications. When certain assumptions are made about the occurrence of interrupts, unexpected situations can result in a crash. When combined with the fact that embedded systems may perform (life)critical tasks, it is desirable that the software can cope with these unusual situations.

Combining an analysis-based method and a testing-based to determine the needed stack size method should result in an embedded system that is "stack safe". This means that it is impossible for a stack to become too large and to overflow into memory used for other purposes. Clearly the ideal system

should just contain enough memory to be stack safe, although in practice an extra margin is used.

Interrupt overload

Interrupt overload happens when an embedded system has to handle so many interrupts, that the main application is starved from CPU cycles. This flow of interrupts is generally caused by an external device generating an unexpectedly high number of interrupts. This can for example be due to a malfunction of this device. Another example is a robot speeding downhill. In that case its sensors will generate more interrupts than when it would ride on a flat surface at full speed. Clearly in this case, the specification of maximum achievable speed of the robot is not a reliable measurement for the real maximum speed.

High interrupt loads do not necessarily mean that an interrupt overload occurs, as the system should be designed to handle that specific load. It is only in the case of unexpectedly high interrupt loads that this problem might occur. The moment when the interrupt overload starts depends on the number of interrupts, the CPU speed and on the length of the interrupt handlers. Due to these different factors the maximum amount of interrupts can vary greatly between different systems and situations.

Because embedded systems interface with the physical world, it is often difficult to determine what the maximum number of interrupts is that a system will receive. It is clear that this number can be higher than what is mentioned in the system's specifications. Simple examples are button presses, where so-called "button bounce" may cause a frequency of interrupts of over 1 kHz when the button makes contact [23]. Also malfunctions of the peripherals on the system board can cause an unexpectedly high number of interrupts. A simple loose wire can already create a 500 Hz signal [23].

The maximum amount of time spent in an interrupt handler is quite easy to compute with the formula in Figure 2.2.

It is clear that limiting the time spent in interrupts by keeping interrupt handlers small is a good idea. Also bounding the arrival rate is clear to reduce the total time spent in interrupt. However, as mentioned before, assumptions about the maximum interrupt arrival rate need to be carefully considered. Another method to reduce the maximum rate is by using smarter peripherals, for example a serial port which does not generate an interrupt

$$\begin{aligned} &\textit{maximum time spent in interrupt handler} \\ &*\textit{maximum interrupt arrival rate} \\ &= \textit{total time spent in an interrupt} \end{aligned}$$

Figure 2.2: Time spent in an interrupt [23]

for every byte it receives. It is also possible to switch to polling when a high rate of interrupts is detected. The overhead associated with polling is caused by checking in vain for events. Certain network chips will switch from interrupts to polling when many packets arrive for an extended period of time [23].

Another method which can be applied is called Restricted Interrupt Discipline (RID) [22]. RID is application code that uses the enable bits of the hardware to enable and disable interrupts as they are needed, which is fairly straightforward. This reduces the changes of unexpected interrupts being fired. It consists of two steps. First the developer needs to initialize the hardware properly in order to disable all requested interrupts. Requested interrupts are interrupts which are caused implicitly by the programmer. This is, for example, an interrupt signalling that serial data was successfully sent. Therefore, the interrupt in question should only be enabled when the application sends serial data. Spontaneous (non-requested interrupts) can be enabled as soon as the application is ready to handle them.

Testing

Finally, interrupts can be the source of serious software errors. However, errors like the aforementioned stack overflow might only appear under very specific conditions. This is because interrupt based software usually contains a very large number of executable paths [22]. This means that certain bugs can be very rare and therefore hard to detect during testing. Interrupts add fine-grained concurrency to embedded applications [22]. This can introduce various race conditions, which are difficult to find. These problems may sound familiar to people developing multi-threaded programs. Also because the number of executable paths increases significantly when introducing interrupts in software, it becomes difficult to reason about the software and its execution. Many embedded systems serve a safety critical role or have to run

without human intervention for an extended period of time, consequentially bugs should be detected during testing.

2.3 Case study: pulse width modulation with wireless XBee control

The following case study illustrates the problems associated with interrupt-based software. It implements a pulse width modulation on a board shown on Figure 2.7.

Pulse width modulation is a technique to create an analog voltage between 0 volt and V_{cc} volt. This is achieved by quickly enabling and disabling a digital output (which can only output 0 or V_{cc} volt). By varying the duty cycle of the output, one can emulate an analog voltage in the supported range. PWM has various applications. One of them is dimming LEDs, instead of using a variable resistor to vary the current through the LED and thus the light intensity. When using PWM, the LEDs are quickly turned on and off, giving the human eye the impression that the LED is dimmed. This task is highly periodic as frequencies of 100Hz and more are recommended to drive LEDs.

PWM can be implemented in software too, however by using dedicated hardware, the application can continue without having to be interrupted 100 times per second to toggle the output of the pin.

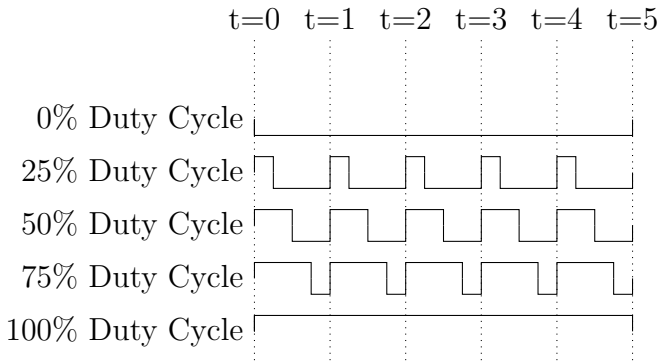


Figure 2.3: Pulse Width Modulation (PWM)

2.3.1 Hardware setup

Two of the boards shown in Figure 2.4 are connected using a wireless module called XBee to synchronize their PWM values. Data is sent back and forth between the XBee module and the micro-controller using serial communication over RS-232 UART. The RS-232 protocol uses three wires: one to send data (tx), one to receive data (rx) and a common ground (gnd). Every bit is sent one by one over a wire, hence the name *serial* communication. The detailed principles are explained in Section 4.6.2.

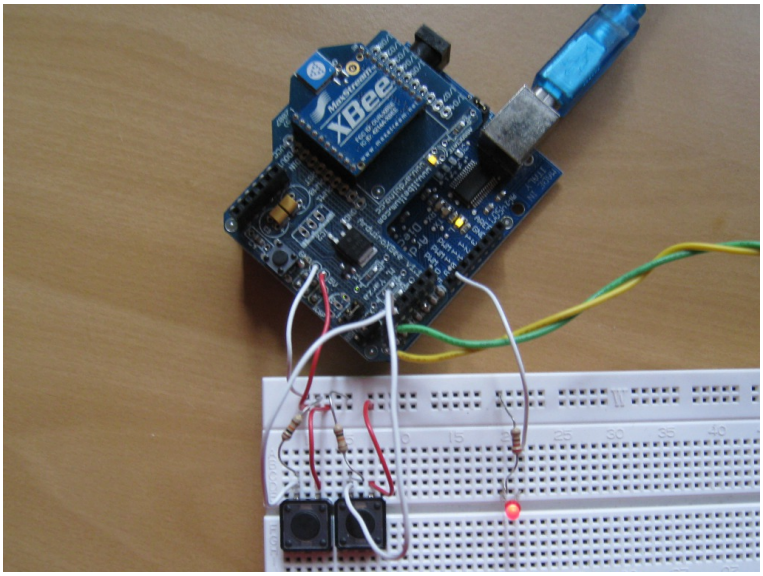
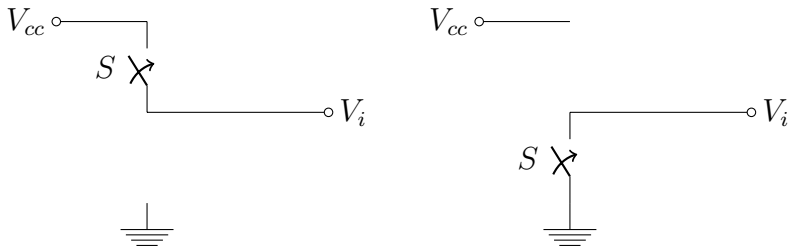


Figure 2.4: Hardware setup

To modify the PWM value, two buttons are used, as shown on Figure 2.4. When connecting a button to a chip one cannot simply connect it as displayed in Figure 2.5. In this case, when the switch S is open, the pin of the chip will be floating, meaning that this pin is not connect to either ground or V_{cc}. This means that it will get an undefined logic level, giving erroneous input to the chip.

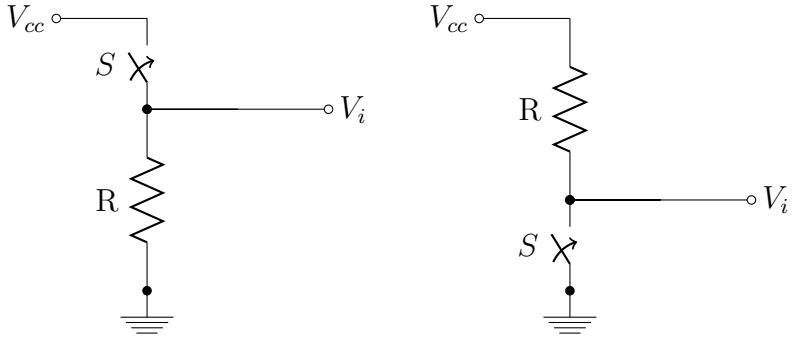
To prevent this problem, the pin needs to be connected to either ground or V_{cc} at all time. Therefore an extra resistor is added as illustrated in Figure 2.6. In case (a), a pull down resistor will pull the voltage V_i to *Ground* when switch S is open. While the second case (b), a pull up resistor, V_i will pull the output voltage to the V_{cc} level when the switch is open. The resistor used needs to have a big resistance to ensure that when closing the



(a) Connected to V_{cc} (b) Connected to Gnd

Figure 2.5: Incorrectly connected buttons

switch S, the voltage remains at the desired logical level. Typically for a V_{cc} of 5V a resistor of over 1000 Ω is used. When the button is closed, the resistor will work as a voltage divider.



(a) Pull down (b) Pull up

Figure 2.6: Pull down and pull up circuits

The needed peripheral electronics for this case study is shown in Figure 2.7. It consists of two buttons with pull down resistors connected to pins 2 and 3. Pin 9 is connected to the LED. The current through the LED is limited by a 220 Ω resistor.

2.3.2 Software

First the hardware needs to be initialized. This is implemented in the `setup` function shown in Listing 2.1. The serial output is initialized at a speed (or baudrate) of 9600 bps. Next the pin connected to the LED is defined as output and gets its default value. The function `analogWrite` will write the PWM value. This function interprets 0 as always off and 255 as always

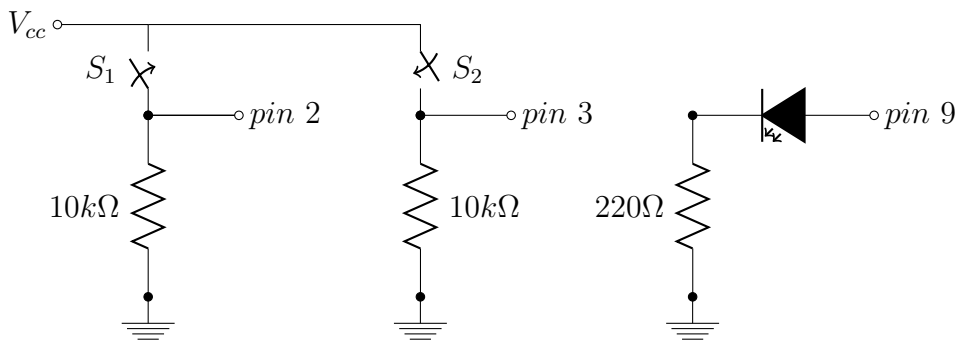


Figure 2.7: Case study hardware setup

on. It configures the hardware accordingly and then returns. The hardware PWM module will then perform its task independently. Finally, the function `increase` and `decrease` are set up as interrupt handlers for the buttons. These external interrupts referenced with numbers 0 and 1 even if they are connected to pins 2 and 3 (pins 0 and 1 are used for the XBee UART communication). When a button is pressed, the appropriate interrupt handler is called. The interrupt handlers will only be called when a button is pressed, due to the extra `RISING` parameter. Due to this parameter an interrupt will only be caused by a rising edge. A rising edge is the moment when the voltage on a wire changes from 0 volt to V_{cc}.

Listing 2.1: Setup

```
1  int led = 9;
2  int pwm = 7;
3
4  void setup()  {
5    Serial.begin(9600);
6
7    pinMode(led, OUTPUT);
8    analogWrite(led, pwm*16);
9
10   attachInterrupt(0, increase, RISING);
11   attachInterrupt(1, decrease, RISING);
12 }
```

When the setup is finished, the main loop shown in Listing 2.2 is entered, which handles the RS-232 input by polling for available data. If there is data available, it is used to update the PWM value. This update is performed by

the `analogWrite` function which will update the PWM pulse set on the pin. The duty cycle is set by varying the second argument from 0 to 255. This function will emulate setting an analog voltage on the pin, which is generated by pulse width modulation by the chip's hardware.

Listing 2.2: Handling the RS-232 input

```
1  int led = 9;
2  int pwm = 7;
3
4  void loop()
5  {
6    if ( Serial.available() > 0)
7    {
8      pwm = (Serial.read()%16);
9      analogWrite(led, pwm*16);
10   }
11 }
```

The buttons are handled by using two interrupts shown in Listings 2.3 and 2.4. When a button is pressed, the appropriate handler is called. In that handler, we will sample the current time to perform a so-called *debounce*-operation. This is needed because when a button is pressed, it will not immediately have a firm contact, but bounce for a brief moment between open and closed. This will effectively generate multiple key presses, therefore it's our task to filter these unwanted interrupts out. If the button hasn't been pushed for half a second, this means that we can interpret the button push as legitimate. In that case the new PWM value will be calculated and sent via serial and the XBee module to the other board.

Listing 2.3: Increase interrupt handler

```
1  int led = 9;
2  int pwm = 7;
3
4  void increase()
5  {
6    static unsigned long last = 0;
7    unsigned long current = millis();
8
9    if (current - last < 500) return;
10
11   last = current;
12
13   if (pwm < 15)
14   {
15     pwm++;
16     analogWrite(led, pwm*16);
17     Serial.write(pwm);
18   }
19 }
```

The `decrease` interrupt handler shown in Listing 2.4 is very similar to the `increase` one, containing debounce code followed by code modifying of the LED PWM.

Listing 2.4: Decrease interrupt handler

```
1  int led = 9;
2  int pwm = 7;
3
4  void decrease()
5  {
6    static unsigned long last = 0;
7    unsigned long current = millis();
8
9    if (current - last < 500) return;
10
11   last = current;
12
13   if (pwm > 0)
14   {
15     pwm--;
16     analogWrite(led, pwm*16);
17     Serial.write(pwm);
18   }
19 }
```

The main function shown in Listing 2.5 is called by the boot loader when the Atmel chip is powered on. It will first initialize and set up the hardware and next enter the main loop. The include `WProgram.h` contains the functions used to modify the hardware. They contain the low level implementation.

When the main loop is running it polls continuously for incoming data from the UART. However, at any time this code can be interrupted by an interrupt generated by a button press. The accompanying interrupt handlers can modify the *PWM* value. This variable is effectively shared memory between the main loop and the handlers, therefore at any time this value can be changed, introducing unwanted changes of this variable in the main loop. It is possible to stop this problem by protecting it using extra code which turns the interrupts question off during the critical sections. Clearly more interrupts will add much more possible code paths. Consequentially, in complex embedded software, problems as the one shown above can be much harder to detect.

Listing 2.5: Main function

```
1  #include "WProgram.h"
2  int led = 9;
3  int pwm = 7;
4
5  void setup();
6  void loop();
7  void increase();
8  void decrease();
9
10 int main(void)
11 {
12   init();
13
14   setup();
15
16   for (;;)
17     loop();
18
19   return 0;
20 }
```

2.4 Conclusion

Interrupts are widely used in todays embedded software. They offer various advantages over polling, like improved power efficiency and more efficient usage of resources. However, they also are the source of various hard to fix problems. These problems often occur only under very special circumstances. Therefore they are particularly hard to find and solve during testing. In a case study we gave an example on how interrupts can introduce bugs in very subtle ways.

Chapter 3

Event-driven embedded software

Due to chips becoming increasingly powerful, it has become feasible to use software for tasks that used to be implemented in hardware. This is an approach advocated by the XMOS company. The event-driven architecture eliminates the need to use interrupts to handle events from the outside world. A thread runs its application code until it has to wait for one or more events. The thread will suspend itself until the event occurs. Once the event has occurred, the thread continues executing.

The multi-core XMOS chip supports 32 threads in hardware which each get a guaranteed minimum amount of CPU cycles. Because these cycles are guaranteed for each thread, it allows writing embedded software with more predictable timing behaviour. The execution of a function will never be delayed due to factors like interrupts. Because the hardware supports multiple threads and thanks to the event-driven architecture, threads can handle their own IO instead of having to rely on interrupts for this. Because all of this is supported by the specifically designed hardware, the chip offers high performance, but can also conserve power if programmed properly.

However, the concept of a chip specifically designed for parallel computing is not new. In the 1980s, the company Inmos produced a chip called the Transputer [1]. The architecture and the accompanying programming language Occam are based upon CSP, which is also the case for the XMOS chips.

3.1 Threads and events

Applications written for XMOS chips are almost always split over multiple threads. The computing power of these chips can only be exploited fully when using threads.

These threads are directly supported by the XMOS chips. However, only a limited number of threads is supported. The amount of threads that can run on the chip used in this dissertation, the XS1-G4, is limited to 32. By limiting the number of threads, it is possible to have a dedicated set of registers for each of them. This allows to quickly switch between threads as the states of the thread being paused and the one being loaded don't have to be stored in and fetched from RAM memory respectively. Each thread is assigned a guaranteed amount of CPU cycles. This increases the predictability of the program's execution time. When a core is executing n threads, each thread can execute its next instruction at most n clock cycles in the future [15]. Timing constraints will therefore always be met. Events that arrive while handling another event, will be handled by a separate thread.

The XMOS programming model does not allow threads to share memory. Two threads can communicate by means of exchanging messages. Concretely, these messages are passed over channels. A channel has exactly two channel ends, connecting exactly two threads. These two threads can bidirectionally communicate over the channel. Communication over this channel is blocking. Therefore, the two communicating functions must reside in separate threads, which run in parallel. If not, the program will stall.

Threads on different cores and different chips have different physical memory. If two threads run on the same core, communication can, in theory, happen through shared memory rather than message passing. Although not recommended, it is therefore possible to disable the compiler's disjointness checks of variables. These disjointness checks are carried out by the compiler to ensure, that no variables are shared between threads. Disabling these enables multi-threaded programming using the shared memory paradigm. A third, even more low-level, way of communication is via registers.

Channels don't have a specified direction in which the communication should happen. This means that two threads can communicate back and forth over the same channel. But communication is blocking, consequentially threads need to be sending and receiving data in the correct order. If the two threads communicating over the same channel try to send or receive at the same time, a deadlock will occur.

Aside from multi-threading, the XMOS chip also offers hardware support for event-driven architectures of embedded systems. Event-driven entails that a thread subscribes to an event and performs the corresponding computations when that event occurs. Events can be time-related (e.g. the completion of a millisecond count), communication-related (e.g the reception of data from another thread) or input/output related (e.g. the pushing on a button). When a thread waits for an event to occur, it is suspended. By suspending itself, it allows other threads to be executed or to reduce the power consumption of the chip in case there are no other threads to run.

3.2 Event-driven XMOS hardware

The event-driven and multi-threaded XMOS programming model requires specifically designed hardware. The XMOS XS1-G4 chip used in this dissertation combines 4 processing cores into one piece of silicium, as depicted on Figure 3.1. These cores are also called XCores. Each of these XCores runs at 400 MHz and has its own dedicated memory and IO. A maximum of 8 threads can run in parallel on each core. As illustrated by Figure 3.1, the XCores are connected using an interconnect or switch which allows threads on different cores to communicate.

Apart from the 4-core chip used in this dissertation, there are other versions available too containing 1 or 2 cores. Both these chips also exists in a faster 500 MHz version, which gives a 25 % speed increase over the standard versions which run at 400 MHz. Trading in parallelism for a faster sequential execution of the individual threads.

3.2.1 The XCore architecture

Processor

Each chip designed and manufactured by XMOS contains one or more XCores. As mentioned before, each XCore can run a maximum of 8 threads. These threads are supported in hardware and are executed interleaved. Due to the round robin scheduler, threads appear to execute in parallel. Figure 3.2 illustrates that an XCore has a dedicated set of registers for each of these 8 threads. Each XCore is equipped with 64 kilobytes of RAM memory, which

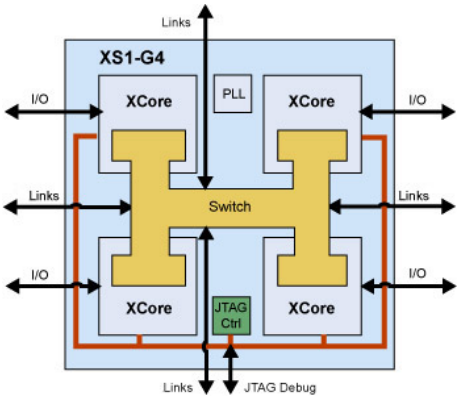


Figure 3.1: XS-G4 chip schematic

is shared among all threads running on that core. This memory is not only used during run-time, but also contains the code for the application itself.

Input and output

Each XCore is connected with up to 64 pins to perform IO. These can be configured either for input or for output purposes. For IO the notion of ports is used. A single port can represent from one up to 32 pins.

Communication

Each XCore features support for XLink channel ends. These allow threads to communicate, even if they reside on a different XCore or even on a different chip.

3.2.2 Thread execution speed

As each XCore on the XS1-G4 chip, runs at a clockspeed of 400 MHz by default, a maximum of 400 million instructions per second (MIPS) can by executed. For the XS1-G4, which contains four XCores, this totals to a maximum of 1600 MIPS for the entire chip. The CPU is RISC based (Reduced Instruction Set Computer), therefore most instructions execute in a single clock tick.

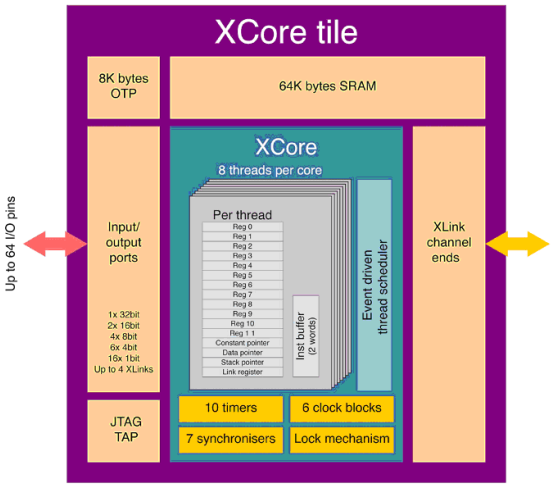


Figure 3.2: XCore architecture

Each thread gets an equal guaranteed minimum amount of processor cycles. However, as illustrated by Figure 3.3, the maximum performance for a thread will only be attained when running four or less threads on a single core. In that case, each thread will be executed at 100 MIPS. When an application needs more threads than those four, the guaranteed minimum CPU time that will be granted to each thread will decrease accordingly. When the maximum of 8 threads per XCore is running, each of them will get 50 MIPS.

The numbers mentioned above are the strict minimum, the exact amount of CPU time each thread gets varies depending on the actual application and thread size. When one thread is suspended (i.e. is waiting for an event to happen), extra CPU cycles become available for the other threads, enjoying an increase in their execution speed.

3.2.3 The memory model

Each core is equipped with its own individual memory. The amount of memory available on a core is limited: only 64 kilobytes of memory. In a four-core chip this results in a total of 256 kilobytes of memory. This memory has to host the entire application code, but also the stack and the heap at runtime. As this memory cannot be shared between threads on different cores, the memory requirements of a single core application cannot exceed 64 kilobytes. To use all 256 kilobytes of memory, the application needs to consist

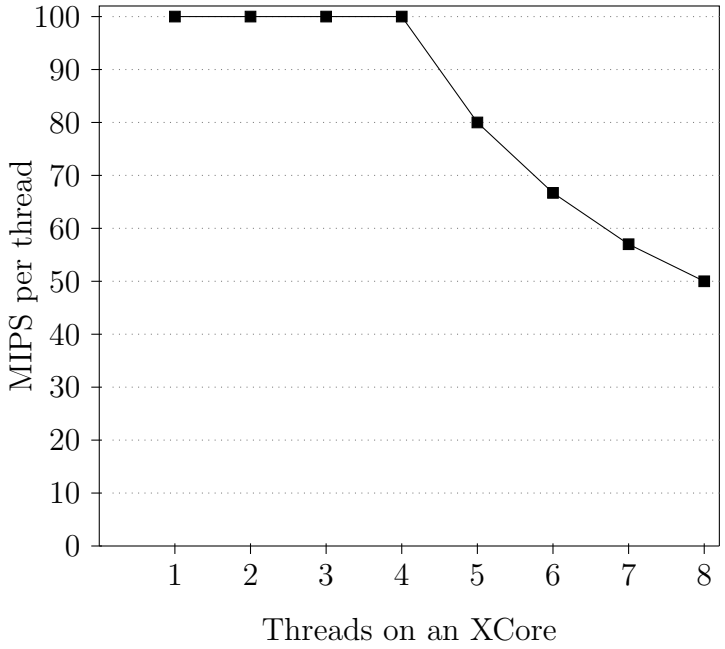


Figure 3.3: Guaranteed minimum MIPS per thread

of at least four threads and each thread has to run on a different core. These threads cannot use shared memory to communicate, all communication happens via message passing. To assist the programmer with fitting a program in this memory, the mapper of the XMOS toolchain can create a report with the memory requirements on each core.

Apart from the RAM there is a small piece of ROM on the chip. This ROM is 8 kilobytes large and contains the startup code for the chip. It can also be used for security purposes as the code inside this ROM can no longer be changed after it is programmed.

3.2.4 Communicating between threads

The XS1-G4 chip contains four of the above-mentioned XCores. As each core has its own memory, shared memory cannot be used for communication between threads on different cores. These cores are connected using an interconnect which allow threads to communicate using message passing. It is also possible to connect multiple chips using the "XLinks".

To communicate with other XMOS chips, each chip is equipped with four of these links. They allow to connect multiple chips in a chain or in a hypercube. A chain can be made by connecting multiple XK-1 development kits [13] while the hypercube is used inside the high performance XK-XMP-64 development kit [14].

As the distance between two communicating threads increases, the communication delay increases accordingly. Communicating between threads on a single XCore takes only a single CPU cycle. This results in a speed of 1Gbps. This increases to 3 clock cycles when communicating between threads residing on different XCores, but still on the same chip. Communicating between two chips takes at least 20 cycles. Just like the execution speed of the threads, the communication delay is fully deterministic.

3.3 Conclusion

To enable event-driven programming of embedded software, the XMOS company has designed a multi-core and multi-threaded chip. Because there is the possibility of running up to 32 threads in parallel, the concept of interrupts is no longer needed. This clearly also keeps out the problems associated with them. Threads are supported directly in hardware and because every thread gets a minimum number of CPU cycles, this architecture can reliably meet timing constraints, as there are no interrupts that can unexpectedly stall the application.

In order to program this new concurrent architecture, a different approach to embedded programming is needed. Applications need to be split up in threads, however embedded software maps quite naturally into multiple threads. These thread communicate via message passing.

Chapter 4

Programming XMOS hardware using XC

To program its chips, XMOS designed a language called XC which is very similar to ANSI C. It contains extra constructs to support the chip's special features which we described in the previous chapter. In contrast to C, XC does not support pointers. However, XC does support passing arguments to functions by reference. To allow returning multiple arguments from a function (which is usually implemented with a pointer in C) XC supports multiple return values.

The XC programming language is based upon Communicating Sequential Processes [9]. Parallelism and IO operations are a fundamental part of CSP. This is also the case for XC, where IO operations are a fundamental part of XC [9]

XC programs are compiled and debugged using a modified version of the GNU toolchain.

4.1 Executing functions in parallel

The `par` statement executes two or more functions in parallel. Each of these functions will be executed in a separate thread. In the case of Listing 4.1, all threads will be started on the same core. In a strict sense, these threads won't therefore run in parallel, but interleaved.

Listing 4.1: Executing functions in parallel

```
1 par{
2   function1();
3   function2();
4 }
```

As illustrated in Listing 4.2, the programmer can specify a core on which the thread has to be executed. This is especially important when a thread is doing IO because not all IO pins are available on a core. This is illustrated by Figure 4.1. IO should therefore be performed on the core where the IO pins are available. Threads that communicate a lot with each other can also be assigned to the same core out of performance considerations (cfr Section 3.2.4). This results in the smallest communication overhead. When a program consists of more than four threads that do heavy calculations, it is recommended to divide them over different cores. This will yield better overall performance. When running more than four threads on the same core, each of them will receive a maximum of 80 MIPS, instead of the maximum of 100 MIPS (cfr Section 3.2.2).

Listing 4.2: Executing functions in parallel on a specified core

```
1 par{
2   on stdcore[0]: function1();
3   on stdcore[1]: function2();
4 }
```

4.2 Communicating between threads

When threads need to communicate, they do so by passing messages over a channel. Each channel has exactly two channel ends. The mapping between channels and channel ends is performed by the XC compiler. As communication is blocking, the two functions communicating need to be inside a `par` statement, in order to be executed in in parallel. If not, the program would stall when the functions try to communicate.

Listing 4.3 illustrates two threads that communicate over a channel. The first thread sends the number 7 over a channel called *c*. The "<:" operator

is the equivalent to the CSP exclamation mark. This thread will now block until the value is read from the other end of the channel. The second thread tries to read a value from the same channel and blocks until it's available. The ":>" operator is equivalent to the question mark operator in CSP.

Listing 4.3: Communicating between concurrently running threads.

```
1  chan c;
2
3  void thread1(chanend c1)
4  {
5    c1 <: 7;
6  }
7
8  void thread2(chanend c2)
9  {
10     int v;
11     c2 :> v;
12 }
13
14 par
15 {
16   thread1(c);
17   thread2(c);
18 }
```

Channels are not directional. This means that two threads can communicate back and forth over the same channel. However, as communication is blocking, threads need to be sending and receiving data in the correct order. If two threads try to send or receive at the same time over the same channel, a deadlock will occur.

The data types of input and output variables must comply with the standard C rules for assignments. The programmer is in charge of casting incompatible data types.

If two threads run on the same core, they can, in theory, communicate through shared memory. However, to preserve the CSP principles, XC always uses the message passing.

In addition to channels, XC also offers streaming channels which have a buffer. When reading from and sending over a streaming channel, threads

won't always block. The sending thread will only block when the buffer is full, while the receiving one will block when that same buffer is empty.

4.3 Performing input and output using ports

Ports (or more specifically the pins they represent) are used to connect to peripheral hardware. A port can represent 1, 2, 4, 8, 16 or 32 pins. This is called the width of the port.

Listing 4.4 depicts a small example performing IO. It will activate LEDs on the development board and subsequently read which buttons are pressed. Two header files are included: the standard C IO library and an XC-specific file which defines the port names used on the development board. Referencing these names, lines 4-6 define two output ports and one input port. Port `PORT_CLOCKLED_SELG` refers to a single pin, while the other two represent multiple pins (cfr Figure 4.1). The operations listed in Table 4.1 are now used to do the actual IO on lines 12-17. Line 16 uses an extra check `pinsneq` on the input port. This will cause the thread to wait until the value of the port is not equal to 15. When the value of the port is no longer 15, the new value is returned.

XC	Description
port <: value;	Immediately write to a port.
port :> int value;	Immediately read from a port.
port when pinseq (data) :> int output	Read from port when the value on the pins equals `data`.
port when pinsneq (data) :> int output	Read from port when the value on the pins differs from `data`.

Table 4.1: IO functions

Listing 4.4: Performing IO operations

```
1  #include <stdio.h>
2  #include <platform.h>
3
4  out port cled0    = PORT_CLOCKLED_0;
5  out port cledg    = PORT_CLOCKLED_SELG;
6  in   port button  = PORT_BUTTON;
7
8  int main()
9  {
10   int b1, b2;
11
12   cledg <: 1;
13   cled0 <: 0x70;
14
15   button :> b1;
16   button when pinsneq(15) :> b2;
17   printf("%d %d\n", b1, b2);
18   return 0;
19 }
```

The mapping between ports and hardware is different for each XMOS (development) board. The mapping for the XC-1A development kit is displayed in Figure 4.1. This figure also illustrates that not all peripheral hardware is accessible form all cores. Most of this hardware is connected to core (or processor) zero. The development board's buttons (`BUTTON [A-D]`) and their accompanying LEDs (`BUTTONLED [A-D]`) are, for instance, only accessible from core zero.

Pin	Port				Processor			
	1b	4b	8b	16b	0	1	2	3
X*n*D0	P1A0				PORT_SPI_MISO	X1PortA Header	X2PortA Header	X3PortA Header
X*n*D1	P1B0				PORT_SPI_SS			
X*n*D2		P4A0	P8A0	P16A0				
X*n*D3		P4A1	P8A1	P16A1				
X*n*D4		P4B0	P8A2	P16A2				
X*n*D5		P4B1	P8A3	P16A3				
X*n*D6		P4B2	P8A4	P16A4				
X*n*D7		P4B3	P8A5	P16A5				
X*n*D8		P4A2	P8A6	P16A6				
X*n*D9		P4A3	P8A7	P16A7				
X*n*D10	P1C0				SPI_CLK			
X*n*D11	P1D0				SPI_MOSI			
X*n*D12	P1E0				CLOCKLED_SELG			X3PortB Header
X*n*D13	P1F0				CLOCKLED_SELR			
X*n*D14		P4C0	P8B0	P16A8	BUTTONLED [A]			
X*n*D15		P4C1	P8B1	P16A9	BUTTONLED [B]			
X*n*D16		P4D0	P8B2	P16A10	BUTTON [A]			
X*n*D17		P4D1	P8B3	P16A11	BUTTON [B]			
X*n*D18		P4D2	P8B4	P16A12	BUTTON [C]			
X*n*D19		P4D3	P8B5	P16A13	BUTTON [D]			
X*n*D20		P4C2	P8B6	P16A14	BUTTONLED [C]			
X*n*D21		P4C3	P8B7	P16A15	BUTTONLED [D]			
X*n*D22	P1G0				TESTPOINT			
X*n*D23	P1H0				UART_TX			
X*n*D24	P1I0				UART_RX			
X*n*D25	P1J0				TESTPOINT			
X*n*D26		P4E0	P8C0	P16B0	PROTO_AREA_4			
X*n*D27		P4E1	P8C1	P16B1	PROTO_AREA_5			
X*n*D28		P4F0	P8C2	P16B2	PROTO_AREA_6			
X*n*D29		P4F1	P8C3	P16B3	PROTO_AREA_7			
X*n*D30		P4F2	P8C4	P16B4	PROTO_AREA_8			
X*n*D31		P4F3	P8C5	P16B5	PROTO_AREA_9			
X*n*D32		P4E2	P8C6	P16B6	PROTO_AREA_10			
X*n*D33		P4E3	P8C7	P16B7	PROTO_AREA_11			
X*n*D34	P1K0				SPEAKER			
X*n*D35	P1L0				TESTPOINT			
X*n*D36	P1M0		P8D0	P16B8	PROTO_AREA_12			
X*n*D37	P1N0		P8D1	P16B9	PROTO_AREA_13			
X*n*D38	P1O0		P8D2	P16B10	PROTO_AREA_14			
X*n*D39	P1P0		P8D3	P16B11	PROTO_AREA_15			
X*n*D40			P8D4	P16B12				
X*n*D41			P8D5	P16B13	CLOCKLED_0 [I]	CLOCKLED_1 [IV]	CLOCKLED_2 [VII]	CLOCKLED_3 [X]
X*n*D42			P8D6	P16B14	CLOCKLED_0 [II]	CLOCKLED_1 [V]	CLOCKLED_2 [VIII]	CLOCKLED_3 [XI]
X*n*D43			P8D7	P16B15	CLOCKLED_0 [III]	CLOCKLED_1 [VI]	CLOCKLED_2 [IX]	CLOCKLED_3 [XII]

Figure 4.1: Port to pin mapping for the XC-1A [12].

Full and detailed information about the mapping between pins and ports for the XC-1A development kit can be found in [12]. Table 4.2 clarifies how a 32-bit value is mapped onto port `BUTTONLED` when written by a thread on core zero. Only the four least significant bits of the 32-bit value map to the port. The 28 most significant bits are not used. This is because the port represents four pins, as can be derived from Figure 4.1.

Data bits	$b_{31} - b_4$	b_3	b_2	b_1	b_0
Port	*not mapped*	BUTTONLED (PORT_4C)			
		P4C3	P4C2	P4C1	P4C0
LEDs		D	C	B	A
Pins		X0D21	X0D20	X0D15	X0D14

Table 4.2: Mapping a 32-bit variable onto ports and pins (on core zero)

It is not possible to immediately address a single pin on a port representing multiple pins. However, the programmer can use the current value of a port and apply a bitmask to it to change a single pin.

4.4 Timing operations

As discussed in Chapter 1 timing is often crucial in embedded software. In serial communication, for instance, the bits need to be put on the line at the exact time defined by the baudrate. Pulse width modulation (PWM), is switching a digital output on and off very often to emulate an analog voltage is another example. In a proper PWM implementation the output needs to be toggled at a frequency of least 100 Hz, making it a highly periodic task. XC offers timers to time operations, for example to pause between bits in serial communication. Timers contain a 32-bit value which is (by default) incremented every clock tick of the processor.

Listing 4.5 depicts a simple PWM implementation which will dim the board's LEDs by enabling them only 10 percent of the time. This program will first read the current time into an integer with the name *time*. After this small setup, the loop doing the actual PWM is entered on line 15. The LEDs are switched on by sending 0x70 to the correct port and turned back off by sending zero. Between switching on and off, there is each time a small delay introduced using the timer.

As the thread runs at 100 MHz, the timer is incremented each 10 ns. This implies that to have a PWM frequency of about 100 Hz, we need a total delay of 1 000 000 clockcycles as illustrated in Figure 4.2. This delay is split in a 1 to 9 ratio between respectively off and on. As the current time is saved into an integer, it is simple to calculate when the next switch of the LEDs

needs to happen. The `timerafter` function will cause an event when the time given as its argument has passed.

$$\frac{clockticks\ per\ second}{PWM\ frequency * PWM\ steps} = delay\ in\ clockticks$$

$$\frac{100 * 10^6}{100 * 10} = 100\ 000\ clockticks$$

Figure 4.2: PWM timing

Listing 4.5: PWM using timers

```
1  #include <platform.h>
2  #define CYCLE 100000
3
4  out port cled0    = PORT_CLOCKLED_0;
5  out port cledg    = PORT_CLOCKLED_SELG;
6
7  int main()
8  {
9    int time;
10   timer tmr;
11
12   cledg <: 1;
13   tmr :> time;
14
15   while(1)
16   {
17     cled0 <: 0x70;
18     time += 1 * CYCLE;
19     tmr when timerafter ( time ) :> void ;
20
21     cled0 <: 0;
22     time += 9 * CYCLE;
23     tmr when timerafter ( time ) :> void ;
24   }
25   return 0;
26 }
```

4.5 Handling multiple events at once

All communication and certain port input operations are blocking. This means that a thread can only check for one event at a time. In order to overcome this limitation, there is the XC `select` statement that allows checking for multiple events at once in a single thread. This `select` statement is illustrated in Listing 4.6

Syntactically the XC `select` is similar to the C `switch` statement. Instead of checking the value of a single variable, `select` allows reacting to events originating from multiple resources. These resources can be ports, channels or timers.

When exactly one event has occurred, the associated action will be executed. When more than one event has occurred, only one action will be executed. The `select` statement is often used inside an endless loop. During each iteration, a different action will be executed. If no events have occured, the thread will block until one of the events occurs. The `select` statement can end with a "default case", which will be executed when not a single event has occured. Clearly, this also implies that the thread will not be suspended when no events are ready.

Listing 4.6 lists a `select` statement with three cases. The first waits for data to become available on the channel with a channel end named `inputchanend`. This data is written to the variable `c` after which the statement's body is executed. The second case waits for a value different from 15 showing up on the port called `buttons`. The last case will become applicable when the timer `tmr` has passed the value of variable `t`.

Each case statement has to end with `break` or `return`. Therefore, contrary to the C `switch` statement, it is not possible to have one case statement continue into the next one.

Listing 4.6: Select statement

```
1  unsigned c, x;
2
3  select
4  {
5    case inputchanend :> c:
6      ...
7      break;
8    case buttons when pinsneq(15) :> x:
9      ...
10     break;
11   case tmr when timerafter(t) :> void:
12     ...
13     break;
14   default:
15     ...
16     break;
17 }
```

It is not possible to use output operations in case statements (a limitation originating from CSP). This could be useful when a thread wants to send data via a channel to another thread. For example when implementing a buffer in a thread with one channel for incoming data and one for outgoing data. However, it is possible to work around this limitation, by having the receiving thread send a ready signal to the buffer (as shown by Figure 4.3). As this ready signal is input, it can be used as a case statement which will perform the output operation.

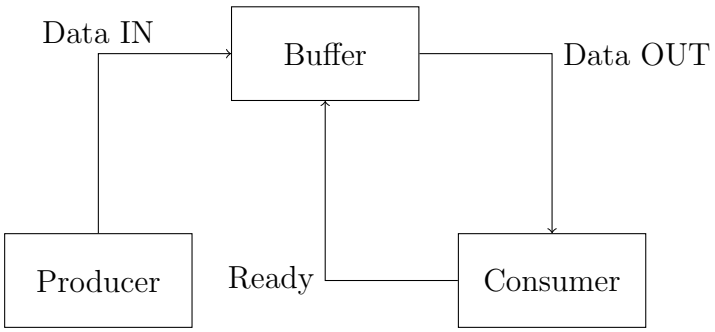


Figure 4.3: Buffer structure

An implementation of this buffer is shown in Listing 4.7. The buffer can store twelve integers. This implementation uses an extra guard in the case statements. Using a guard, the channel is only read from when the expression before the "=>" evaluates to true.

Listing 4.7: Buffer implementation

```
1  void boundedbuffer(chanend producer, chanend consumer){
2    int moreSignal;
3    int buffer[12];
4    int inp = 0;
5    int outp = 0;
6
7    while(1){
8      select{
9      case inp<outp+12 => producer:>buffer[inp % 12]:
10       inp++;
11       break;
12     case outp<inp => consumer:>moreSignal:
13       consumer <: buffer[outp % 12];
14       outp++;
15       break;
16     }
17   }
18 }
```

4.6 Case study revisited: a low-level event-driven implementation in XC

We will revisit the case study introduced in the Chapter 2. Concretely we ported it to the XMOS platform and hence moved from an interrupt-driven to an event-driven architecture. Some adaptations are needed as the two hardware platforms are vastly different.

We implemented a program through which users can adjust the color of the LEDs on two XC-1A development boards. The colors of the LEDs range from red over yellow to green. Each development board has four buttons through which the colors of the LEDs can be changed. The colors are synchronized

between the two boards. This synchronization is done wirelessly via XBee modules.

The program on each board contains both its own color and the one shown on the other board. These values are kept synchronized. When the user presses one of the buttons to change the LEDs color, the new values are sent to the other board.

Figure 4.4: Closeup of the LEDs

As shown in closeup on Figure 4.4, the XC-1A development board red-green LEDs (twelve in total). Each of these red-greed LEDs actually contains two separate LEDs, one for each color. By switching between the red and green LEDs quickly we can fool the human eye to believe that it sees a single color. Figure 4.5 depicts an example where the red led is on for 60 % of the time, while the green one is on for 40 %. This cycle should run at a frequency of about 100 Hz. This frequency will ensure that a human observer will not notice any flickering caused by the rapid switching.

...	G	G	R	R	R	R	R	R	G	G	G	G	R	R	R	...

Figure 4.5: Showing a color with 60% red and 40 % green

The application is split up in four threads as depicted in Figure 4.6. Two of these threads are responsible for the UART communication, concretely the receiver (UART RX) and the transmitter (UART TX). A third performs the pulse width modulation of the LEDs The fourth contains the application logic and handles the user input from the buttons.

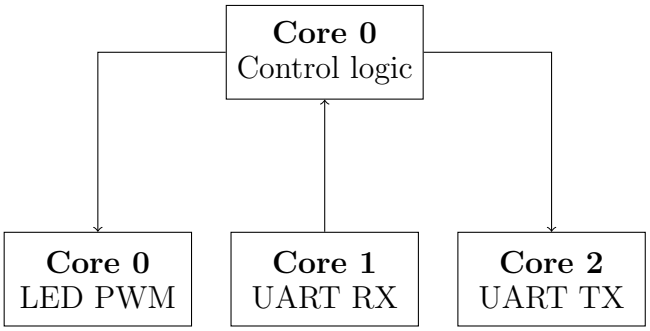


Figure 4.6: Structure of the case study application

4.6.1 Hardware setup

Figure 4.7 displays the hardware setup used for this case study. The wiring of this setup is schematically displayed on Figure 4.8. It connects the XMOS XC-1A development board on the left with an XBee ZigBee receiver on the right. The former is used to power the latter with its built in 5 volt power supply. However, the XBee module itself works on 3.3 volt [10]. Therefore, an Arduino XBee shield is used as it contains the needed hardware [20] to do the voltage conversion for both the power supply and the signal levels.

The wires used to communicate with the XBee module are connected to `PORT_1A` on both cores one and two. Figure 4.1 displays that this port is connected to the pin called XnD0. This particular pin is mapped to the `X1PortA` and `X2PortA` headers respectively on cores one and two. More information about the exact location of these connections on the development board can be found in its documentation [12].

For debugging purposes an XBee module can be connected to a PC, using another ArduinoXBee shield. This XBee sheeld is plugged into an Arduino board which has its ATMEL CPU removed. The microcontroller-less Arduino board contains an FTDI chip which performs the RS-232-to-USB conversion. On a PC with correctly installed drivers, the XBee module will then show up as a serial device. After configuring a 9600 baudrate and the 8N1 data format, any serial terminal emulator can be used to send data wirelessly to the XMOS board.

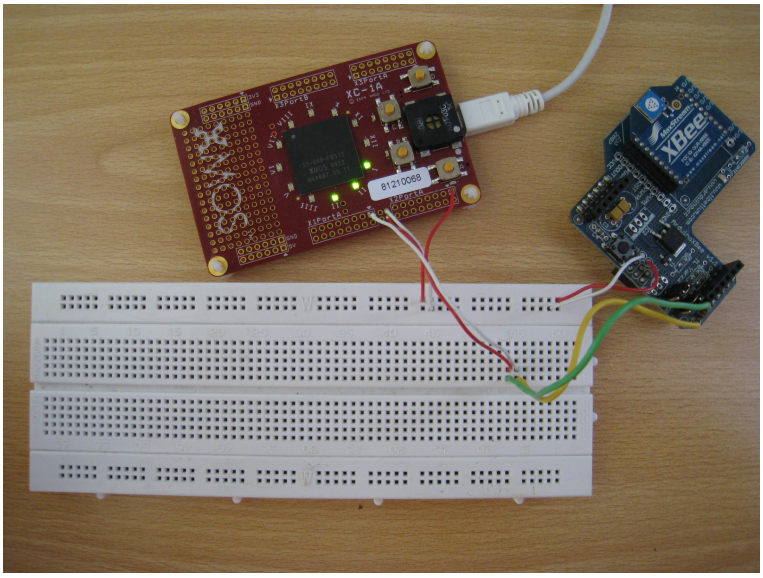


Figure 4.7: Hardware setup

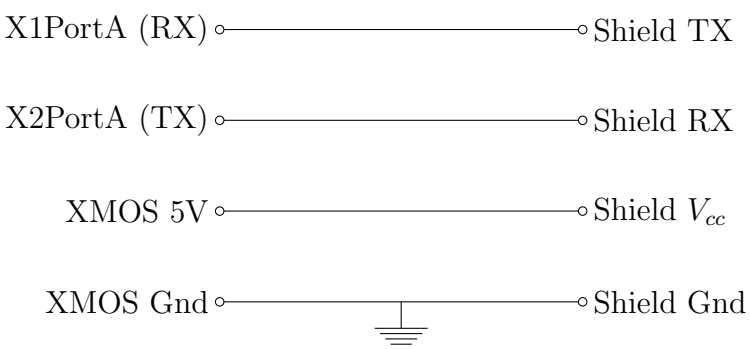


Figure 4.8: Schematic hardware setup

4.6.2 UART communication

As the XMOS hardware can be used to implement functionality that is usually implemented in hardware, we implement the serial communication over UART in software instead of using the dedicated hardware implementation. This means that we will implement a low level, meaning that each bit will be sent over the wire manually. There are many different standards and variations of how to do serial communication. In this case study, the 3-wire RS-232 protocol is used. As the name suggests, it uses 3 wires: two for carrying data (one for each direction) and a wire to have a common ground

between the communicating devices. The ground is needed in order that both devices have the same reference to measure the 0 volt and 5 volt levels. There is no error checking or flow control, meaning that its implementation can be kept relatively simple.

There are various speeds and formats to send data. The transmission speed (also called baudrate) varies between 300 and 115200 bits per second. In this case study, we will use a baudrate of 9600. There are various formats, but the one which will be used sends 8 data bits at a time end ends it with a stop bit. As previously mentioned, there is no error checking on the transmission in this example, although a parity bit could be supported to allow basic error detection. The used format is usually abbreviated to "8N1": 8 data bits, no parity, one stop bit. The format and the speed need to be configured correctly by the programmer. If that person fails to do so, the output will be incorrect due to reading the data at the wrong moment and/or misinterpreting of the received bitstream.

As visualized by Figure 4.9, a wire used for transmission is by default at 0 volt. A transmission is announced by a logical zero as start bit (which is represented using 5 volt). After that, the data is put on the line — the least significant bit first. The levels used to send the data are inverted. A logical one is represented by 0 volt, while 5 volt represents a logical zero. A transmission is terminated by a stop bit, which consists of a logical one, putting the line back at 0 volt.

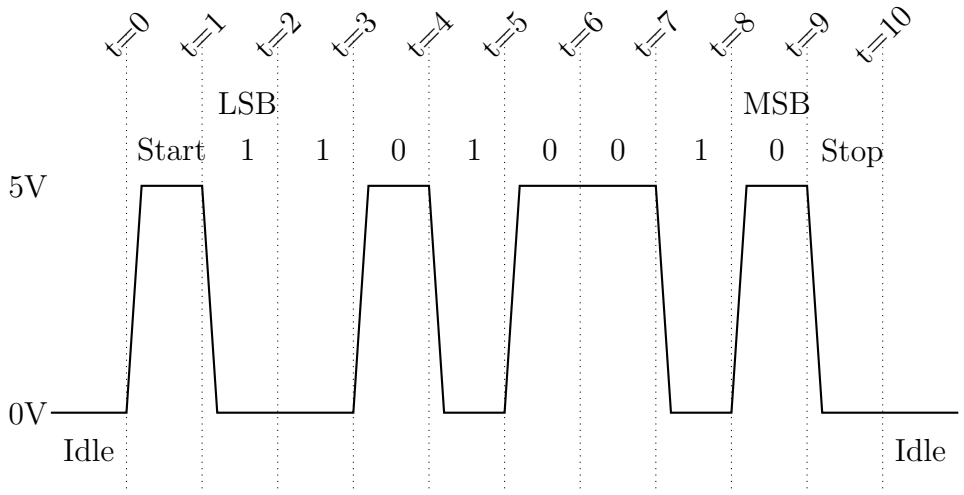


Figure 4.9: RS-232 signal levels

Because the communicating devices cannot negotiate about the speed during the transmission, timing is essential to perform a correct transmission of the data. When a sender puts the start bit on the line it is crucial that the right data is placed correctly on the line at the correct moment. On the receiver side, the receiving software will listen for a rising edge on the line. This rising edge is the transition from the default 0 volt to the 5 volt of the start bit. The receiver will subsequently need to read the signal of the line at the appropriate time. In addition, it will need to ensure that the measurement is only performed when the signal on the line is stable[1]. Therefore, the level is measured with a small offset compared to the moment the sender put the signal on the line. This offset is usually half the bit time. UART communication clearly requires strict timing. Because the XMOS chip gives each thread a guaranteed amount of CPU cycles, we are sure that timing constraints will always be met. The same implementation on an interrupt driven chip cannot do that, as the UART code can be interrupted at any moment.

Listing 4.8 depicts the code implementing the UART transmitter. The statement on line 11 causes the thread to wait until it receives a byte to be sent from the channel end. To calculate the time between bits, the current time needs to be stored. This time is used as a base to calculate when each next bit needs to be put on the line. Figure 4.10 illustrates how such a delay between bits is calculated. The timer is incremented with each clock tick and the processor runs at 100 MHz and a baudrate of 9600 is used.

$$\frac{clockticks\ per\ second}{baudrate} = \frac{100 * 10^6}{9600}$$

Figure 4.10: Delay between bits during serial communication

The code on line 15 starts the transmission by putting the start bit on the line. After this bit, we wait a brief moment. Lines 20 to 24 put the eight data bits on the line, with the same delay between them. The transmission is completed by putting a logical one on the line. The length of the wait depends on the baudrate used during communication and is essential to have successful communication.

[1] A line is stable when the voltage on it will not change during sampling, ensuring that a correct value is read.

Listing 4.8: UART transmitter

```
1  #define BIT_RATE 9600
2  #define BIT_TIME 100000000 / BIT_RATE
3
4  on stdcore[2]: out port TXD = XS1_PORT_1A;
5
6  void transmitter (chanend transmit) {
7    unsigned byte, time;
8    timer t;
9
10   // Initial stop level
11   TXD <: 1;
12
13   while (1) {
14     /* get next byte to transmit */
15     transmit :> byte;
16     t :> time ;
17
18     /* output start bit */
19     TXD <: 0;
20     time += BIT_TIME ;
21     t when timerafter ( time ) :> void ;
22
23     /* output data bits */
24     for ( int i=0; i <8; i ++) {
25       TXD <: >> byte ;
26       time += BIT_TIME ;
27       t when timerafter ( time ) :> void ;
28     }
29
30     /* output stop bit */
31     TXD <: 1;
32     time += BIT_TIME ;
33     t when timerafter ( time ) :> void ;
34   }
35 }
```

The receiver code, which is illustrated in Listing 4.9, exhibits a lot of similarities with the transmitter. It will wait for a start bit to be put on the line. Next, each subsequent bit is put on the line after a time `BIT_TIME`. However,

an extra delay of half the bit time is added. This will ensure that the bit sampled in the middle of the time is on the line. At that moment, the voltage level is stable ensuring a correct sampling. After the start bit, eight data bits will be sampled. Finally, a stop bit is sampled, but not saved. Now the data is sent to another thread via the channel connected to the "received" channel end.

Listing 4.9: UART receiver

```
1  on stdcore[1]: in port RXD = XS1_PORT_1A;
2
3  void receiver (chanend received) {
4    unsigned byte, time;
5    unsigned levelTest;
6    timer t;
7
8    while (1) {
9      /* wait for negative edge of start bit */
10     RXD when pinseq (1) :> void ;
11     RXD when pinseq (0) :> void ;
12
13     /* move time into centre of bit */
14     t :> time ;
15     time += BIT_TIME /2;
16     t when timerafter ( time ) :> void ;
17
18     /* Ensure start bit wasn't a glitch */
19     RXD :> levelTest;
20     if (levelTest == 0) {
21
22       /* input data bits */
23       for ( int i=0; i <8; i ++) {
24         time += BIT_TIME ;
25         t when timerafter ( time ) :> void ;
26         RXD :> >> byte ;
27       }
28
29       /* input stop bit */
30       time += BIT_TIME ;
31       t when timerafter ( time ) :> void ;
32       RXD :> levelTest ;
33
```

```
34          /* Send rx data if stop bit valid */
35          if (levelTest == 1) {
36            byte = byte >> 24;
37            received <: byte;
38          }
39        }
40      }
41  }
```

4.6.3 Pulse Width Modulation

The third thread, displayed in Listing 4.10, implements the pulse width modulation.

First, an initialization is performed. It initializes a default PWM value, the timer and it enables all LEDs by sending a value of 0x70 to the appropriate port. Next, an endless loop is entered. This loop contains a select statement which will either read an incoming value from the channel or perform a PWM operation. To perform the PWM, either the red or green LEDs (via the `cledR` and `cledG`) are selected at the appropriate moment to give the impression of different colors.

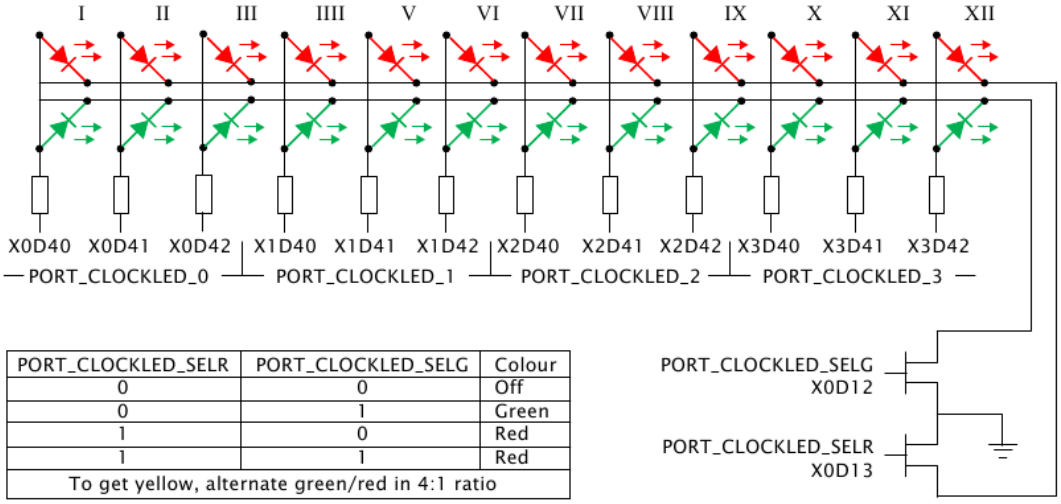
Listing 4.10: Pulse Width Modulation

```
1  #define FLASH_PERIOD 100000
2  #define PWMMAX 15
3  #define PWMSTART 7
4
5  out port cled0 = PORT_CLOCKLED_0;
6  out port cledG = PORT_CLOCKLED_SELG;
7  out port cledR = PORT_CLOCKLED_SELR;
8
9  void led (chanend pwm) {
10   int red = PWMSTART;
11
12   timer tmr;
13   unsigned t;
14   tmr :> t;
15
16   unsigned ledGreen = 0x1;
17   cled0 <: 0x70;
18
19   while (1)
20   {
21     select
22     {
23       case pwm :> red:
24         break;
25       case tmr when timerafter(t) :> void:
26         t += FLASH_PERIOD*(ledGreen?(PWMMAX-red):red);
27         cledG <: ledGreen;
28         cledR <: !ledGreen;
29         ledGreen = !ledGreen;
30         break;
31     }
32   }
33 }
```

Because of the particular hardware layout of the XC-1A development board, we also need to do LED multiplexing[2]. Clearly there needs to be a method

[2]Multiplexing is used to share a resource over multiple devices (in this case a pin which is connected to two LEDs). This reduces the required number of pins from 24 to 14, as depicted on Figure 4.11

to select the required device too, which in this case is two extra lines. As depicted in Figure 4.11, both the red and green LEDs are connected to the same ports. However, using the `CLOCKLED_SELR` and `CLOCKLED_SELG` one can select which banks of LEDs the circuit needs to be closed of by connecting them to ground.



PORT_CLOCKLED_SELR	PORT_CLOCKLED_SELG	Colour
0	0	Off
0	1	Green
1	0	Red
1	1	Red
To get yellow, alternate green/red in 4:1 ratio		

Figure 4.11: LED configuration on the XC-1A [12]

A fourth thread, listed in Listing 4.11, handles the user input using buttons and the incoming data from the serial port. It contains a select statement which is executed continuously. This select either handles a button press or incoming serial data. In the former case, the new PWM values are calculated, which are subsequently sent to the other device. Next, we wait until the button is released and pause for a while. In the latter case, the received PWM values are used to update the current values.

Listing 4.11: Control logic

```
1  #define CORE_FREQ 100000000
2  #define BUTTON_TIME CORE_FREQ / 4
3
4  int handle_press(unsigned buttons, unsigned shift,
       unsigned pwm_value) {
5    buttons = (buttons >> shift) & 3;
6    if(buttons == 3) return pwm_value;
7    if(buttons & 1)
8      if(pwm_value < PWMMAX) pwm_value++;
9    else
10     if(pwm_value > 0) pwm_value--;
11   return pwm_value;
12 }
13 void button (chanend pwm, chanend commands, chanend
       response) {
14   timer tmr;
15   unsigned t, value;
16   unsigned local_pwm = PWMSTART;
17   unsigned remote_pwm = PWMSTART;
18
19   while (1) {
20     select {
21     case buttons when pinsneq(15) :> value:
22       local_pwm = handle_press(value, 0, local_pwm);
23       remote_pwm = handle_press(value, 2, remote_pwm);
24
25       pwm <: local_pwm;
26       response <: (int)(remote_pwm + (local_pwm << 4));
27
28       buttons when pinseq(15) :> void;
29       tmr :> t;
30       t += BUTTON_TIME;
31       tmr when timerafter(t) :> void;
32       break;
33     case commands :> value:
34       remote_pwm = (value >> 4) & 0xF;
35       local_pwm = value & 0xF;
36       pwm <: local_pwm;
37       break;
38 } } }
```

4.6.4 Distributing threads over cores

The individual components discussed above need to work together to create a properly functioning program. Each of the above-mentioned functions is started in their own thread and run in parallel. These threads occupy three of the four cores of the XMOS chip. They communicate using three channels (as illustrated by the schema in Figure 4.6).

Listing 4.12: Application structure

```
1  int main()
2  {
3    chan rx, tx, pwm;
4
5    par
6    {
7      // UART RX thread
8      on stdcore[1]: receiver(rx);
9      // UART TX thread
10     on stdcore[2]: transmitter(tx);
11     // PWM thread
12     on stdcore[0]: led(pwm);
13     // Button handler and control logic
14     on stdcore[0]: button(pwm, rx, tx);
15   }
16   return 0;
17 }
```

This case study illustrates how to program the multi-core and event-driven XMOS architecture. Embedded applications map quite naturally to the multi-core architecture, as there are multiple tasks to be executed in parallel. Thanks to the guaranteed amount of CPU cycles each thread gets, we can be certain that our timing constraints will be met at all times. This is especially important when performing serial communication, as it requires strict timing.

Compared to the interrupt-driven implementation of the same program discussed in Section 2.3, this code is easier to understand. It does not contain hidden code paths due to the interrupts and cannot contain issues due to shared variables between the interrupts and the main application logic.

4.7 Conclusion

To program its chips, the XMOS company has designed a programming language called XC. XC is similar to ANSII C but contains the needed extra syntax to create event-driven software. It is possible to start parallel threads using the `par` statement. These parallel threads can communicate via message passing using channels. To perform input and output, XMOS uses the concept of ports. These ports represent one or more physical pins. Events can also be added to ports, allowing a thread to suspend until a certain value is available on the port. Timers allow to time operations.

In the case study we show how to program the XMOS chip. The resulting source code is easier to read and understand than its interrupt-driven counterpart. Due to the chip's guarantees concerning the execution speed of threads, the application will always meet its timing constraints.

Chapter 5

High-level event-driven programming in Scheme

In this chapter, we port the high-level programming language Scheme to the XMOS architecture and extend the language with abstractions for the concurrency model of XMOS. Programming embedded systems in Scheme has the advantage of not having to work in low-level languages such as C and XC which simplifies the work of the programmer as he or she does not has to think about issues concerning, memory management, et cetera. Concretely, we will port and extend an existing Scheme interpreter to the XMOS platform. Next, we extend this interpreter to be able to use features specific to the XMOS hardware. These features include evaluating functions in different threads and a message passing mechanism between threads.

5.1 Selecting a suitable Scheme system

As previously discussed, the 400 MHz XS1-G4 chip used provides considerable processing power. However, it is severely limited by the amount of memory, on each core only 64 kilobytes. As listed in Table 5.1, the size of most minimalistic Scheme interpreters exceed the available 64 kilobytes, consequentially we had to look into different techniques.

Implementation	Size of interpreter
fools 1.3.2	288 KB
minischeme 0.85	95 KB
scm 4e1	368 KB
siod 3.0	166 KB
bit (interpreter, with full library)	72 KB
bit (interpreter only)	22 KB

Table 5.1: Size of different small Scheme implementations[5]

5.1.1 Implementation constraints

Small memory footprint As mentioned before, the memory footprint of the interpreter has to be very small, as each core is equipped with only 64 kilobytes of memory. In this 64 kilobytes we need to fit the interpreter itself and leave enough space for the runtime memory requirements.

Evaluated or byte code based architecture In evaluated interpreters, the Scheme code is entirely evaluated on the chip, at runtime. This means that the interpreter has no prior knowledge about the functionality used by the Scheme application. Consequentially, all primitives and functions need to be present in memory at all time. This can become troublesome considering the limited memory. When the Scheme code would be directly interpreted on the chip, all functions the application could possibly call need to be available in memory. This significantly increases the needed memory.

Another approach is by using a bytecode based interpreter. As the compiler knows which functions from the library will be used by the Scheme application, he can leave out the functions which are not required by the application. This significantly reduces the memory footprint compared to evaluation based interpreters. However, the extra compilation step to bytecode makes developing applications more tedious.

Byte code interpreters can offer more functionality in a low memory footprint than evaluation based interpreters. This is due to the bytecode compilation phase which can remove the functions not used by the Scheme application.

Uses pointers As XC doesn't support pointers, it is recommended to use an interpreter which doesn't use pointers either. Consequently the interpreter can be extended with XC specific code more easily.

Real-time garbage collector The XMOS hardware guarantees the execution speed of threads and therefore also the applications being interpreted. However, when a garbage collector can stop the application code for an arbitrary period, this guarantee becomes useless. Therefore, we need a garbage collector can preserve the execution speed guarantees of the XMOS architecture.

RAM and ROM Many chips used in the embedded domain use ROM memory to store the program. The RAM memory is only used for the runtime memory. However, this is not the case for the XMOS chips where the application and the runtime requirements are located in the same memory.

Already used in embedded domain It is an advantage that an interpreter already has been used in the embedded domain. As its functionality is more likely to be targeted to this domain.

5.1.2 Comparing different interpreters

We compared different interpreters to find the most suitable one. We took into account the criteria discussed in Section 5.1.1. The most important and restricting criteria, is the limited amount of memory available. Most interpreters require an order of magnitude more RAM than what is provided by the XMOS chip (cfr Table 5.1). Table 5.2 shows an overview of interpreters which show interesting characteristics making them suitable to be ported to the XMOS chip.

Ypsilon is a Scheme interpreter which claims having a
mostly concurrent garbage collection", which is optimized for the multi-core CPU system [7]. However, the interpreter is badly documented and is an order of magnitude too large for our target device.

MiniScheme is an evaluating interpreter. Its functionality is rather limited and is quite large to fit in the available memory. However, by removing more functions, it might fit into the XMOS chip's memory. Its garbage collector cannot offer timing guarantees, which is highly desirable for this project.

	Memory (KB)	Byte Code	Pointers	RT GC	Embedded	RAM-ROM	
Bit Scheme	22	•	•	•	•	–	[5]
MiniScheme	95	–	•	–	–	–	[17][19][5]
PIC Bit	23	•	•	–	•	•	[4]
PICO Bit	17.5	•	•	–	•	•	[26]
Ypsilon	1600	–	•	•	–	–	[7]

Table 5.2: Scheme interpreters

Bit Scheme, PIC Bit and PICO Bit are three very similar byte code based Scheme interpreters. However only Bit Scheme contains a real-time garbage collector and isn't specifically designed for systems equipped with ROM memory. This makes Bit Scheme the most suitable interpreter to port to the XMOS chip.

Both \ˈskēm\[1] and Pico[2] were also considered, however, they were too large to fit in the memory available on the XMOS chip.

5.2 Exploiting the XMOS concurrency model in Scheme

There are multiple approaches that can be chosen to exploit the concurrency made available by the multi-threaded architecture of the XMOS chip.

A possibility is to run multiple independent interpreters in parallel. Preferably distributed over all available cores. This gives access to all IO pins and all available memory.

It also is possible to create one virtual memory space by combining the memory of all four cores into one. This would result in a memory of 256 KB.

[1] http://soft.vub.ac.be/soft/skem
[2] http://pico.vub.ac.be/

As shown on Figure 5.1, all interpreters would be running concurrently on the same core. However, as illustrated on Figure 4.1, an IO pin is only accessible on one core. This means that when all threads are running on one core, only a fourth of the IO pins would be accessible. To solve this limitation, we could implement dedicated IO threads on each core. That way the interpreters can reach all pins from a single core by sending messages to the IO thread on the appropriate core. Because of the architecture of the XMOS chip, we can run up to four threads concurrently on a single core without losing execution performance (cfr Figure 3.3). Running more than four threads on a single XCore would result in a performance decrease, compared to what the chip is actually capable of when the threads are distributed over all four cores. Another issue we would face is the increased latency when accessing memory on a different core than where the interpreter is running. Each memory access needs to be encapsulated in a message which is then passed over the interconnect to the other core. That core will then reply with the content of the requested memory address. It is clear that this will add a significant overhead.

Another possible approach is to run a concurrent garbage collector. For threads on the same core, we can use shared memory to run a concurrent garbage collector. This, however, breaks the basic principles of the XMOS programming model which is based on threads communicating solely by means of message passing.

A different method is by using the Scheme thread model which is based on shared memory and can spawn threads during runtime [21]. However that doesn't map properly on the XMOS hardware where threads communicate with each other by means of message passing. Also, on XMOS chips, threads are implemented in hardware and the maximum number of threads that can run concurrently is limited. Finally, these threads are mapped on the cores during compile time.

Termite Scheme is a Scheme variant developed for distributed computing [8]. It is based on message passing between processes which run on physically separated systems. This is very similar to the XMOS approach where threads have their own memory, although it is only physically separate when threads are running on separate cores. In Termite Scheme the processes exchange messages over an unreliable network. This is not entirely the same inside the XMOS chip, where the interconnect between the cores is considered reliable. XMOS thread are not lightweight as opposed to the Termite processes. The XMOS chip only supports a limited number of threads, while Termite Scheme programs can consist of hundreds of processes. Both XMOS and Termite use

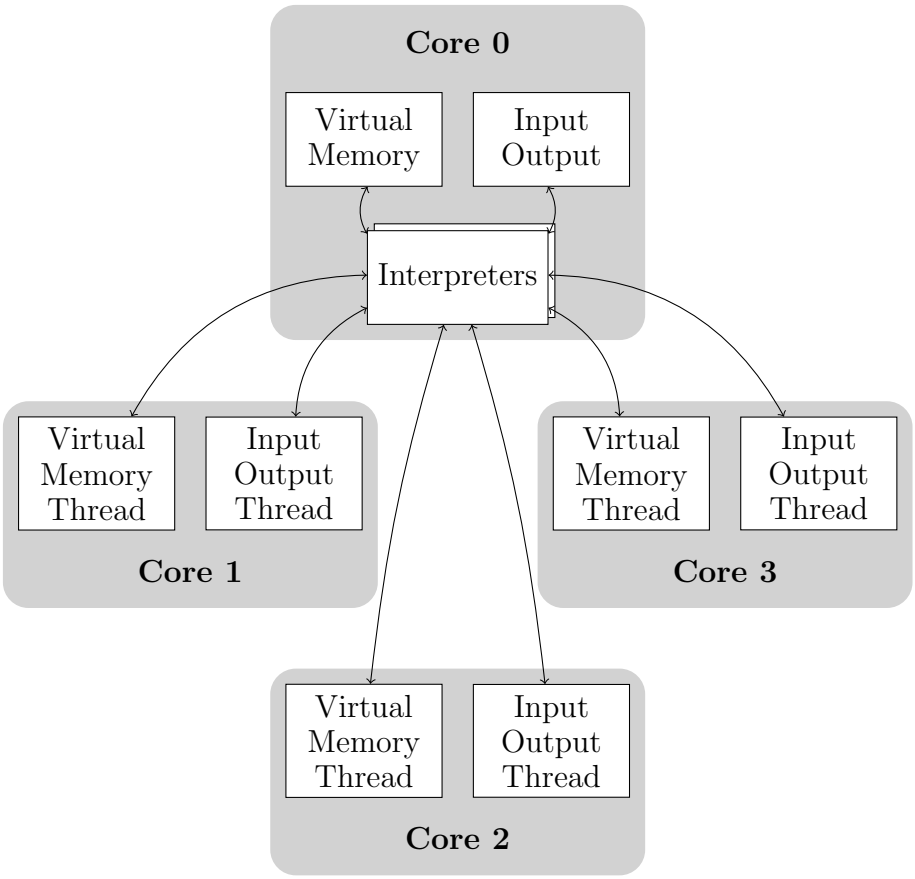


Figure 5.1: Virtual memory architecture

message passing, however, they don't behave the same. In XMOS chips, both sending and receiving messages is blocking, while in Termite sending is asynchronous and receiving is blocking. This clearly has to do with the fact that in Termite Scheme sending messages is unreliable, consequentially a message sent may never arrive at its destination.

5.3 Bit Scheme

After evaluating the criteria against existing Scheme implementations, Bit Scheme was found to be the most suitable Scheme interpreter to serve as a basis for a port to the XMOS architecture.

Clearly the interpretation of the bytecode comes with a performance overhead compared to running compiled applications written C and XC. After porting and modifying the Bit interpreter to run on the XMOS, the interpreter itself takes about 24 kilobytes of memory per core, as displayed in 5.1. This does not include any bytecode, nor runtime memory requirements.

This Scheme interpreter is byte-code based which implements R^4RS [2]. Bit Scheme was originally written for the Motorola 68HC11 [5], but was also ported to PIC microcontrollers, two radically different architectures [4]. It is bytecode based, which allows it to be very small. Its bytecode instruction set is also created with memory saving in mind, allowing to make the bytecode even smaller. This is also due to the lack of runtime checks, Bit Scheme assumes that the bytecode it runs is error-free. This lack of runtime checks however can complicate debugging as the interpreter does not give error messages, it just crashes. Due to the very low memory requirements, it fits in the tiny memory of the XMOS chip, while leaving enough space for the bytecode and the runtime memory.

Bit Scheme also contains a real-time garbage collector. In this context real-time means that the garbage collector is guaranteed to return within a fixed amount of time.[5] This is especially useful when timing constraints need to be met. This is often the case in embedded software. It comes with a compiler, implemented in Scheme, that translates the Scheme source code into bytecode specific for the Bit interpreter. We ported the Bit Scheme interpreter to the XMOS platform and extended to support functions specific to the XMOS hardware. Extending the interpreter is needed to be able to use all the available cores on the XMOS chip and to be able to create event-driven applications.

To exploit the parallelism provided by the XMOS chip, we chose to run one bytecode interpreter per core. This allows to use the entire available memory. Each core has only access to a port of the ports for doing IO, therefore this approach also allows to use all IO possibilities available on the chip. The message passing is now only needed when threads explicitly need to communicate.

In this project the number of threads was limited to one per core. In theory, one could perfectly run more than one interpreter on a core. However, the available memory space is already very small and it would further divide it between the interpreters on that core. This would only allow very small programs on each interpreter.

5.4 XMOS Bit Scheme: bytecode interpreter

The bytecode interpreter of XMOS Bit Scheme uses the layered architecture depicted in Figure 5.2. We have placed the original bytecode interpreter between 2 layers of XC code. This layered architecture is necessary because XC does not support pointers, which are extensively used in the interpreter. As a result XC code cannot be used within the implementation of the bytecode interpreter. However, the linker in the XMOS toolchain can link object files compiled from XC on the one hand and C source code on the other hand together into one file. This entails that XC and C (and also pointers) can be used together in an application — just not together in the same source file.

XC bindings	(XC)
Bytecode interpreter	(C)
Initialization	(XC)

Figure 5.2: Interpreter architecture

As Bit Scheme is implemented in ANSI C, it compiles fine using the XMOS toolchain. However, by default it doesn't use any of the features concerning events and parallelism that are available in the hardware. To be able to use all the functions of the XMOS chips, the bytecode interpreter needs to be adapted and extended. The four cores each need to get the correct piece of bytecode. We will discuss our solution to this problem in the next section. In addition, we implement communication primitives that allow message passing between the interpreters. This newly added functionality will be implemented in the layers above and beneath the bytecode interpreter.

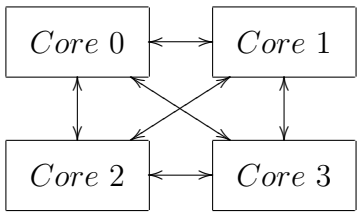


Figure 5.3: Channels between the interpreters

The lowest layer is ran when the XMOS chip powers on and is called "Initialization". We have implemented this component in XC. It starts an interpreter on each core and sets up the necessary channels to allow the interpreters on the different cores to communicate with each other. Channels

allow bidirectional communication, therefore only one channel is needed to connect two interpreters. As illustrated by Figure 5.3, we create a static network of point-to-point channel between the four interpreters. Each interpreter has 3 channels, over which it can communicate directly with the other interpreters. Each of these channels is defined at compile time.

The top layer, "XC bindings" contains the implementation of the primitives we added. These primitives provide IO, timing and message passing functionality to the interpreter.

5.5 XMOS Bit Scheme: bytecode instruction set

XMOS Bit Scheme features several new primitives that expose the functionality of the underlying XMOS hardware. We implemented these primitives in the "XC bindings" layer. Table 5.3 shows an overview. The primitives will be described in detail in Sections 5.7 and 5.8. These primitives are implemented by extending the compiler to have these primitives compiled to new bytecode instructions. Clearly, the bytecode interpreter needed the appropriate modifications to support these new instructions. Bytecode instructions 27 till 34 expect one argument, while the instructions in the range from 49 to 55 expect two arguments. As a result of these modifications, programs written in the original Bit interpreter are not bytecode-compatible with this version.

5.6 XMOS Bit Scheme: distributing bytecode across cores

5.6.1 First compilation phase

The compilation process consists of 2 distinct phases. In the first one, the Scheme program is compiled to bytecode. During the second phase the resulting bytecode is compiled together with the bytecode interpreter into an executable to run on the XMOS chip. This phase is performed by the Bit Scheme compiler, which itself is written in Scheme. We have modified this compiler to support the XMOS platform. As displayed in Listing 5.1, the

Function name	Bytecode	Arguments	Description
get_time	5	*none*	returns current time
after	27	time	pause thread until *time*
pin	28	port	read value from *port*
pon	29	port	enable *port*
poff	30	port	disable *port*
pconf_in	31	port	configure *port* as input
cin	32	core	receive data from core *core*
select_cin	33	port	used by *select* statement
select_after	34	time	used by *select* statement
pout	49	port, value	write value to *port*
peq	50	port, value	pause thread until port *port* equals value *value*
pne	51	port, value	pause thread while port *port* equals value *value*
pconf_out	52	port, value	configure *port* as output, with *value* as default value
cout	53	core, value	send data to core *core*
select_peq	54	port, value	used by *select* statement
select_pne	55	port, value	used by *select* statement

Table 5.3: Added primitives

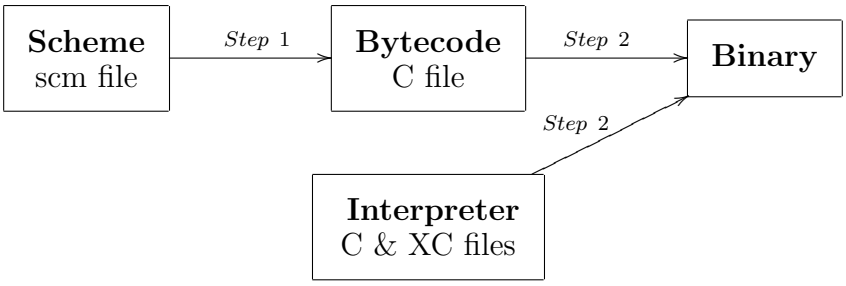


Figure 5.4: Compilation of a Bit Scheme application into a binary for XMOS devices

compiler expects to be invoked with two arguments, the first being the source Scheme file, the second the name of the output file.

Listing 5.1: Compiler invocation

```
1 (byte-compile "input.scm" "output.c")
```

During this compilation from Scheme to bytecode, parallel Scheme programs are divided into independent pieces of bytecode. Each piece will be assigned to an interpreter. The `par` statement itself is only used by the Scheme compiler. It is not translated into a bytecode equivalent. Distinct bytecode will be generated for each core, as illustrated by Figure 5.5. This is needed because each core has its own individual memory and because one interpreter is installed on each core. Therefore no bytecode can be shared between interpreters.

5.6.2 Second compiler phase

The result of the first compilation phase is a single C file. This C file contains the bytecode, the constants and the global variables for each core. In the second compilation phase, the C file containing the bytecode for the scheme program is compiled together with the source code for the bytecode interpreter. This is carried out using the compiler and linker of the XMOS toolchain. The result is a monolithic binary file containing all bytecode for all cores together with the interpreter.

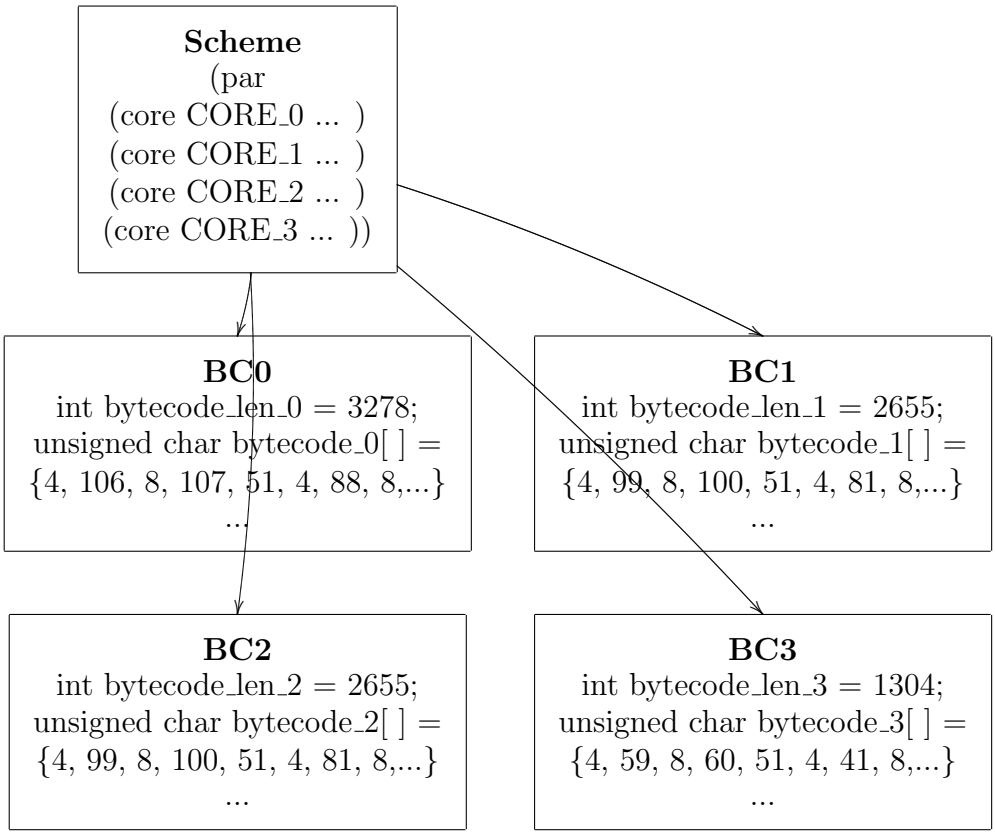


Figure 5.5: Compilation of a parallel Scheme program into bytecode

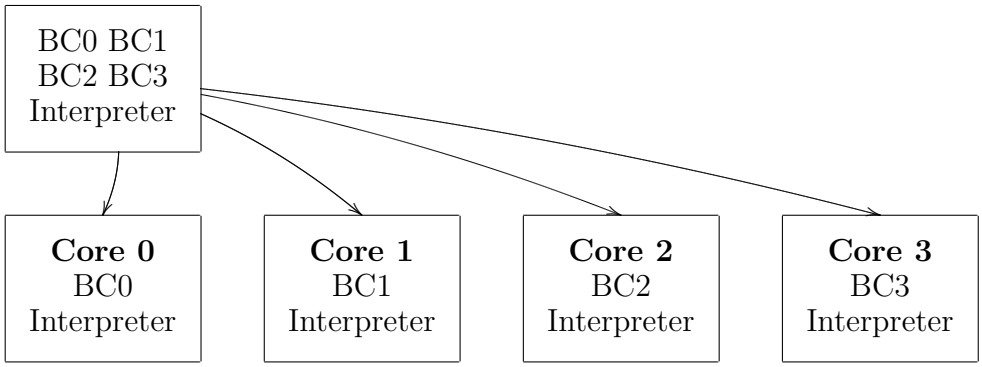


Figure 5.6: Mapping of the code on the different cores

5.6.3 Mapping bytecode to specific cores

Next, the XMOS mapper will map the appropriate code to the appropriate cores. It performs an analysis to remove code that will not be reached on a certain core. Concretely, this will result in four chunks of binary each containing bytecode for a core plus the interpreter (cfr Figure 5.6).

Of course the XMOS mapper will need some clear instructions about which code should end up on which core. To this end, we use the code shown in Listing 5.2. It relies on a get_core_id() function which returns the ID of the core on which the code runs. This gives the mapper a clear indication that a piece of bytecode can only be accessed on exactly one core. As a result bytecode which will be executed on a different core than the current one, will be stripped away. This is highly desirable as each core only has 64 kb at its disposal to run the program and store its bytecode in addition to the interpreter. The interpreter itself takes about 24 kb of memory. This leaves about 40 kb for the bytecode and the runtime memory. Clearly the available runtime memory is highly dependent on the size of the applications bytecode.

When four cores/threads used, the code from Listing 5.2 needs to be repeated four times. Note that in the actual code, we implemented this functionality as a C preprocessor macro.

Listing 5.2: Assigning bytecode to core zero

```
1  if(get_core_id() == 0)
2  {
3    extern int bytecode_len_0;
4    extern unsigned char bytecode_0[];
5    extern int const_desc_len_0;
6    extern unsigned char const_desc_0[];
7    extern int nb_scm_globs_0;
8    extern int scm_globs_0[];
9
10   bytecode_len = &bytecode_len_0;
11   bytecode = &bytecode_0[0];
12   const_desc_len = &const_desc_len_0;
13   const_desc = &const_desc_0[0];
14   nb_scm_globs = &nb_scm_globs_0;
15   scm_globs = &scm_globs_0[0];
16 }
```

5.7 XMOS Bit Scheme: primitives for IO

We extended the original Bit scheme with several XMOS specific primitives. Table 5.3 lists these primitives together with the bytecode instructions they are compiled into. We will first discuss the primitives that concern IO operations involving the physical pins on the XMOS chip. These are summarized in Table 5.4 together with their XC equivalent. As discussed in Section 3.2 ports can either be a pin or a group of up to 32 pins. We will refer to these ports using Scheme numbers. The numbers used are resource identifiers, these are the low level identification for a port. They can be found inside the header files used by the XMOS toolchain.

pin and pout immediately read from, respectively write to a port without performing checks.

peq and pne wait until the value on a port equals to (peq) or differs from certain value (pne). Once this has happened, the new value is returned. The waiting is performed by the hardware, which means that the thread will not have to check the value manually. However, there is no pos-

Scheme	XC equivalent	Description
(pout port value)	port <: value	Write to a port.
(pin port)	port :>int value	Read from a port.
(peq port value)	p when pinseq(v):>w	Wait until the value on the port equals to v, then write it to w.
(pne port value)	p when pinsneq(v):>w	Wait until the value on the port differs from v, then write it to w.
(pon port)	set_port_use_on(p)	Enables a port for usage.
(poff port)	set_port_use_on(p)	Disables a port.
(pconf_out port value)	configure_out_port (p, clk, v)	Configure a port as output, with the provide value as default.
(pconf_in port)	configure_in_port (p, clk)	Configure a port as input.

Table 5.4: Overview of IO primitives

sibility for the garbage collector to run while the thread is waiting in hardware.

pon and poff turn a port on or off. This needs to happen before the port is used or configured. Otherwise, the port remains in a high impedance state [16]. Note that this action is not required in XC. The XC compiler knows which ports are used, and enables them at compile time. When using ports from Scheme, the used ports are not know by the XMOS compiler. As they are "hidden" in the bytecode corresponding to the Scheme program.

pconf_in and pconf_out These primitives are used to configure a port either as input or output. Again, this is not required in XC.

Listing 5.3 displays a typical example of configuring a port. A variable `PORT_1A` is defined containing a number. This number is the low-level resource identifier of the port we want to use. Next, the port needs to be initialized using the primitive `pon`. Finally, the port is configured as input by calling `pconf_in`. Once the port has been configured, it is ready for use and can be read from using the `pin` primitive.

Writing to ports is very similar as illustrated in Listing 5.4. First a variable representing the port by its resource identifier is defined, which is then used to initialize the port. To configure a port as output, an extra parameter is required. This parameter is the default value put on the pin(s). Once the port has been configured, it can be written to using the `pout` primitive.

Listing 5.3: Reading from a port

```
1 (define PORT_1A 66048)
2 (pon PORT_1A)
3 (pconf_in PORT_1A)
4 (pin PORT_1A)
```

Listing 5.4: Writing to a port

```
1 (define PORT_CLOCKLED 525056)
2 (pon PORT_CLOCKLED)
3 (pconf_out PORT_CLOCKLED 0)
4 (pout PORT_CLOCKLED 15)
```

5.8 XMOS Bit Scheme: time-related primitives

In order to add the notion of time in applications, we added two primitives shown in Table 5.5. The primitive `timer` evaluates to the current time, which is represented using a number. This number is increased every clock tick. In order to suspend a thread for a certain period, the `after` primitive is used.

Listing 5.5 illustrates how these primitives are used together in order to suspend a thread for 5 seconds. It samples the current time. This value is used to calculate the appropriate delay based on the clock speed of 100 MHz. The resulting value is used as an argument for the `after` primitive, which suspends the thread.

Scheme	XC	Description
(timer)	*timer* :>int time	Return the current time.
(after *time*)	*timer* when timerafer (*time*) :>void	Suspend thread until *timer* has past *time*.

Table 5.5: time-related primitives

Scheme	XC	Description
(cin core)	channel :>value	Read data from a channel. Returns the received value.
(cout core value)	channel <: value	Write data to a channel.

Table 5.6: Communication primitives

Listing 5.5: Writing to a port

```
1 (define now (timer))
2 (define clock 100000000)
3 (define delay (* 5 clock))
4 (after delay)
5 (display "5 seconds later")
```

5.9 XMOS Bit Scheme: message passing primitives

XMOS threads communicate through message passing. Because there is one Scheme interpreter running on each core, we opted to use the core ID to identify the destination/origin of a communication operation. Each interpreter has three channels, one to each of the other interpreters (cfr Figure 5.3). Depending on the destination core, the interpreter selects the correct channel to complete the operation. Because channels are bidirectional, only one channel is needed between two cores.

There are two primitives that concern communication between interpreters as depicted in Table 5.6. Both these primitives are blocking. When sending data to a certain interpreter, that thread will block until the data is read from the channel by the receiving interpreter. The core argument is a number

between 0 and 3, identifying each core. Sending data from an interpreter to itself is not supported as communication is blocking.

Listing 5.6 illustrates how to use the communication primitives between two interpreters. The interpreter running on core zero sends and integer to core one. It will block until the data is received on core one. On that destination core, the thread waits until data arrives from core zero. When the data is received, a variable is defined with the received integer.

Listing 5.6: Communicating between threads

```
1 (par
2  (core CORE_0
3   (cout CORE_1 80))
4  (core CORE_1
5   (define input (cin CORE_0))))
```

Our current implementation is limited to primitive types such as integers and characters can be sent to another interpreter. These values are not modified during transmission and are represented as integers internally. Compound data structures such as lists are currently not supported by the XMOS Bit Scheme interpreter itself. However, we have created a proof-of-concept implementation in Scheme which supports sending trees and lists to other cores. In order to do so the trees are traversed in a depth first way, in order to send each item individually over the channel. On the receiving core, similar code will reconstruct the original data structure for the data it receives. It is clear that the time to send a data structure across a channel is proportional to its size.

5.10 XMOS Bit Scheme: handling multiple events at once

As discussed in Section 4.5, an XC thread has to use the XC select statement to check for multiple events at the same time. The aforementioned `pne`, `peq` and `cin` primitives cannot be used as these are blocking.

In XC, the select statement is compiled to assembler. This requires that all case statements need to be known at compile-time, which makes it impossible

to use the XC select statement to implement its Scheme equivalent. What is needed is what could be called a "dynamic select" statement, which is configurable at run-time. That way the interpreter can check for multiple inputs and perform the needed configuration at run-time. We identified two ways to implement a dynamic select statement. One is by implementing it directly in assembler. Another way is by programming it in Scheme. Both approaches have their advantages and disadvantages.

Assembler It is possible to program a dynamic select in assembler. The XC compiler supports inline assembler, meaning it is possible to embed the needed assembler instructions directly into XC source files. That way assembler code can be called using XC functions. We need to manually register the events we want to wait for as illustrated by Listing 5.7.

Listing 5.7: The select statement in assembler [18]

```
1  clre
2  setc res[r0], COND_EQ
3  setd res[r0], 1
4  eeu res[r0]
5  setv res[r0], case0
6  setc res[r1], COND_AFTER
7  setd res[r1], r2
8  setv res[r1], case1
9  eeu res[r1]
10 waiteu
```

First, the `clre` instruction clears all existing events for this thread. Then each event we want to listen for, needs to be registered. This is implemented using the `eeu` instruction. Next, a vector is registered which points to the code handling this event. This is accomplished using the `setv` opcode. After all events are set, the `waiteu` instruction will cause the thread to pause until one of the events fires [18]. This needs to be implemented at the lowest possible level, by directly using the correct instruction. This approach is hard and time-consuming to debug and test. But when implemented, it uses the built-in event mechanisms of the XMOS hardware. This approach will therefore yield the highest performance and/or the lowest power usage.

Scheme A totally different possibility is to implement the same functionality in Scheme. Only for checking the events, a call to an XC function is needed. This is the approach taken in our implementation. A downside of this approach is that it requires additional primitives in the Scheme interpreter. This approach still needs extra functions in the interpreter because all functions mentioned above regarding input, timing and communication are blocking. They can therefore not be used to do the checks, as this would block the interpreter, which is the opposite of the desired behaviour. We therefore added four extra primitives `select_pne`, `select_peq`, `select_after` and `select_cin`. These allow to do the "port not equal", "port equal", "time after" and "channel input" operations in a non-blocking way. Internally, they are embedded in an XC select statement. If the channel contains data waiting, this will be returned. If not, the default case will return a zero. Note that the default return value (zero) will not be confused with a zero in the interpreter, as the latter are identified with a bitflag.

```
1  int xc_select_cin(chanend c)
2  {
3    select
4    {
5      case c :> int i:
6        return i;
7      default:
8        return 0;
9    }
10 }
```

Clearly a high level implementation is not as hard as an assembler one. However, this approach in Scheme has some disadvantages. It is not possible to block until one of multiple events occurs. This is the behaviour of the select statement in XC, when it does not contain a "default case". Because it does not block, this implementation in Scheme cannot use all the hardware's features to maximize execution speed and minimize power consumption.

We decided to choose the high level Scheme implementation. The implementation in assembler would take much more effort, with very little added value for our case study.

To do this implementation, we modified the Bit Scheme compiler to recognize the `select` statement. The XC select statement has a lot of similarities

Listing 5.8: Scheme select

```
1  (select
2    ((select_pne buttons 15)
3     (lambda (buttonsvalue)
4       (display buttonsvalue)))
5    ((select_peq port1A 1)
6     (lambda (value)
7       (display value)))
8    ((select_cin 1)
9     display)
10   (else (display "default")))
```

with the C/C++ switch statement. We chose to let the `select` statement be syntactically similar to the `cond` primitive.

Listing 5.8 illustrates a usage example of this select statement. Each case statement contains a lambda as body. This function is executed when the test evaluates to true. Contrary to a Scheme conditional, tests can return a value, for example when reading data from a channel. This value is then passed as an argument to the function.

5.11 XMOS Bit Scheme: 32-bit integer support

Bit scheme was originally written for 16 bit microcontrollers, which causes some problems when using Bit Scheme on a 32-bit chip, like the XMOS XS1-G4. In particular, the integers caused some problems.

5.11.1 Representation of integers

Integers in the original interpreter uses only 16 bit of the 32 bits supported by the XMOS hardware. As shown in Figure 5.7, in the bytecode generated by the original Bit Scheme compiler, there are four bytecodes to indicate integers. The first two represent integers larger than -255 and smaller than 255. These can be represented using a single byte. The next two are used

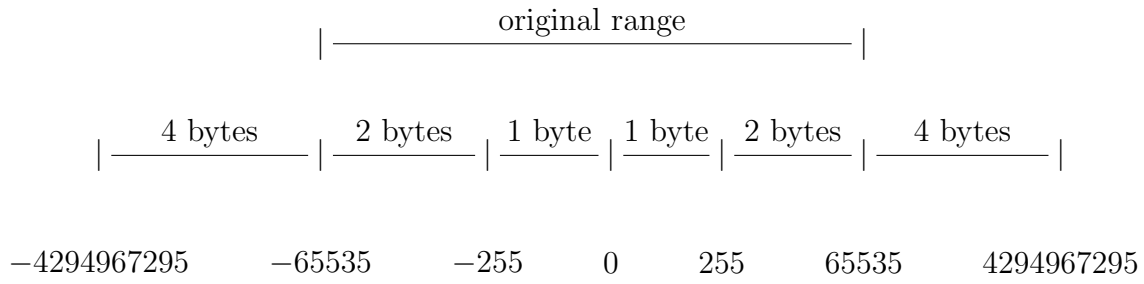


Figure 5.7: Extending integer range

to extend this range to 16 bit. This means a range starting from -65535 to 65535, which still doesn't use the XMOS hardware properly.

We added a third instruction which allows to use the full 32 bit range, as depicted in Figure 5.7. We chose not to extend the range of the integers by modifying the semantics of the existing bytecodes. This way, we can keep the bytecode small as smaller integers still only use the bytes needed to represent them. Moreover our approach limits the possibility of introducing bugs due to the extended range. The interpreter can retain its original assumptions about the bytecode representation of integers.

The Bit Scheme interpreter uses a one-bit flag to mark data stored in RAM as integers. This means that of the original 16 bits representing an integer, only 15 can effectively be used to store data. In our XMOS Bit Scheme interpreter this means that only 31 bits can be used. Consequentially, while the byte code allows representing the full range 32 bit integers, they cannot be stored entirely in RAM memory. This causes some problems described in the next section.

5.11.2 Using timers

Even when extending the range for representing integers to 32 bits, not all problems are solved. As there is the one-bit flag, only 31 bits can effectively be used. This is no problem for purely arithmetic operations. However, it poses problems when using the timers in XMOS as timers use the full 32-bit range. It is not possible to configure the maximum possible timer value. We circumvent this problem by using the current time to determine which value is the lost most significant bit. We assume that the desired time, is the one

closest to the current time and fill in the missing information appropriately. That way, 32-bit timers can be used in XMOS Bit Scheme.

5.11.3 Floats and unsigned integers

Bit Scheme does not support floating point numbers because its original hardware platform has no support for that. This might seem like a big shortcoming. However, it doesn't pose a problem, as all the IO used in the case study is digital. Floating points are only needed for analog IO. Therefore, the need for floating point support is limited in this case.

There is no support for unsigned integers in XMOS Bit Scheme too. These are heavily used by XC to read from and write to ports. For example when driving some LEDs, using unsigned integers for them makes the most sense. However, when using the full available 31-bit range, we will need to have a negative number — due to the two-complement — to drive the LED connected to the most significant bit. This neither is a big handicap of the interpreter.

5.12 Case study revisited: a high-level event-driven implementation in Scheme

In this section, we reimplement the case study from the previous chapter (cfr Section 4.6) in XMOS Bit Scheme. It is very similar to the previous case studies in Chapters 2 and 4 where we connect two board performing PWM via wireless over XBee. On these two boards we will perform PWM operations which can be altered by the user via the four buttons on the development board. The program starts four threads as shown in Listing 5.9. One thread contains the application logic and handles the button presses. Thread one and two handle the UART communication (RX and TX respectively). The third thread performs the pulse width modulation of the LEDs required for the core. The placement of the threads is slightly different from the case study in the previous chapter. Because of the hardware layout, the PWM thread needs to be on core zero. This is due to the `CLOCKLED_SELR` and `CLOCKLED_SELG` ports residing on core zero. However, because only one thread can run on each core, the internal LEDs cannot be used. Therefore, equivalent hardware is connected to core three, which circumvents this problem. Another possible solution was to merge the code of both threads into

one, more complex, thread. However this would defeat the purpose of the multi-threaded architecture.

Listing 5.9: Program structure

```
(par
 (core CORE_0
  ; Application logic and button handling
  (define (logic) ... )
  (logic))
 (core CORE_1
  ; UART RX
  (define (uartrx) ... )
  (uartrx))
 (core CORE_2
  ; UART TX
  (define (uarttx) ... )
  (uarttx))
 (core CORE_3
  ; LED PWM
  (define (pwm) ... )
  (pwm)))
```

Core zero contains the thread with the application logic and which handles the button presses. This code is displayed in Listing 5.10. First, the default PWM values are initialized, together with the delay needed after a button is pressed. Next, the hardware, more in particular, the port connected to the buttons is initialized. The last element to define is a function called `handle_press` which handles the button and modifies the PWM value appropriately.

Next is the main loop of this thread. It contains a select statement which either handles a button press or incoming data from core one (which contains the UART RX code). The of the former case is executed when the value on the `buttons` port is not 15. The value 15 originates from the four buttons which are connected with a pull up resistor (cfr Section 2.3.1). When the user pushes a button, the appropriate bit will be a logic zero, effectively changing the value on the port. When this happens, the `handle_press` function will update the PWM values accordingly. After this is finished, the local value is sent to the thread running on core three, where the code resides to shown it on the LEDs. Next, the remote value and the local value are combined into

a single byte and then fed to core two which will send it via serial port to the other board.

The above-mentioned `select` statement also handles data input from core one. The thread running on this core handles incoming serial data. When data arrives, the PWM values are updated. Finally the local value is used to update the shown color.

Listing 5.11 contains the code for reading a byte from the serial port. It based on the low level XC implementation discussed in Section 4.6.2. First the port used to send the data over, being port 1A, needs to be initialized (lines 2-4). The delay used during transmission is also calculated based on the baudrate of 1200.

Next is the main loop of this thread, which first waits for data to arrive from thread zero. When data arrives that needs to be sent, first a start bit is put on the line in the form of a zero. Then we add a delay depending on the baud rate. After this delay, each data bit is sent, starting with the least significant bit (lines 14-21). When all eight data bits are sent, the transfer is finished with a stop bit.

Reading data from the serial port obviously has a fair share of resemblances with the sending code. To read a data byte, we first wait until the start bit is put onto the RXD pin as displayed in Listing 5.12. After the start bit is received, the current time is sampled and saved. To ensure that the bit is sampled when its signal is stable, half of the bit time is added. Next, each bit is sampled with `bit_time` time in between. Because XMOS Bit Scheme does not support bit shift operations, a small calculation is performed that emulates a right bit shift. Each newly read bit is put at the position of the most significant bit and then shifted to the right using a division by two. This is illustrated by Table 5.7. As the least significant bit is sent first, all bits will be at the appropriate position after the entire data byte has been read. After all data bits are read, we wait for the stop bit and then send the data over the channel to core zero.

Listing 5.10: Application logic

```
1  (define (logic)
2    (define remote_pwm 7)
3    (define local_pwm 7)
4    (define button_time (/ 100000000 4))
5
6    (define   buttons 262912)
7    (pon      buttons)
8    (pconf_in buttons)
9
10   (define (handle_press buttons shift pwm_value)
11     (set! buttons (modulo (/ buttons (expt 2 shift) )
         4))
12       (cond
13         ((= buttons 3)
14           pwm_value)
15         ((= (modulo buttons 2) 1)
16           (if (< pwm_value 15) (+ pwm_value 1)
               pwm_value))
17         ((if (> pwm_value 0) (- pwm_value 1) pwm_value)
            )))
18
19   (let loop ()
20     (select
21       ((select_pne buttons 15)
22         (lambda (value)
23           (set! local_pwm  (handle_press value 0
               local_pwm))
24           (set! remote_pwm (handle_press value 2
               remote_pwm))
25           (cout 3 local_pwm)
26           (cout 2 (+ remote_pwm (* local_pwm 16)))
27           (peq buttons 15)
28           (after (+ (timer) button_time))))
29       ((select_cin 1)
30         (lambda (value)
31           (set! remote_pwm (modulo (/ value 16)16))
32           (set! local_pwm (modulo value 16))
33           (cout 3 local_pwm))))
34       (loop)))
```

Listing 5.11: Sending a byte over the UART

```
1  (define (uartrx)
2    (define TXD 66048) ; 1A
3    (pon TXD)
4    (pconf_out TXD 1)
5
6    (define bit_rate 1200)
7    (define bit_time (/ 100000000 bit_rate))
8
9    (let loop ((value (cin 0)))
10     (let ((time (timer)))
11       (pout TXD 0)
12       (set! time (+ time bit_time))
13       (after time)
14         (let dataloop ((pos 0))
15           (if (< pos 8)
16             (begin
17               (pout TXD value)
18               (set! value (/ value 2))
19               (set! time (+ time bit_time))
20               (after time)
21               (dataloop (+ pos 1)))))
22       (pout TXD 1)
23       (set! time (+ time bit_time))
24       (after time))
25     (loop (cin 0))))
```

0	b_0000	0000
1	b_1b_000	0000
2	$b_2b_1b_00$	0000
3	$b_3b_2b_1b_0$	0000
4	$b_4b_3b_2b_1$	b_0000
5	$b_5b_4b_3b_2$	b_1b_000
6	$b_6b_5b_4b_3$	$b_2b_1b_00$
7	$b_7b_6b_5b_4$	$b_3b_2b_1b_0$

Table 5.7: Reading in the serial bits

Listing 5.12: Getting a byte from the UART

```
1  (define (uarttx)
2    (define TXD 66048) ; 1A
3    (pon TXD)
4    (pconf_out TXD 1)
5
6    (define bit_rate 1200)
7    (define bit_time (/ 100000000 bit_rate))
8
9    (let loop ((value (cin 0)))
10       (let ((time (timer)))
11         (pout TXD 0)
12         (set! time (+ time bit_time))
13         (after time)
14           (let dataloop ((pos 0))
15             (if (< pos 8)
16               (begin
17                 (pout TXD value)
18                 (set! value (/ value 2))
19                 (set! time (+ time bit_time))
20                 (after time)
21                 (dataloop (+ pos 1)))))
22       (pout TXD 1)
23       (set! time (+ time bit_time))
24       (after time))
25     (loop (cin 0))))
```

Configuring the XBee modules

To use the XBee modules with XMOS Bit Scheme, they need to be reconfigured. The interpreter can't output data at a baudrate of 9600, which is the standard baudrate of the XBee modules. Because the interpreter is too slow to output the bits at that speed, the module is reconfigured to run at a speed 1200 bits per second.

To do so, the module needs to be in *configuration mode*. This can be achieved by sending `+++` to the module. Before doing so, there needs to a one second delay before and after which no data is sent to the module. This prevents data from being misinterpreted.

When the module is in configuration modus, it will reply with `OK`, followed by a carriage return. We can now send commands to reconfigure the module. This is done using AT commands. These consists of `AT`, followed by a two letter command describing the desired command. To set the baudrate, the command is `ATBD`. The XBee module supports various baud rates as shown in table 5.8.

In order to configure the needed 1200 baudrate, we send the command `ATBD0`, followed by a carriage return. To check that the settings are applied correctly, the command `ATBD` can be sent without an argument. This will return the current setting, which should be zero in this case.

To make the current settings permanent, we can write them to the non-volatile memory by issueing the `ATWR` command. After this, the *configuration mode* can be left using the `ATCN` command. If this is not done manually, the module will automatically leave the *configuration mode* after a few seconds.

Pulse Width Modulation

The PWM code runs on core three. Therefore, we cannot use the LEDs on the development board, as these are only accessible from core zero (cfr Figure 4.1). Instead pins `PORT_1A` and `PORT_1B` are used to perform the PWM by using some extra hardware. This hardware consists of a red-green LED in combination with a current limiting resistor. As depicted on Figure 5.8, the LED internally consists of two separate LEDs, one in each direction. When `PORT_1A` outputs a logical one (having voltage of 5 volt) and `PORT_1B` is at a logical zero (effectively being connected to ground), the red LED will light up. When the outputs are switched, it effectively changes the direction of the

Baudrate	AT
1200	0
2400	1
4800	2
9600	3
19200	4
38400	5
57600	6
115200	7

Table 5.8: Baudrates supported by the XBee module

current, and enables the green LED. Clearly by switching between these two states rapidly using pulse width modulation, we can create various shades.

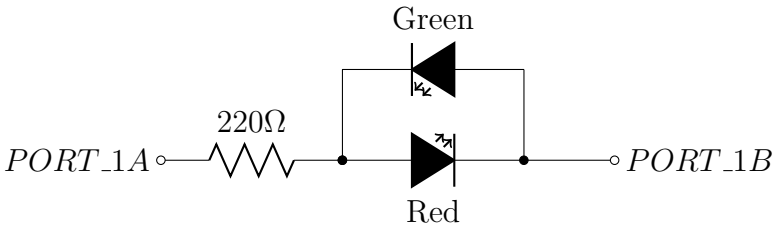


Figure 5.8: LED setup

This modulation is shown in Listing 5.13. On lines 2 to 8, the ports `PORT_1A` and `PORT_1B` are defined using their resource identifier, next they are initialized. After defining an initial PWM value on line 10, the main loop is entered. This loop contains a `select` statement which either updates the PWM value using incoming data from the application logic core zero or will perform the PWM. The latter PWM case, either enables the red or green led according to the time-interval defined by the `pwm` variable.

Listing 5.13: Pulse Width Modulation

```
1  (define (pwm)
2    (define redplus 66048) ; 1A
3    (pon redplus)
4    (pconf_out redplus 1)
5
6    (define greenplus 65536) ; 1B
7    (pon greenplus)
8    (pconf_out greenplus 0)
9
10   (define pwm 7)
11
12   (let loop ((current 15))
13     (select
14       ((select_cin 0)
15         (lambda (value) (set! pwm value)))
16       (else
17         (pout greenplus (if (< current pwm) 1 0)
18         (pout redplus (if (< current pwm) 0 1)))))
19     (loop (if (> current 0) (- current 1) 15))))
```

5.13 Discussion

The XMOS Bit Scheme implementation allows to write embedded software in a high-level language. This improves the clearness of the application even more over its XC equivalent. However there are some trade-offs. The execution speed of Scheme application is much lower than their XC counterparts. The UART code can serve as an example. The Scheme implementation works up to a speed of 1200 bps, while the native XC implementation can reach much higher speeds. But due to the real-time garbage collector in Bit Scheme, the performance is predictable. Consequentially, transmission will not fail due to unexpected garbage collection.

Bit Scheme also lacks typical bit level operations which are frequently used in embedded operations. These need to be converted to arithmetic operations which are less convenient. The added XMOS related primitives integrate very well with the existing Scheme primitives, therefore using the hardware features is very straightforward.

The XMOS compiler does not know which ports are accessed by the byte-code, consequentially the programmer needs to do that manually, which is inconvenient as it's not needed when writing plain XC applications. Also the lack of runtime error handling makes debugging XMOS Bit Scheme applications a tedious job.

The XMOS Bit Scheme implementation only supports four threads. In the case study two of the four available threads are already occupied by code to perform the serial communication. This could be circumvented by implementing serial communication as a separate threat in immediately XC and make its functionality available as a primitive in XMOS Bit Scheme. This would also allow serial communication at much higher speeds. Clearly, it would be possible to implement other communication protocols and/or abstraction layers for peripheral hardware.

5.14 Conclusion

In this chapter, we have shown how to port the Bit Scheme interpreter to the XMOS hardware and how to extend it to use the architecture specific features. This way, programmers can develop event-driven and multi-threaded embedded applications in Scheme. We have exported hardware functionality into XMOS Bit Scheme primitives. That way Scheme applications can immediately access the chip's hardware functionality. The programmer can work in a high-level language, therefore he or she can work without having to worry about memory management. The interpreter works reasonably fast while it fits inside the very limited available memory. We have also reimplemented the case study from the previous chapters. The implementation is easier to understand, but there is a significant speed penalty which became apparent when implementing the UART code. However, real-time nature of the garbage collector ensures that no unexpected delays happen due to the garbage collection process.

Chapter 6

Conclusion

Embedded software becomes increasingly important and is all around us in a variety of systems. As embedded software has to interact with the physical world, it differs significantly from PC or server application software [6]. Most microchips and the accompanying embedded software is based on the concept of interrupts. These asynchronous signals indicate to the CPU that its attention is needed. The CPU can then execute the appropriate handler and afterwards it continues executing. However as discussed in Chapter 2, this approach has several drawbacks.

The chip designed by the XMOS company contains an event-driven and multi-threaded architecture. In this architecture, no interrupts are used, consequentially it does not inherit the drawbacks of interrupt driven software. The architecture supports multiple threads in hardware. Concretely, each thread has its dedicated set of registers and gets a guaranteed amount of CPU cycles [16]. A thread can communicate with other threads through message passing and can subscribe to events. These events implement a similar functionality as interrupts. When a thread subscribes to an event, it is suspended until the event occurs. That way, other threads can run or, when few threads are running, power can be conserved. As most embedded software is inherently concurrent, the mapping of embedded software on multiple threads is more natural than on traditional desktop software. The concurrency is introduced due to the various interactions from the outside world that need to be handled in combination with the application logic.

6.1 Contributions

In this dissertation, we have ported a Scheme interpreter to a new platform. The interpreter of choice is "Bit Scheme". This bytecode based interpreter is very small, which is needed to fit in the tiny memory of the XMOS chips. The interpreter is entirely designed for a small memory footprint, however, it does not cut down on features. The bytecode interpreter is equipped with a real-time garbage collector. In this context, real-time means that the garbage collector will return in a predefined amount of time. This is especially useful in embedded software which usually contains time critical functions.

In addition to porting the Bit Scheme interpreter, we also extended Scheme with several primitives that allow using the XMOS specific hardware features. As a result, the XMOS hardware can now be programmed in Scheme. In order to support the concurrent programming on the four cores of the XMOS hardware, each having a separate memory, we opted for an architecture in which each core runs an independent bytecode interpreter. Therefore, during Scheme compilation, the program is split up in separate pieces of bytecode, which get downloaded to a specific core. This ensures that the small amount of available memory is used as good as possible. To allow the different interpreters to communicate, message passing primitives were added. While Bit Scheme is entirely optimized for size and not for speed, its execution is very reliable in terms of timing. This is due to the real-time garbage collector and the guarantees the XMOS hardware gives in terms of execution speed. To demonstrate the feasibility of programming embedded systems using event-driven and multi-threaded Scheme programs, we performed a case study for which the timing of the interactions with the physical world is critical. Concretely it involved UART communication and pulse width modulation, both are usually implemented in hardware.

6.2 Limitations and future work

Passing complex data structures across channels

We have made a proof-of-concept to send complex data structures such as trees and lists across channels. However, sending closures is currently not possible. In order to do so, the needed bytecode would need to be sent to the destination core. However the XMOS Bit Scheme compiler removes

unneeded functionality from each core. As a consequence, a closure sent to another core might call functionality which has been stripped from that core's bytecode during compile time.

Unsigned and floating point numbers

Support for unsigned and floating point numbers is missing in Bit Scheme itself. Especially floating point numbers are useful when dealing with analog input and output.

More thread flexibility

Our current implementation allows one thread per core. This is only a fraction of the available eight per core. However, when increasing the number of threads, one will encounter memory issues. Each thread needs a fixed chunk of memory to be allocated. In case we allow running more than one interpreter per core, the available memory will further need to split up.

When increasing the number of threads one will also need another model to allocate the available channels to communicate between threads. The current all-to-all model will run out of channels quickly, therefore the Bit Scheme compiler would need to detect which threads communicate. The channels could be allocated during the second compilation phase.

Optimize the interpreter

In future work, we intend to implement a select statement at assembler level (cfr Section 5.10). This would allow to better use the possibilities of the hardware than the current Scheme implementation. Doing so would both increase the execution performance and the power efficiency of applications written in Scheme.

Introduce parallelism in the interpreter

Clearly the current implementation of Bit Scheme is not the most efficient one, further optimizations could be made here. For example, performing the garbage collection in a separate thread.

Improve development cycle

The lack of runtime error checking allows Bit Scheme to be very small. While this saves memory, it is a big strain on the developer and is clearly very troublesome during debugging, as the programmer does not get useful information about software crashes. This could be worked around by creating an emulation environment which allows applications to be developed and debugged on a desktop. That way, the developer is provided with useful debugging information during development. At the same time, the runtime memory requirements are not increased by error checking code.

Create a library with all ports and their names in

This would simplify the programmers task, as it would no longer be needed to look up the correct integer identifying a certain resource. Preferably, this library could be (re)generated automatically, as the resource identifiers are different for each device.

Chapter 7

Samenvatting

Embedded software wordt steeds belangrijker. Digitale horloges, microgolfovens, auto's bevatten allemaal ingebedde software. In de meeste gevallen is een embedded systeem opgebouwd uit hardware en software die ontwikkeld zijn voor een specifieke taak. Meer dan 98 % van de processoren [25] worden tegenwoordig gebruikt in ingebedde systemen. Ingebedde software verschilt echter significant van software die ontwikkeld is voor PC of voor servers. Naast applicatielogica moet ingebedde software ook interacties met de fysieke wereld afhandelen. Deze interacties zijn ondermeer sensors uitlezen, een motor of een licht aansturen, communiceren met andere systemen, enzovoort. Bepaalde protocollen en hardware die door de chip worden aangestuurd vereisen een strikte timing. In dit geval dient het systeem te reageren binnen een bepaald tijdsinterval. Indien een ingebed systeem meerdere van deze tijdskritische interacties concurrent dient af te handelen, worden de zaken nog complexer.

De meeste microchips en bijhorende ingebedde software zijn gebaseerd op interrupts. Een interrupt is een asynchroon signaal dat aan de CPU signalleert dat zijn aandacht vereist is. In dat geval zal de CPU stoppen met de taak die momenteel wordt uitgevoerd en de geassocieerde interrupt handler uitvoeren. Nadien zal de CPU verder gaan met het uitvoeren van de taak die werd onderbroken toen de interrupt zich voordeed. Deze interrupt gebaseerde aanpak is wijdverspreid en veelvuldig gebruikt om een chip snel te laten reageren op verschillende signalen van buitenaf. Nochtans zijn er verschillende nadelen aan verbonden aan de interrupt gebaseerde aanpak [23][22]. We zullen deze nadelen bespreken. Hierna behandelen we eventgebaseerde architecturen die een compleet andere aanpakken hebben naar

ingebedded systemen en software toe. Elke aanpak zal besproken worden door middel van een representatieve case study.

7.1 Interrupt gebaseerde ingebedde systemen

Op chips worden interrupts vaak gebruikt om signalen van buitenaf af te handelen. In interrupt gebaseerde systemen zullen deze interrupts de applicatiecode stoppen en wordt de inhoud van de verschillende registers door de chip op de stack bewaard. Na het uitvoeren van de passende interrupt handler, worden deze registers terug gevuld met hun oorspronkelijke waarde om de uitvoer van de applicatie code verder te kunnen zetten. Indien we interrupt gebaseerde systemen vergelijken met systemen die gebruik maken van polling, dan zullen de eerstgenoemde een vermindering van de vertraging en de overhead kunnen bewerkstelligen. Systemen die polling gebruiken dienen voortdurend een bepaalde conditie te controleren. Indien deze controle niet frequent genoeg wordt uitgevoerd kan de vertraging tussen het event en het afhandelen ervan problematisch worden. Echter, indien veelvuldig gepolld wordt, kan dit een significante overhead met zich meebrengen. Interruptgebaseerde software kan ook gebruikt worden om het energieverbruik van het ingebedde systeem te verlagen. In dat geval kan de CPU in een energiebesparende slaapstand gebracht worden tot een interrupt zich voordoet. In minder krachtige chips, wordt specifieke hardware vaak gebruikt om bepaalde tijds- en rekenintensieve IO taken uit te voeren zoals bijvoorbeeld seriële communicatie. Interrupts worden dan gebruikt voor de synchronisatie tussen deze specifieke hardware en de chip die de applicatiecode uitvoert. Deze hardware kan bijvoorbeeld signaleren dat er data ontvangen is van de seriële poort.

Signalen van buitenaf kunnen op elk moment voorkomen, dus geldt hetzelfde voor interrupts. Dit kan een bron zijn van verschillende moeilijk opspoorbare problemen aangezien ze enkel voorkomen onder specifieke en zeldzame condities. Ondermeer kan er een stack overflow voorkomen wanneer er te veel interrupts op hetzelfde moment aankomen in de chip. Dit veroorzaakt excessief gebruik van de stack door de verschillende interrupt handlers. Indien er een uitzonderlijk hoog aantal interrupts voorkomt, kan de hoofdapplicatie ook toegang tot CPU-tijd worden ontzegd. Bovendien veroorzaakt iedere interrupt een onderbreking van de applicatie, wat het voldoen aan real-time voorwaarden kan bemoeilijken. Bestaande methodes om deze problemen te voorkomen proberen meestal empirisch de benodigde sys-

teembronnen te bepalen die nodig zijn om alle mogelijke combinaties van interrupts aan te kunnen. Deze methodes zijn echter niet perfect en kunnen de zaken complex maken. In deze verhandeling zullen we de bovenstaande problemen illustreren door middel van een case study die de hierboven aangehaalde problemen weergeeft. Concreet zullen we een case study maken die de problemen illustreert verbonden met het gebruik van polling en interrupts in ingebedde software. In deze case studies implementeren we een applicatie die pulse width modulatie (PWM) uitvoert op een LED. Twee van deze apparaten zullen draadloos verbonden worden door middel van een XBee module, waardoor ze hun PWM waarde kunnen synchroniseren.

7.2 Event gebaseerde ingebedde systemen

Door de steeds krachtiger wordende chips is het mogelijk geworden om software te gebruiken voor taken die voordien werden geïmplementeerd in hardware zoals seriële communicatie. Deze aanpak wordt aangeprezen door het bedrijf XMOS. Hun chips bevatten een event gebaseerde architectuur. Een thread, die direct in hardware is geïmplementeerd, zal een event onderschrijven en de bijhorende berekeningen uitvoeren als dit event zich voordoet. Events kunnen voorkomen door wijzigingen in timers, communicatie of gerelateerd zijn met in- en uitvoer operaties. Threads hebben geen gedeeld geheugen, maar kunnen communiceren door middel van het uitwisselen van berichten. Indien een thread een event wil afhandelen zal het na het onderschrijven ervan zichzelf pauzeren tot het event in kwestie zich voordoet. Door zichzelf te pauzeren geeft een thread andere threads de kans om uitgevoerd te worden. Indien weinig threads worden uitgevoerd, wordt het energieverbruik verminderd. In de XMOS architectuur heeft elke thread een eigen set van registers en krijgt elke thread een gegarandeerde hoeveelheid CPU-tijd [16]. Dit betekent dat tijdskritische voorwaarden steeds gehaald zullen worden, ongeacht de events die door andere threads worden behandeld.

In traditionele desktop software is het vaak nodig om een applicatie op te splitsen in delen die concurrent kunnen draaien zodoende de uitvoer in meerdere threads mogelijk te maken. Echter, dit is niet het geval voor ingebedde software. De meeste ingebedde software is immers inherent concurrent aangezien het applicatiecode bevat naast code die de interacties met de buitenwereld behandeld. Dit resulteert in relatief natuurlijke verdeling van ingebedded software over verschillende threads, zoals voor applicaties op de XMOS architectuur aangewezen is.

Op een chip met slechts één core is het mogelijk om parallelle uitvoer van meerdere threads te emuleren. Dit is bijvoorbeeld geïmplementeerd in occam-π dat toelaat multi-threaded programma's te ontwikkelen op een gelijkaardige manier als voor XMOS chips. Echter, applicaties ontwikkeld in occam-π worden uitgevoerd op interrupt gebaseerde platformen. Dit kan door bijvoorbeeld een scheduler te implementeren die elke thread voorziet van een bepaalde hoeveelheid CPU-tijd. Echter, deze aanpak kan geen garanties bieden in verband met tijdskritische voorwaarden en uitvoeringssnelheid, iets wat wel het geval is voor de XMOS architectuur. Een thread die gescheduled is kan immers nog steeds onderbroken worden door events van buitenaf. Dit maakt het onmogelijk om met zekerheid te bepalen wanneer een berekening zal beëindigd zijn. Aangezien threads interleaved worden uitgevoerd kan het bovendien moeilijk zijn om real-time voorwaarden na te komen.

Een andere mogelijke aanpak is deze van het besturingssyteem TinyOS [11]. Dit besturingssysteem laat het schedulen van taken (tasks) toe.

Tenslotte zullen we de eerder vermeldde case study herzien om aan te tonen welke problemen door de XMOS architectuur worden opgelost en om de event-gebaseerde architectuur te vergelijken met een interrupt-gebaseerde architectuur.

7.3 High-level event gebaseerd programmeren in Scheme

Momenteel kunnen XMOS chips enkel worden geprogrammeerd in een low-level programmeertaal afgeleid van C. In deze verhandeling zullen we daarom onderzoeken of de XMOS architectuur ook kan geprogrammeerd worden door middel van de high-level programmeertaal Scheme en of het gebruik van Scheme op deze architectuur de taak van de ontwikkelaar nog meer vereenvoudigt.

We hebben de bytecode gebaseerde interpreter Bit Scheme geport naar het XMOS platform. Deze interpreter is zeer klein en past dus in het geheugen op de chip, terwijl er genoeg plaats over is voor de bytecode van de Scheme applicatie en het benodigde runtime geheugen van de applicatie. Bit Scheme heeft met een bijbehorende compiler die Scheme broncode kan vertalen naar bytecode. De bytecode interpreter bevat bovendien een real-time garbage collector. Dit is een belangrijk voordeel in het domein van ingebedde systemen.

In deze context betekent real-time dat de garbage collector zijn taak zeker zal afronden binnen een vast tijdsinterval [5]. Dit is uitermate handig als er aan tijdskritische voorwaarden moet worden voldaan. Naast het porten van de Bit Scheme interpreter naar het XMOS platform, hebben we ook de interpreter en compiler uitgebreid met specifieke mogelijkheden van de XMOS hardware te kunnen gebruiken. Om multi-core ingebedde systemen te ondersteunen, voeren we vier Scheme interpreters uit in parallel. We hebben ook nieuwe primitieven toegevoegd aan deze Scheme interpreter zodoende dat de interpreters kunnen communiceren door middel van het uitwisselen van berichten. De Bit Scheme interpreter is ook uitgebreid met XMOS specifieke IO abstracties. Tenslotte illustreren we de voordelen van deze high-level aanpak in een case study.

Bibliography

[1] Alison Carling. *Parallel Processing - Occam and the Transputer*. Sigma Press, Water Lane, Wilmslow, England, 1988.

[2] William D. Clinger and Jonathan Rees. The revised^4 report on the algorithmic language scheme. http://www.cs.indiana.edu/scheme-repository/R4RS/r4rs_toc.html, November 1991.

[3] Atmel Corporation. Atmel atmega48/88/168 avr microcontroller datasheet. http://www.atmel.com/dyn/-resources/prod_documents/doc2545.pdf, 2009.

[4] Danny Dubé and Marc Feeley. Picbit: A scheme system for the pic microcontroller. In *Proceedings of the Fourth Workshop on Scheme and Functional Programming*, pages 7–15, Boston, Massachusetts, USA, November 2003.

[5] Danny Dubé and Feeley Marc. Bit: A very compact scheme system for microcontrollers. *Higher-Order and Symbolic Computation*, 18(3-4):271–298, 2005.

[6] Lee Edward A. Embedded software. *Advances in Computers*, 56:56–97, Augustus 2002.

[7] Yoshikatsu Fujitax and LittleWing Company Limited. Ypsilon — the implementation of r6rs scheme programming language for real-time applications. http://code.google.com/p/ypsilon/, December 2008.

[8] Guillaume Germain, Marc Feeley, and Stefan Monnier. Concurrency oriented programming in termite scheme. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, Portland, Oregon, USA, September 2006.

[9] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1985.

[10] Digi International. Xbee and xbee-pro zigbee datasheet. http://www.digi.com/products/wireless/zigbee-mesh/xbee-zb-module.jsp#docs, 2010.

[11] Philip Levis and David Gay. *TinyOS Programming*. Cambridge University Press, The Edinburgh Building, Cambridge, 2009.

[12] XMOS Ltd. Xc-1a hardware manual. http://www.xmos.com/published/xc-1a-hardware-manual?ver=xc1ahw.pdf, 2009.

[13] XMOS Ltd. Xk-1 hardware manual. http://www.xmos.com/published/xk-1-hardware-manual?ver=xk1hw.pdf, 2009.

[14] XMOS Ltd. Xk-xmp-64 hardware manual. http://www.xmos.com/published/xmp-64-hardware-manual?ver=xk-xmp-64-hardware.pdf, 2009.

[15] XMOS Ltd. Xmos technology whitepaper. http://www.xmos.com/published/xmos-technology-whitepaper?ver=xmos-technology-whitepaper.pdf, 2010.

[16] David May and XMOS Ltd. The xmos xs1 architecture. http://www.xmos.com/published/xmos-xs1-architecture-0?ver=xs1_en.pdf, 2009.

[17] Atsushi Moriwaki and Akira Kida. Minischeme version 0.85k4 sourcecode. ftp://ftp.cs.indiana.edu/pub/scheme-repository/imp/minischeme.tar.gz, 1994.

[18] Richard Osborne. Targeting xcore resources from llvm. http://llvm.org/devmtg/2009-10/Osborne_TargetingXCoreResources.pdf, 2009.

[19] MiniScheme Project. Minischeme sourcecode. http://sourceforge.net/projects/minischeme/, 2010.

[20] The Arduino project. Arduino xbee shield schematics. http://www.arduino.cc/en/uploads/Main/XbeeShieldSchematic.pdf, 2007.

[21] Racket and PLT Scheme. Reference guide, chapter 10: Concurrency. http://docs.racket-lang.org/reference/concurrency.html?q=thread, 2010.

[22] John Regehr. Random testing of interrupt-driven software. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 290–298, Jersey City, New Jersey, USA, September 2005.

[23] John Regehr. Safe and structured use of interrupts in real-time and embedded software. In *Handbook of Real-Time and Embedded Systems*, chapter 16. Chapman & Hall/CRC, 2007.

[24] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(4):751–778, November 2005.

[25] Erwin Schoitsch. Embedded systems - introduction. *European Research Consortium for Informatics and Mathematics News*, (52):10–11, January 2003.

[26] Vincent St-Amour and Marc Feeley. Picobit: A compact scheme system for microcontrollers. In *Implementation and Application of Functional Languages*, pages 1–11, Seton Hall University, South Orange, New Jersey, USA, September 2009.