App note: Writing in C for the coprocessor

Introduction

This note describes how the popular 'C' programming language can be used to author programs for the ARM coprocessor. Writing in a higher level language has many advantages over writing in assembler. Often the program's goal can be achieved in a much shorter development time, using existing 'C' functions written by other people, with fewer hard to track down bugs introduced.

Please note that this application note is not suitable as a general introduction to those wishing to learn to write programs in 'C', existing knowledge of the language is assumed.

Conventions used in this manual

The following typographical conventions are used throughout this guide:

Hexadecimal numbers are prefixed with ampersand.

Decimal numbers have no prefix.

Binary numbers may be denoted with a leading percent and given in decending bit significant order (ie.for an eight bit number they will be written in the order %76543210).

Multibyte data is stored in memory in little endian form.

Copyright

History

V0.10 First draft.

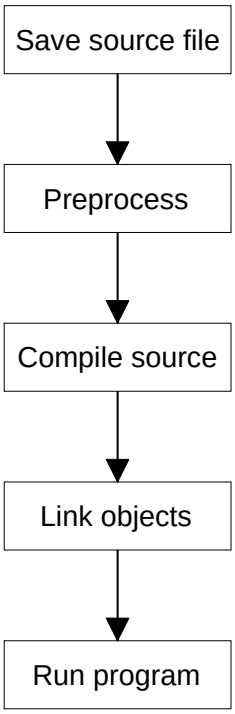
V0.20 Added a third compiler, the Yagarto GCC tools.

V0.21 Moved -lc -lm switches in Yagarto link step.

Tool chain

Outline of steps

Unlike interpreted languages such as BASIC, a 'C' program must be compiled before it can be run. This process generally involves the following steps



The author writes and saves the source file as a text file.

During preprocessing, any macros are expanded header files included, and substitutions made.

The compiler then turns the source file into an object file which contains any function and variable names along with a partly complete program.

Next the linker looks at all of the object files that the compiler produced and joins them all together matching up function and variable names used.

Finally, the program is ready to run.

each of these steps must be performed successfully for an executable to be produced, with each tool forming a link in the chain.

Due to the ARM's popularity there are many different vendors of tool chains, some will have the preprocessor as part of the compiler, some compilers might automatically run the linker for you, and so on. In this application note three tool chains will be covered, though it should be possible to study the general steps involved and produce 'C' programs using your preferred tool chain.

Norcroft C compiler

This is the compiler developed by Acorn to write desktop programs for their computers.
Version used in these examples: 5.54
Hereafter referred to as the 'Norcroft tool chain'.

ARM C compiler

This is the compiler which is part of the RealView Development Suite (RDS) from ARM.
Version used in these examples: 2.0.1 build 359
Hereafter referred to as the 'ARM tool chain'.

GCC compiler

This is the compiler which is part of the 'Yagarto' open source GNU compiler collection.
Version used in these examples: 4.4.2
Hereafter referred to as the 'GCC tool chain'.

Libraries

Calling SWIs

All compilers come with a default set of libraries which are handy functions written by someone else which are supplied as object files (sometimes the source code is also supplied too), and header file (so the compiler knows how many parameters each function takes and what results it returns).

By default ANSI libraries are supplied which are standard functions defined by the ANSI committee:

assert	- assertion tests
ctype	- common types and type conversions
errno	- the global error number
locale	- country or region specific tailoring
math	- floating point functions such as logarithm
setjmp	- context switching support
signal	- error handling
stdarg	- multiple argument processing
float	- standard definitions
limits	- standard definitions
stddef	- standard definitions
stdio	- input output functions
stdlib	- integer support functions
string	- string manipulation
time	- calendar related

though often only one or two libraries will be actually required and the rest can be ignored.

What is not usually supplied with a compiler are libraries which relate to interfacing to the underlying hardware or operating system. In the case of the ARM coprocessor these are both interfaced to via SWI calls (documented seperately).

The Norcroft tool chain provides the SWIs library for this purpose, comprising a header file with function prototypes in and all the core SWI numbers. For the purposes of working with the GCC and ARM tool chains these two functions must be reimplemented.

Function: `int _swi(int swi_no, unsigned int mask, ...);`

Purpose: A function to call a given SWI from C and get its results

The `swi_no` is the SWI number to call, and the `mask` defines how many parameters are passed into the SWI and how many come back and into which registers. Formulating this mask is made easier by a set of macros which are simply OR'd together.

Function: `_kernel_oserror *_swix(int swi_no, unsigned int mask, ...);`

Purpose: A function to call a given SWI trapping and returning any error it causes

The `swi_no` is the SWI number to call, and the `mask` defines how many parameters are passed into the SWI and how many come back and into which registers. Formulating this mask is made easier by a set of macros which are simply OR'd together.

This differs from the `_swi()` function as it will automatically call the X version of the SWI so that any errors that occur during the call do not cause your program to quit. Instead they are returned in an error message block (defined here as `_kernel_oserror`).

The example programs later in this application note call SWIs for output, and can be studied as a starting point.

Connecting up getc() and putc()

Any time some results from your program are to be displayed on the screen, for example using formatted printing in the `printf()` function, the input output library ('stdio') will break down the string to be printed into a stream of single characters. These ultimately are printed using the single character put function, `putc()`.

Similarly when requesting key presses from the user, the input output library will call an underlying `getc()` function.

Due to the ARM coprocessor supporting all of the SWIs required by the Norcroft tool chain for displaying text on the screen and reading characters from the keyboard, all that is required for `getc()` and `putc()` to work is to link the program with the preexisting library. Ordinarily this would be the Shared C Library, this makes programs smaller and easier to distribute on floppy disc amongst other reasons since the Shared C Library module is loaded just once from ROM on most Acorn computers - whenever a C program is started it just locates the library and shares all of its functions. As the ARM Tube OS does not include a Shared C Library module any C programs you create must instead be linked with the standalone ANSI library ('ansilib'), which makes the program about 60kB bigger typically, though as the ARM coprocessor has many megabytes of RAM this is unlikely to be a major issue.

As the ARM tool chain is used by many developers on products from mobile phones to computer games it can't make any assumptions about where `putc()` and `getc()` go to, sometimes they target hardware doesn't have a screen and a serial port is used instead. Therefore, filling in `putc()` and `getc()` is simply a matter of calling the corresponding SWI which will in turn ask the host to display the character on screen or read from the keyboard.

The GCC tool chain is generally expecting to target a Unix environment with a console for input and output. For more embedded use, such as the ARM coprocessor, a cut down library ('newlib') is available which is both smaller and also includes hooks to override the `putc()` and `getc()` by writing replacements called `_read()` and `_write()`.

Runtime memory

Memory that is declared to hold variables at compile time, whether static or global, appears after the code when loaded into the coprocessor to run. Those which are initialised to a value go in the "read/write" region, and those which are initialised to zero go in the "zero init" region.

The Norcroft tool chain knows that the rest of application RAM is free to be used for a heap and stack, whereas the GCC and ARM tool chains have to be told. For this purpose a short routine is required which reads the application RAM limits from the ARM Tube OS and uses these values to determine where the user stack and heap can go just before the `main()` function is reached.

Floating point mathematics

Floating point maths is useful where real life numbers are involved, for example measuring a temperature which can't conveniently be expressed as a whole number.

When the ARM processor was originally designed its instruction set was left open ended, and any time an undefined instruction was encountered in a program it would be offered around on a coprocessor interface - not to be confused with the Tube! This allows for specialised number crunching chips to sit alongside the ARM processor - remember though that the ARM used here does not have any such on board hardware coprocessors.

The first such coprocessor was the FPA ('floating point accelerator') which dealt with floating point numbers in single and double precision. Often the FPA hardware was omitted and its capabilities emulated in

software instead, the last hardware FPA was included in the ARM7500FE in 1996. Other more recent coprocessors include the Piccolo (for DSP work) and VFP (for vector floating point work).

Whenever floating point numbers are used the Norcroft tool chain translates these directly into the FPA instruction set, knowing that even if an FPA is not present the operation will be emulated in software instead by the Floating Point Emulator module. As the ARM Tube OS does not include the Floating Point Emulator module and it is not possible to force the Norcroft tool chain to do floating point maths any differently it effectively means that floating point must be avoided.

The GCC and ARM tool chains on the other hand allow the choice of floating point unit ('fpu') to be made when compiling your program. Choosing the VFP coprocessor would cause it to output VFP instructions for example, and there is one additional option which is to select software floating point.

Software floating point selects a software floating point library which includes all of the normal floating point operations but written using only normal ARM instructions, so for floating point work this is the approach which will be adopted here.

Putting it all together

Setup

Commands for the 3 tool chains considered for compiling the examples given at the end of this application note. To save time, it is more normal to put the following commands into a script of some sort (such as a batch file) so that refinements can be made without having to tediously reenter the commands by hand.

Source files are assumed to be in a directory called 'c', with the header files in a directory called 'h', and the object files output into a directory called 'o'.

Running the program

Application space on the ARM coprocessor starts at &00008000, so the load and execution addresses need to be set to this number before the executable can be run. The free utility 'SETATTR' is available for this purpose.

Then all that is required is

```
*RUN myprogram
```

alternatively the ARM Tube OS supervisor can run the program regardless of its load and execution addresses

```
*GOS
*LOAD myprogram 8000
*GO
```

consult the documentation on *RUN and *GO for details of how to pass command line parameters into your program if this is required (they can be read with OS_GetEnv once the program has started).

Norcroft tool chain specifics

For the Norcroft tool chain the ".c" and ".h" file extensions are not required and should be removed.

Norcroft tool chain (example 1)

```
cc -c alphabet -o o.alphabet -apcs 3/32bit
link -o alphabet -aif o.alphabet C:o.ansilib
```

Norcroft tool chain (example 3)

```
cc -c sieve -o o.sieve -apcs 3/32bit
link -o sieve -aif o.sieve C:o.ansilib
```

Norcroft tool chain (general case)

```
cc -c file1 -o o.file1 -apcs 3/32bit
cc -c file2 -o o.file2 -apcs 3/32bit
:
cc -c fileN -o o.fileN -apcs 3/32bit
link -o myprogram -aif o.file1 o.file2 ... o.fileN C:o.ansilib
```

ARM tool chain specifics

The ARM assembler and C source code is provided for the functions `_swi()`, `_swix()`, `getc()`, and `putc()` as described earlier. For simplicity the object files will be used in these examples, but if any changes are made they were produced using the following commands:

```
armasm -cpu ARM7TDMI -apcs /inter -i .\h -g
       -o .\o\armtubeswis.o .\c\armtubeswis.s
armcc -c -cpu ARM7TDMI -apcs /inter/noswst -i .\h -arm -fpu none
       -o .\o\armtubeio.o .\c\armtubeio.c
```

The ARM linker ordinarily outputs ELF format files which cannot be run directly by the ARM coprocessor, so a utility is used to convert the output ELF into a binary file that the coprocessor can run.

ARM tool chain (example 1)

```
armcc -c -cpu ARM7TDMI -apcs /inter/noswst -i .\h -arm -fpu none
       -o .\o\alphabet.o .\c\alphabet.c
armlink -elf -nodebug -nolocals -ro 0x00008000
       -o .\alphabet.elf .\o\alphabet.o .\o\armtubeswis.o
fromelf --bin -o .\alphabet.bin .\alphabet.elf
```

ARM tool chain (example 2)

```
armcc -c -cpu ARM7TDMI -apcs /inter/noswst -i .\h -arm -fpu none
       -o .\o\circle.o .\c\circle.c
armlink -elf -nodebug -nolocals -ro 0x00008000
       -o .\circle.elf .\o\circle.o  .\o\armtubeswis.o
fromelf --bin -o .\circle.bin .\circle.elf
```

ARM tool chain (example 3)

```
armcc -c -cpu ARM7TDMI -apcs /inter/noswst -i .\h -arm -fpu softvfp
       -o .\o\sieve.o .\c\sieve.c
armlink -elf -nodebug -nolocals -ro 0x00008000
       -o .\sieve.elf .\o\sieve.o .\o\armtubeio.o .\o\armtubeswis.o
fromelf --bin -o .\sieve.bin .\sieve.elf
```

ARM tool chain (general case)

```
armcc -c -cpu ARM7TDMI -apcs /inter/noswst -i .\h -arm -fpu softvfp
       -o .\o\file1.o .\c\file1.c
armcc -c -cpu ARM7TDMI -apcs /inter/noswst -i .\h -arm -fpu softvfp
       -o .\o\file2.o .\c\file2.c
:
armcc -c -cpu ARM7TDMI -apcs /inter/noswst -i .\h -arm -fpu softvfp
       -o .\o\fileN.o .\c\fileN.c
armlink -elf -nodebug -nolocals -ro 0x00008000 -o .\myprogram.elf
       .\o\file1.o .\o\file2.o ... .\ofileN.o
       .\o\armtubeio.o .\o\armtubeswis.o
fromelf --bin -o .\myprogram.bin .\myprogram.elf
```

GCC tool chain specifics

The ARM assembler and C source code is provided for the functions `_swi()`, `_swix()`, `_read()`, and `_write()` as described earlier. For simplicity the object files will be used in these examples, but if any changes are made they were produced using the following commands:

```
arm-elf-gcc -x assembler-with-cpp -march=armv4t -g
       -c .\c\armtubeswis.s -o .\o\armtubeswis.o
arm-elf-gcc -march=armv4t -mthumb-interwork -mfpu=fpa -g -I.\h
       -c .\c\armtubeio.c -o .\o\armtubeio.o
```

The GCC linker ordinarily outputs ELF format files which cannot be run directly by the ARM coprocessor, so a utility is used to convert the output ELF into a binary file that the coprocessor can run.

The switches "-lc" and "-lm" force the linker to scan "libc" and "libm" from 'newlib' before the larger built in "libgcc". However, there appears to be a fault with floating point maths in 'newlib' version 1.18.0 so the earlier version 1.17.0 must be used. When downloading the tools from

http://sourceforge.net/projects/yagarto/files

specifically look for the text "gcc-4.4.2" and "nl-1.17.0" in the filename to make sure a known good combination is used. Download the archive, and run the install wizard to install it on the host PC.

The examples also use "-mthumb-interwork" and "-mfpu=fpa" together for convenience because the precompiled copy of 'newlib' that the install wizard comes with were also compiled with these options, so to avoid the linker faulting any differences all of the object files should be made with these switches too. The last important step is to make sure these specific libraries are used in preference to any other, which requires copying all the files from

C:\Program Files\yagarto\arm-elf\lib\interwork\

up to

C:\Program Files\yagarto\arm-elf\lib\

and similarly all the files in

C:\Program Files\yagarto\lib\gcc\arm-elf\4.4.2\interwork\

up to

C:\Program Files\yagarto\lib\gcc\arm-elf\4.4.2\

GCC tool chain (example 1)

```
arm-elf-gcc -march=armv4t -mthumb-interwork -mfpu=fpa -g -I.\h
       -c .\c\alphabet.c -o .\o\alphabet.o
arm-elf-gcc -nostartfiles -entry=0x8000 -o alphabet.elf
       .\o\armtubeswis.o .\o\alphabet.o -lc
arm-elf-objcopy -Obinary alphabet.elf alphabet.bin
```

GCC tool chain (example 2)

```
arm-elf-gcc -march=armv4t -mthumb-interwork -mfpu=fpa -g -I.\h
       -c .\c\circle.c -o .\o\circle.o
arm-elf-gcc -nostartfiles -entry=0x8000 -o circle.elf
       .\o\armtubeswis.o .\o\circle.o -lc -lm
arm-elf-objcopy -Obinary circle.elf circle.bin
```

GCC tool chain (example 3)

```
arm-elf-gcc -march=armv4t -mthumb-interwork -mfpu=fpa -g -I.\h
       -c .\c\sieve.c -o .\o\sieve.o
arm-elf-gcc -nostartfiles -entry=0x8000 -o sieve.elf
       .\o\armtubeswis.o .\o\armtubeio.o .\o\sieve.o -lc
arm-elf-objcopy -Obinary sieve.elf sieve.bin
```

GCC tool chain (general case)

```
arm-elf-gcc -march=armv4t -mthumb-interwork -mfpu=fpa -g -I.\h
       -c .\c\file1.c -o .\o\file1.o
arm-elf-gcc -march=armv4t -mthumb-interwork -mfpu=fpa -g -I.\h
       -c .\c\file2.c -o .\o\file2.o
:
arm-elf-gcc -march=armv4t -mthumb-interwork -mfpu=fpa -g -I.\h
       -c .\c\fileN.c -o .\o\fileN.o
arm-elf-gcc -nostartfiles -entry=0x8000 -o myprogram.elf
       .\o\file1.o .\o\file2.o ... .\o\fileN.o
       .\o\armtubeio.o .\o\armtubeswis.o -lc -lm
arm-elf-objcopy -Obinary myprogram.elf myprogram.bin
```

Example program (1)

This simple program prints all of the letters in the alphabet before quitting. There are of course many other ways of printing out the alphabet: using the very flexible `printf()` function; without a loop; using `putc()` with `stdout` as the file handle; and so on. This is presented here mainly as a parallel to the example sideways ROM covered in another application note in this series.

```
#include <stdio.h>
#include "swis.h"

int main(void)
{
        char    i;

        /* Print the alphabet then return */
        for (i = 'A'; i <= 'Z'; i++)
        {
                _swi(OS_WriteC, _IN(0), i);
        }
        _swi(OS_NewLine, 0);

        return 0;
}
```

Example program (2)

The following listing is called 'Sieve', it looks for prime numbers between 1 and a given limit. It is often used as a benchmark for testing the output of a compiler.

```
#include <stdio.h>

/*
 * Sieve of Erastosthenes - a standard C benchmark.
 * Try timing the 100 iterations of counting all primes less than 8190.
 */

#define TRUE    1
#define FALSE   0
#define SIZE    8190

static char flags[SIZE+1];

int main(void)
{
        unsigned int i, prime, k, count, iter;

        printf("Sieve of Erastosthenes - 100 iterations...\n");

        for (iter = 1; iter <= 100; iter++)
        {
                count = 0;
                for (i = 0; i <= SIZE; i++) flags[i] = TRUE;
                for (i = 0; i <= SIZE; i++)
                {
                        if (flags[i])
                        {
                                prime = i + i + 3;
                                for (k = i + prime; k <= SIZE; k += prime)
                                {
                                        flags[k] = FALSE;
                                }
                                count++;
                        }
                }
        }
        printf("There are %u primes less than %u\n", count, SIZE);

        return 0;
}
```

Example program (3)

The last example listing is called 'Circle', a very simple test of the use of floating point number representation where the compiler can use software floating point as opposed to outputting machine code instructions for a hardware floating point unit. It will draw a 400 pixel wide circle on the centre of the screen.

```
#include <stdio.h>
#include <math.h>
#include "swis.h"

#define PI      (3.1415927)
#define RADIUS  (400.0)

int main(void)
{
        double  angle;

        /* Change to mode 0 */
        _swi(OS_ScreenMode, _IN(0) | _IN(1), 0, 0);

        /* Do some trig */
        for (angle = 0.00; angle < 360.0; angle = angle + 1.0)
        {
                double  x, y;

                x = RADIUS * sin((angle/180.0) * PI);
                y = RADIUS * cos((angle/180.0) * PI);

                _swi(OS_Plot, _IN(0) | _IN(1) | _IN(2),
                     64 + 5,         /* Point absolute foreground colour */
                     640 + (int)x,   /* X coordinate centred on the screen */
                     512 + (int)y);  /* Y coordinate centred on the screen */
        }

        return 0;
}
```