Howto: Porting `newlib`

A Simple Guide

Jeremy Bennett
Embecosm

Application Note 9. Issue 1
Publication date July 2010

Legal Notice

Embecosm is the business name of Embecosm Limited, a private limited company registered in England and Wales. Registration number 6577021.

Table of Contents

EMBECOSM

Chapter 1. Introduction

Newlib is a C library intended for use on embedded systems. It is a conglomeration of several library parts, all under free software licenses that make them easily usable on embedded products.

1.1. Target Audience

Porting newlib is not difficult, but advice for the beginner is thin on the ground. This application note is intended for software engineers porting newlib for the first time.

The detail of all the steps needed are covered here, and have been tested using newlib versions 1.17.0 and 1.18.0 with the *OpenRISC 1000*.

For those who already have some experience, the entire porting process is summarized in the final chapter, with links back to the main document (see Chapter 9). It's a useful checklist when carrying out a new port.

1.2. Examples

This application note includes examples from the port of newlib to the *OpenRISC 1000* architecture, originally by Chris Bower, then of Imperial College, London, and subsequently extensively updated by Jeremy Bennett of Embecosm.

The examples are two Board Support Packages (BSP) for use with the *OpenRISC 1000* architectural simulator *Or1ksim* by the same two authors.

At the time of writing the *OpenRISC 1000* implementation is not part of the main newlib distribution. It can be downloaded from OpenCores (www.opencores.org).

1.3. Further information

The main source of information is the newlib website (sourceware.org/newlib). This includes a FAQ, which has brief instructions on porting newlib and documentation for libc [1] and libm, the two libraries making up newlib. The libc documentation is particularly useful, because it lists the system calls which must be implemented by any new port, including minimal implementations.

The newlib **README** is another source of information. Key header files within the source also contain useful commenting, notably **ieeefp.h** and **reent.h**.

There is also a mailing list, <newlib@sourceware.org> where questions can be asked, or new ports submitted.

This application note does not cover the detail of testing newlib on physical hardware. That subject is well covered by Dan Kegel's *Crosstool* project [2].

This application note has drawn heavily on these sources, and the author would like to thank the providers of that original information.

1.4. About Embecosm Application Notes

Embecosm is a consultancy specializing in hardware modeling and open source tool chains for the embedded market. If we can ever be of help, please get in touch at <sales@embecosm.com>.

As part of its commitment to the open source community, Embecosm publishes a series of free and open source application notes, designed to help working engineers with practical problems.

Feedback is always welcome, which should be sent to <info@embecosm.com>.

Chapter 2. `newlib` within the GNU Tool Chain

`Newlib` is intended for use with the GNU tool chain. If `newlib` is included within the build of the GNU tool chain, then all the libraries will be built and installed in the correct places to be found by GCC

2.1. The Unified Source Tree

The three separate packages, `binutils`, GCC and GDB are all taken from a common source tree. GCC and GDB both use many libraries from `binutils`. It is convenient to reassemble that source tree and make a single build of all the tools together.

The easiest way to achieve this is to link all the top level directories in each package into a single unified directory, leaving out any duplicated files or directories.

The following **bash** script will take unpacked distributions of `binutils` GCC and GDB and link them into a single directory, **`srcw`**.

```
#!/bin/bash

component_dirs='binutils-2.18.50 gcc-4.2.2 gdb-6.8'
unified_src=srcw

cd ${unified_src}
ignore_list=". .. CVS .svn"

for srcdir in ${component_dirs}
do
    echo "Component: $srcdir"
    case srcdir
        in
        /* | [A-Za-z]:[\\/]*)
            ;;

        *)
            srcdir="../${srcdir}"
            ;;
    esac

    files=`ls -a ${srcdir}`

    for f in ${files}
    do
        found=

        for i in ${ignore_list}
        do
            if [ "$f" = "$i" ]
            then
                found=yes
            fi
        done

        if [ -z "${found}" ]
        then
            echo "$f             ..linked"
            ln -s ${srcdir}/$f .
        fi
    done

    ignore_list="${ignore_list} ${files}"
done

cd ..
```

The entire tool chain can then be configured and built in a separate directory. The **`configure`** script understands to pass on top level arguments to subsidiary configuration scripts. For example to configure to build a C only tool chain for the 32-bit *OpenRISC 1000* architecture to be installed in **`/opt/or32-elf`**, the following would be appropriate.

```
mkdir build
cd build
../src/configure --target=or32-elf --enable-languages=c --prefix=/opt/or32-elf
cd ..
```

Each tool can be built with its own specific target within that build directory

```
cd build
make all-build all-binutils all-gas all-ld all-gcc all-gdb
cd ..
```

Note

The initial **make** target, **all-build** is used to build some of the baseline libraries and tools used throughout the tool chain.

Similarly the tools can be installed using the following:

```
cd build
make install-build install-binutils install-gas install-ld install-gcc \
     install-gdb
cd ..
```

2.1.1. Incorporating Newlib within the Tool Chain Build

Newlib can be linked into the unified source directory in the same fashion. All that is needed is to add newlib to the component directories in the linking script.

```
#!/bin/bash

component_dirs='binutils-2.18.50 gcc-4.2.2 newlib-1.18.0 gdb-6.8'
unified_src=srcw
...
```

The configuration command should also specify that this is a build using newlib

```
mkdir build
cd build
../src/configure --target=or32-elf --enable-languages=c --with-newlib \
                 --prefix=/opt/or32-elf
cd ..
```

Two new targets are needed for newlib, one to build newlib itself, and one to build any board support packages using libgloss (see Chapter 3 for an explanation of how libgloss is used with newlib).

```
cd build
make all-build all-binutils all-gas all-ld all-gcc all-target-newlib \
     all-target-libgloss all-gdb
cd ..
```

Similarly additional targets are needed for installation.

```
cd build
make install-build install-binutils install-gas install-ld install-gcc \
     install-target-newlib install-target-libgloss install-gdb
cd ..
```

Chapter 3. Overview of newlib

3.1. The relationship between libgloss and newlib

Newlib is now divided into two parts. The main **newlib** directory contains the bulk of the code for the two main libraries, **libc** and **libm**, together with any *architecture* specific code for particular targets.

The **libgloss** directory contains code specific to particular platforms on which the library will be used, generally referred to as the Board Support Package (BSP). Any particular target architecture may have multiple BSPs, for example for different hardware platforms, for a simulator etc.

The target architecture specific code within the **newlib** directory may be very modest - possibly as little as an implementation of **setjmp** and a specification of the IEEE floating point format to use.

The board support package is more complex. It requires an implementation of eighteen system calls and the definition of one global data structure, although the implementation of some of those system calls may be completely trivial.

Note

The separation of BSP implementation into libgloss is relatively recent. Consequently the source tree contains a number of older target implementations where the BSP is entirely within newlib. When looking for examples, be sure to choose an architecture which has been implemented through libgloss. The *OpenRISC 1000* implementation is one such architecture.

3.2. The C Namespace and Reentrant Functions

The BSP implements the system calls—functions like **close**, **write** etc. It is possible for the BSP to implement these directly, but these will then be defined in the main C namespace. It is perfectly permissible for the user to replace these functions, and the user versions take precedence, which requires some care at link time.

Newlib allows the implementer instead to provide namespace clean versions of these functions by prefixing them with an underscore. Newlib will ensure that the system calls map to these namespace clean version (i.e. a call to **close** becomes a call to **_close**) unless the user has reimplemented that function themselves.

A reentrant function may be safely called from a second thread, while a first thread of control is executing. In general a function that modifies no static or global state, will be reentrant.

Many system calls are trivially reentrant. However for some calls, reentrancy is not easy to provide automatically, so reentrant versions are provided. Thus for **close**, there is the reentrant version **close_r**. The reentrant versions take an extra argument, a *reentrancy structure*, which can be used to ensure correct behavior, by providing per-thread versions of global data structures.

It is worth noting that use of the global error value, **errno** is a common source of non-reentrancy. The standard reentrancy structure includes an entry for a per-thread value of **errno**.

For many systems, the issue of reentrancy does not arise. If there is only ever one thread of control, or if separate threads have their own address space there is no problem.

However it's worth remembering that even a bare metal system may encounter issues with reentrancy if event handlers are allowed to use the system calls.

Newlib gives considerable flexibility, particularly where namespace clean versions of the basic system calls are implemented. The implementer can choose to provide implementations of the reentrant versions of the functions. Alternatively newlib can provide reentrancy at the library level, but mapping the calls down the system calls, which are not themselves reentrant. This last can often prove a practical solution to the problem.

3.3. Adding a new Target to Newlib

Adding a new architecture to newlib requires the following steps.

1. Provide a machine specific directory within the **newlib** directory for architecture specific code, notably the **setjmp** implementation.
2. Provide a platform directory for BSP implementation(s) within the **libgloss** directory. The code implementing systems calls for each BSP is placed in this directory.
3. Update the **configure.host** file in the **newlib** directory to point to the machine and platform directories for the new target.

3.3.1. Extending configure.host for a New Target

The **configure.host** file needs changes in two places, to identify the architecture specific machine directory and the platform directory for BSP implementations.

The machine name is specified in a **case** switch on the **${host_cpu}** early on in the file. Add a new **case** entry defining **machine_type** for the architecture. Thus for *OpenRISC 1000* 32-bit architecture we have:

```
or32)
      machine_dir=or32
      ;;
```

This specifies that the machine specific code for this architecture will be found in the directory **newlib/libc/machine/or32**.

The platform directory and details are specified in a subsequent **case** switch on **${host}** (i.e. the full triplet, not just the CPU type). For the 32-bit *OpenRISC 1000* we have the following.

```
or32-*-*)
      syscall_dir=syscalls
      ;;
```

This is the simplest option, specifying that the BSPs for all *OpenRISC 1000* 32-bit targets will implement namespace clean system calls, and rely on newlib to map reentrant calls down to them. The directory name for the BSP implementations will match that of the machine directory, but within the **libgloss** directory. So for *OpenRISC 1000* 32-bit targets; the BSP implementations are in **libgloss/or32**.

There are four common alternatives for specifying how the BSP will be implemented.

1. The implementer defines reentrant namespace clean versions of the system calls. In this case, **syscall_dir** is set to **syscalls** as above, but in addition, **-**

DREENTRANT_SYSCALLS_PROVIDED is added to **newlib_cflags** in **configure.host**. For the *OpenRISC 1000* 32-bit target we could have done this with:

```
or32-*-*)
      syscall_dir=syscalls
      newlib_cflags="${newlib_cflags} -DREENTRANT_SYSCALLS_PROVIDED"
      ;;
```

For convenience, stub versions of the reentrant functions may be found in the **libc/reent** directory. These are in fact the functions used if the reentrant system calls are not provided, and map to the non-reentrant versions.

2. The implementer defines non-reentrant, but namespace clean versions of the system calls. This is the approach we have used with the *OpenRISC 1000* and all the implementer needs to do in this case is to set **syscall_dir** to **syscalls** in **configure.host**. newlib will map reentrant calls down to the non-reentrant versions.

3. The implementer defines non-reentrant, regular versions of the system calls (i.e. **close** rather than **_close**). The library will be neither reentrant, not namespace clean, but will work. In this case, **-DMISSING_SYSCALL_NAMES** is added to **newlib_cflags** in **configure.host**. For the *OpenRISC 1000* we could have done this with:

```
or32-*-*)
      newlib_cflags="${newlib_cflags} -DMISSING_SYSCALL_NAMES"
      ;;
```

Note in particular that **syscall_dir** is not defined in this case.

4. The implementer defines non-reentrant, regular versions of the system calls (i.e. **close** rather than **_close**). The reentrant system calls are mapped onto these functions. The library will not be namespace clean, but will offer reentrancy at the library level. In this case, **-DMISSING_SYSCALL_NAMES** and **-DREENTRANT_SYSCALLS_PROVIDED** are both added to **newlib_cflags** in **configure.host**. For the *OpenRISC 1000* we could have done this with:

```
or32-*-*)
      newlib_cflags="${newlib_cflags} -DMISSING_SYSCALL_NAMES"
      newlib_cflags="${newlib_cflags} -DREENTRANT_SYSCALLS_PROVIDED"
      ;;
```

Note in particular that **syscall_dir** is not defined in this case.

Chapter 4. Modifying `newlib`

Changes that depend on the architecture, and not the particular platform being used, are made in the **`newlib`** directory. These comprise changes to standard headers and custom code for the architecture.

4.1. The Machine Directory

Within the **`newlib`** directory, machine specific code is placed in a target specific directory, **`libc/machine/`*`arch`***.

The only code that has to be there is the implementation of **`setjmp`** and **`longjmp`**, since the implementation of these two functions invariably requires target specific machine code. However any other target specific code may also be placed here.

4.1.1. Updating the Main Machine Directory Configuration files

The machine directory uses GNU *autoconf* and *automake* for configuration. There is a configuration template file (**`configure.in`**) and Makefile template (**`Makefile.am`**) in the main machine directory (**`libc/machine`** within the **`newlib`** directory).

`configure.ac` contains a **`case`** statement configuring the target specific subdirectories. This must be updated to configure the subdirectory for the new target. Thus for the *OpenRISC 1000* we have the following.

```
if test -n "${machine_dir}"; then
  case ${machine_dir} in
        a29k) AC_CONFIG_SUBDIRS(a29k) ;;
        arm) AC_CONFIG_SUBDIRS(arm) ;;

 <other machines not shown>

        necv70) AC_CONFIG_SUBDIRS(necv70) ;;
        or32) AC_CONFIG_SUBDIRS(or32) ;;
        powerpc) AC_CONFIG_SUBDIRS(powerpc) ;;

 <other machines not shown>

        xstormy16) AC_CONFIG_SUBDIRS(xstormy16) ;;
        z8k) AC_CONFIG_SUBDIRS(z8k) ;;
  esac;
fi
```

`Makefile.am` is standard and will not need to be changed. Having changed the configuration template, the configuration file, **`configure`**, will need to be regenerated. This only requires running *autoconf*

```
autoconf
```

Since **`Makefile.am`** has not been changed there is no need to run *automake*

4.1.2. Implementing the `setjmp` and `longjmp` functions.

setjmp and **longjmp** are a pair of C function facilitating cross-procedure transfer of control. Typically they are used to allow resumption of execution at a known good point after an error.

Both take as first argument a buffer, which is used to hold the machine state at the jump destination. When **setjmp** is called it populates that buffer with the current location state (which includes stack and frame pointers and the return address for the call to **setjmp**, and returns zero.

longjmp takes a buffer previously populated by **setjmp**. It also takes a (non-zero) second argument, which will ultimately be the result of the function call. **longjmp** restores the machine state from the buffer. It then jumps to the return address it has just restored, passing its second argument as the result. That return address is the return address from the original call to **setjmp**, so the effect will be as if **setjmp** has just returned with a non-zero argument.

setjmp and **longjmp** are typically used in a top level function in the following way.

```
#include <setjmp.h>

...

  jmp_buf  buf;

  if (0 == setjmp (buf))
    {
      normal processing passing in buf
    }
  else
    {
      error handling code
    }

...
```

During normal processing if an error is found, the state held in **buf** can be used to return control back to the top level using **longjmp**.

```
#include <setjmp.h>

...

  if (error detected)
    {
      longjmp (buf, 1);
    }

...
```

The program will behave as though the original call to **setjmp** had just returned with result 1.

It will be appreciated that this is behavior that cannot usually be written in C. The *OpenRISC 1000* implementation is given as an example. This processor has 32 registers,

r0 through **r31**, each of 32-bits. **r0** is always tied to zero, so need not be saved. **r11** is the function result register, which is always set by **setjmp** and **longjmp**, so also need not be saved. In addition we should save and restore the machine's 32-bit supervision register, which holds the branch flag.

Thus we need the buffer to be 31 32-bit words long. This is defined in the **setjmp** header (see Section 4.2.2).

In the Application Binary Interface (ABI) for the *OpenRISC 1000*, function arguments are passed in registers **r3** through **r8** and the function return address is in **r9**.

When defining these two functions, in assembler, be aware of any prefix conventions used by the C compiler. It is common for symbols defined in C to have an underscore prepended (this is the case for the *OpenRISC 1000*). Thus in this case the assembler should define **_setjmp** and **_longjmp**.

This is the implementation of **setjmp**.

```
        .global _setjmp
_setjmp:
        l.sw    4(r3),r1                /* Slot 0 saved for flag in future */
        l.sw    8(r3),r2
        l.sw    12(r3),r3
        l.sw    16(r3),r4
        l.sw    20(r3),r5
        l.sw    24(r3),r6
        l.sw    28(r3),r7
        l.sw    32(r3),r8
        l.sw    36(r3),r9
        l.sw    40(r3),r10              /* Skip r11 */
        l.sw    44(r3),r12
        l.sw    48(r3),r13
        l.sw    52(r3),r14
        l.sw    56(r3),r15
        l.sw    60(r3),r16
        l.sw    64(r3),r17
        l.sw    68(r3),r18
        l.sw    72(r3),r19
        l.sw    76(r3),r20
        l.sw    80(r3),r21
        l.sw    84(r3),r22
        l.sw    88(r3),r23
        l.sw    92(r3),r24
        l.sw    96(r3),r25
        l.sw    100(r3),r26
        l.sw    104(r3),r27
        l.sw    108(r3),r28
        l.sw    112(r3),r29
        l.sw    116(r3),r30
        l.sw    120(r3),r31

        l.jr    r9
        l.addi  r11,r0,0                /* Zero result */
```


In this simplified implementation, the status flags are not saved—that is a potential future enhancement. All the general registers, with the exception of **r0** (always zero) and **r11** (result register) are saved in the buffer, which, being the first argument, is pointed to by **r3**.

Finally the result register, **r11** is set to zero and the function returns using **r9** (the *OpenRISC 1000* has delayed branches, so the setting of **r11** is placed after the branch to return.).

The implementation of **longjmp** is slightly more complex, since the second argument will be returned as the effective result from **setjmp**, *unless* the second argument is zero in which case 1 is used.

The result must be dealt with first and placed in the result register, **r11**, because the second argument, in **r4** will be subsequently overwritten when the machine state is restored. Similarly we must ensure that **r3**, which holds the first argument pointing to the restore buffer must itself be the last register restored.

```
        .global _longjmp
_longjmp:
        /* Sort out the return value */
        l.sfne  r4,r0
        l.bf    1f
        l.nop

        l.j     2f
        l.addi  r11,r0,1                /* 1 as result */

1:      l.addi  r11,r4,0                /* val as result */

        /* Restore all the other registers, leaving r3 to last. */
2:      l.lwz   r31,120(r3)
        l.lwz   r30,116(r3)
        l.lwz   r29,112(r3)
        l.lwz   r28,108(r3)
        l.lwz   r27,104(r3)
        l.lwz   r26,100(r3)
        l.lwz   r25,96(r3)
        l.lwz   r24,92(r3)
        l.lwz   r23,88(r3)
        l.lwz   r22,84(r3)
        l.lwz   r21,80(r3)
        l.lwz   r20,76(r3)
        l.lwz   r19,72(r3)
        l.lwz   r18,68(r3)
        l.lwz   r17,64(r3)
        l.lwz   r16,60(r3)
        l.lwz   r15,56(r3)
        l.lwz   r14,52(r3)
        l.lwz   r13,48(r3)
        l.lwz   r12,44(r3)
        l.lwz   r10,40(r3)              /* Omit r11 */
        l.lwz   r9,36(r3)
        l.lwz   r8,32(r3)
        l.lwz   r7,28(r3)
        l.lwz   r6,24(r3)
        l.lwz   r5,20(r3)
        l.lwz   r4,16(r3)
        l.lwz   r2,8(r3)                /* Skip r3 */
        l.lwz   r1,4(r3)                /* Slot 0 saved for flag in future */
        l.lwz   r3,12(r3)               /* Now safe */

        /* Result is already in r11. Having restored r9, it will appear as
           though we have returned from the earlier call to _setjmp. The
           non-zero result gives it away though. */
        l.jr    r9
        l.nop
```

The return address, stack pointer and frame pointer having been restored, the return from the function, will place the execution point immediately after the original call to **setjmp**.

The following is a simple test program, which can be used to verify that **setjmp** and **longjmp** are working correctly.

```
#include <setjmp.h>
#include <stdio.h>


void
testit (jmp_buf  env,
        int      prev_res)
{
  int  res = (0 == prev_res) ? prev_res : prev_res + 1;

  printf ("Long jumping with result %d\n", res);
  longjmp (env, res);

}       /* testit () */


int
main (int   argc,
      char *argv[])
{
  jmp_buf  env;

  int  res = setjmp (env);

  printf ("res = 0x%08x\n", res);

  if (res > 1)
    {
      return  0;
    }

  testit (env, res);

  return  256;                  /* We should never actually get here */

}       /* main () */
```

4.1.3. Updating the Target Specific Machine Directory Configuration files

configure.in and **Makefile.am** files also be needed for the target specific directory (i.e. **libc/machine/*target***) within the **newlib** directory). These are generally quite standard, and the easiest approach is to copy the versions used for the *fr30* architecture. Modern practice is to use the file name **configure.ac** rather than **configure.in**, but either will be accepted by *autoconf*.

Makefile.am should be modified if necessary to specify the source files (for example **setjmp.S** and **longjmp.S**). More complex implementations may require modifications to **configure.in** as well.

For example the *OpenRISC 1000* machine directory (**libc/machine/or32** within the **newlib** directory) contains the following.

```
AUTOMAKE_OPTIONS = cygnus

INCLUDES = $(NEWLIB_CFLAGS) $(CROSS_CFLAGS) $(TARGET_CFLAGS)

AM_CCASFLAGS = $(INCLUDES)

noinst_LIBRARIES = lib.a

lib_a_SOURCES = longjmp.S setjmp.S
lib_a_CCASFLAGS=$(AM_CCASFLAGS)
lib_a_CFLAGS=$(AM_CFLAGS)

ACLOCAL_AMFLAGS = -I ../../.. -I ../../../..
CONFIG_STATUS_DEPENDENCIES = $(newlib_basedir)/configure.host
```

After any changes it will be necessary to run *autoconf* and/or *automake* to generate new versions of **configure** and **Makefile.in**. *autoconf* requires a number of newlib specific macros. These can be generated from the main newlib include file (**acinclude.m4**) by running *aclocal*. The full set of commands would be.

```
aclocal -I ../../..
autoconf
automake --cygnus Makefile
```

aclocal only need to be run the first time the directory is created, or when moving the directory to a new release of newlib. *autoconf* need only be run each time **configure.in** (or **configure.ac**) is changed. *automake* need only be run each time **Makefile.am** is changed.

4.2. Changing Headers

There are two places, where header definitions must be modified for a new target architecture: the specification of the IEEE floating point format used, and the specification of the **setjmp** buffer size.

4.2.1. IEEE Floating Point

The floating point format is specified within the **newlib** directory in **libc/include/machine/ieeefp.h**. Details of how the IEEE 754 format is implemented, and variations from the standard, are specified by defining a number of C macros.

- **__IEEE_BIG_ENDIAN**
  Define this macro if the floating point format is big endian.

  

  Caution
  One, and only one of **__IEEE_BIG_ENDIAN** and **__IEEE_LITTLE_ENDIAN** must be defined.

- **__IEEE_LITTLE_ENDIAN**

Define this macro if the floating point format is little endian.

Caution

One, and only one of **__IEEE_LITTLE_ENDIAN** and **__IEEE_BIG_ENDIAN** must be defined.

- **__IEEE_BYTES_LITTLE_ENDIAN**
  Define this macro in addition to **__IEEE_BIG_ENDIAN**, where the words of a multi-word IEEE floating point number are in big endian order, but the bytes within each word are in little endian order.

- **_DOUBLE_IS_32BITS**
  Define this if double precision floating point is represented using the 32-bit IEEE representation.

- **_FLOAT_ARG**
  Floating point arguments are usually promoted to double when passed as arguments. If this is not the case, then this macro should be defined to the type actually used to pass floating point arguments.

- **_FLT_LARGEST_EXPONENT_IS_NORMAL**
  Define this if the floating point format uses the largest exponent for finite numbers rather than NaN and infinities. Such a format cannot represent NaNs or infinities, but it's **FLT_MAX** is twice the standard IEEE value.

- **_FLT_NO_DENORMALS**
  Define this if the floating point format does not support IEEE denormalized numbers. In this case, every floating point number with a zero exponent is treated as a zero representation.

Caution

Two of these macros (**_FLT_LARGEST_EXPONENT_IS_NORMAL** and **_FLT_NO_DENORMALS**) specify deviations from IEEE 754. These macros only work with single-precision floating point and may not work correctly if hardware floating point support is used (enabled by configuring with **--enable-newlib-hw-fp**).

For most targets it is sufficient to define just one of **__IEEE_BIG_ENDIAN** or **__IEEE_LITTLE_ENDIAN**. The definitions should always be surrounded by a conditional, so they are only used when the target architecture is selected. For example the *OpenRISC 1000* is big-endian, so we add the following to the header file.

```
#if defined(__or32__)
#define __IEEE_BIG_ENDIAN
#endif
```

4.2.2. `setjmp` Buffer Size

The implementation of **setjmp** and **longjmp** made use of a buffer to hold the machine state. The size of that buffer is architecture dependent and specified within the **newlib** directory in **libc/include/machine/setjmp.h**.

The header specifies the number of entries in the buffer and the size of each entry (as a C type). So for the *OpenRISC 1000* we use the following.

```
#if defined(__or32__)
/* Enough space for all regs except r0 and r11 and the status register */
#define _JBLEN 31
#define _JBTYPE unsigned long
#endif
```

As before, the definition is within a conditional, so it is only used when the target is the *OpenRISC 1000* 32-bit architecture.

The type **jmp_buf** used with **setjmp** and **longjmp** is then defined as:

```
typedef _JBTYPE jmp_buf[_JBLEN];
```

4.2.3. Miscellaneous System Definitions

Various system wide constants are specified within the **newlib** directory in **libc/include/sys/config.h**.

Very often the system default values are quite sufficient (this is the case for the *OpenRISC 1000*). However target specific overrides of these values can be provided at the end of this file. This file is included in all other source files, so can be used to redefine any of the constants used in the system. The existing file gives numerous examples from different machines.

Caution

A number of the constants defined here mirror those in GCC's **limits.h** file. They should be kept consistent.

4.2.4. Overriding Other Header Files

If other headers must be overridden (not usually necessary with a simple port), then the new versions can be placed in **libc/machine/*arch*/machine** within the **newlib** directory. These header files will be used in preference to those in the standard distribution's **machine** header directory.

Chapter 5. Modifying `libgloss`

Any target architecture may need multiple implementations, suited to different platforms on which the code may run. The connection between the library and a specific platform is known as a Board Support Package (BSP). In recent versions of `newlib`, BSPs are separated out into their own library, `libgloss`, the source for which is in the top level **`libgloss`** directory.

For `newlib` the BSP within `libgloss` comprises an implementation of the C runtime initialization, **`crt0.o`**, a definition of one global data structure, and implementation of eighteen system calls for each platform.

Note
`libgloss` is a relatively new addition to `newlib`. Some older ports still have the BSP code within the **`newlib`** directory.

5.1. The Platform Directory

A directory is created in the **`libgloss`** directory corresponding to the machine directory created in the **`newlib/libc/machine`** directory (see Chapter 4).

This directory will hold the source code for the C runtime initialization (**`crt0.o`**) and for the system calls for each BSP

5.1.1. Ensuring the Platform Directory is Configured

`configure.in` within the **`libgloss`** directory includes a **`case`** statement configuring the target for each target platform. This should be extended to add the new platform directory. The *OpenRISC 1000* 32-bit target requires the following change.

```
case "${target}" in
  i[[3456]]86-*-elf* | i[[3456]]86-*-coff*)
        AC_CONFIG_SUBDIRS([i386])
        ;;
  m32r-*-*)
        AC_CONFIG_SUBDIRS([m32r])
        ;;

  <Other targets not shown>

  spu-*-elf)
        AC_CONFIG_SUBDIRS([spu])
        config_testsuite=false
        config_libnosys=false
        ;;
  or32-*-*)
        AC_CONFIG_SUBDIRS(or32)
        ;;
  iq2000-*-*)
        AC_CONFIG_SUBDIRS([iq2000])
        ;;
esac
```

After making this change the **configure** file should be regenerated by running *autoconf*.

5.2. The C Runtime Initialization, crt0.o

The C Runtime system must carry out the following tasks.

- Set up the target platform in a consistent state. For example setting up appropriate exception vectors.
- Initialize the stack and frame pointers
- Invoke the C constructor initialization and ensure destructors are called on exit.
- Carry out any further platform specific initialization.
- Call the C **main** function.
- Exit with the return code supplied if the C **main** function ever terminates.

The code is invariably assembler, although it may call out to C functions, and is best illustrated by example from the *OpenRISC 1000*. This is a BSP designed for use with a fast architectural simulator. It comes in two variants, one providing just standard output to the console, the other implementing a simulated UART with both standard input and standard output. The **crt0.0** is common to both BSPs and found in **crt0.S**.

5.2.1. Exception vector setup

The first requirement is to populate the exception vectors. The *OpenRISC 1000* uses memory from **0x0** to **0x1fff** for exception vectors, with vectors placed **0x100** bytes apart. Thus a reset exception will jump to **0x100**, a bus error exception to **0x200** and so on.

In this simple BSP, the vast majority of exceptions are not supported. If they are received, they print out (using **printf**) identification of the exception and the address which caused it to the simulator console, and then exit. We provide a macro for that assembly code, since it will be reused many times.

```
#define UNHANDLED_EXCEPTION(str)                                        \
        l.addi  r1,r1,-20               /* Standard prologue */         ;\
        l.sw    16(r1),r2                                               ;\
        l.addi  r2,r1,20                                                ;\
        l.sw    12(r1),r9                                               ;\
                                                                        ;\
        l.movhi r3,hi(.Lfmt)            /* printf format string */      ;\
        l.ori   r3,r3,lo(.Lfmt)                                         ;\
        l.sw    0(r1),r3                                                ;\
        l.movhi r4,hi(str)              /* Name of exception */         ;\
        l.ori   r4,r4,lo(str)                                           ;\
        l.sw    4(r1),r4                                                ;\
        l.mfspr r5,r0,SPR_EPCR_BASE     /* Source of the interrupt */   ;\
        l.jal   _printf                                                 ;\
        l.sw    8(r1),r5                                                ;\
                                                                        ;\
        l.ori   r3,r0,0xffff            /* Failure RC */                ;\
        l.jal   _exit                                                   ;\
        l.nop                                                           ;\
                                                                        ;\
        l.rfe                           /* Never executed we hope */
```

The call to **printf** is expected to use a standard format string (at the label **.Lfmt**) which requires two other arguments, an identification string (labeled by the parameter **st** to the macro) and the program counter where the exception occurred (loaded from Special Purpose Register **SPR_EPCR_BASE**). Return from exception is provided as a formality, although the call to **exit** means that we should never execute it.

Note that compiled C functions have their names prepended by underscore on the *OpenRISC 1000*. It is these names that must be used from the assembler code.

The format and identification strings are read only data.

```
        .section .rodata
.Lfmt:  .string "Unhandled %s exception at address %08p\n"
.L200:  .string "bus error"
.L300:  .string "data page fault"
.L400:  .string "instruction page fault"
.L500:  .string "timer"
.L600:  .string "alignment"
.L700:  .string "illegal instruction"
.L800:  .string "external interrupt"
.L900:  .string "data TLB"
.La00:  .string "instruction TLB"
.Lb00:  .string "range"
.Lc00:  .string "syscall"
.Ld00:  .string "floating point"
.Le00:  .string "trap"
.Lf00:  .string "undefined 0xf00"
.L1000: .string "undefined 0x1000"
.L1100: .string "undefined 0x1100"
.L1200: .string "undefined 0x1200"
.L1300: .string "undefined 0x1300"
.L1400: .string "undefined 0x1400"
.L1500: .string "undefined 0x1500"
.L1600: .string "undefined 0x1600"
.L1700: .string "undefined 0x1700"
.L1800: .string "undefined 0x1800"
.L1900: .string "undefined 0x1900"
.L1a00: .string "undefined 0x1a00"
.L1b00: .string "undefined 0x1b00"
.L1c00: .string "undefined 0x1c00"
.L1d00: .string "undefined 0x1d00"
.L1e00: .string "undefined 0x1e00"
.L1f00: .string "undefined 0x1f00"
```

The first executable code is for the exception vectors. These must go first in memory, so are placed in their own section, **.vectors**. The linker/loader will ensure this this code is placed first in memory (see Section 7.3).

The reset vector just jumps to the start code. The code is too large to sit within the **0x100** bytes of an exception vector entry, and is placed in the main text space, in function **_start**.

```
        .section .vectors,"ax"

        /* 0x100: RESET exception */
        .org    0x100
_reset:
        /* Jump to program initialisation code */
        l.movhi r2,hi(_start)
        l.ori   r2,r2,lo(_start)
        l.jr    r2
        l.nop
```

The second vector, at address **0x200** is the bus error exception vector. In normal use, like all other exceptions it it causes exit and uses the **UNHANDLED_EXCEPTION** macro.

However during start up, the code tries deliberately to write out of memory, to determine the end of memory, which will trigger this bus exception. For this a simple exception handler, which just skips the offending instruction is required.

The solution is to place this code first, followed by the unhandled exception code. Once the end of memory has been located, the initial code can be overwritten by **l.nop** opcodes, so the exception will drop through to the **UNHANDLED_EXCEPTOON** code.

```
        .org    0x200
_buserr:
        l.mfspr r24,r0,SPR_EPCR_BASE
        l.addi  r24,r24,4               /* Return one instruction on */
        l.mtspr r0,r24,SPR_EPCR_BASE
        l.rfe

_buserr_std:
        UNHANDLED_EXCEPTION (.L200)
```

No effort is made to save the register (**r24**) that is used in the handler. The start up code testing for end of memory must not use this register.

The next exception, data page fault, at location 0x300, like all other exceptions is unhandled.

```
        .org    0x300
        UNHANDLED_EXCEPTION (.L300)
```

5.2.2. The `_start` Function and Stack Initialization

The *OpenRISC 1000* ABI uses a falling stack. The linker will place code and static data at the bottom of memory (starting with the exception vectors). The heap then starts immediately after this, while the stack grows down from the end of memory.

The linker will supply the address for the start of heap (it is in the global variable **end**). However we must find the stack location by trying to write to memory above the heap to determine the end of memory. Rather than write to every location, the code assumes memory is a multiple of 64KB, and tries writing to the last word of each 64KB block above **end** until the value read back fails.

This failure will trigger a bus error exception, which must be handled (see Section 5.2.1). The address used for the start of the stack (which is also the last word of memory) is stored in a global location, **_stack** (which C will recognize as **stack**).

```
        .section .data
        .global _stack
_stack: .space  4,0
```

_start is declared so it looks like a C function. GDB knows that **_start** is special, and this will ensure that backtraces do not wind back further than **main**. It is located in ordinary text space, so will be placed with other code by the linker/loader.

```
        .section .text
        .global _start
        .type   _start,@function
_start:
```

The first memory location to test is found by rounding the **end** location down to a multiple of 64KB, then taking the last word of the 64KB above that. **0xaaaaaaaa** is used as the test word to write to memory and read back.

```
        l.movhi r30,hi(end)
        l.ori   r30,r30,lo(end)
        l.srli  r30,r30,16              /* Round down to 64KB boundary */
        l.slli  r30,r30,16

        l.addi  r28,r0,1                /* Constant 64KB in register */
        l.slli  r28,r28,16

        l.add   r30,r30,r28
        l.addi  r30,r30,-4              /* SP one word inside next 64KB? */

        l.movhi r26,0xaaaa              /* Test pattern to store in memory */
        l.ori   r26,r26,0xaaaa
```

Each 64KB block is tested by writing the test value and reading back to see if it matches.

```
.L3:
        l.sw    0(r30),r26
        l.lwz   r24,0(r30)
        l.sfeq  r24,r26
        l.bnf   .L4
        l.nop

        l.j     .L3
        l.add   r30,r30,r28             /* Try 64KB higher */

.L4:
```


The previous value is then the location to use for end of stack, and should be stored in the **_stack** location.

```
        l.sub   r30,r30,r28             /* Previous value was wanted */
        l.movhi r26,hi(_stack)
        l.ori   r26,r26,lo(_stack)
        l.sw    0(r26),r30
```

The stack pointer (**r1**) and frame pointer (**r2**) can be initialized with this value.

```
        l.add   r1,r30,r0
        l.add   r2,r30,r0
```

Having determined the end of memory, there is no need to handle bus errors silently. The words of code between **_buserr** and **_buserr_std** can be replaced by **l.nop**.

```
        l.movhi r30,hi(_buserr)
        l.ori   r30,r30,lo(_buserr)
        l.movhi r28,hi(_buserr_std)
        l.ori   r28,r28,lo(_buserr_std)
        l.movhi r26,0x1500              /* l.nop 0 */
        l.ori   r26,r26,0x0000

.L5:
        l.sfeq  r28,r30
        l.bf    .L6
        l.nop

        l.sw    0(r30),r26              /* Patch the instruction */
        l.j     .L5
        l.addi  r30,r30,4               /* Next instruction */

.L6:
```

Note

It is essential that this code is before any data or instruction cache is initialized. Otherwise more complex steps would be required to enforce data write back and invalidate any instruction cache entry.

5.2.3. Cache Initialization

The OpenRISC 1000 has optional instruction and data caches. If these are declared (in the **or1ksim-board.h** header), then they must be enabled by setting the appropriate bit in the supervision register.

This is an example of machine specific initialization.

```
        /* Cache initialisation. Enable IC and/or DC */
.if IC_ENABLE || DC_ENABLE
        l.mfspr r10,r0,SPR_SR
.if IC_ENABLE
        l.ori   r10,r10,SPR_SR_ICE
.endif
.if DC_ENABLE
        l.ori   r10,r10,SPR_SR_DCE
.endif
        l.mtspr r0,r10,SPR_SR
        l.nop                           /* Flush the pipeline. */
        l.nop
        l.nop
        l.nop
        l.nop
.endif
```

5.2.4. Clearing BSS

BSS is the area of memory used to hold static variables which must be initialized to zero. Its start and end are defined by two variables from the linker/loader, **__bss_start** and **end** respectively.

```
        l.movhi r28,hi(__bss_start)
        l.ori   r28,r28,lo(__bss_start)
        l.movhi r30,hi(end)
        l.ori   r30,r30,lo(end)

.L1:
        l.sw    (0)(r28),r0
        l.sfltu r28,r30
        l.bf    .L1
        l.addi  r28,r28,4
```

5.2.5. Constructor and Destructor Handling

GCC may require constructors to be initialized at start up and destructors to be called on exit. This behavior is captured in the GCC functions **__do_global_ctors** and **__do_global_dtors**. There is some complexity associated with this functionality, since there may be separate lists for the main code and shared libraries that are dynamically loaded.

It is usual to wrap this functionality in two functions, **init** and **fini**, which are placed in their own sections, **.init** and **.fini**. The **.init** section is loaded before all other text sections and the **.fini** section after all other text sections.

The start up code should call **init** to handle any constructors.

```
        l.jal   init
        l.nop
```

The **fini** function is passed to the library function **_atexit** to ensure it is called on a normal exit.

```
        l.movhi r3,hi(fini)
        l.jal   _atexit
        l.ori   r3,r3,lo(fini)          /* Delay slot */
```

5.2.6. C Initialization Functions

Now that the C infrastructure is set up, it is appropriate to call any C functions that are used during initialization. In the *OpenRISC 1000* case this is a function to initialize a UART. Only one version of the library actually has a UART. However it is easiest to substitute a dummy version of the initialization function in the version of the library without a UART, rather than making this function conditional.

```
        l.jal    __uart_init
        l.nop
```

5.2.7. Invoking the main program

The final stage is to call the main program. In this simple implementation there is no mechanism to pass arguments or environments to **main**, so the arguments **argc**, **argv** and **env** (in **r3**, **r4** and **r5**) are set to **0**, **NULL** and **NULL** respectively.

```
        l.or    r3,r0,r0
        l.or    r4,r0,r0
        l.jal   _main
        l.or    r5,r0,r0                /* Delay slot */
```

If the main program returns, its result (held in **r11** on the *OpenRISC 1000*) will be a return code from the program, which we pass to the **exit**.

```
        l.jal   _exit
        l.addi  r3,r11,0                /* Delay slot */
```

exit should not return, but just in case, we can put the processor in a tight loop at this stage, in order to ensure consistent behavior.

```
.L2:
        l.j     .L2
        l.nop
```

5.3. Standard System Call Implementations

The simplest way to provide a board support package is to implement the 18 system calls in non-reentrant fashion. For many bare metal implementations this is sufficient.

The simplest possible BSP supports just output to standard output and non input. We give the minimal implementation for such a system.

Where appropriate, we also show the *OpenRISC 1000* implementation as a practical example.

This section duplicates much of the information found in the `newlib libc` documentation [1]. It is included here for completeness.

5.3.1. Error Handling

Many functions set an error code on failure in the global variable, **`errno`**.

There is a slight complication with `newlib`, because **`errno`** is not implemented as a variable, but a macro (this make life easier for reentrant functions).

The solution for standard system call implementations, which must return an error code is to undefine the macro and use the external variable instead. At the head of such functions use the following.

```
#include <errno.h>
#undef errno
extern int errno;
```

Note

`errno` is a global variable, so changing it will immediately make a function non-reentrant.

5.3.2. The Global Environment, `environ`

The global variable, **`environ`** must point to a null terminated list of environment variable name-value pairs.

For a minimal implementation it is sufficient to use an empty list as follows.

```
char *__env[1] = { 0 };
char **environ = __env;
```

5.3.3. Exit a program, `_exit`

Exit a program without any cleanup.

The *OpenRISC 1000*s implementation makes use of the **`l.nop`** opcode. This opcode takes a 16-bit immediate operand. Functionally the operand has no effect on the processor itself. However a simulator can inspect the operand to provide additional behavior external to the machine.

When executing on *Or1ksim*, **`l.nop 1`** causes a tidy exit of the simulator, using the value in **`r3`** as the return code.

```
void
_exit (int  rc)
{
  register int  t1 asm ("r3") = rc;

  asm volatile ("\tl.nop\t%0" : : "K" (NOP_EXIT), "r" (t1));

  while (1)
    {
    }
}       /* _exit () */
```

Note the use of **volatile**. Otherwise there is a strong possibility of an optimizing compiler recognizing that this opcode does nothing (we are relying on a simulation side-effect) and removing it.

Caution

The name of this function is already namespace clean. If a namespace clean implementation of the system calls has been specified in **configure.host** (see Section 3.3.1), then this function is still named **_exit**, not **__exit**.

5.3.4. Closing a file, close

For a namespace clean function, implement **_close**, otherwise implement **close**. The detailed implementation will depend on the file handling functionality available.

In the minimal implementation, this function always fails, since there is only standard output, which is not a valid file to close. This implementation is sufficient for the *OpenRISC 1000*.

```
#include <errno.h>

#undef errno
extern int  errno;

int
_close (int   file)
{
  errno = EBADF;

  return -1;                    /* Always fails */

}       /* _close () */
```

5.3.5. Transfer Control to a New Process, execve

For a namespace clean function, implement **_execve**, otherwise implement **execve**. The implementation of this functionality will be tightly bound to any operating infrastructure for handling multiple processes.

A minimal implementation, such as that for bare metal coding, only offers a single user thread of control. It is thus impossible to start a new process, so this function always fails.

```
#include <errno.h>

#undef errno;
extern int  errno;

int
_execve (char  *name,
         char **argv,
         char **env)
{
  errno = ENOMEM;
  return -1;                    /* Always fails */

}       /* _execve () */
```

The choice of **errno** is somewhat arbitrary. However no value for "no processes available" is provided, and **ENOMEM** is the closest in meaning to this.

5.3.6. Create a new process, fork

For a namespace clean function, implement **_fork**, otherwise implement **fork**. The implementation of this functionality will be tightly bound to any operating infrastructure for handling multiple processes.

A minimal implementation, such as that for bare metal coding, only offers a single user thread of control. It is thus impossible to start a new process, so this function always fails.

```
#include <errno.h>

#undef errno
extern int  errno;

int
_fork ()
{
  errno = EAGAIN;
  return -1;                    /* Always fails */

}       /* _fork () */
```

The choice of **errno** is again somewhat arbitrary. However no value for "no processes available" is provided, and **EAGAIN** is the closest in meaning to this.

5.3.7. Provide the Status of an Open File, fstat

For a namespace clean function, implement **_fstat**, otherwise implement **fstat**. The detailed implementation will depend on the file handling functionality available.

A minimal implementation should assume that all files are character special devices and populate the status data structure accordingly.

```
#include <sys/stat.h>

int
_fstat (int          file,
        struct stat *st)
{
  st->st_mode = S_IFCHR;
  return  0;

}       /* _fstat () */
```

The *OpenRISC 1000* implementation requires two versions of this, one for the BSP using the console for output and one for the BSP using a UART and supporting both standard input and standard output.

Without a UART, the implementation still checks that the file descriptor is one of the two that are supported, and otherwise returns an error.

```
#include <errno.h>
#include <sys/stat.h>
#include <unistd.h>

#undef errno
extern int  errno;

int
_fstat (int          file,
        struct stat *st)
{
  if ((STDOUT_FILENO == file) || (STDERR_FILENO == file))
    {
      st->st_mode = S_IFCHR;
      return  0;
    }
  else
    {
      errno = EBADF;
      return  -1;
    }
}       /* _fstat () */
```

The implementation when a UART is available is almost identical, except that **STDIN_FILENO** is also an acceptable file for which status can be provided.

5.3.8. Get the Current Process ID, `getpid`

For a namespace clean function, implement **_getpid**, otherwise implement **getpid**. The implementation of this functionality will be tightly bound to any operating infrastructure for handling multiple processes.

For a minimal implementation, with no processes, this can just return a constant. It is perhaps safer to return one rather than zero, to avoid issue with software that believes process zero is something special.

```
int
_getpid ()
{
  return  1;                            /* Success */

}       /* _getpid () */
```

5.3.9. Determine the Nature of a Stream, `isatty`

For a namespace clean function, implement **_isatty**, otherwise implement **isatty**. The detailed implementation will depend on the file handling functionality available.

This specifically checks whether a stream is a terminal. The minimal implementation only has the single output stream, which is to the console, so always returns 1.


```
int
_isatty (int   file)
{
  return  1;

}       /* _isatty () */
```

Caution

Contrary to the standard `libc` documentation, this applies to any stream, not just output streams.

The *OpenRISC 1000* version gives a little more detail, setting **`errno`** if the stream is not standard output, standard error or (for the UART version of the BSP) standard input.

```
#include <errno.h>
#include <unistd.h>

#undef ERRNO
extern int  errno;

int
_isatty (int   file)
{
  if ((file == STDOUT_FILENO) || (file == STDERR_FILENO))
    {
      return  1;
    }
  else
    {
      errno = EBADF;
      return  -1;
    }
}       /* _isatty () */
```

The UART version is almost identical, but also succeeds for standard input.

5.3.10. Send a Signal, `kill`

For a namespace clean function, implement **`_kill`**, otherwise implement **`kill`**. The implementation of this functionality will be tightly bound to any operating infrastructure for handling multiple processes.

A minimal implementation has no concept of either signals, nor of processes to receive those signals. So this function should always fail with an appropriate value in **`errno`**.

```
#include <errno.h>

#undef errno
extern int  errno;

int
_kill (int  pid,
       int  sig)
{
  errno = EINVAL;
  return -1;                    /* Always fails */

}       /* _kill () */
```

5.3.11. Rename an existing file, `link`

For a namespace clean function, implement **`_link`**, otherwise implement **`link`**. The detailed implementation will depend on the file handling functionality available.

A minimal implementation has no file system, so this function must always fail, with an appropriate value set in **`errno`**.

```
#include <errno.h>

#undef errno
extern int  errno;

int
_link (char *old,
       char *new)
{
  errno = EMLINK;
  return -1;                    /* Always fails */

}       /* _link () */
```

5.3.12. Set Position in a File, `lseek`

For a namespace clean function, implement **`_lseek`**, otherwise implement **`lseek`**. The detailed implementation will depend on the file handling functionality available.

A minimal implementation has no file system, so this function can return 0, indicating that the only stream (standard output) is positioned at the start of file.

```
#include <errno.h>

#undef errno
extern int  errno;

int
_lseek (int   file,
        int   offset,
        int   whence)
{
  return  0;

}       /* _lseek () */
```

The *OpenRISC 1000* version is a little more detailed, returning zero only if the stream is standard output, standard error or (for the UART version of the BSP) standard input. Otherwise -1 is returned and an appropriate error code set in **errno**.

```
#include <errno.h>
#include <unistd.h>

#undef errno
extern int  errno;

int
_lseek (int   file,
        int   offset,
        int   whence)
{
  if ((STDOUT_FILENO == file) || (STDERR_FILENO == file))
    {
      return  0;
    }
  else
    {
      errno = EBADF;
      return  (long) -1;
    }
}       /* _lseek () */
```

The UART version is almost identical, but also succeeds for standard input.

5.3.13. Open a file, open

For a namespace clean function, implement **_open**, otherwise implement **open**. The detailed implementation will depend on the file handling functionality available.

A minimal implementation has no file system, so this function must always fail, with an appropriate error code set in **errno**.


```
#include <errno.h>

#undef errno
extern int  errno;

int
_open (const char *name,
       int         flags,
       int         mode)
{
  errno = ENOSYS;
  return -1;                    /* Always fails */

}       /* _open () */
```

5.3.14. Read from a File, `read`

For a namespace clean function, implement **_read**, otherwise implement **read**. The detailed implementation will depend on the file handling functionality available.

A minimal implementation has no file system. Rather than failing, this function returns 0, indicating end-of-file.

```
#include <errno.h>

#undef errno
extern int  errno;

int
_read (int   file,
       char *ptr,
       int   len)
{
  return  0;                            /* EOF */

}       /* _read () */
```

The *OpenRISC 1000* BSP without a UART is very similar to the minimal implementation, but checks that the stream is standard input before returning 0. For all other streams it returns an error.

```
#include <errno.h>
#include <unistd.h>

#undef errno
extern int  errno;

int
_read (int   file,
       char *ptr,
       int   len)
{
  if (STDIN_FILENO == file)
    {
      return  0;                          /* EOF */
    }
  else
    {
      errno = EBADF;
      return  -1;
    }
}       /* _read () */
```

The *OpenRISC 1000* BSP with a UART is more complex. In this case, if the stream is standard input, a character is read (and optionally echoed) from the UART.

```
#include <errno.h>
#include <unistd.h>

#undef errno
extern int  errno;

int
_read (int   file,
       char *buf,
       int   len)
{
  if (STDIN_FILENO == file)
    {
      int  i;

      for (i = 0; i < len; i++)
 {
   buf[i] = _uart_getc ();
#ifdef UART_AUTO_ECHO
   _uart_putc (buf[i]);
#endif
   /* Return partial buffer if we get EOL */
   if ('\n' == buf[i])
     {
       return  i;
     }
 }

      return  i;   /* Filled the buffer */
    }
  else
    {
      errno = EBADF;
      return  -1;
    }
} /* _read () */
```

Caution

The *Or1ksim* UART implementation only returns data when carriage return is hit, rather than as each character becomes available, which can lead to some unexpected behavior.

5.3.15. Allocate more Heap, `sbrk`

For a namespace clean function, implement **`_sbrk`**, otherwise implement **`sbrk`**. This is one function for which there is no default minimal implementation. It is important that it is implemented wherever possible, since **`malloc`** depends on it, and in turn many other functions depend on **`malloc`**. In this application note, the *OpenRISC 1000* implementation is used as an example.

As noted earlier (Section 5.2.2), the heap on the *OpenRISC 1000* grows up from the end of loaded program space, and the stack grows down from the top of memory. The linker defines

the symbol **_end**, which will be the start of the heap, whilst the C runtime initialization places the address of the last work in memory in the global variable **_stack**.

Caution
_end is a symbol defined by the linker, not a variable, so it is its *address* that must be used, not its value.

Within a C program these two variables are referred to without their leading underscore—the C compiler prepends all variable names with underscore.

```
#include <errno.h>

#undef errno
extern int  errno;

#define STACK_BUFFER  65536     /* Reserved stack space in bytes. */

void *
_sbrk (int nbytes)
{
  /* Symbol defined by linker map */
  extern int  end;              /* start of free memory (as symbol) */

  /* Value set by crt0.S */
  extern void *stack;           /* end of free memory */

  /* The statically held previous end of the heap, with its initialization. */
  static void *heap_ptr = (void *)&end;         /* Previous end */

  if ((stack - (heap_ptr + nbytes)) > STACK_BUFFER )
    {
      void *base  = heap_ptr;
      heap_ptr   += nbytes;

      return  base;
    }
  else
    {
      errno = ENOMEM;
      return  (void *) -1;
    }
}       /* _sbrk () */
```

The program always tries to keep a minimum of 65,536 (2^{16}) bytes spare for the stack.

Note
This implementation defines **_sbrk** as returning type **void *****. The standard newlib documentation uses return type **caddr_t**, which is defined in **unistd.h**. The author believes that **void *** is now the recommended return type for this function.

Important
sbrk has to return the previous end of the heap, whose value is held in the static variable, **heap_ptr**.

The problem is that this now makes the function non-reentrant. If the function were interrupted after the assignment to **base**, but before the following assignment to **heap_ptr**, and the interrupt routine itself also called **sbrk**, then the heap would become corrupted.

For simple systems, it would be sufficient to avoid using this function in interrupt service routines. However the problem then knowing which functions might call **malloc** and hence **sbrk**, so effectively all library functions must be avoided.

The problem cannot even be completely avoided by using reentrant functions (see Section 5.4), since just providing a per thread data structure does not help. The end of heap is a single global value. The only full solution is to surround the update of the global variable by a semaphore, and failing the allocation if the region is blocked (we cannot wait, or deadlock would result).

5.3.16. Status of a File (by Name), `stat`

For a namespace clean function, implement **_stat**, otherwise implement **stat**. The detailed implementation will depend on the file handling functionality available.

A minimal implementation should assume that all files are character special devices and populate the status data structure accordingly.

```
#include <sys/stat.h>

int
_stat (char        *file,
       struct stat *st)
{
  st->st_mode = S_IFCHR;
  return 0;

}       /* _stat () */
```

The *OpenRISC 1000* implementation takes a stricter view of this. Since no named files are supported, this function always fails.

```
#include <errno.h>
#include <sys/stat.h>

#undef errno
extern int  errno;

int
_stat (char        *file,
       struct stat *st)
{
  errno = EACCES;
  return  -1;

}       /* _stat () */
```

5.3.17. Provide Process Timing Information, `times`

For a namespace clean function, implement **_times**, otherwise implement **times**. The implementation of this functionality will be tightly bound to any operating infrastructure for handling multiple processes.

A minimal implementation need not offer any timing information, so should always fail with an appropriate value in **errno**.

```
#include <errno.h>
#include <sys/times.h>

#undef errno
extern int  errno;

int
_times (struct tms *buf)
{
  errno = EACCES;
  return  -1;

}       /* _times () */
```

5.3.18. Remove a File's Directory Entry, `unlink`

For a namespace clean function, implement **_unlink**, otherwise implement **unlink**. The detailed implementation will depend on the file handling functionality available.

A minimal implementation has no file system, so this function should always fail, setting an appropriate value in **errno**.

```
#include <errno.h>

#undef errno
extern int  errno;

int
_unlink (char *name)
{
  errno = ENOENT;
  return -1;                    /* Always fails */

}       /* _unlink () */
```

5.3.19. Wait for a Child Process, `wait`

For a namespace clean function, implement **_wait**, otherwise implement **wait**. The implementation of this functionality will be tightly bound to any operating infrastructure for handling multiple processes.

A minimal implementation has only one process, so can wait for no other process and should always fail with an appropriate value in **errno**.

```
#include <errno.h>

#undef errno
extern int  errno;

int
_wait (int *status)
{
  errno = ECHILD;
  return -1;                    /* Always fails */

}       /* _wait () */
```

5.3.20. Write to a File, `write`

For a namespace clean function, implement **`_write`**, otherwise implement **`write`**. The detailed implementation will depend on the file handling functionality available.

A minimal implementation only supports writing to standard output. The core of the implementation is:

```
int
_write (int   file,
        char *buf,
        int   nbytes)
{
  int i;

  /* Output character at at time */
  for (i = 0; i < nbytes; i++)
    {
      outbyte (buf[i]);
    }

  return nbytes;

}       /* _write () */
```

The function **`outbyte`** must use the functionality of the target platform to write a single character to standard output. For example copying the character to a serial line for display. There can be no standard implementation of this function.

For the *OpenRISC 1000* two versions are needed one for the BSP without a UART one for the BSP with a UART.

Without a UART the implementation uses the **`l.nop`** opcode with a parameter, as with the implementation of **`_exit`** (Section 5.3.3). In this case the parameter 4 will cause the simulator to print out the value in register **`r3`** as an ASCII character.


```
#include "or1ksim-board.h"

static void
outbyte (char  c)
{
  register char  t1 asm ("r3") = c;

  asm volatile ("\tl.nop\t%0" : : "K" (NOP_PUTC), "r" (t1));

}       /* outbyte () */
```

We also use a stricter implementation of the main **write** function, only permitting a write if the standard output or standard error stream is specified.

```
#include <errno.h>
#include <unistd.h>

#undef errno
extern int  errno;

int
_write (int   file,
        char *buf,
        int   nbytes)
{
  int i;

  /* We only handle stdout and stderr */
  if ((file != STDOUT_FILENO) && (file != STDERR_FILENO))
    {
      errno = EBADF;
      return -1;
    }

  /* Output character at at time */
  for (i = 0; i < nbytes; i++)
    {
      outbyte (buf[i]);
    }

  return nbytes;

}       /* _write () */
```

For the BSP supporting a UART, all that is needed is to change the **outbyte** function to use the routines to drive the UART

```
static void
outbyte (char  c)
{
  _uart_putc (c);

}       /* outbyte () */
```

The UART support routines are provided separately, driving the interface via its memory mapped registers.

5.4. Reentrant System Call Implementations

Reentrancy is achieved by providing a global reentrancy structure, **struct _reent** for each thread of control, which holds thread specific versions of global data structures, such as **errno**.

For a fully reentrant system, the BSP should implement the reentrant versions of the system calls, having defined **syscall_dir=syscalls** and added **-DREENTRANT_SYSCALLS_PROVIDED"** to **newlib_cflags** in **configure.host** (see Section 3.3.1).

16 of the system calls have reentrant versions, which take the suffix **_r** and are passed an additional first argument, which is a pointer to the reentrancy structure, **struct reent** for the thread of control. Thus **_close** is replaced by **_close_r**. The reentrant functions are **_close_r**, **_execve_r**, **_fcntl_r**, **_fork_r**, **_fstat_r**, **_getpid_r**, **_link_r**, **_lseek_r**, **_open_r**, **_read_r**, **_sbrk_r**, **_stat_r**, **_times_r**, **_unlink_r**, **_wait_r** and **_write_r**.

Two system calls do not need reentrant versions, **_kill** and **_exit**, which are provided as with non-reentrant versions.

For many of the reentrant functions, the behavior is almost identical to that of the non-reentrant versions, beyond ensuring the thread specific version of **errno** in the reentrancy structure is used. Template versions can be found in the **libc/reent** directory under the **newlib** directory.

There are two ways in which the end user can be supported with these reentrancy functions. In the first it is up to the user to manage per thread reentrancy data structures and to call the reentrant functions explicitly.

However the more powerful solution is for the system to manage the reentrancy structure itself. The end user can call the standard functions, and they will be mapped to reentrant calls, passing in a reentrancy structure for the thread.

For this approach to be used, **-D__DYNAMIC_REENT__** must be added to **newlib_cflags** and the BSP must define the function **__getreent**, to return the reentrancy structure for the current thread.

5.5. BSP Configuration and Make file;

There is little documentation for the configuration and **make** files for the BSPs. The general guideline is to copy the baseline versions of these files in the default platform library, **libnosys**, which is based on the minimal implementations described in Section 5.3.

This application note uses the configuration and **make** files for the *OpenRISC 1000* to illustrate the key principles.

Building the BSP only uses *autoconf* and *autoheader*, but not *automake*. So there is a **configure.in** (or **configure.ac**) and **Makefile.in**, but no **Makefile.am**. After making any

changes it is important to run *autoconf* and *autoheader* to regenerate the **configure** script and header files. It will also need a **aclocal.m4** to give the local macro definitions, which can be regenerated from the main **libgloss** **acinclude.m4** using *aclocal*. The command needed are:

```
aclocal -I ..
autoheader
autoconf
```

aclocal need only be run the first time the directory is created. *autoheader* is only needed if the BSP needs configuration parameters from the system in a local **config.h** file.

5.5.1. configure.in for the BSP

The **configure.in** for the *OpenRISC 1000* is closely based on the version in **libnosys**.

The initial declarations just need modifying to change the name of the package.

```
AC_PREREQ(2.59)
AC_INIT(libor32.a,0.2.0)
AC_CONFIG_HEADER(config.h)
```

There is then code to print a warning if the user has asked for shared library support (not available) and to locate the auxiliary tools for *autoconf*.

The script makes use of **AC_CANONICAL_SYSTEM** to determine the system type and set appropriate variables. This is now obsolete, and is replaced by **AC_CANONICAL_TARGET** in the *OpenRISC 1000* version. The installed program names may be changed (for example by **--prefix**), so we need **AC_ARG_PROGRAM** and we locate the install program.

```
AC_CANONICAL_TARGET

AC_ARG_PROGRAM
AC_PROG_INSTALL
```

The assumption is made that we are using GNU *ld*, so we define **HAVE_GNU_LD**. The script in **libnosys** does this in an obsolete way, which is fixed in the *OpenRISC 1000* script.

```
AC_DEFINE(HAVE_GNU_LD, 1, [Using GNU ld])
```

The standard script tests the canonical target name to determine if this is an ELF target. For *OpenRISC 1000* this is always the case, so the test can be replaced by a simple declaration.

```
AC_DEFINE(HAVE_ELF, 1, [Using ELF format])
```

The script in **libnosys** then tests for the presence of various features. Most of those are not relevant to *OpenRISC 1000* so can be left out. However we do need to determine what the

symbol prefix is. We could just define this as being '_', but instead we let the script work it out, using the standard script's code.

```
AC_CACHE_CHECK([for symbol prefix], libc_symbol_prefix, [dnl
cat > conftest.c <<\EOF
foo () { }
EOF

libc_symbol_prefix=none
if AC_TRY_COMMAND([${CC-cc} -S conftest.c -o - | fgrep "\$foo" > /dev/null]);
then
  libc_symbol_prefix='$'
else
  if AC_TRY_COMMAND([${CC-cc} -S conftest.c -o - | fgrep "_foo" > /dev/null]);
  then
    libc_symbol_prefix=_
  fi
fi
rm -f conftest* ])
if test $libc_symbol_prefix != none; then
  AC_DEFINE_UNQUOTED(__SYMBOL_PREFIX, "$libc_symbol_prefix", [symbol prefix])
else
  AC_DEFINE(__SYMBOL_PREFIX, "", [symbol prefix])
fi
```

The code to define the various host tools used is standard. However it will expect to find an **aclocal.m4** file in the directory. This can be regenerated, or simply copied from the **libnosys** directory. The variable **host_makefile_frag** refers to standard **make** script defining how compilation is carried out for the various source files.

Finally the new **Makefile** can be generated in a suitably initialized environment.

```
AC_CONFIG_FILES(Makefile,
                ac_file=Makefile . ${libgloss_topdir}/config-ml.in,
                srcdir=${srcdir}
                target=${target}
                with_multisubdir=${with_multisubdir}
                ac_configure_args="${ac_configure_args} --enable-multilib"
                CONFIG_SHELL=${CONFIG_SHELL-/bin/sh}
                libgloss_topdir=${libgloss_topdir}
)
AC_OUTPUT
```

5.5.2. Makefile.in for the BSP

The first part of **Makefile.in** is just transferring values from **configure** and is used unchanged. The first potential variation is in multilib handling. If your GCC implements multilibs, then that may need to be mirrored in the BSP implementation. If not, then there is no need to set **MULTIDO** and **MULTICLEAN** to **true** and these lines can be removed.

The **Makefile.in** in **libnosys** includes an option to use *new* versions of the loader and assembler. However for most implementations, the plain tool is all that is needed, so simple transfer of the configured values is sufficient.


```
CC = @CC@
AS = @AS@
AR = @AR@
LD = @LD@
RANLIB = @RANLIB@
```

The main tools will already have been transformed to take account of any prefix (for example using **or32-elf-gcc** rather than **gcc**). However this has not been done for **objdump** and **objcopy**, so these are transformed here.

This is the point at which we define the BSPs to be built. Any custom flags for the compilation can be added to **CFLAGS** here.

```
CFLAGS = -g
```

We specify the C start up file(s) and BSP(s) to be built.

```
CRT0     = crt0.o
BSP      = libor32.a
BSP_UART = libor32uart.a

OUTPUTS  = $(CRT0) $(BSP) $(BSP_UART)
```

Important

It is important to define **OUTPUTS**. This is the complete set of programs and libraries being built. It is used in the **clean** and **install** targets.

For each BSP we specify the object files from which it is built. For the plain *OpenRISC 1000* BSP we have:

```
OBJS = _exit.o      \
       close.o      \
       environ.o    \
       execve.o     \
       fork.o       \
       fstat.o      \
       getpid.o     \
       isatty.o     \
       kill.o       \
       link.o       \
       lseek.o      \
       open.o       \
       read.o       \
       sbrk.o       \
       stat.o       \
       times.o      \
       uart-dummy.o \
       unlink.o     \
       wait.o       \
       write.o
```

For the BSP with UART support we use many of the same files, but also have some different files.

```
UART_OBJS = _exit.o       \
            close.o       \
            environ.o     \
            execve.o      \
            fork.o        \
            fstat-uart.o  \
            getpid.o      \
            isatty-uart.o \
            kill.o        \
            link.o        \
            lseek-uart.o  \
            open.o        \
            read-uart.o   \
            sbrk.o        \
            stat.o        \
            times.o       \
     uart.o        \
            unlink.o      \
            wait.o        \
            write-uart.o
```

At this point, the version of **Makefile.in** in **libnosys** specifies explicitly the rules for compiling object files from C and assembler source. However it is better to incorporate a standard set of rules, using the **host_makefile_frag** reference from the configuration.

```
@host_makefile_frag@
```

This is the point at which to specify the first **make** rule to create the C runtime start up files and BSPs.

```
all: ${CRT0} ${BSP} ${BSP_UART}
```

The object files (including **crt0.o**) will be built automatically, but we need rules to build the libraries from them.

```
$(BSP): $(OBJS)
        ${AR} ${ARFLAGS} $@ $(OBJS)
        ${RANLIB} $@

$(BSP_UART): $(UART_OBJS)
        ${AR} ${ARFLAGS} $@ $(UART_OBJS)
        ${RANLIB} $@
```

The remainder of **Makefile.in** is standard. It provides rules to clean the build directory, to install the generated BSP(s) and C start up file(s), and rules to ensure **configure** and **Makefile** are regenerated when necessary.

There also hooks to create, clean and install any documentation (as **info** files), which are empty by default.

Very often these rules are sufficient, so long as all the entities created have been listed in **OUTPUTS**. They should be modified if necessary.

5.6. The Default BSP, libnosys

Newlib also builds a default BSP **libnosys.a**. This can be used with the **-lnosys** flag, and provides a convenient way of testing that code will link correctly in the absence of a full BSP

The code can be found in the **libnosys** sub-directory of the main libgloss directory.

For completeness, the configuration template file, **configure.in**, in this directory should be updated for any new target that is defining namespace clean versions of the functions. Each such system is selected using a **case** statement. The new entry for the *OpenRISC 1000* is as follows.

```
  or32-*-*)
;;
```

Having updated the configuration template, run *autoconf* to regenerate the **configure** script file.

Chapter 6. Configuring, Building and Installing Newlib and Libgloss

Having made all the changes it is not time to configure, build and install the system. The examples in this chapter for the *OpenRISC 1000* assume a unified source tree in **srcw** and a build directory, **bld-or32**, with the installation directory prefix **/opt/or32-elf-new**.

6.1. Configuring Newlib and Libgloss

Newlib is configured as follows.

```
cd bld-or32
../srcw/configure --target=or32-elf --with-newlib --prefix=/opt/or32-elf-new
```

Note

Other options may be needed on the command line if other GNU tools are being built. However these are the options relevant to newlib

6.2. Building Newlib and Libgloss

The system is built using **make** from within the **bld-or32** directory.

```
make all-target-newlib
make all-target-libgloss
```

6.3. Testing Newlib and Libgloss

Testing newlib and libgloss requires further configuration. The details are discussed later in this application note (see Chapter 8). For now this step can be skipped.

6.4. Installing Newlib and Libgloss

The system is installed using **make** from within the **bld-or32** directory.

```
make install-target-newlib
make install-target-libgloss
```

Chapter 7. Modifying the GNU Tool Chain

7.1. Putting `Newlib` in a Custom Location

Normally `newlib` will be installed in a standard place with the rest of the tool chain. Its headers will go in the **`include`** directory within the target specific installation directory. The C runtime start up file, the `newlib` libraries themselves and BSP libraries will go in the **`lib`** directory within the target specific installation directory.

This arrangement ensures that GCC will pick up the headers and libraries automatically and in the correct sequence.

However if `newlib` is not the only C library, then this may be inconvenient. For example the *OpenRISC 1000* usually uses `uClibc`, and only uses `newlib` when regression testing the GNU tool chain.

The solution is to move the `newlib` headers and libraries to a custom location and modify GCC to search there when `newlib` is being used (see Section 7.2).

This is achieved with a simple script at the end of build and install. For example with the *OpenRISC 1000* the following command will suffice, where the prefix used for the entire tool chain build is in **`${install_dir}`**.

```
mkdir -p ${install_dir}/or32-elf/newlib
rm -rf ${install_dir}/or32-elf/newlib-include
mv ${install_dir}/or32-elf/include ${install_dir}/or32-elf/newlib-include
mv ${install_dir}/or32-elf/lib/*.a ${install_dir}/or32-elf/newlib
mv ${install_dir}/or32-elf/lib/crt0.o ${install_dir}/or32-elf/newlib
```

7.2. Changes to GCC

In general GCC will work with `newlib` with no change. All that is needed is to include the BSP library on the command line.

However it is convenient to modify GCC so that it picks up the BSP automatically. This is particularly useful when `newlib` has been installed in a custom location (see Section 7.1).

This is achieved by adding machine specific options to GCC, and modifying the Spec definitions to pick up the `newlib` libraries when the relevant option is in effect.

All the relevant files are found in the **`gcc/config/`*`target`*** directory of GCC. For the 32-bit *OpenRISC 1000* this is **`gcc/config/or32`**.

7.2.1. Adding Machine Specific Options for `Newlib`

Machine specific options are described in the ***`target`*`.opt`** file. By convention machine specific options begin with 'm'.

For the *OpenRISC 1000* we define two options, **`-mor32-newlib`** and **`-mor32-newlib-uart`** for the plain and UART enabled versions of the BSP respectively.

For each option we provide its name on one line, any parameters on subsequent lines and a final line of description. In this case the only parameter is to say that the parameter can only appear in its positive form (i.e. **`--mno-or32-newlib`** is not permitted).

```
mor32-newlib
Target RejectNegative
Link with the OR32 newlib library

mor32-newlib-uart
Target RejectNegative
Link with the OR32 newlib UART library
```

These parameters can then be used elsewhere.

7.2.2. Updating Spec Definitions

GCC calls a number of subsidiary programs (the compiler itself, the assembler, the linker etc). The arguments to these are built up from the parametrized strings, known as *Spec* strings.

This application note cannot describe the huge range of possible parameters. However we will use one example to show what is possible. The changes are all made to the definitions of the strings in ***target*.h**. In the case of the *OpenRISC 1000* this is **or32.h**.

We need to make four changes.

1. We need to tell the C preprocessor to look for headers in the relocated newlib library directory.
2. We need to tell the linker to pick up the newlib C runtime start up file.
3. We need to tell the linker where to find the `newlib` libraries.
4. We need to tell the linker to include the BSP library in the right place.

The Target Specific Installation Directory

All of these changes will require knowing the location of the target specific installation directory. Unfortunately there is no Spec parameter giving this. However we can construct it from two definitions available when compiling GCC. **STANDARD_EXEC_PREFIX** is the directory where the GCC executables will be found. Two directories up from that will be the main prefix directory. The target machine is specified in **DEFAULT_TARGET_MACHINE**. So concatenating the three strings yields the target specific directory.

```
STANDARD_EXEC_PREFIX "/../../" DEFAULT_TARGET_MACHINE
```

The newlib headers are in the subdirectory **newlib-include** and the C runtime start up and libraries in **newlib**.

We define a new string, **TARGET_PREFIX** based on the concatenation.

```
#define CONC_DIR(dir1, dir2) dir1 "/../../" dir2
#define TARGET_PREFIX CONC_DIR (STANDARD_EXEC_PREFIX, DEFAULT_TARGET_MACHINE)
```

Defined constants cannot be used directly in Spec strings, but we can make them available by defining the macro **EXTRA_SPECS**.

```
#define EXTRA_SPECS                                     \
  { "target_prefix", TARGET_PREFIX }
```


The Spec string **target_prefix** is now available to be used in other Spec strings.

Specifying the header directory.

Additional arguments to the C preprocessor are defined in **CPP_SPEC**. The newlib header directory should we searched after any user specified header directories (from **-I** arguments) and after the GCC system headers. So it is specified using the **-idirafter** option.

```
#undef CPP_SPEC
#define CPP_SPEC "%{mor32-newlib*:-idirafter %(target_prefix)/newlib-include}"
```

This specifies that any option beginning **-mor32-newlib** should be replaced by the string **-idirafter** followed by the **newlib-incldue** subdirectory of the **target_prefix** directory.

So so for example, if we build the *OpenRISC 1000* GCC with **--prefix=/opt/or32-elf-new**, we would have **STANDARD_EXEC_PREFIX** set to **/opt/or32-elf-new/lib/gcc** and **DEFAULT_TARGET_MACHINE** set to **or32-elf**. The Spec variable **target_prefix** would therefore be **/opt/or32-elf-new/lib/gcc/../../or32-elf** and thus the C preprocessor would have the following added to its option list.

```
-idirafter /opt/or32-elf-new/lib/gcc/../../or32-elf/newlib-include"
```

This substitution only occurs when **-mor32-newlib** or **-mor32-newlib-uart** is specified, which is exactly the behavior desired.

Note

If newlib is not relocated as described in Section 7.1, then the headers will be in a standard location, which GCC will search anyway, so there is no need to define **CPP_SPEC**.

Specifying the C Start up File

crt0.o should be the first object file or library specified to the linker. This is covered by **STARTFILE_SPEC**.

This string already has a partial definition, to look for **crt0.o** in a standard place, and to include the **crtinit.o** file from a standard place.

```
#undef STARTFILE_SPEC
#define STARTFILE_SPEC "%{!shared:crt0%s crtinit.o%s}"
```

So long as **-shared** is not specified as an option, this looks for **crt0.o** and **crtinit.o** in standard directories and substitutes them on the command line (the suffix **%s** indicates that the preceding file should be searched for in standard directories, and its name expanded to include the directory name).

This needs changing to indicate that if **-mor32-newlib** or **-mor32-newlib-uart** is specified, then **crt0.o** should be taken from the newlib directory.

```
#define STARTFILE_SPEC \
    "%{!shared:%{mor32-newlib*:%(target_prefix)/newlib/crt0.o} \
     %{!mor32-newlib*:crt0.o%s} crtinit.o%s}"
```

Note that we must also include the case that when neither of the `newlib` options is specified, then **`crt0.o`** will be searched for in standard directories.

Note

If `newlib` is not relocated as described in Section 7.1, then **`crt0.o`** will be in a standard location, which GCC will search anyway, so there is no need to modify **`STARTFILE_SPEC`**.

Specifying the `Newlib` library location

We need to tell the linker where to look for `newlib` libraries. This is achieved in a similar manner to the search for the headers, but using the **`-L`** option and **`LINK_SPEC`**.

```
#undef LINK_SPEC
#define LINK_SPEC "%{mor32-newlib*:-L%(target_prefix)/newlib}"
```

Note

If `newlib` is not relocated as described in Section 7.1, then the `newlib` libraries will be in a standard location searched by GCC, so there is no need to specify **`LINK_SPEC`**.

Adding a BSP to the link line.

The libraries searched by GCC are by default specified to be **`-lgcc -lc -lgcc`**, with variants if profiling is being used. When a BSP is used, it must be searched after `libc`, but that can leave references unresolved, so `libc` must be searched again afterward.

The sequence of libraries to be searched between the two searches of `libgcc` is given in **`LIB_SPEC`**. It already has a definition.

```
#define LIB_SPEC "%{!p:%{!pg:-lc}}%{p:-lc_p}%{pg:-lc_p}
```

This specifies a variant library when profiling is in place. `newlib` does not offer profiling support, but it does have a debugging version of the library (`libg`).

```
#undef LIB_SPEC
#define LIB_SPEC "%{!mor32-newlib*:%{!p:%{!pg:-lc}}%{p:-lc_p}%{pg:-lc_p}} \
                  %{mor32-newlib:%{!g:-lc -lor32 -lc}                     \
                                 %{g:-lg -lor32 -lg}}                     \
                  %{mor32-newlib-uart:%{!g:-lc -lor32uart -lc}            \
                                 %{g:-lg -lor32uart -lg}}"
```

This ensures that the correct BSP library will be used, according the the option selected, and that if **`-g`** is specified on the command line, the debugging version of the C library (`libg`) will be used instead.

Even if the `newlib` is not relocated as described in Section 7.1, then this Spec change is required in order to ensure the correct libraries are picked up.

7.3. Changes to the GNU Linker

In general changes to the linker are not needed. Instead the BSP should make use of information provided by the standard linker. For example in the definition of **`sbrk`** (see

Section 5.3.15) the code uses the **_end** symbol defined by the linker at the end of the loaded image to be the start of the heap.

Chapter 8. Testing Newlib and Libgloss

Newlib and libgloss both come with *DejaGnu* test infrastructures, although as noted in Section 8.2, the libgloss infrastructure is non-functional.

The total number of tests is modest (24 tests in release 1.18.0). In practice much of the testing is achieved through the GCC test suite (40,000+ tests) and the GDB test suite (5,000+ tests).

8.1. Testing Newlib

Like all tools, newlib can be tested with a *DejaGnu* test suite. *DejaGnu* must be installed on the test machine.

If you already have testing set up for other tools in the GNU tool chain on your target, then you can skip the remainder of this section, and just test newlib from the build directory with the following.

```
cd bld-or32
make check-target-newlib
```

If this is the first time you have tried testing, then you'll need to set up your system appropriately. Once this is done, you will be able to test all the GNU tool chain components.

The tests require a target on which to run the tests. This can be a physical machine, or it can be a simulator for the target architecture.

The details of the target are provided in an *expect* board configuration file. This is referenced from the *DejaGnu* global configuration file. The environment variable **DEJAGNU** should point to the global configuration file.

For the *OpenRISC 1000*, the global configuration file is in **site.exp** and a subdirectory, **boards** contains **or32-sim.exp**, which is the board configuration file for the OpenRISC simulator target.

The **site.exp** file has two functions. First, it must add the **boards** directory to the list of board directories to search. Secondly, it must ensure that the target triplet name is mapped to the name of the board configuration file.

This **site.exp** file can be reused for checking other targets in the GNU tool chain, which may have a different test suite hierarchy. We cannot therefore just reference the **boards** directory relative to the test directory. All we know is that it will be in one of the directories above, and there is no other boards directory in the hierarchy, so we add all the possible directories. Not elegant, but effective.

```
#Make sure we look in the right place for the board description files
if ![info exists boards_dir] {
        set boards_dir {}
}

# Crude way of finding the boards directory
lappend boards_dir "${tool_root_dir}/../boards"
lappend boards_dir "${tool_root_dir}/../../boards"
lappend boards_dir "${tool_root_dir}/../../../boards"
lappend boards_dir "${tool_root_dir}/../../../../boards"

global target_list
  case "$target_triplet" in {
    { "or32-*-elf" } {
        set target_list { "or32-sim" }
    }
  }
```

Within the **boards** directory, the board configuration file, **or32-sim.cfg** gives all the details required for the configuration.

The tool chains supported by this board are specified first. In the case of the *OpenRISC 1000*, only one is supported.

```
set_board_info target_install {or32-elf}
```

We then need to load some generic routines, and the generic board configuration.

```
load_generic_config "sim"
load_base_board_description "basic-sim"
```

The default settings assume that a program is executed on the target by a command named **run**, built in a target specific subdirectory of the top level **sim** directory. In the case of the *OpenRISC 1000* this directory would be **sim/or32**.

At a minimum, **run** takes as argument an executable to run, and returns the exit code from that executable as its result.

The **sim** directory is usually distributed as part of GDB. Simulators may be derived from *CGEN* specifications of the architecture, or by integrating third party simulators. The latter is the case for the *OpenRISC 1000*.

The default settings for a target are obtained using the **setup_sim** procedure.

```
setup_sim or32
```

The remainder of the file is used to configure variations on the default settings. This is done using the **set_board_info** procedure.

The *OpenRISC 1000* simulator needs an additional argument, which is a configuration file for the simulator. We know that file will be in the **libgloss** target directory and named **sim.cfg**.

We can use the **lookfor_file** procedure to search up from the current source directory to locate the file.

```
set cfg_file [lookfor_file ${srcdir} libgloss/or32/sim.cfg]
set_board_info sim,options "-a \"-f ${cfg_file}\""
```

A number of helpful procedures make it easy to locate parts of the tool chain and their default arguments. For the *OpenRISC 1000* we make one change, which is to specify **-mor32-newlib** for the linker flags, so that the `newlib` BSP will be used.

```
process_multilib_options ""
set_board_info compiler "[find_gcc]"
set_board_info cflags "[libgloss_include_flags] [newlib_include_flags]"
set_board_info ldflags "[libgloss_link_flags] -mor32-newlib [newlib_link_flags]"
set_board_info ldscript ""
```

Not all targets have the same functionality, and the remaining options specify those limitations. This is a generic board specification, so some of these apply to testing components other than `newlib`. The limitations specified will mean that some tests, which are inappropriate do not run.

For the *OpenRISC 1000* we specify that the simulator is fast, that programs it runs cannot be passed arguments, that it does not support signals (for testing GDB) and that the maximum stack size is 64KB (for testing GCC).

```
set_board_info slow_simulator 0
set_board_info noargs  1
set_board_info gdb,nosignals  1
set_board_info gcc,stack_size 65536
```

We can now set **DEJAGNU** to point to the global configuration directory, change to the build directory and run the **make** command to check newlib.

```
export DEJAGNU=`pwd`/site.exp
cd bld-or32
make check-target-newlib
```

The good thing is that this set up is generic across all the GNU tool chain, so all the other tools can be checked in the same way.

8.1.1. Checking Physical Hardware

The same technique can be used to run the tests against physical hardware rather than a simulator. The setup of the board configuration is rather more complicated, with considerable variation for different arrangements.

The detail is beyond the scope of this application note, but is well described in Dan Kegel's *Crosstool* project [2].

8.2. Testing `Libgloss`

In principle, having set up `newlib` testing, testing `libgloss` should be as simple as:


```
cd bld-or32
make check-target-libgloss
```

Unfortunately, the current `newlib` release (at the time of writing 1.18.0) does not implement testing for `libgloss`. The **`testsuite`** subdirectory exists, but the code to configure it is currently commented out in **`configure.in`**.

It should not be difficult to build the infrastructure. However as noted at the start of this chapter, testing of `newlib` and `libgloss` is as much achieved through GCC and GDB testing as through the modest number of tests within `newlib`

Chapter 9. Summary Checklist

This summary can be used as a checklist when creating a new port of `newlib`. The configuration and build steps are typically encapsulated in a simple shell script, which builds, tests and installs the entire GNU tool chain as well as `newlib` and `libgloss`

Throughout this checklist, the new target architecture is referred to as ***`target`***. It is recommended `newlib` and `libgloss` are built as part of a unified source tree including the `newlib` distribution (see Section 2.1).

1. Edit **`newlib/configure.host`** adding entries for the new target (Section 3.3).
   - Decide whether to implement reentrant or non-reentrant system calls and whether to use namespace clean system call names (Section 3.2).
2. Add a `newlib` machine subdirectory for the new target, **`newlib/libc/machine/`*`target`*** (Section 4.1).
   - Modify **`configure.in`** in **`newlib/libc/machine`** to configure the new target subdirectory and run *autoconf* in **`newlib/libc/machine`** to regenerate the **`configure`** script (Section 4.1.1).
3. Implement **`setjmp`** and **`longjmp`** in the target specific machine directory, **`newlib/libc/machine/`*`target`*** (Section 4.1.2).
   - Copy and modify **`Makefile.am`** and **`configure.in`** from the *fr30* directory and run *aclocal*, *autoconf* and *automake* in **`newlib/libc/machine/`*`target`*** to regenerate the **`configure`** script and Makefile template. (Section 4.1.3).
4. Modify `newlib` header files (Section 4.2).
   - Add entries in **`newlib/libc/include/ieeefp.c`** (Section 4.2.1)
   - Add entry in in **`newlib/libc/include/setjmp.c`** (Section 4.2.2)
   - Add entries in **`newlib/libc/include/sys/config.h`** (Section 4.2.3).
   - Optionally add other custom headers in **`newlib/libc/machine/target/machine`** (Section 4.2.4).
5. Add a `libgloss` platform directory, **`libgloss/`*`target`*** (Section 5.1).
   - Modify **`libgloss/configure.in`** to configure the platform subdirectory and run *autoconf* in the **`libgloss`** directory to regenerate the **`configure`** script (Section 5.1.1).
6. Implement the Board Support Package(s) for the target (Chapter 5).
   - Implement the C Runtime start up, **`crt0.o`** for each BSP (Section 5.2).
   - Implement the environment global variable and 18 system call functions for each BSP following the convention namespace and reentrancy conventions specified in **`newlib/configure.host`** (Section 5.3 and Section 5.4).
   - Create **`libgloss/`*`target`*`/Makefile.in`** and **`libgloss/`*`target`*`/configure.ac`**, based on the versions in the **`libgloss/libnosys`** directory and run *aclocal* and *autoconf* in **`libgloss/`*`target`*** (Section 5.5).
7. If necessary update **`libgloss/libnosys/configure.in`** to indicate the target is using namespace clean system calls and run *autoconf* in **`libgloss/libnosys`** (Section 5.6).
8. Modify GCC for `newlib` (Section 7.2).

- Optionally add target specific option(s) to specify `newlib` BSP(s) (Section 7.2.1).
- Optionally specify the location of `newlib` headers, the BSP C runtime start up file and the newlib libraries, if they have been moved from their standard locations and/or names (Section 7.2.2)
- Specify the `libgloss` BSP library to be linked, ensuring **`malloc`** and **`free`** are linked in if required (the section called “ Adding a BSP to the link line. ”).

9. Ensure the linker scripts are suitable for use with `newlib` (Section 7.3).
10. Configure and build `newlib` and `libgloss` (Chapter 6).
    - Optionally move the `newlib` header directory, libraries, C start-up and BSP(s) to a custom location (Section 7.1).
    - Rebuild GCC
    - Rebuild *ld* if any linker scripts have been changed.
11. Test `newlib` (Section 8.1).
12. Install `newlib` and `libgloss` (Chapter 6).
    - Reinstall GCC
    - Reinstall *ld* if any linker scripts have been changed.

You should now have a working `newlib` implementation integrated within your GNU tool chain.

Glossary

Application Binary Interface (ABI)
: The definition of how registers are used during function call and return for a particular architecture.

big endian
: A multi-byte number representation, in which the most significant byte is placed first (i.e. at the lowest address) in memory.
See also: little endian

Block Stated by Symbol (BSS)
: Universally known by its acronym (the full name is a historical relic), this refers to an area of storage used for holding static variables and initialized to zero.

Board Support Package (BSP)
: The low level interface between an operating system or library and the underlying physical platform.

little endian
: A multi-byte number representation, in which the least significant byte is placed first (i.e. at the lowest address) in memory.
See also: big endian

reentrant
: A function which is *reentrant* may be safely called from another thread of control while an initial thread's flow of control is still within the function.
In general a function will be reentrant if it changes no static state.

special purpose register (SPR)
: A set of up to 2^{16} 32-bit registers used to hold additional information controlling the operation of the *OpenRISC 1000*

supervision register
: An *OpenRISC 1000* special purpose register holding information about the most recent test result, whether the processor is in supervisor mode, and whether certain functions (cache etc) are enabled.
See also: special purpose register

References

[1] The Red Hat `Newlib` C Library Available at sourceware.org/newlib/libc.html.

[2] The Crosstool Project, Dan Kegel. Available at www.kegel.com/crosstool.