Title:	Serial data communication Part 2
Author:	Craig Duffy 17/11/21 13/12/23
Module:	Embedded Systems Programming
Awards:	BSc Computer Science.
Prerequisites:	Basic computer architecture and some C

In the previous worksheet we looked are setting up, initialising and simple use of a serial port. If we want to do more complex things then we will need to employ some more resources.

The story so far…..

We have, I hope, managed to get simple character input and output working – putchar() and getchar(). Just to get a clear idea of how limiting this is I want you to try the following 2 exercises. Firstly, write a piece of code – in your main.c program – that outputs as ASCII characters the numbers 1 to 12 – you can use a loop like

```
for ( int i=1; i != 12; i++)
```

Doing the obvious putchar(i) won't have the desired effect – try it and see. You will have to convert the integer value of i into the ASCII value for 1, 2 and so forth. You can use man ascii from the command line to find out the integer values of the characters 1 to 12.

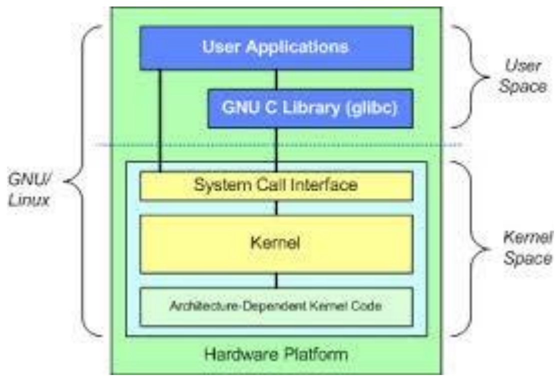
This can get tricky – and output is easier than input! Now try to write a version of the C library function strlen() - use man strlen as your guide. It should take a pointer to a string and return the integer value of its length - int strlen(const char *s). You can simply call it using string constants in your code – IE n=strlen("Hello World"); - if you want to print out the length then you can use your code from the previous exercise to get ASCII output.

Now imagine if you want to recreate all of the libc string functions – namely stpcpy, strcasecmp, strcat, strchr, strcmp, strcoll, strcpy, strcspn, strdup, strfry, strlen, strncat, strncmp, strncpy, strncasecmp, strp-brk, strrchr, strsep, strspn, strstr, strtok, strxfrm, index, rindex. You can use man string to see the details. This is only a tiny part of libc's functionality.

Rather than writing a library from scratch it is better to try to port one. First let's consider what a library is.

libc the standard C library

In our previous worksheet we developed some low level IO facilities. This has allowed us to do simple input and output, however if we want to either write or port more sophisticated applications then we will require much more functionality. This functionality is normally provided by both the operating system on the board and a series of libraries. The standard library for the C language is libc (called glibc with the GNU C compiler), this provides many standard features and functions.



Applications, Libraries and Operating System

As we don't yet have an operating system on our board we won't be able to offer many features, in fact the two we can provide will be memory allocation, which is obviously important for any code, and IO. In later worksheets we will look at the provision of more features through the combination of an operating system kernel and specialist libraries. These libraries will support features such as TCP/IP and networking protocols.

Currently we have developed a very simple device driver for our serial port/UART, and we have some simple applications. If we could provide more library features then we might be able to port more complex code onto the board.

Note on library types.

The diagram above of the relationship of application code, libraries and operating systems is commonly seen in books and articles on the subject, however it is really over simplified. As well as the standard library – libc (or glibc) there are quite a few others. I will briefly go over them, giving an explanation of where they are found and what they do.

crts0.o or cstart.o – These object files are not really libraries but have a very functionally important role in building executives. The are the initialisation code to get the processes running. If you look at your code it seems to start at main(), however quite a few things need to happen before main() can be called. On larger systems, ones running Linux or Windows, these things will have been done at system start up or by the operating system when a new process is created. On our boards we have to take responsibility for this work. So, at a minimum a stack needs setting up and various data areas need initialising. On the code we have been using this code is in startup_stm32l475xx.c.

libgcc The compiler itself may need a library to execute code. For example, if the language expects to be able to do things like 32 or 64 multiplication and that isn’t natively supported by the processor then that code will have to be supplied for the compiler.

libc This is the set of files that is normally thought of as the library. This will include all of the standard input and output functionality, string handling and so on. There is a standard library for mathematical operations called libm.

Requirements for Embedded Systems Libraries

There already exist some very stable and fully featured C libraries around – for example the GNU C compiler, **gcc** comes with **glibc** as a default. This provides support for most things a programmer would need, and many more they may never have thought of! However, glibc is really targeted at the PC market and doesn't really take account of code size, so although there is source code available and many examples of versions being built for ARM processors it is just too large for our board. Another candidate might be **uclibc**, this is a stripped down version of the C library that was developed to support uClinux and busybox, the Linux system for smaller processors and micro-controllers. This is the library system you may well have used if you installed Linux from boot CDs or if you have ever rooted an Android device. The main problem with uclibc is that it requires a version of Linux to be running on the target and our machine doesn't really have enough memory to do that – you would require somewhere in the region of 4MB. There are a number of smaller and specialist libraries for embedded targets see http://www.etalabs.net/compare_libcs.html for a comparison of some of them – note the size of the glibc code in the comparisons.

The library we will use is called **newlib**, it has been developed especially for small, MMUless systems and does not depend upon their being either a specific operating system available or, which is very good for us, there being no operating system available. Newlib has been around for quite a while – over 20 years – is stable and is backed by many large industry players – it is currently owned by Redhat and is used by Google and Mentor Graphics (CodeSourcery).

This will possibly be your first experience of downloading and configuring a reasonably large open source project. So, some of the territory will be unfamiliar but the experience will help you with later projects.

Note

If you are building this on your own machine or caddy drive you may well have to install lots of applications to help build the libraries. Exactly how you do this will depend upon your installed OS.

Getting Newlib

There a quite a few ways of getting newlib sources. The easiest is to use git to take a copy (AKA clone). Use

```
git clone git://sourceware.org/git/newlib-cygwin.git
```

This will create a newlib-cygwin directory with all of the source and various other files. It is worth putting this directory somewhere sensible as you will want to refer to it later on through your Makefiles – I put mine in the directory with all my lab session repos.

You can also get the newlib source from their ftp site which will allow you to download one of the earlier releases of it, this is useful if you are looking to download a specific release. This will be in **tar** (tape archive) format and will require the **tar** command to extract the files (see **man tar** for details). The ftp site can be found at

```
ftp://sourceware.org/pub/newlib/index.html
```

This will require you to untar the tar file into a convenient directory. Where you place the code is important as you will be using it later on. This means that makefiles will have to refer to it.

Finally, you can get newlib in source form from a CVS repository – **CVS** is the Concurrent Versions System which is a free open source client server code version control system. This allows you to check out various versions of code source trees.

```
cvs -z 9 -d :pserver:anoncvs@sourceware.org:/cvs/src login
{enter "anoncvs" as the password}
cvs -z 9 -d :pserver:anoncvs@sourceware.org:/cvs/src co newlib
```

The source tree will be under a **src/** directory and will contain all of the source, configuration and makefiles to configure and build newlib.

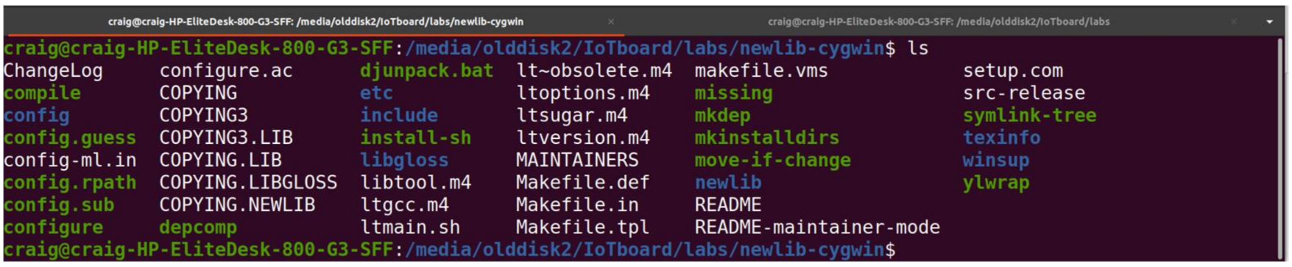
Once you have unpacked your source, if necessary, we will need to explore the source code tree a bit before going on to configure it. There is some pretty good documentation of newlib at:

https://sourceware.org/newlib/

Note: The version of newlib at the time of writing the worksheet was newlib-4.1.0. It is likely there will be many newer versions available to download over time. Normally one would want to download the newest version but for the purposes of this worksheet I am recommending working with version 4.1.0 so that the worksheet is easier to complete. Once that has been done it should be reasonably straight forward to upgrade to the newest version. The reason for favouring the newer version is that any bugs may well have been fixed in the most recent version and developers won't give you any help debugging old versions of the library, for obvious reasons.

You can find the version in the README file in the newlib directory.

The source directory (default name newlib-cygwin/) is made of a number of files and sub-directories:



```
craig@craig-HP-EliteDesk-800-G3-SFF:/media/olddisk2/IoTboard/labs/newlib-cygwin$ ls
ChangeLog      configure.ac      djunpack.bat  lt~obsolete.m4  makefile.vms           setup.com
compile        COPYING           etc           ltoptions.m4    missing                src-release
config         COPYING3          include       ltsugar.m4      mkdep                  symlink-tree
config.guess   COPYING3.LIB      install-sh    ltversion.m4    mkinstalldirs          texinfo
config-ml.in   COPYING.LIB       libgloss      MAINTAINERS     move-if-change         winsup
config.rpath   COPYING.LIBGLOSS  libtool.m4    Makefile.def    newlib                 ylwrap
config.sub     COPYING.NEWLIB    ltgcc.m4      Makefile.in     README
configure      depcomp           ltmain.sh     Makefile.tpl    README-maintainer-mode
craig@craig-HP-EliteDesk-800-G3-SFF:/media/olddisk2/IoTboard/labs/newlib-cygwin$
```

The config and etc directories contain general housekeeping information. The documentation is kept in texinfo and the main code directories are newlib and libgloss. Newlib contains all of the source code for the library functions, so you will find a libc directory, for the standard C library, libm, for the maths library and so on. In each of those directories there are various sub-directories and C programs that implement the library functionality. An important directory is in libc/syscalls which contains all of the operating system interface code. This is the main bit that we will need to change.

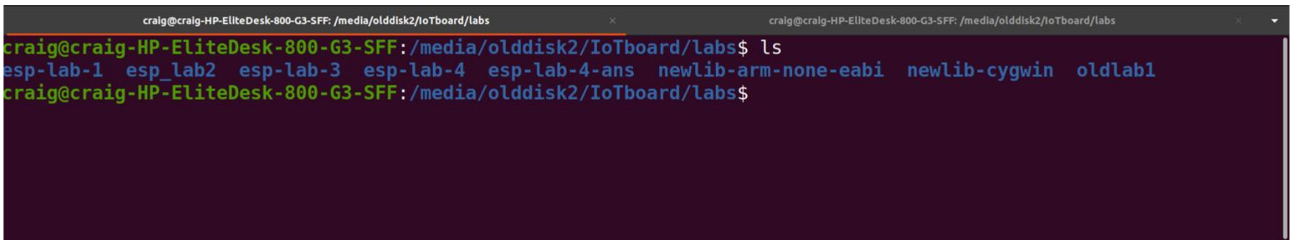
Configuring Newlib

As with many of this type of open source package there is a configuration phase, a build phase and then the installation. In order to understand what we are trying to do read the installation notes at the newlib web site – also look in the file newlib/README from your newlib-cygwin driectory.

As with many installations and configurations you create a target directory – newlib-arm-none-eabi and make sure it is separate from the original sources. So, create a new directory to hold the built version of newlib – call this **newlib-arm-none-eabi**. This directory should be in the same parent directory as the newlib sources. In the directory which holds the source directory newlib-cygwin/ type:

```
%mkdir newlib-arm-none-eabi
```

So once done you should see something along the lines of



```
craig@craig-HP-EliteDesk-800-G3-SFF:/media/olddisk2/IoTboard/labs$ ls
esp-lab-1  esp_lab2  esp-lab-3  esp-lab-4  esp-lab-4-ans  newlib-arm-none-eabi  newlib-cygwin  oldlab1
craig@craig-HP-EliteDesk-800-G3-SFF:/media/olddisk2/IoTboard/labs$
```

As we are cross compiling the library on a PC host we need to tell newlib where to find the correct compiler and we may want to pass some other configuration commands. Check the README document to see if you understand the various options, you can find out the various option by typing

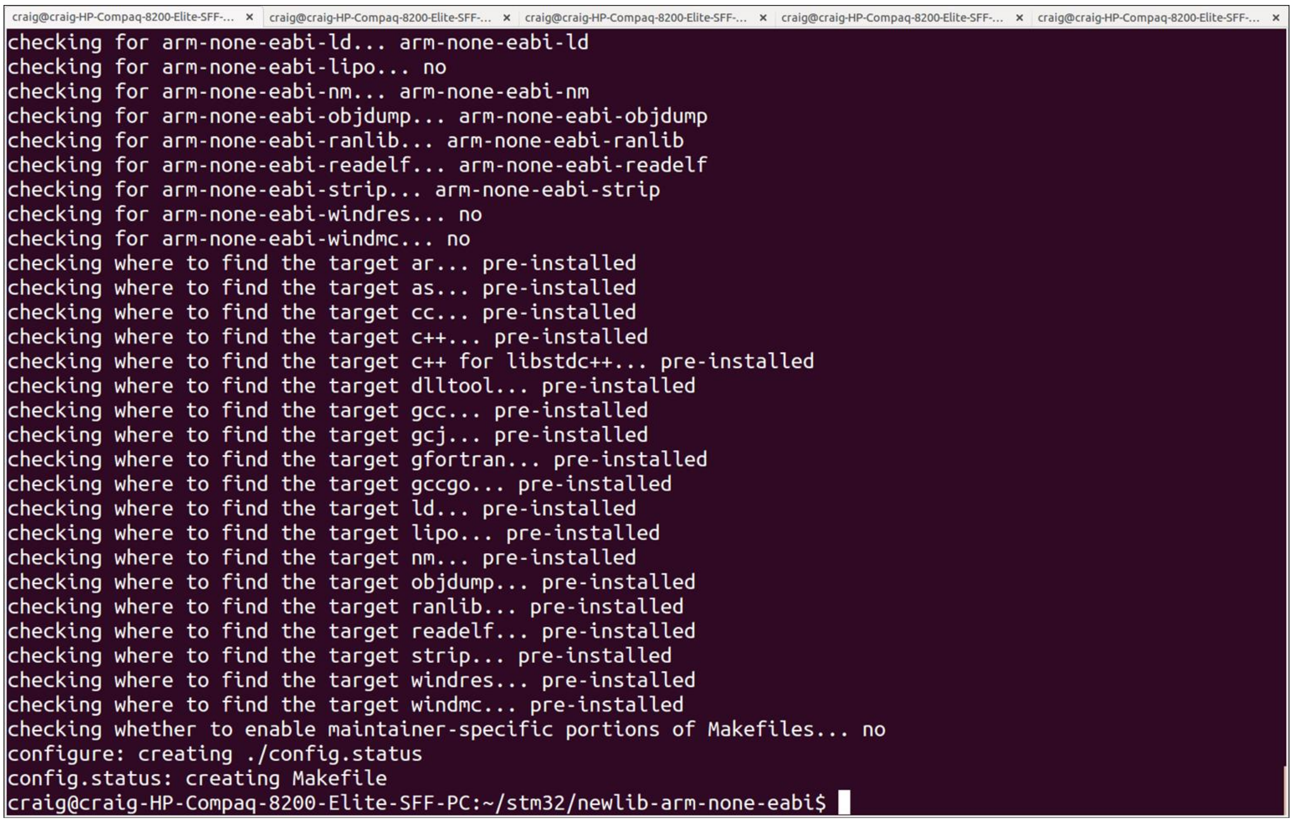
```
../newlib-cygwin/configure -help | more
```

in your newlib-arm-none-eabi directory. The configuration command is issued in the newlib-arm-none-eabi directory but is actually a command held in the original sources directory:

```
%../newlib-cygwin/configure --target arm-none-eabi --disable-newlib-supplied-syscalls -
    srcdir=../newlib-cygwin -prefix=`pwd` --with-gnu-as --with-gnu-ld --enable-multilib=no
```

The command line is pretty complex as we are doing quite a number of different things. Obviously, we are cross compiling - **-target arm-none-eabi**. As we don't have any operating system we won't be able to use any of the standard newlib supplied system calls - **--disable-newlib-supplied-syscalls**. We tell it where the source code for the library is - **-srcdir=../newlib-cygwin.** We are specifying both the gnu assembler (**as**) and linker (**ld**). We don't want to build lots of versions – **enable-multilib=no**.

This should result in a small amount of output, mainly a series of checks to make sure that the tools and environment are OK and to create some configuration files.



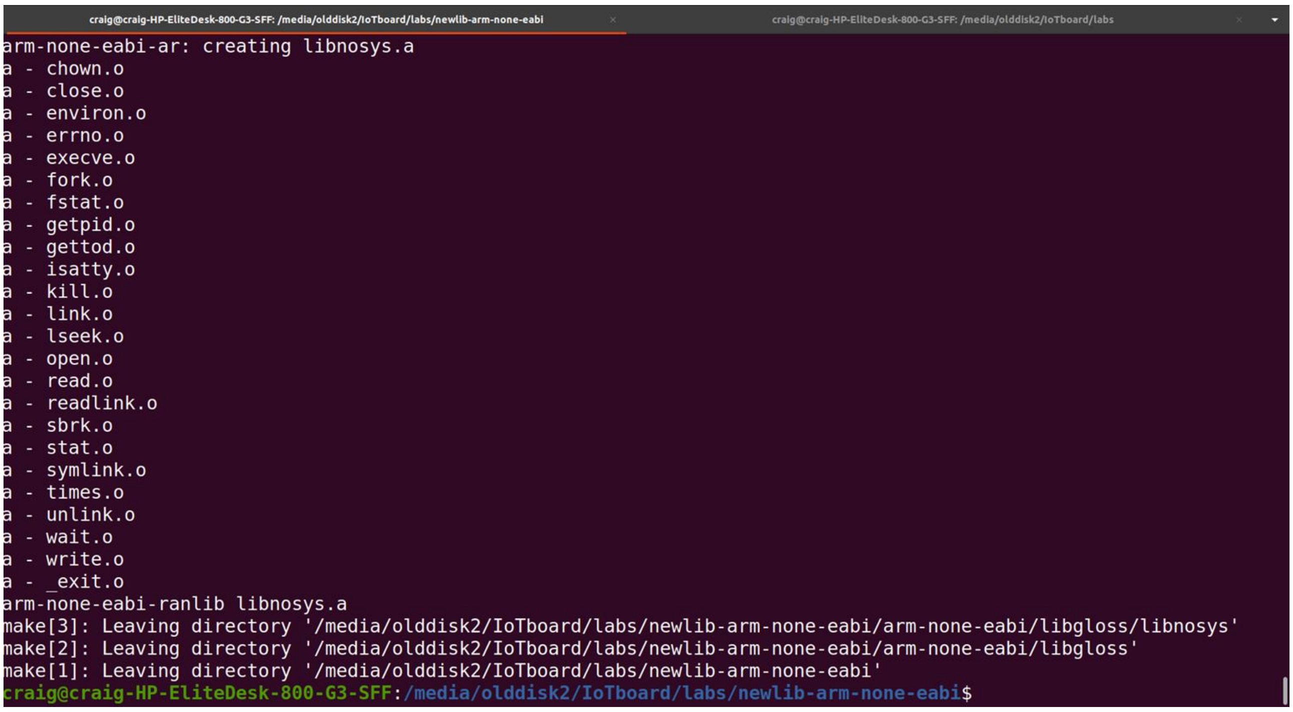
```
checking for arm-none-eabi-ld... arm-none-eabi-ld
checking for arm-none-eabi-lipo... no
checking for arm-none-eabi-nm... arm-none-eabi-nm
checking for arm-none-eabi-objdump... arm-none-eabi-objdump
checking for arm-none-eabi-ranlib... arm-none-eabi-ranlib
checking for arm-none-eabi-readelf... arm-none-eabi-readelf
checking for arm-none-eabi-strip... arm-none-eabi-strip
checking for arm-none-eabi-windres... no
checking for arm-none-eabi-windmc... no
checking where to find the target ar... pre-installed
checking where to find the target as... pre-installed
checking where to find the target cc... pre-installed
checking where to find the target c++... pre-installed
checking where to find the target c++ for libstdc++... pre-installed
checking where to find the target dlltool... pre-installed
checking where to find the target gcc... pre-installed
checking where to find the target gcj... pre-installed
checking where to find the target gfortran... pre-installed
checking where to find the target gccgo... pre-installed
checking where to find the target ld... pre-installed
checking where to find the target lipo... pre-installed
checking where to find the target nm... pre-installed
checking where to find the target objdump... pre-installed
checking where to find the target ranlib... pre-installed
checking where to find the target readelf... pre-installed
checking where to find the target strip... pre-installed
checking where to find the target windres... pre-installed
checking where to find the target windmc... pre-installed
checking whether to enable maintainer-specific portions of Makefiles... no
configure: creating ./config.status
config.status: creating Makefile
craig@craig-HP-Compaq-8200-Elite-SFF-PC:~/stm32/newlib-arm-none-eabi$
```

A successful configure output

In the new directory we should have some configure logs and a makefile. Still in the new **newlib-arm-none-eabi** directory we now need to issue a **make** command to create the libraries. This command is quite long and complex. Before issuing this command it is a good idea to type in the **script** command, this will save all of the terminal's input and output into a file called **typescript**. As there will be a lot of output saving it will help us if we need to debug and look at the messages and output. Once the **make** command has finished, and it will take some time, then you type the **exit** command to stop the output being copied to the typescript file. Here is the **make** command:

```
% make CFLAGS_FOR_TARGET="-ffunction-sections -fdata-sections -DPREFER_SIZE_OVER_SPEED -D__OPTIMIZE_SIZE__ -Os -fomit-frame-pointer -march=armv7e-m+fp -mcpu=cortex-m4 -mthumb -mthumb-interwork -D__thumb2__ -D__BUFSIZ__=256" CCASFLAGS="-march=armv7e-m+fp -mcpu=cortex-m4 -mthumb -mthumb-interwork -D__thumb2__"
```

If the command has been successful, then you should see something like:


craig@craig-HP-EliteDesk-800-G3-SFF: /media/olddisk2/IoTboard/labs/newlib-arm-none-eabi
craig@craig-HP-EliteDesk-800-G3-SFF: /media/olddisk2/IoTboard/labs

```
arm-none-eabi-ar: creating libnosys.a
a - chown.o
a - close.o
a - environ.o
a - errno.o
a - execve.o
a - fork.o
a - fstat.o
a - getpid.o
a - gettod.o
a - isatty.o
a - kill.o
a - link.o
a - lseek.o
a - open.o
a - read.o
a - readlink.o
a - sbrk.o
a - stat.o
a - symlink.o
a - times.o
a - unlink.o
a - wait.o
a - write.o
a - _exit.o
arm-none-eabi-ranlib libnosys.a
make[3]: Leaving directory '/media/olddisk2/IoTboard/labs/newlib-arm-none-eabi/arm-none-eabi/libgloss/libnosys'
make[2]: Leaving directory '/media/olddisk2/IoTboard/labs/newlib-arm-none-eabi/arm-none-eabi/libgloss'
make[1]: Leaving directory '/media/olddisk2/IoTboard/labs/newlib-arm-none-eabi'
craig@craig-HP-EliteDesk-800-G3-SFF:/media/olddisk2/IoTboard/labs/newlib-arm-none-eabi$
```

A successful make command output

The command should also result in the creation of some new directories – **etc** and **arm-none-eabi**. In the latter directory there should be 2 further sub-directories – **newlib** and **libgloss**. **Libgloss** contains all of the 'messy' code that is required to interface to boards, operating systems and CPUs – the type of stuff that is normally glossed over in discussions of creating libraries. **Newlib** contains all of the compiled library code, specifically the main C library **libc** and maths library **libm**. Have a look around these directories to see what code they contain. For example the libc/stdio/ sub-directory contains the standard input and output commands – printf, scanf, puts and so on.

Note: If you get any problems with make there can be a number of causes. Firstly, if you cut and pasted the commands from the PDF it is likely that the PDF format will causes lots of problems – for example it may well add extra new line characters. These changes will totally mess up the compilation. Generally it is better to cut and paste the commands into an edit buffer to remove any format and then read the commands carefully to see if they are correct. If a partial or unsuccessful make or configure command has been issued it is best to delete the **newlib-arm-none-eabi** directory, reconfigure and rerun the, corrected, make command.

If you get errors saying, for example, **texinfo** has not been installed and you will then need to install the missing package. Then you will need to remake and configure the **newlib-arm-none-eabi** directory as it will have been configured with a faulty version of the system. This shouldn't really happen on the UWE systems but can well happen if you are trying this on your own machine.

Test the new newlib library

The observant amongst you might have noticed the changes between worksheet 2 and worksheets 3 & 4. 2 used printf output – so it required some extra object files and a different build command. This is because it was using the libc library to do the output – on worksheets 3 &

4 the only IO was either digital IO – LED and button, or single character input and output. So, these binaries need less code within them. You might have noticed that they didn't need syscalls.o and in the linker command line they had –nolibc and –nostdlib. If you compare the size of the binaries for demo.elf in worksheet 2 with those in worksheet 3 & 4 you will find the latter are quite a bit smaller.

We are now in a position to try to test the new library. However, if we were to try to build with the library it would fail as we haven't yet resolved the issue of the system calls – remember we disabled the default ones above. We now need to provide a number of bits of code for these system calls. The majority of these bits of code will be what are called stubs – meaning bits of code that don't really do very much but are place holders for when we can implement the code properly. In the newlib documentation the list of required system calls is quite small, which is why it is ideal for porting.

Process commands	**File commands**
Execve	close
Fork	fcntl
Getpid	fstat
Kill	isatty
Sbrk	link
Times	lseek
Wait	mkdir
	open
	read
	stat
	unlink
	write

Newlib system calls

As we don't have an operating system we won't be able to provide any of the process commands – apart from **sbrk** which allocates/de-allocates memory to a process. The file system commands will mainly be restricted to our input and output routines as we don't have a file system. In UNIX/Linux parlance we will be restricted to standard input, output and error – stdin, stdout and stderr. These are just terminal IO.

The stubs for this code are provided in the syscalls.c program which is in the Lab4.1 repo – this file is also given in the appendices to this worksheet. Most of the routines just gracefully handle the fact that we can't really do what is requested – so if a new process is asked for then an error message – EAGAIN, is returned. At the start of the code a global variable **errno** is created to hold and communicate these errors. You will notice that the code also gets rid of any earlier versions of **errno** just in case.

The 2 functions we need to focus on are **write** and **sbrk**. Write is pretty simple, although it has a notion of file handles, it can in fact only use stdout and stderr, both attached to the terminal. Write is called with a pointer to a buffer (**char * ptr**) of data and the number of items in that buffer (**int len**). It then loops calling __io_putchar with each character in turn. You will notice that file only refers to STDOUT and STDERR – anything else is considered wrong and ignored.

```
int _write(int file, char *ptr, int len)
        {
                int DataIdx;

                switch (file)
                {
                        case STDOUT_FILENO: /* stdout */
                        case STDERR_FILENO: /* stderr */
                                for (DataIdx = 0; DataIdx < len; DataIdx++)
                                {
                                        __io_putchar(*ptr++);
                                }
                        default:        /* error no such file stream */
                                errno = EBADF;
                                return -1;
                }
                return len;
        }
```

The write system call

Sbrk is a bit more complicated. This routine is related to **malloc/free** and other memory allocators, for example constructors and destructors in C++. This routine must manage the free memory between the end of the code's data segment and the stack, this region is known as the heap. As the stack changes size as the code is executed and there may be multiple calls for memory allocation, this code must be careful. If it gets it wrong then the program may start to get random values on its stack, if we are lucky it will crash!

```
caddr_t _sbrk(int incr)
        {
                extern char end asm("end");
                static char *heap_end;
                char *prev_heap_end;

                if (heap_end == 0)
                        heap_end = &end;
                prev_heap_end = heap_end;
                if (heap_end + incr > stack_ptr)
                {       //write(1, "Heap and stack collision\n", 25);    //abort();
                        errno = ENOMEM;                 return (caddr_t) -1;    }heap_end
                += incr;return (caddr_t) prev_heap_end;}
```

The sbrk system call

This code is monitoring the run time memory to check if the stackj and the heap are not colliding. It uses 2 assembler calls to find out where the current stack pointer is

```
register char * stack_ptr asm("sp");
```

And also where the end of the stack is

```
extern char end asm("end");
```

It then use those value to see if there is enough space for the heap. If there isn't it terminates – there are commented calls to an error message, although it is unlikely that this message would succeed as it will require stack and heap itself!

In order to use these files we will need to make quite a few alterations to our Makefile. Firstly, we will need to tell it where to find the newlib library files and also some header files for any linked in code to use. You will notice that the include files are the includes from the original source files not the newly created versions. If we were to build a release version we would need to clean this up and create directories for the library and include files. The STARTUP variables will also need to include the syscalls.o source file for the syscalls. Finally, we will need some slight changes to the C compiler flags. We need to tell it not to include any standard libraries - **-nostdlib** and we also need to put the newlib includes into the compiler's search paths. The changes that you need to make are given below. However, we want it to include the compiler's library, **libgcc**, as this contains all of the bits of code that the compiler needs to generate code – these tend to be functions like 32 bit multiplication that don't exist in the instruction set and require macros. You will notice in the invocation line, it calls **-lc -lm -lgcc** both at the beginning and the end of the line to make sure that those libraries are searched when necessary.

```
# Library path

HAL = ../Drivers/STM32L4xx_HAL_Driver
CMSIS = ../Drivers/CMSIS/Core
DEVICE = ../Drivers/CMSIS/Device/ST/STM32L4xx/Include
Common_BSP= ../Drivers/BSP/Components/Common
APP

……..

#Search path for peripheral library

VPATH=$(HAL)/Src:$(APP_CODE)

vpath %.c $(APP_CODE)
vpath %.s $(APP_CODE)
vpath %.c $(HAL)/Src

…..

PTYPE           = STM32L475xx
LDSCRIPT        = STM32L475VGTx_FLASH.ld
STARTUP         = startup_stm32l475xx.o system_stm32l4xx.o stm32l4xx_hal_msp.o
                  stm32l4xx_it.o syscalls.o

……

#CompilationFlags

FULLASSERT=-DUSE_FULL_ASSERT
LDFLAGS+=-T$(LDSCRIPT) -mthumb -mcpu=cortex-m4 –nolibc -nostdlib
CFLAGS+=-mcpu=cortex-m4 –mthumb -D$(BOARD) -D$(BOARDTYPE) -D$(PTYPE)
CFLAGS+= -I$(APP)/Inc -I$(HAL)/Inc -I$(CMSIS)/Include -I$(Common_BSP)  -I$(DEVICE)
CFLAGS+= -DUSE_USB_OTG_FS  -D$(BOARD) -DUSE_STDPERIPH_DRIVER $(FULLASSERT)
```

The changes to the makefiles

Now all we need is a program or two to test our new library!

Exercises.

Pass exercise. Create a new main.c which will be a reworking of the one in worksheet 4.1. It will need the comport init code, and the __io_putchar code. Have a small loop that does printf("hello world\n"); a few times.

Pass exercise. Create a piece of code for the read system call. This will need to use the __io_getchar() routine you created in worksheet 4. Write some code to test this routine.

Credit exercise. Write a small maths test program that generates random maths questions and asks the user to input the correct answers. Reward or punish the user as you see fit on their performance. This program will require the use of srand() and rand() to seed and generate the random numbers. Also the input and output should use the formatting features of printf and scanf. You will need to read the newlib documentation to see how these functions are used.

Credit exercise. Write some debug code that prints out data, stack and heap values as the program runs.

A smaller version – newlib_nano

Sys_calls.c

```
/*
 * A set of newlib system call stubs to be used with the arm-none-eabi Olimex
 * board.  This is a hacked version of ones found on line by nanoage and
 * balau82.
 * Craig November 2014
 *
 */


#include <errno.h>
#include <sys/stat.h>
#include <sys/times.h>
#include <sys/unistd.h>
#include <stm32f10x.h>
#undef errno
extern int errno;


/*
 environ
 A pointer to a list of environment variables and their values.
 For a minimal environment, this empty list is adequate:
 */
char *__env[1] = { 0 };
char **environ = __env;

int _write(int file, char *ptr, int len);

void _exit(int status) {
  _write(1, "exit", 4);
  while (1) {
    ;
  }
}

int _close(int file) {
  return -1;
}


/*
 execve
 Transfer control to a new process. Minimal implementation (for a system
 without processes):
 */
int _execve(char *name, char **argv, char **env) {
    errno = ENOMEM;
    return -1;
}


/*
 fork
 Create a new process. Minimal implementation (for a system without processes):
 */
int _fork() {
  errno = EAGAIN;
  return -1;
}
```

```
/*
  fstat
  Status of an open file. For consistency with other minimal implementations
  in these examples, all files are regarded as character special devices.
  The `sys/stat.h' header file required is distributed in the `include' sub-
  directory for this C library.
 */
int _fstat(int file, struct stat *st) {
  st->st_mode = S_IFCHR;
  return 0;
}


/*
  getpid
  Process-ID; this is sometimes used to generate strings unlikely to conflict
  with other processes.
  Minimal implementation, for a system without processes:
 */
int _getpid() {
  return 1;
}


/*
  isatty
  Query whether output stream is a terminal. For consistency with the other
  minimal implementations,
 */
int _isatty(int file) {
  switch (file){
    case STDOUT_FILENO:
    case STDERR_FILENO:
    case STDIN_FILENO:
      return 1;
    default:
      //errno = ENOTTY;
      errno = EBADF;
      return 0;
  }
}


/*
  kill
  Send a signal. Minimal implementation:
 */
int _kill(int pid, int sig) {
  errno = EINVAL;
  return (-1);
}


/*
  link
  Establish a new name for an existing file. Minimal implementation:
 */
int _link(char *old, char *new) {
  errno = EMLINK;
  return -1;
```

```
}



/*
 lseek
 Set position in a file. Minimal implementation:
 */
int _lseek(int file, int ptr, int dir) {
 return 0;
}



/*sbrk
 Increase program data space.
 Malloc and related functions depend on this - requires core_cm3.c from
 the ST preipheral library to  use the __getMSP routine
*/
caddr_t _sbrk(int incr) {
 extern char _ebss; // Defined by the linker
 static char *heap_end;
 char *prev_heap_end;

 if (heap_end == 0) {
   heap_end = &_ebss;
 }
 prev_heap_end = heap_end;

 char * stack = (char*) __get_MSP();
 if (heap_end + incr >  stack)
 {
   _write (STDERR_FILENO, "Heap and stack collision\n", 25);
   errno = ENOMEM;
   return  (caddr_t) -1;
   //abort ();
 }

 heap_end += incr;
 return (caddr_t) prev_heap_end;
}



/*
 read
 Read a character to a file.   returns -1 as no read routine supplied
 */
int _read(int file, char *ptr, int len) {
 errno = EBADF;
 return -1;
}



/*
 stat
 Status of a file (by name). Minimal implementation:
 int    _EXFUN(stat,( const char *__path, struct stat *__sbuf ));
 */
int _stat(const char *filepath, struct stat *st) {
 st->st_mode = S_IFCHR;
```

```
  return 0;
}




/*
  times
  Timing information for current process. Minimal implementation:
 */

clock_t _times(struct tms *buf) {
  return -1;
}


/*
  unlink
  Remove a file's directory entry. Minimal implementation:
 */
int _unlink(char *name) {
  errno = ENOENT;
  return -1;
}


/*
  wait
  Wait for a child process. Minimal implementation:
 */
int _wait(int *status) {
  errno = ECHILD;
  return -1;
}


/*
  write
  Write a character to a file. `libc' subroutines will use this system routine
  for output to all files, including stdout
  Uses outbyte routine
  Returns -1 on error or number of bytes sent
 */
int _write(int file, char *ptr, int len) {
  int n;

  switch (file) {
    case STDOUT_FILENO: /*stdout*/
      for (n = 0; n < len; n++) {
        outbyte(*ptr++ & (u16)0x01FF);
      }
      break;
    case STDERR_FILENO: /* stderr */
      for (n = 0; n < len; n++) {
        outbyte(*ptr++ & (u16)0x01FF);
      }
      break;
    default:
      errno = EBADF;
      return -1;
  }
  return len;
```

```
}
```