COMS20001 lab. worksheet: week #13

- Both the hardware and software in MVB-2.11 is managed by the IT Services Zone E team. If you encounter a problem (e.g., a workstation that fails to boot, an error when you try to use some software, or you just cannot log into your account), they can help: either talk to them in room MVB-3.41, submit a service request online via

  `http://servicedesk.bristol.ac.uk`

  or talk to the dedicated CS Teaching Technologist, Richard Grafton, in room MVB-2.07.
- We intend this worksheet to be attempted, at least partially, in the associated lab. session. Your attendance is important, since this session represents a central form of feedback and help for COMS20001. Perhaps more so than in units from earlier years, *you* need to actively ask questions of and seek help from either the lectures and/or lab. demonstrators present: passively expecting them to provide solutions is less ideal.
- The questions are roughly classified as either L (for coursework related questions that should be completed in the lab. session), or A (for additional questions that are entirely optional). Keep in mind that we only *expect* you to complete the first class of questions: the additional content has been provided *purely* for your benefit and/or interest, so there is no problem with nor penalty for totally ignoring it (since it is not directly assessed).

Before you start work, download and unarchive[a] the file

`http://www.ole.bris.ac.uk/bbcswebdav/courses/COMS20001_2015/csdsp/os/sheet/lab/lab-1_q.tar.gz`

somewhere secure[b] in your file system: it is intended to act as a starting point for your own work, and will be referred to in what follows.

[a]Execute the command `tar xvfz lab-1_q.tar.gz` from a BASH shell (e.g., in a terminal window), or use the archive manager GUI (available by using the menu `Applications`→`Accessories`→`Archive Manager` or directly executing `file-roller`) if you prefer.

[b]For example, the `Private` sub-directory in your home directory.

Q1[L]. This question introduces the emulated development platform used by the coursework assignment, and concludes with a set of challenges designed to help you explore it experimentally. The experience gained by your carefully working through what is, admittedly, a detailed[1] worksheet is crucial: doing so means you can rapidly engage with the intellectual vs. engineering challenges in the coursework assignment.

Q1–§1 The PB-A8 target platform

As in the lecture(s) we focus on a specific target platform, namely the RealView Platform Baseboard for Cortex-A8 [4] (which will be referred to as PB-A8 for short). The PB-A8 is an *entire* computer system, so the complexity hinted at in [4, Figure 3-1], for example, should come as no surprise. However, do not let this put you off: we are interested in a (very) limited sub-set of the components illustrated, namely

- the ARMv7-A [6], Cortex-A8 [3] processor core (labelled the "test chip" in [4]),
- the Universal Asynchronous Receiver/Transmitter (UARTs) whose ARM part number is PL011 [2],
- the timers whose ARM part number is SP804 [1]

plus a specific implementation of the ARM Generic Interrupt Controller (GIC) [7] design used on this platform.

In the first half of the unit, you were challenged to develop software using an XMOS-based development board as the target platform. Contrast this board with Figure 1, which illustrates a physical PB-A8: it should seem (and is) more cumbersome to use due to the size and extra equipment needed to power and develop software for it. In an effort to a) make development less intimidating, and b) address the issues of scale when used in the context of the unit (i.e., where there are > 100 students), we shift approach a little in this, second half: rather than a physical PB-A8, we use QEMU[2] to *emulate* the same target platform with development carried out using (and the emulator executed on) a standard Linux-based lab. workstation.

[1]In an attempt to manage the level of detail, extended explanation of some topics is deferred to Appendix A and Appendix B. Having an Appendix in a worksheet might seem, and possibly is ridiculous, but the goal is to offer help to those who need it *without* also cluttering the worksheet itself too much. Hopefully this seems a reasonable compromise.

[2]`http://www.qemu.org`

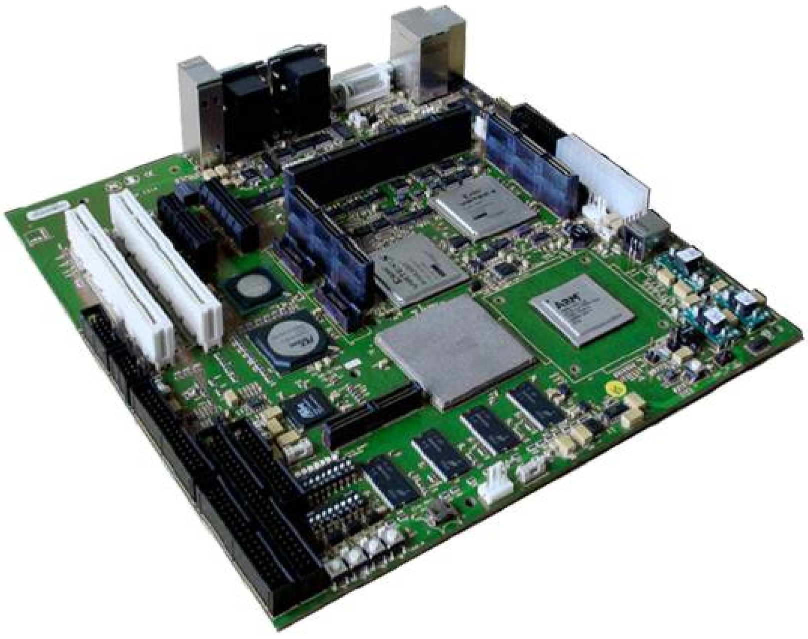


Figure 1: *A PB-A8 target platform.*

Q1–§2 The “hello world” kernel image

As suggested by the introduction, this worksheet is intended as a guided exploration of the development platform: it intentionally does *not* require you to develop any software for it! Essentially, the goal is that you understand an example provided and how it is executed by the (emulated) platform under control of `gdb`; at least some elements of this are likely to be new to everyone.

The example is a “hello world” kernel image. Note that although we term it a *kernel* image, it clearly it is *not* an operating system kernel! Even so, there are a few reasons for using this term: a) it is the terminology QEMU uses for the image (or program) provided for it to execute, but, perhaps most importantly, c) it stresses the fact execution will be on bare-metal, and therefore in a privileged processor mode akin to a kernel.

Q1–§2.1 Explore the archive content

The first step is to explore the example. Figure 2 shows the content and structure of the archived material provided, which can be viewed as two parts:

a some source code organised in three directories, namely

- `device`, which supports[3] access to pertinent devices on the PB-A8 using various definitions to model the memory map,
- `kernel`, which contains kernel mode related source code,
- `user` which contains user mode related source code,

plus the linker script `image.ld`, which are combined to produce a kernel image QEMU can execute, plus

b a build system for the source code, namely a `Makefile` with six targets:

- `build` uses `gcc` et al. to compile and link all source code files into a kernel image ready for use,
- `launch-qemu` launches an instance of `qemu-system-arm`, which loads the (pre-)compiled kernel image and waits for a remote debugger to connect,
- `launch-gdb` launches an instance of `gdb`, connecting to the waiting instance of `qemu-system-arm` st. it then controls execution,
- `kill-qemu` terminates any/all instances of `qemu-system-arm`,
- `kill-gdb` terminates any/all instances of `gdb`, and

[3]This source code is provided as a layer of abstraction, in an attempt to minimises the amount of low-level detail you are exposed to. You *could* attempt to understand and even extend it to suit your requirements, but equally there is no problem with simply using it as a form of “platform API” and ignoring how it works.

- `clean` removes any compiled material (e.g., any intermediate object files, plus the kernel image).

It is crucial to keep in mind that the `Makefile` is written *assuming* the above structure, and launches QEMU and components in the development tool-chain (e.g., `gcc`) via hard-coded paths. *If* you want to use `Makefile` as is for *your* work, which is by no means a requirement, you need to retain the same structure.

Q1–§2.2 Understand the archive content

In an attempt to understand what the source code itself *does*, the natural next step is to examine the archive content in some detail. In this example there are only a few files of interest, each of which is explained line-by-line in what follows.

`image.ld` Figure 4 illustrates the linker script `image.ld`. It controls how `ld` produces the kernel image from object files, which, in turn, stem from compilation of the source code files; the resulting layout in memory is illustrated by Figure 3.

- Line # 3 states the kernel image should be loaded starting at address $70010000_{(16)}$, matching what QEMU expects: the reason for this choice is outlined in more detail below.
- Lines # 3 to # 9 place the text, data and bss segments of *all* object files in a subsequent region of addresses.
- Line #11 aligns the current address to an 8-byte boundary, matching what the AAPCS function calling convention expects.
- Lines #13 and #14 assign the symbol `tos_svc` a value equal to the current address, having incremented it by 4KiB. This is intended to allocate a 4KiB region for use as a stack when the processor is in SVC mode: note that the acronym Top of Stack (ToS) is used throughout.

`interrupt.[sh]` Figure 5 and Figure 7 illustrate the header file `interrupt.h` and source code `interrupt.s`. The former is uninteresting, in the sense it is empty! The idea is that it will (in some later worksheets) declare a function prototype for any public assembly language function in `interrupt.s`. By doing so, `kernel.h` can then include the header file and therefore `kernel.c` can "see" and hence invoke said functions. The latter implements a function `handler_rst`:

- Line #1 moves the literal $D3_{(16)}$ into the CPSR register st. $\mathsf{CPSR}[M] = 10011_{(2)}$, $\mathsf{CPSR}[F] = 1_{(2)}$, and $\mathsf{CPSR}[I] = 1_{(2)}$. That is, the processor is switched into SVC mode with both FIQ and IRQ interrupts disabled.
- Given the processor is now in SVC mode, remember that subsequent references to `sp` resolve to the banked version of that register. Therefore, line #2 initialises the SVC mode stack by setting the stack pointer to where the linker script specified the SVC mode ToS should be, i.e., to `tos_svc`.
- Line #4 invokes the C function `kernel_handler_rst` defined in `kernel.c`.
- If or when `kernel_handler_rst` returns, there is nowhere sane for `handler_rst` return to: this is reflected by the instruction after invocation of `kernel_handler_rst`, i.e., line #5, realising an infinite loop (the `b` instruction will branch to itself) to approximate halting the processor.

When QEMU is executed, it always copies the kernel image into memory at address $70010000_{(16)}$ and transfers control to it via a short bootloader[4] placed at address $7000000_{(16)}$ (which is where execution begins). A short explanation of why is that it stems from launching QEMU using the `-kernel` option: this is a) useful in the sense that it automates various tasks, but b) fairly special-purpose in the sense it is intended for use with a Linux kernel image. For a longer, more concrete explanation you could read through

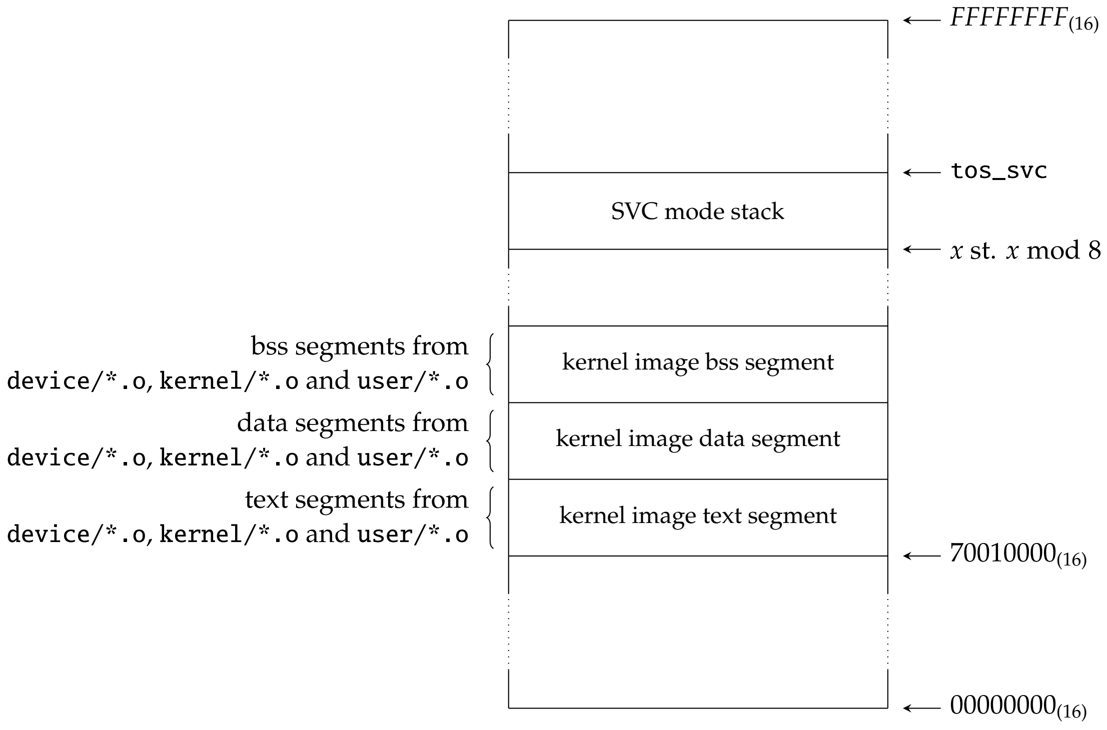
`http://www.github.com/qemu/qemu/blob/master/hw/arm/realview.c`

e.g., working backwards from line #366! Either way, this all means `handler_rst` will be invoked because it was placed at address $70010000_{(16)}$ by the linker script; `kernel_handler_rst` is then invoked once the stack pointer for SVC mode is initialised. Keep in mind that after `kernel_handler_rst` is invoked, the processor is in SVC mode; unlike functions you might normal write, it therefore a) has *total* control of the PB-A8, but b) has no support, e.g., in the form of system calls it might otherwise make into a kernel.

[4] `http://wiki.osdev.org/Bootloader`

```
${ARCHIVE}/lab-1_q
         ├── device
         │     ├── GIC.h
         │     ├── GIC.c
         │     ├── PL011.h
         │     ├── PL011.c
         │     ├── SP804.h
         │     └── SP804.c
         ├── kernel
         │     ├── interrupt.h
         │     ├── interrupt.s
         │     ├── kernel.h
         │     └── kernel.c
         ├── user
         ├── image.ld
         └── Makefile
```

Figure 2: *A diagrammatic description of the material in* `lab-1_q.tar.gz`*.*



FFFFFFFF(16)
tos_svc
SVC mode stack
x st. x mod 8 ≡ 0
bss segments from device/*.o, kernel/*.o and user/*.o
kernel image bss segment
data segments from device/*.o, kernel/*.o and user/*.o
kernel image data segment
text segments from device/*.o, kernel/*.o and user/*.o
kernel image text segment
70010000(16)
00000000(16)

Figure 3: *A diagrammatic description of the memory layout realised by* `image.ld`*.*

`kernel.[ch]` Figure 6 and Figure 8 illustrate the header file `kernel.h` and source code `kernel.c`. The former is uninteresting, bar the fact it includes headers st. each device on the PB-A8 can be accessed via memory-mapped I/O. The latter implements a function `kernel_handler_rst`:

- Lines #7 to #9 realise an inner `for` loop which iterates through each `i`-th character in the string `x`: each one is transmitted via the `PL011_t` instance `UART0` by invoking `PL011_putc`.
- Lines #6 to #10 realise an infinite outer `while` loop, each iteration of which thus transmits `x`, i.e., "hello world", via `UART0` per the above; this obviously implies that the function never returns.

The `launch-qemu` target in `Makefile` associates an emulated UART with the standard streams (i.e., `stdin` and `stdout`) of the resulting QEMU process. Put simply, this allows software executing on the emulated target platform to interact with the development platform. For example, transmitting a character via the `PL011_t` instance `UART0` results in writing it via `stdout` and hence the emulator terminal.

Q1–§2.3 Experiment with the archive content

Now, finally, you are ready to actually do something! Each step in what follows relates to executing the example kernel image under control of `gdb`, and in doing so get some valuable hands-on experience. Start by launching

```
SECTIONS {
  /* assign address (per  QEMU)  */
  .        =     0x70010000;
  /* place text segment(s)       */
  .text : { kernel/interrupt.o(.text) *(.text .rodata) }
  /* place data segment(s)       */
  .data : {                         *(.data        ) }
  /* place bss  segment(s)       */
  .bss  : {                         *(.bss         ) }
  /* align  address (per AAPCS)  */
  .        = ALIGN(8);
  /* allocate stack for svc mode */
  .        = . + 0x00001000;
  tos_svc = .;
}
```

Figure 4: *image.ld*

```
#ifndef __INTERRUPT_H
#define __INTERRUPT_H

#endif
```

Figure 5: *kernel/interrupt.h*

```
#ifndef __KERNEL_H
#define __KERNEL_H

#include <stddef.h>
#include <stdint.h>

#include   "GIC.h"
#include "PL011.h"
#include "SP804.h"

#include "interrupt.h"

#endif
```

Figure 6: *kernel/kernel.h*

```
handler_rst: msr   cpsr, #0xD3              @ enter SVC mode with no interrupts
             ldr   sp, =tos_svc             @ initialise SVC mode stack

             bl    kernel_handler_rst       @ invoke C function
             b     .                        @ halt
```

Figure 7: *kernel/interrupt.s*

```
#include "kernel.h"

void kernel_handler_rst() {
  char* x = "hello world\n";

  while( 1 ) {
    for( int i = 0; i < 12; i++ ) {
      PL011_putc( UART0, x[ i ] );
    }
  }
}
```

Figure 8: *kernel/kernel.c*

three terminals which will be referred to as the

- development terminal (this is where you edit source code, e.g., using `emacs`),
- debugging terminal (this is where you execute `gdb`), and
- emulation terminal (this is where you execute QEMU, i.e., `qemu-system-arm`)

respectively.

Challenge #1: build the image Build the kernel image by issuing the command

```
make build
```

in the development terminal: you should find that `image.bin` and `image.elf` have been produced as a result.

Challenge #2: execute the image

a First launch QEMU, by issuing the command

```
make launch-qemu
```

in the emulation terminal. QEMU has been instructed to wait for a connection from a debugger, so next issue the command

```
make launch-gdb
```

in the debugging terminal. A `gdb` prompt replaces the shell prompt, indicating that `gdb` is ready to accept commands and hence control QEMU.

b At this point you are ready to have QEMU execute the kernel image, so can issue the command

```
continue
```

in the debugging terminal: `gdb` resumes execution from whatever address the program counter holds (e.g., where execution was started initially, or was last stopped). You *should* see "hello world" written (continuously) to the emulation terminal, matching what we expect from having examined the source code.

c Once you are fed up, the final step is to instruct `gdb` to stop execution. However, there is currently no `gdb` prompt shown: QEMU has not yet returned control to `gdb` after you last instructed it to continue execution. So you need to forcibly interrupt execution by pressing (and holding) Ctrl-C in the debugging terminal until a `gdb` prompt is again shown. Now issue the command

```
quit
```

in the debugging terminal so `gdb` terminates, and you get a shell prompt back. Finally, force termination of the QEMU instance by issuing the command

```
make kill-qemu
```

again in the debugging terminal *or* pressing (and holding) Ctrl-C in the emulation terminal.

Challenge #3: execute the image under more control

a Launch QEMU and `gdb` by repeating associated steps from above.

b Rather than continue indefinitely, as above, `gdb` also enables you to continue execution until a specific breakpoint (an instruction or statement) is reached *or* a condition is met. To see this in action, issue the commands

```
break kernel_handler_rst
```

then

```
continue
```

in the debugging terminal. The first instructs `gdb` to stop execution once it reaches `kernel_handler_rst`, and the second resumes execution as before. However, rather than again having to interrupt execution to get the `gdb` prompt back, this time `gdb` offers the prompt: the breakpoint was reached, so execution stops. Note that you can have more than one breakpoint active at any one time, and that

```
info breakpoints
```

will list them for you; deleting a breakpoint from the list can be accomplished with

```
delete breakpoints 1
```

where the numeric argument identifies said breakpoint (in this case #1, corresponding to that created above).

c `gdb` enables you to single-step execution, meaning it it offers the prompt after executing either the next instruction or statement: this is accomplished via the `stepi` and `step` commands respectively. Try this out: execution was stopped at `kernel_handler_rst`, so issue the command

```
step
```

in the debugging terminal to single-step one statement further. You can repeat the previous command simply by issuing an empty command, i.e., by pressing return, so do this say 10 times. You should see execution progress through `PL011_putc`, writing the first few characters of "hello world". Note that the command

```
step 10
```

will single-step though the next 10 statements in one go if you prefer, with an analogous syntax for `stepi`.

d Terminate QEMU and `gdb` by repeating associated steps from above.

Challenge #4: execute the image displaying more information

a Launch QEMU and `gdb` by repeating associated steps from above.

b Using a breakpoint as above, control execution st. it stops once the `kernel_handler_rst` function is invoked.

c `gdb` supports various mechanisms for what could be generically described as inspecting the state of execution.

- The `x` (or "examine") command is used to show the content of memory starting at a given address: various formats can be used, e.g., interpreting said content as a sequence of 8-bit decimal bytes, or 32-bit hexadecimal words. Try this out: issue the command

  ```
  x/16i 0x70010000
  ```

  and then

  ```
  x/16x 0x70010000
  ```

  in the debugging terminal. The two commands show memory content at address $70010000_{(16)}$, yielding 16 (disassembled) instructions in the first instance and then 16 hexadecimal words (i.e., the encoded instructions) in the second instance.
- Rather than inspect memory via an address, it can be more convenient to display the value of a variable (as defined in some C function, for example: there is no analogy in assembly language). Issue the command

  ```
  print x
  ```

  in the debugging terminal: this shows both the address of the variable `x`, *and* the value. Note that the alternative

  ```
  display x
  ```

  does more or less the same, except that it is "sticky" in the sense the output is repeated after every command.
- The `disassemble` command is what it sounds like: it disassembles the machine code in memory, and shows what the assembly language equivalent would look like. Try this out: issue the command

  ```
  disassemble kernel_handler_rst
  ```

  in the debugging terminal: you should be able to identify both the outer (infinite) `while` loop and inner `for` loop, plus the call to `PL011_putc` in the ~ 11 instruction assembly language disassembly of the `kernel_handler_rst` function.

- Since execution is stopped, gdb can also inspect the state of the processor. For example, one might issue the command

  ```
  info registers
  ```

  to show the content of the ARM register file, or

  ```
  info stack
  ```

  to describe the entire stack, or

  ```
  info frame
  ```

  to describe the current frame on that stack.

d Terminate QEMU and gdb by repeating associated steps from above.

Q1–§2.4 Next steps

There are various things you could (optionally) do next: here are some ideas.

a gdb is an extremely powerful tool; previous Sections only covered a small sub-set of the functionality it provides. As such, time invested now in learning about gdb is sure to pay off later in terms of time *saved* when debugging your own programs. Among numerous resources you could use, some examples include the online documentation at

http://www.gnu.org/software/gdb/

or various online tutorials such as

http://www.youtube.com/watch?v=sCtY--xRUyI

or even dedicated books [9, 8].

b You now have a way to execute ARM assembly language programs via QEMU: essentially you could replace handler_rst with *anything*. As such, this represents an ideal opportunity to start experimenting with your *own* ARM assembly language programs.

If you need a specific challenge, what about attempting to write a solution for a previous coursework from COMS10002? Or, try to solve something from

http://projecteuler.net/

using assembly language!

References

[1] ARM Limited. ARM Dual-Timer Module (SP804) Technical Reference Manual. Technical Report DDI-0271D, 2004. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0271d/index.html.

[2] ARM Limited. PrimeCell UART (PL011) Technical Reference Manual. Technical Report DDI-0183F, 2005. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0183f/index.html.

[3] ARM Limited. Cortex-A8 Technical Reference Manual. Technical Report DDI-0344K, 2010. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/index.html.

[4] ARM Limited. RealView Platform Baseboard for Cortex-A8. Technical Report HBI-0178, 2011. http://infocenter.arm.com/help/topic/com.arm.doc.dui0417d/index.html.

[5] ARM Limited. Procedure Call Standard for the ARM Architecture. Technical Report IHI-0042E, ver. 2.09, 2012. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/index.html.

[6] ARM Limited. ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition. Technical Report DDI-0406C, 2014. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406c/index.html.

[7] ARM Limited. ARM Generic Interrupt Controller Architecture Specification (GIC architecture version 3.0 and version 4.0). Technical Report IHI-0048B, 2015. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0048b/index.html.

[8] N. Matloff and P.J. Salzmann. *Art of Debugging with GDB and DDD*. No Starch Press, 2008.

[9] R.M. Stallman, R. Pesch, and Shebs S. *Debugging with GDB: The GNU Source-Level Debugger*. GNU Press, 2002.

A Models of software development

Some people find it hard to understand what emulation *is*, or, put another way, what relationship exists between the physical and emulated target platforms. In an *attempt* to explain, and although heavily dependant on the specific context and objectives, one can consider (at least) three scenarios when developing some software artefact:

- Figure 9 illustrates scenario #1, which you are likely most familiar with: the idea is that one first a) develops a program, say `foo.c`, which is b) compiled into an executable `foo` then c) executed. Each step is performed on the same platform, which is to say the development and target platforms are the same. You might be *so* used to using this approach that various subtle yet important facts seem trivial. For example, note that

  - the tool-chain used for compilation is native, which means it generates machine code for the same platform it executes on,
  - whenever a program is executed, an associated process will be created and managed by (i.e., will execute under control of) an operating system kernel, and
  - such a process can interact with a rich set of resources: it can usually write to and read from a file system for example, and use standard streams (e.g., `stdout`) as a way to communicate with both the user and other processes.

  Such an approach has no obvious disadvantages per se, in the sense it is, by design, a very convenient way to develop software.

- Figure 10 illustrates scenario #2, where target and development platforms are now physically separate. A common reason for this separation is they differ in type: if the target platform is an embedded device, for example, it may be too constrained (there is often less memory available, and the processor is less computationally able) or lack resources required to host a kernel. As a solution, it is common to develop software on the development platform and then transfer this for bare-metal execution on the separate target platform. A range of implications stem from this difference:

  - the tool-chain used will differ somewhat: the compiler used is now a cross-compiler (in the diagram, a compiler for X st. we use `x-gcc` not `gcc`) meaning it executes on the development platform but generates machine code for the target platform,
  - because there is no kernel, the programmer is tasked with how a) the program is loaded into memory on the target platform, b) execution is then involved, *and* c) any subsequent communication with the development platform, and
  - the executing program has full control over the platform, but no support (other than from standard libraries linked to it): for example, there simply is no `stdout` to use in the same way as scenario #1.

  This approach provides some advantages: bare-metal execution allows use of instructions or resources that may otherwise be protected by the kernel, for example. However, it can also present significant development challenges, such as the necessity for use of a remote debugger.

- Figure 11 illustrates scenario #3, which represents a compromise between scenarios #1 and #2. In a sense scenario #3 is the same as scenario #2, because the target and development platforms differ. However, the physical target platform is replaced by an emulated (or simulated) alternative.

  The advantage of doing so is that the (software) emulator executes on, and can therefore interact with the development platform as with scenario #1. So although we still need a cross-compiler and so on to develop software, some of the development challenges presented by scenario #2 are lessened. The (potential) disadvantage is realism, in the sense that accuracy of the emulator is now crucial. If it were *in*accurate our software might behave differently on the physical and emulated target platforms, which is less than ideal. Provided it *is* accurate, however, the software we develop is no less valid than in scenario #2. Rather, it simply removes the need for a physical target platform until the software is then later deployed.

B Developing software for the PB-A8 using QEMU

Developing then executing and debugging software for the PB-A8 can be challenging, in the sense doing so will differ from your normal approach. As such, the subsequent Sections attempt to clearly explain what exactly each difference is, what options exist for coping with said differences, and which option is assumed here, and therefore recommended for use in the coursework assignment.

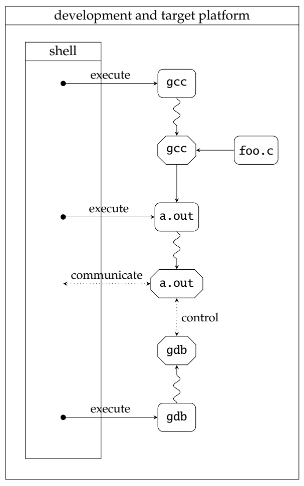


Figure 9: *A block diagram attempting to illustrate development scenario #1: since the development and target platforms are the same, this should reflect your normal compile-execute-debug cycle as invoked from a shell or via an IDE.*

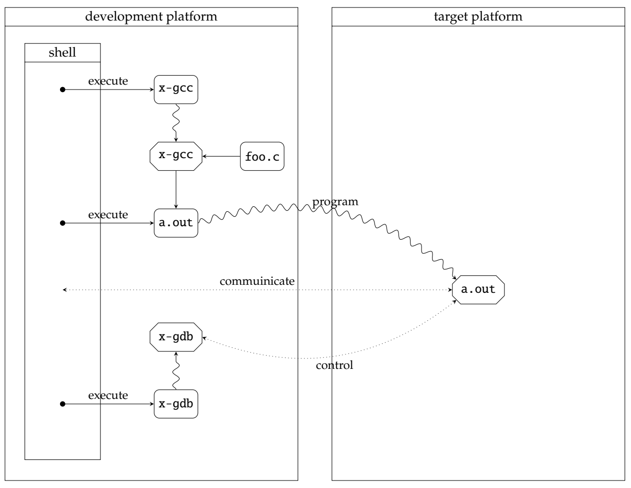


Figure 10: *A block diagram attempting to illustrate development scenario #2: since the development and target platforms differ, there needs to be a) an explicit programming step (to transfer* `a.out` *into memory on the target platform), and b) extra physical connectivity (e.g., via USB or similar) to allow communication and remote debug.*

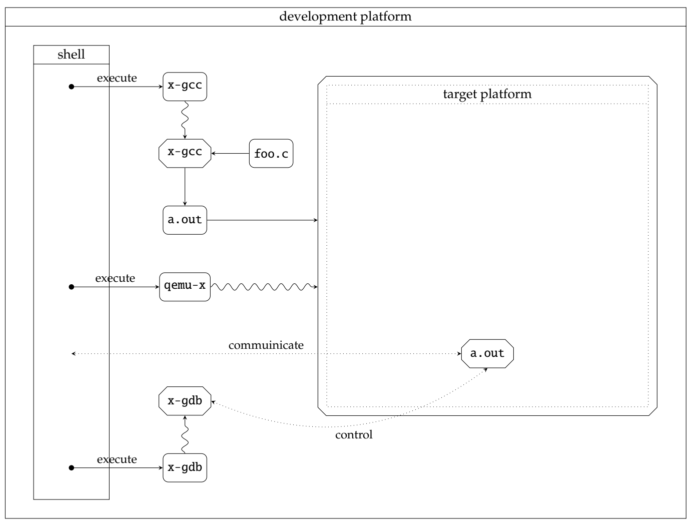


Figure 11: *A block diagram attempting to illustrate development scenario #3: the emulated target platform is now shown as a process executing on the development platform, thus acting as a form of intermediate solution that blends features of Figure 9 and Figure 10.*

Writing programs The nature of software we are interested in developing is different from the norm. More specifically, it will often *require* the use assembly language, e.g., to guarantee control over what is executed, and/or access a low-level processor feature which lacks support at a higher level. Faced with this requirement, (at least) two strategies are possible: you could

- write a high-level program in C, and link it to any low-level part(s) written in *raw* assembly language, or
- write a high-level program in C, and embed any low-level part(s) in that program using *inline*[5] assembly language.

We will a) assume use of the first strategy, since it enforces a clearer separation between the high- and low-level parts, but b) try to minimise the amount of assembly language *you* have to write yourself.

Compiling programs Either way, compiling the resulting program demands more care than usual. Whereas normally the default tool-chain options will suffice, we need to carefully set those options to ensure the compilation process matches our requirements. With this in mind, we assume `gcc` is invoked as follows:

- It is important to be clear about which processor will be targeted, since the compiler may behave (e.g., make an optimisation decision) differently for one vs. another. As such, we use

  ```
  -mcpu=cortex-a8
  ```

  to specify the exact processor model.
- It is important to be clear about which function calling convention will be used, otherwise compiled and hand-written assembly language programs cannot interact correctly. This is achieved by
  - *forcing* use of the ARM-specified AAPCS [5] by using `-mabi=aapcs`, and
  - assuming the `-O` optimisation flag is used, st. `-fomit-frame-pointer` is implied; the frame pointer register, optional under AAPCS, is not used and the calling convention is simpler as a result.

Linking programs

1. As already outlined by Appendix A, a bare-metal, unhosted target platform differs from a hosted alternative wrt. the support it provides during execution of software. Within this context, the C standard library[6] represents an central component. In general, it could be thought of as performing two roles: it
   (a) provides a range of kernel-agnostic functionality relating to the C language (e.g., `stdint.h`, which just defines types such as `uint32_t`), and
   (b) provides a range of kernel-specific functionality, and thus a software layer to abstract interaction with the kernel (e.g., `stdio.h`, which defines I/O functionality and thus depends on the implementation of I/O in the kernel).

   The latter is a problem for an unhosted platform since there is no kernel to interact with, but clearly *some* functionality will be useful even in this case. Therefore, various pared down (or limited) implementations of the library exist; these are commonly designed for and used on embedded platforms. An example is

   http://www.sourceware.org/newlib/

   which we assume use of here.
2. The kernel is normally tasked with management of each process which results from execution of an associated program. An important part of this task, as performed by the loader, is initialisation of the address space: an obvious example is the text segment, populated by instructions from the executable image that constitute the program, which may or may not be subjected to some form of relocation.

   On an unhosted target platform, there is no kernel so *we* must take responsibility for this task. Doing so involves provision of extra information to the linker `ld`, in the shape of an explicit (rather than implicit, default) linker script[7]. We assume `ld` is invoked with the

   ```
   -T image.ld
   ```

   option where `image.ld` is the linker script in question.

[5] http://wiki.osdev.org/Inline_Assembly
[6] http://wiki.osdev.org/C_Library
[7] http://wiki.osdev.org/Linker_Scripts

Debugging programs Debugging software can be difficult at the best of times, so even when you are faced with this task normally a debugger can be extremely useful. For example, executing a program under control of `gdb` will allow single-stepping through individual instructions or direct inspection of their influence on registers and memory.

`gdb` supports local or remote debugging, capturing cases where the development and target platforms are the same or differ respectively. In the latter case, a `gdb`-based interface (or shell) executed on the development platform can "remote control" the target platform via a communication link (e.g., a physical cable, or across a network). We will assume this strategy is employed, and specifically that:

- Compilation and assembly using `gcc` and `as` uses the `-g` flag to produce output with debugging information (e.g., symbols) included; this allows `gdb` to form a correspondence between the source code and machine code.
- `qemu-system-arm` is executed using
  - the `-gdb tcp:127.0.0.1:1234` option so a `gdb`-friendly remote debugging interface is presented on TCP port 1234 of the development platform, and
  - the `-S` flag to halt execution once the emulated platform is initialised (i.e., it waits st. `gdb` can connect before any instructions are actually executed)

  meaning `gdb` can subsequently connect to and remotely debug the program *it* is executing.