AppNote 2 - Programming SpiNNaker with ARM and GNU tools

SpiNNaker Group, School of Computer Science, University of Manchester

Steve Temple - 25 Nov 2011 - Version 1.00

Introduction

This document is a quick introduction to programming SpiNNaker with software tool-chains from ARM and GNU. It is expected that most low-level programming of SpiNNaker will be done using the C language occasionally supplemented with small sections of assembly language code. The tools required to do this are the C compiler, the assembler and the linker.

Although the description outlined below of the various tools and options seems complex, for most routine programming just standard C language will be used and the details of running the tools will be hidden in pre-prepared scripts and makefiles.

Which Tools?

We have been using the RVDS 4.0 release of the ARM RealView Development System and GCC 4.5.2 from Code Sourcery

```
http://www.codesourcery.com/sgpp/lite/arm/portal/release1802
```

These are installed at Manchester in */home/amulinks/spinnaker/tools*. This directory also contains some locally developed tools (mostly Perl scripts) which are also needed. You can set up to use the GNU tools as follows

```
set -a
TOOLS=/home/amulinks/spinnaker/tools
INC_DIR=/home/amulinks/spinnaker/code/include
PATH=$TOOLS/gnu/bin:$TOOLS/bin:$PATH
```

The GNU binaries begin with the string *arm-none-linux-gnueabi-*. So the compiler is *arm-none-linux-gnueabi-gcc* and the linker is *arm-none-linux-gnueabi-ld*, etc

The ARM tools are set up as follows

```
set -a
TOOLS=/home/amulinks/spinnaker/tools
INC_DIR=/home/amulinks/spinnaker/code/include
ARMBIN=$TOOLS/RVDS40/RVCT/Programs/4.0/400/linux-pentium
ARMLIB=$TOOLS/RVDS40/RVCT/Data/4.0/400/lib
ARMINC=$TOOLS/RVDS40/RVCT/Data/4.0/400/include/unix
PATH=$ARMBIN:$TOOLS/bin:$PATH

LM_LICENSE_FILE=NNNN@licence_server.your.domain
```

Note that the ARM tools are licenced (via FLEX) and require the LM_LICENSE_FILE variable to point to a valid server. Contact your local licence administrator to set this up.

Running the Assembler

Both ARM and GNU assemblers support the same syntax for instructions. The assembler directives are quite different however and it is not possible to write assembly language programs

which can be processed by both assemblers. However, it is possible to automatically translate assembly directives and a simple Perl script *arm2gas* has been developed which translates ARM assembly language programs into a form acceptable to the GNU assembler. To allow a single source file to be maintained, it is recommended that assembly language coding be done in ARM assembler syntax.

The ARM assembler is *armasm* and it normally reads *program.s* and writes *program.o* which is an ELF format object file. Various flags need to be given to configure the assembler. For example

```
armasm --keep --cpu=5te --apcs=/interwork program.s
```

The flags are as follows

```
--keep            # retain local labels in output (aids debugging)
--cpu=5te         # indicates the correct architecture for SpiNNaker
--apcs=/interwork # support ARM/Thumb inter-working
```

The GNU assembler is *arm-none-linux-gnueabi-as*. The extension *.gas* is used for GNU assembler files to distinguish them from the *.s* used for ARM assembler files. The assembler is used as follows

```
arm-none-linux-gnueabi-as -defsym GNU=1 -mthumb-interwork \
   -march=armv5te -o program.o program.gas
```

The flags are as follows

```
-defsym GNU=1     # flags that we are building under GNU
-march=armv5te    # indicates the correct architecture for SpiNNaker
-mthumb-interwork # support ARM/Thumb inter-working
```

Running the Compiler

Both the ARM and GNU C compilers support the C language standard and this means that most coding can be done using standard C. Both compilers support ARM-specific language extensions but in different ways. For example, both allow interrupt handlers to be coded in C but use different syntax to flag the C function as a handler.

In general the ARM C compiler is the more mature product and copes with more ARM-specific coding situations than the GNU compiler. However, by carefully arranging the source language files it is possible to code for both compilers with minimal use of conditional compilation.

The ARM compiler is *armcc* and it normally reads *program.c* and writes *program.o* As with the assembler, various flags are used to control the compilation

```
armcc -c --cpu=5te --c99 --apcs=/interwork -I $INC_DIR program.c
```

The flags are as follows

```
-c                # just compile (no linking)
--cpu=5te         # indicates the correct architecture for SpiNNaker
--c99             # compile C99 standard source
--apcs=/interwork # support ARM/Thumb inter-working
-I $INC_DIR       # specify header file directory (see below)
```

The GNU C compiler is *arm-none-linux-gnueabi-gcc* and it is used as follows

```
arm-none-linux-gnueabi-gcc -c -O1 -nostdlib -mthumb-interwork \
   -march=armv5te -std=gnu99 -I $INC_DIR program.c
```

The flags are as follows

```
-c                # just compile (no linking)
-O1               # optimisation level (use 1 for now)
-nostdlib         # don't compile for standard library
-mthumb-interwork # support ARM/Thumb inter-working
-march=armv5te    # indicates the correct architecture for SpiNNaker
-std=gnu99        # compile GNU C99 standard source
-I $INC_DIR       # specify header file directory (see below)
```

The compilers can generate ARM or Thumb code with ARM being the default. To generate Thumb code the following flags should be added for the ARM compiler

```
--thumb           # cause Thumb code generation
-DTHUMB           # flag used in source to indicate Thumb generation
```

For GCC, add these flags

```
-mthumb           # cause Thumb code generation
-DTHUMB           # flag used in source to indicate Thumb generation
```

Various SpiNNaker-specific header files are available and these are kept in this directory at Manchester

```
/home/amulinks/spinnaker/code/include
```

You should normally set the INC_DIR variable above to this value.

Building sources for both ARM and GNU

If you are sure that you will only ever work with one of the ARM or GNU tool-chains then you can write code which uses features specific to your chosen tool-chain. However, it is possible to write code which works with either tool-chain by using conditional compilation and by understanding the limitations of each tool-chain. It is recommended that you do this wherever possible. Conditional compilation is driven by defining a variable. For the C compilers the variable *__GNUC__* is defined automatically by the GNU compiler. For the assemblers, defining the variable *GNU* on the GNU assembler command line allows assembler programs to be conditionally compiled.

Assembler programs are not particularly problematic to build for ARM or for GNU. However, there are three main areas in C programs where care needs to be taken - interrupt handlers, Thumb code and inline assembly.

Interrupt Handlers

Both ARM and GNU compilers allow interrupt handlers to be coded directly in C. The syntax is different between compilers but easily handled by conditional compilation. For example

```
#ifdef __GNUC__
void __attribute__ ((interrupt ("IRQ"))) interrupt_handler (void)
#else
__irq void interrupt_handler (void)
#endif
{
  ... interrupt handler code
}
```

Thumb Code

Both compilers allow an entire source file to be compiled for either ARM or Thumb using a command line flag. The ARM compiler also allows individual parts of a file specified as ARM or Thumb using *#pragma*. This facility should not be used if ARM/GNU compatibility is needed. Separate source files should be used to define ARM or Thumb compiled sections of code

Where compilation (or assembly) needs to be conditional on whether ARM or Thumb code is being built, the variable *THUMB* may be used as described above.

Inline Assembler

This is the area where the differences between the compilers are greatest. Very different syntax is used meaning that conditional compilation is required and each inline routine must be written twice, once for each compiler. For very small routines this may be acceptable but in general it is better to code these routines separately in an assembler source file and link them using the linker. This also makes it easier to compile the C source for ARM or Thumb while keeping the assembly language routines in the intended ARM or Thumb format.

Running the Linker

Both C compilers and assemblers produce object files in ELF format and these can be linked together by either the ARM or GNU linker programs.

The SpiNNaker memory architecture with separate and small instruction and data memories means that linking programs is not straightforward and linker script files are required to define the architecture and ensure that code and data end up in the correct place. The ARM linker *armlink* and the GNU linker *arm-none-linux-gnueabi-ld* have incompatible linker script syntax and so different linker script files are needed. In many cases, the same script is applicable to many different programs so it's not too difficult to maintain two separate scripts.

The linkers produce output files in ELF format and these need to be further processed to make binary images suitable for loading into SpiNNaker. For the ARM tool-chain the *fromelf* utility performs this step while for GNU the *arm-none-linux-gnueabi-objcopy* utility makes the binary image.

Local Support Tools

To assist with the compilation/assembly process, a couple of programs have been produced. *h2asm* reads a C header file and creates an ARM assembler file. It is mainly intended to translate *"#define NAME VALUE"* into *"NAME equ VALUE"* so that only a single header file needs to be maintained for both C and assembler programs.

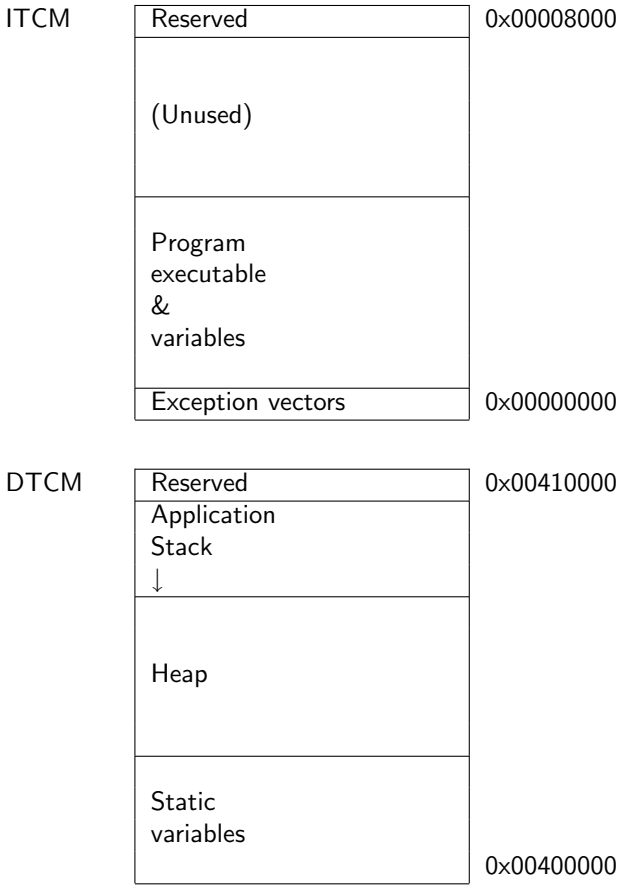
The *arm2gas* program reads assembler language programs in *armasm* format and converts them to GNU assembler format by translating the assembler directives.

SpiNNaker Memory Model

Applications which are loaded onto SpiNNaker require the use of various parts of memory. The executable code will normally be loaded into ITCM. Some variables may be stored in ITCM but generally they will be kept in DTCM. These variables will normally fall into one of three

categories. Stack-based (auto) variables will be kept on the run-time stack. This is usually at or near the top of DTCM and grows downwards. Static (or global) variables will normally be stored near the bottom of DTCM. The area between the static variables and the lower limit of the stack will normally be given over to heap storage and this is where the third type of variables will be kept. These are dynamic storage maintained by a malloc/free mechanism.

Not all SpiNNaker applications will conform to this model and many applications will also require access to other memories such as SDRAM and the System RAM. However, this model is close to that expected by C programs and it forms a useful starting point.



Change log:

- *1.00 - 25nov11- ST* - initial release - comments to *temples@cs.man.ac.uk*