Porting and Using Newlib in Embedded Systems

William Gatliff

Table of Contents

$Revision: 1.5 $

Although technically not a GNU product, the C runtime library newlib is the best choice for many GNU-based embedded systems that require a modest C runtime environment. With some minor modifications, newlib can also be used in embedded systems that are not built using GNU tools. The following shows you how to do this, and more.

TODO: how to use newlib as a linux runtime.

Copyright

Newlib

Newlib is a freely-available C runtime library with a portable and flexible architecture that makes it suitable for use in resource-constrained embedded systems. Newlib is an actively supported and mature product, and is the preferred choice for GNU-based embedded environments, except perhaps in some Linux-based devices that require the full functionality supplied by GNU's glibc[1].

This whitepaper discuss some of newlib's functionality and portabilty features, and provides examples on how to integrate newlib into an embedded environment based on a popular Real Time Operating System (RTOS).

Newlib Licenses

Newlib is actually a collection of source code, assembled by Cygnus Solutions, Inc. (now a part of Red Hat, Inc.)[2]. Because of this, newlib is actually distributed under the terms of several different licenses--- all of which are either public domain or BSD-like. Proprietary applications can use newlib because these licenses do not require distribution of the final article's source code.

All of newlib's licenses are gathered into the file `COPYING.NEWLIB`, which is included with the newlib source code. You should read this file before you decide to use newlib, the same way you do with the license for any other software you use.

Newlib Features

Some of newlib's functionality provides useful enhancements to a "typical" embedded setting (whatever that is), while others allow newlib to be about as POSIX-like as a compact C runtime setting can be. These capabilities can be a big help when porting desktop-tested applications to an embedded environment.

`Printf()` vs. `iprintf()`

Newlib contains a complete implementation of the C standard `printf()` and family. By the implementation's own admission, "this code is large and complicated" [3], but essential for systems that need full ANSI C input and output support, including capabilities for representing and parsing floating point numbers.

Many embedded systems do not use floating point math, however, and great pains are taken in most embedded runtime libraries to cull this code-bloating functionality whenever possible. Newlib approaches this problem in two ways: a `FLOATING_POINT` macro that allows selective disabling of floating point support in each of the library functions that can offer it, and an `iprintf()` function that only knows how to display integer objects.

If an embedded system needs floating point support in only a few of the standard input and output functions, then newlib can be rebuilt to exclude floating point from places where it isn't needed. You can omit floating point for everything except `scanf()`, for example, by either undefining the FLOATING_POINT macro everywhere except in the `scanf.c` source file, or by modifying newlib's `Makefile` to do the same thing.

For situations where only integer output is required, newlib provides the `iprintf()` function: a version of the `printf()` function built with the `FLOATING_POINT` macro undefined. It behaves exactly like `printf()`, except that it does not understand the `%f`, `%F`, `%g`, and `%G` type specifiers, and therefore has a much smaller code footprint.

More on stdio

Newlib's standard input and output facilities are surprisingly complete, even beyond the `printf()` et al implementations. The complete C file API is also provided, complete with read and write buffering, seeking, and stream flushing capabilities. Variations like `sprintf()`, `fprintf()` and `vfprintf()` (takes va_list arguments) are also included, which makes a newlib environment look strikingly similar to one you'd expect to see in a more workstation-oriented programming environment.

An unfortunate limitation of newlib's stdio library is that it requires at least a minimal `malloc()` for complete and proper operation. Fortunately, newlib includes a pretty good dynamic memory allocator that is straightforward to set up and use. One can also build a `malloc()` based on a fixed size memory block allocator, to eliminate fragmentation worries in systems where this is a concern.

If you constrain your use of stdio to just `iprintf()`, you do not need a working `malloc()`.

UNIX API

Newlib includes a lot of the familiar UNIX API functions like `open()` and `write()`. Most of these functions map directly to the code stubs newlib uses to invoke external system resources, so their simplicity eliminates the need for `malloc()`. A notable omission is the `ioctl()` function, which may be easily corrected by providing your own stub, modeled after one of the existing functions.

Libm

Newlib includes a complete IEEE math library called libm. In addition to offering the standard math functions like `exp()`, `sin()` and `pow()`, this library also provides `matherr()`: a modifiable math error handler invoked whenever a serious math-related error like an underflow or loss of precision is detected. By customizing this function, you can handle these situations in whatever way is appropriate for your application.

As a surprising bonus, libm also includes functions that take float parameters, instead of double. These extensions are named after their full precision equivalents, i.e. `sinf()` is the single precision version of the `sin()` function. The reduced precision functions have a considerable speed advantage over their IEEE-compliant double precision counterparts, which can put some floating point operations within reach of hardware that is too weak for full double precision computations.

Reentrancy

Newlib's C and math libraries are reentrant when properly integrated into a multi-threaded environment. The implementation is not obvious at first glance, so the next paragraphs describe how it works. Once you know the details, it will be clear how to set it up properly in your system.

The `errno` variable

The ANSI C standard specifies a global integer called `errno`, that the runtime library asserts when an error occurs. Once asserted, `errno`'s value persists until the application clears it. This simplifies error notification by the library, but can create reentrancy problems when multiple execution threads are working in the library at the same time: even if the error occurs in only one processing context, both threads see the error code that results.

Making `errno` reentrant

Newlib encloses `errno` and several related values into a structure of type struct _reent, and redefines the symbol `errno` as a macro that references a global _reent* pointer named `__impure_ptr`. As a result, when a statement refers to the value of `errno`, it is actually doing an indirect structure lookup that resolves to the `errno` field in a data structure.[4]

The code in Figure 1 describes in general how `errno` is modified under newlib. The code in Figure 2 is a common example of how to use `errno` in an ANSI C environment; because the reimplementation of `errno` is transparent to the application, this code works without modification under newlib.

```
#define errno (*__errno())
extern int *__errno _PARAMS ((void));

#define _REENT _impure_ptr
#define _REENT_INIT(var) \
  { 0, &var.__sf[0], &var.__sf[1], &var.__sf[2], 0, "", 0, "C", \
    0, NULL, NULL, 0, NULL, NULL, 0, NULL, { {0, NULL, "", \
```

```
    { 0,0,0,0,0,0,0,0}, 0, 1} } }

static struct _reent impure_data = _REENT_INIT (impure_data);
struct _reent * _impure_ptr = &impure_data;

int *
__errno ()
{
  return &_REENT->_errno;
}
```

Figure 1. How `errno` is modified under newlib.

```
fp = fopen( "myfile.txt", "rw" );
if( fp == NULL ) {
  switch( errno ) {
     case EACCES:
        /* we don't have permissions */
        ...
```

Figure 2. How to use `errno`.

Managing _reent structures

Newlib declares one _reent structure and aims `_impure_ptr` at it during initialization, so everything starts out correct for situations where only one thread of execution will be in the library at a time. To provide the capability to have multiple library processing contexts, allocate multiple _reent structures, and move `_impure_ptr` between them during context switches.

The _reent structure also contains fields for the *standard input* (stdin), *standard output* (stdout), and *standard error* (stderr) descriptors. This allows each task to define its own set of streams for reading and writing data: tasks A and B could both use `printf()` simultaneously, with each task's output going to different locations.

Reentrancy in memory management

To permit multiple processing contexts in newlib's `malloc()` implementation, you must also provide the functions `__malloc_lock()` and `__malloc_unlock()` to protect your memory pool from corruption during simultaneous allocations. If you are using an RTOS's reentrant memory pool implementation for dynamic memory allocation, however, this heap protection is unnecessary--- the RTOS protects the heap itself.

Designed for portability

All of newlib's functionality builds on a set of seventeen stub functions that newlib uses to hook into the host's execution environment. By modifying this integration layer, you can adapt newlib to just about any system imaginable, from one with no

operating system at all, to one based on an embedded RTOS, to one with a complete POSIX operating system.

Newlib's documentation provides details on which stubs are needed for each library function, and you only need to provide stubs for the portions of newlib that you intend to use. For example, newlib has a stub called `_fork`, but you don't need to do anything with it unless you intend to use `system()` or `fork()`.

An embedded filesystem?!

Newlib does not include a filesystem, but it may seem like it requires one for proper operation--- especially considering that it provides file-oriented functions like `fprintf()` and `fseek()`. However, although newlib likes to think that there is a stream-oriented filesystem working behind the scenes, its integration layer has been conveniently organized to not require this.

Do not dismiss the utility of a filesystem-like abstraction in an embedded system, even in the most minimal designs. In addition to enabling greater portability between workstation and embedded environments, a file-oriented device API gives a consistent look and feel to your applications, regardless of the target system's underlying implementation. This helps make code more reusable, more cleanly defines the logical boundaries between application-specific and platform-specific code in a system, and provides a more familiar environment for new developers.

Building Newlib

Building newlib for a supported target is a straightforward process that follows the conventions adhered to by most open source and Free Software projects. After downloading and decompressing the source code, you simply configure and build it using a cross compiler like gcc.

You start the build process using the commands shown in Figure 3, and then go get a copy of your favorite caffeinated beverage. When you get back, the build process will have produced the files `libc.a`, `libg.a` (a debugging-enabled libc), and `libm.a`, in the directory `/usr/local/<target-name>`. If the target you specify has several variants, the build process will produce multiple files, each with compilation settings specific to each variant. Link one or more of these files with your application, and there you have it: a free C runtime environment.

Note that if you provided a `--prefix` option when building your GNU cross compiler, then you must provide the same `--prefix` here for newlib, if you want everything to compile and link seamlessly. The default value for `--prefix` is `/usr/local/`.

```
$ tar xzvf newlib-1.9.0.tar.gz
$ mkdir build-newlib && cd build-newlib
$ ../newlib-1.9.0/configure --target=$TARGET --prefix=$PREFIX
$ make all install info install-info
```

Figure 3. Building newlib

Newlib does not require use of the GNU compiler collection, but if you use something else then you will have to make adjustments to newlib's Makefiles after the configu-

ration step, to provide the name of the compiler you intend to use. You may also have to tweak your environment *before* configuring newlib, so that newlib's setup process can properly identify your cross compiler.

In extreme cases, you may end up discarding newlib's automated configuration process entirely, and construct Makefiles by hand. More about this is available in the mailing lists at newlib's website.

Newlib's build process produces documentation, in the files `libc.info` and `libm.info`. By default these files go into `/usr/local/info`[5], and they can be browsed using **info**, a documentation browser included with most Linux distributions. Just change to the directory containing these files, and type:

```
$ info -f ./libc.info
```

Newlib's configuration script supports several options, not the least of which are the definition of the target system (what CPU and OS the library will run under), and where to put the files generated during the build. As an example, the following command will set up a build for the Hitachi SH CPU using the ELF" output file format, and put the library files in `/home/bgat/newlib-install`. Since the command doesn't specify any target operating system, the build script assumes that the target system does not have one.

```
$ ../newlib-1.9.0/configure --target=sh-elf \
    --prefix=/home/bgat/newlib-install
```

Use the `--help` option a complete list of command line options. The following are some of the `--target` specifications that newlib supports.

- arm-elf
- m68k-coff
- h8300-coff
- sh-elf
- z8k-coff
- mn10300-elf
- i386-elf

Tweaks

Newlib's source code has a few configuration points, and you will want to use them to eliminate unneeded stubs, to optimize for code size instead of speed, or to remove floating point support. Don't spend much time with this in the early stages of a project, before you have enough of a system in place to evaluate the benefits of your changes, but tinker liberally once you can test your refinements. If you add any useful configuration points, be sure to mention them in the newlib mailing lists so that they may be considered for inclusion in the next version of the library.

To modify a configuration point, change its value in the `Makefile` generated by the **configure** command, before you type **make**. Look for the variable called `CFLAGS_FOR_TARGET`, and add flags there like:

```
-DINTEGER_ONLY
```

to build an integer-only library, or

```
-DPREFER_SIZE_OVER_SPEED
```

to enable a few small changes that reduce library code size.

You can also adjust the value of the `CFLAGS` setting, to affect the way the library is compiled. For example, if you are using the GNU C compiler then use:

```
-Os (instead of -O2)
```

to tell it to optimize for code size over performance, or:

```
-O3 (instead of -O2)
```

to tell the compiler to optimize for raw performance over everything else.

Add:

```
-fomit-frame-pointer
```

to tell the compiler to not build stack frames for functions in the library that do not need them, which saves some space and boosts library performance. Don't eliminate stack frames if you intend to step through code inside of newlib itself, however, because it likely won't work--- most debuggers need a valid stack frame at all times.

You can discover additional, minor source code configuration points by using the **find** program on the library source code, to locate sections of conditional compilation. Here is one way to do it:

```
$ find . -name "*[ch]" -type f | xargs grep "#if"
```

One other thing: if you decide you don't like your changes and want to try again, you don't need to repeat the entire configuration process. Instead, simply edit `Makefile` and then do a clean rebuild, like this:

```
$ make clean all install
```

Porting Newlib

Newlib supports more than a dozen target CPU architectures, but it doesn't come with code to connect it to very many operating systems or target hardware platforms. This makes the odds almost certain that you will need to do some work to get newlib running in your system. Fortunately, the process is both straightforward and relatively painless.

All of newlib's functionality sits on top an integration layer of seventeen *stubs* of code that supply capabilities that newlib cannot provide itself: low-level filesystem access, requests to enlarge its memory heap, getting the time of day, and various types of context management like process forking and killing. Newlib supplies templates for each of these stubs, which either return "not implemented", or fail silently.

The requirements for each stub are fully documented in newlib's `libc.info` file, in the section called *Syscalls*. The key to a successfully ported newlib is providing stubs

that bridge the gap between the functionality newlib needs, and what your target system can provide.

To demonstrate this, the following sections show how to use newlib in an embedded system that also uses uC/OS[6], a pragmatic, superbly written RTOS that features a reentrant memory pool implementation, but lacks any concept of a device driver or filesystem API. Despite the operating system's modest feature set, the combination of uC/OS and newlib yields a practical and perfectly usable system, and offers insights into how you would use newlib in both larger and smaller settings.

Reentrant vs. nonreentrant stubs

In many places newlib offers two types of stubs: reentrant ones, and nonreentrant ones. The only difference between the two is that the reentrant stubs include a _reent structure pointer in their signatures, which allows the implementer to carry context-specific information between the library and the target operating environment.

In order for newlib to use reentrant stubs, you add `-DREENTRANT_SYSCALLS_PROVIDED` to the `CFLAGS_FOR_TARGET` variable in the top level `Makefile` before building newlib. This is a strongly encouraged configuration point, and the example stubs that follow will assume that you have done so.

If you choose not to use the reentrant versions of the stubs, then eliminate the *_r* from each stub's name (`_fork_r` becomes `_fork`) in the code in the following sections, and eliminate the portions of the stubs that relate to the _reent structure. The result is an implementation that is not reentrant when the nonreentrant stubs are invoked.

`_fork_r`

Newlib calls upon this stub to do the work for the `fork()` system call, which in POSIX environments is used to create a clone of the current processing context. A hardcore POSIX enthusiast could implement this stub with help from uC/OS's `OSTaskCreate()` function, but that is a challenging exercise because the semantics of the conventional `fork()` do not coexist peacefully with uC/OS's way of managing task creation and identification.

In fact, trying to implement `fork()` in uC/OS is probably a bad idea, because it raises task priority and synchronization issues that uC/OS already addresses quite well on its own. So save joining the two for another day, and leave this stub essentially unimplemented. The code is in Figure 4.

```
int
_fork_r ( struct _reent *ptr )
{
   /* return "not supported" */
   ptr->errno = ENOTSUP;
   return -1;
}
```

Figure 4. The `_fork_r` stub.

Take this approach for several other context management-related stubs, including `_execve`, `_kill`, `_wait_r` and `_getpid_r`.

_write_r and _read_r

These stubs are a bit more interesting to implement, because uC/OS does not provide any type of device driver or filesystem model--- we must provide one ourselves.

Newlib calls `_write_r` any time it wants to send data to a device, be it due to a `write()` call, `printf()` or `fprintf()`, or anything similar. The _reent parameter provides a place for the stub to communicate errors should they occur, and the file descriptor parameter, `fd`, tells the stub which device is being addressed. The remaining arguments supply a source data buffer and number of bytes to write.

The tricky part here is the semantics. The stub doesn't need to write all the bytes that newlib asks it to, but if it doesn't then newlib will simply invoke it again with the remaining data. So if the return value never eventually equals the number of bytes requested, newlib will misbehave.

Furthermore, newlib doesn't call `open()` for file descriptors 0, 1, or 2, which means that the `_write_r` call is the first activity the stub will see on those streams. Stream zero is defined by convention to be the "standard input" stream, which newlib uses for the `getc()` and similar functions that don't otherwise specify an input stream. Stream number one is "standard output", the destination of `printf()` and `puts()`. Stream number two refers to standard error", the destination conventionally reserved for messages of grave importance. You may use any other positive integers you like for file descriptors.

To implement `_write_r`, start by defining a simple "device operations" table, with function pointers for all the kinds of activities you would expect a stream-like device driver to support. The structure for this table is shown in Figure 5.

```
typedef struct {
   const char *name;
   int  (*open_r  )( struct _reent *r, const char *path,
                     int flags, int mode );
   int  (*close_r )( struct _reent *r, int fd );
   long (*write_r )( struct _reent *r, int fd,
                     const char *ptr, int len );
   long (*read_r  )( struct _reent *r, int fd,
                     char *ptr, int len );
} devoptab_t;
```

Figure 5. The devopttab_t structure.

Each device driver will supply its own operations table:

```
/* devoptab for an example stream device called "com1" */
const devoptab_t devoptab_com1 = { "com1",
                                   com1_open_r,
                                   com1_close_r,
                                   com1_write_r,
                                   com1_read_r };
```

Each driver provides its own implementations of `open_r`, `close_r`, `write_r` and `read_r` functions to handle device initialization and shutdown, and data movement to and from the physical device hardware. In the sample declaration above, these functions are named `com1_open_r()`, etc.

Somewhere in the application, gather up all the devoptab_t declarations into on place, sorted by file descriptor:

```
const devoptab_t *devoptab_list[] = {
   &dotab_com1,  /* standard input */
   &dotab_com1,  /* standard output */
   &dotab_com1,  /* standard error */
   &dotab_com2,  /* another device */
   ... ,         /* and so on... */
   0             /* terminates the list */
};
```

With all of that done, the `_write_r` stub is straightforward to implement because all it has to do is map a file descriptor to the proper set of device operations. Figure 6 shows how to do this.

```
long
_write_r ( struct _reent *ptr,
           int fd,
           const void *buf,
           size_t cnt )
{
   return devoptab_list[fd].write_r( ptr, fd, buf, cnt );
}
```

Figure 6. The `_write_r` stub.

The `_read_r` stub is identical, except that it calls the driver's `read_r` method.

The devoptab_t strategy leaves device drivers free to use whatever uC/OS services they need in order to manage reentrancy, mutual exclusion and performance issues. For example, a driver's `write_r` function could use a semaphore as a mutex to prevent two concurrent write requests, or it could use a message queue send the data to a pending task. Such details are well beyond newlib's concern, of course, but they illustrate the flexibility that is possible.

`_open_r`

This stub translates a device or file "name" to a file descriptor. With the exception of the standard input, standard output and standard error devices, this function can also be used to provide advance notice of an impending `write()` or `read()` request.

Continuing with our approach utilizing device operation tables, the `_open_r` stub can be very simple. The code is shown in Figure 7.

```
int
_open_r ( struct _reent *ptr,
          const char *file,
          int flags,
          int mode )
{
   int which_devoptab = 0;
   int fd = -1;
```

```
   /* search for "file" in dotab_list[].name */
   do {
      if( strcmp( devoptab_list[which_devoptab].name, file ) == 0 ) {
         fd = which_devoptab;
         break;
      }
   } while( devoptab_list[which_devoptab++] );

   /* if we found the requested file/device,
      then invoke the device's open_r() method */
   if( fd != -1 ) devoptab_list[fd].open_r( ptr, file, flags, mode );

   /* it doesn't exist! */
   else ptr->errno = ENODEV;

   return fd;
}
```

Figure 7. The `_open_r` stub.

You can choose to ignore the `_open_r` stub's flags and mode parameters, unless you want to add enhanced functionality like read-only or write-only file descriptors.

`_close_r`

This stub is almost a clone of `_write_r` and `_read_r`, as shown in Figure 8.

```
long
_close_r ( struct _reent *ptr,
           int fd )
{
   return devoptab_list[fd].close_r( ptr, fd );
}
```

Figure 8. The `_read_r` stub.

`_sbrk_r`

Newlib calls this stub whenever `malloc()` runs out of heap space and wants more. As it turns out, this happens frequently--- newlib's memory allocator will only ask for incremental chunks of memory, a benign artifact of its UNIX heritage.

Assuming a reserved a heap memory area using a character array called `_heap`, the `_sbrk_r` stub would look like the code in Figure 9.

```
unsigned char _heap[HEAPSIZE];

caddr_t _sbrk_r ( int incr )
{
   static unsigned char *heap_end;
   unsigned char *prev_heap_end;

   /* initialize */
```

```
   if( heap_end == 0 ) heap_end = heap;

   prev_heap_end = heap_end;

   if( heap_end + incr - heap > HEAPSIZE ) {

      /* heap overflow--- announce on stderr */
      write( 2, "Heap overflow!\n", 15 );
      abort();
   }

   heap_end += incr;

   return (caddr_t) prev_heap_end;
}
```

Figure 9. The `_sbrk_r` stub.

Each time `malloc()` calls `_sbrk_r` the heap end grows by `incr` bytes. When it encounters the end of the allocated heap space (which hopefully never occurs), the stub sends a message to the standard error stream, then forcibly terminates the program. Another approach to a heap overflow would be to return `NULL`, and let the application find a way to muddle through on its own.

`__malloc_lock` and `__malloc_unlock`

Newlib's memory management routines like `malloc()` call these functions when they need to manipulate the memory heap. By implementing mutual exclusion in them, you make newlib's memory management code reentrant--- or at least thread safe.

Portions of newlib's memory management code are recursive, so you will often see the following sequence of invocations in response to a `malloc()` function call:

```
__malloc_lock, __malloc_lock, __malloc_unlock, __malloc_unlock
```

The tricky part here is that, if you aren't careful, the second `__malloc_lock` will cause itself to wait for a lock that it already holds from the first `__malloc_lock`.

There are two ways to solve this problem. The first is to simply punt, and reimplement `malloc()` in its entirety using uC/OS's reentrant memory pool API. The second option is to really implement a working `__malloc_lock` and `__malloc_unlock`. Both approaches have their advantages, and which one you choose will depend on how your application needs to use dynamic memory.

Figure 10 is an example of how to use uC/OS memory pools to implement `malloc()`. In this code, each allocation request consumes one block from the memory pool, whether the allocation needs that much space or not. Furthermore, if the allocation size exceeds the block size then the request fails, because uC/OS's memory block manager does not permit this.

```
/* number of bytes per allocation */
#define HEAPBLKSIZE   64

/* number of allocations available */
#define HEAPBLKS    1024

/* our heap */
OS_MEM *heap;
unsigned char heapmem[HEAPBLKS * HEAPBLKSIZE];

void *malloc ( size_t size )
{
   INT8U err = OS_NO_ERR;
   void *alloc = 0;

   /* initialize, if necessary */
   OS_ENTER_CRITICAL();
   if( !heap )
      heap = OSMemCreate( heapmem, HEAPBLKS,
                          HEAPBLKSIZE, &err );
   OS_EXIT_CRITICAL();

   if( heap && err == OS_NO_ERR ) {

      /* if the request fits the heap block length,
         then make the allocation from the heap */
      if( size <= HEAPBLKLEN )
         alloc = OSMemGet( heap, &err );

      /* otherwise, we're sunk */
      else err = OS_MEM_NO_FREE_BLKS;
   }

   /* deny the allocation on errors */
   if( err != OS_NO_ERR )
      alloc = 0;

   return alloc;
}
```

Figure 10. Implementing `malloc()` with a memory pool.

Using uC/OS's memory pools eliminates fragmentation worries and makes `malloc()` reentrant, but wastes memory if the pool's block size doesn't match up with the typical allocation request. You reduce some of the waste by providing buffer pools of several different sizes (perhaps corresponding to the sizes of data structures you know you will be frequently allocating memory for), but this approach is hardly generic--- particularly when a distribution of sizes is needed.

For situations where you need a range of allocation sizes, or the size of the largest potential allocation request is unknown, use newlib's memory allocator and implement `__malloc_lock` and `__malloc_unlock` functions. Figure 11 shows how to do that.

```
/* semaphore to protect the heap */
static OS_EVENT *heapsem;

/* id of the task that is
   currently manipulating the heap */
static int lockid;

/* number of times
   __malloc_lock has recursed */
static int locks;


void
__malloc_lock ( struct _reent *_r )
{
   OS_TCB tcb;
   OS_SEM_DATA semdata;
   INT8U err;
   int id;


   /* use our priority as a task id */
   OSTaskQuery( OS_PRIO_SELF, &tcb );
   id = tcb.OSTCBPrio;

   /* see if we own the heap already */
   OSSemQuery( heapsem, &semdata );
   if( semdata.OSEventGrp && id == lockid ) {

      /* we do; just count the recursion */
      locks++;
   }

   else {
      /* wait on the other task to yield the
         heap, then claim ownership of it */
      OSSemPend( heapsem, 0, &err );
      lockid = id;
   }

   return;
}

void
__malloc_unlock ( struct _reent *_r )
{
   /* release the heap once the number of
      locks == the number of unlocks */
   if( (--locks) == 0 ) {
      lockid = -1;
      OSSemPost( heapsem );
   }
}
```

Figure 11. The `__malloc_lock` and `__malloc_unlock` functions.

`__env_lock` and `__env_unlock`

These stubs protect the application's environment memory space, similar to what `__malloc_lock` and `__malloc_unlock` do for heap space. They are related to newlib's `setenv()` and `getenv()` functions; you can ignore them if you don't use environment variables, or you can duplicate the strategy used for heap memory protection.

`_exit`

This stub forcibly terminates the application in response to the `exit()` or `system()` functions. There are several possiblities here, from allowing a watchdog timeout, to passing control to some kind of secondary application, to simulating a powerup reset in software. The Hitachi SH-2 CPU reads its initial program counter and stack pointer from the first eight bytes of memory, so the code in Figure 12 can be used to simulate a powerup reset.

```
mov #-1, r0     ; disable interrupts
ldc r0, sr

mov #4, r0      ; reset the stack pointer
mov.l @r0, r15

mov #0, r0      ; reset the program counter
mov.l @r0, r0
jmp @r0
nop
```

Figure 12. Simulating a powerup reset (Hitachi SH).

The same approach can be used for most other processors, but you have to be careful here: this technique does not restore all of the target CPU's registers and peripherals to their powerup states, so application code can not depend on initial values for proper operation. In particular, device drivers cannot enable device interrupts prior to clearing any pending interrupt requests, or a spurious interrupt will result.

`_stat_r`, `_fstat_r`, `_link_r`, `_unlink_r`, and `_lseek_r`

These stubs implement newlib's `stat()`, `fstat()`, `link()`, `unlink()` and `lseek()` functions. These functions all involve files, so they're of little importance when the target environment lacks an underlying filesystem.

For `_stat_r` and `_fstat_r`, just tell the caller that the requested file or descriptor is a character device. This code is shown in Figure 13.

```
int
_stat_r ( struct _reent *_r, const char *file,
          struct stat *pstat )
{
   pstat->st_mode = S_IFCHR;
   return 0;
}
```

```
int
_fstat_r ( struct _reent *_r, int fd, struct stat *pstat )
{
   pstat->st_mode = S_IFCHR;
   return 0;
}
```

Figure 13. The `_stat_r` and `_fstat_r` stubs.

For `_link_r` and `_unlink_r`, claim that the operation always fails. See Figure 14.

```
int
_link_r ( struct _reent *_r, const char *oldname,
              const char *newname )
{
   r->errno = EMLINK;
   return -1;
}

int
_unlink_r ( struct _reent *_r, const char *name )
{
   r->errno = EMLINK;
   return -1;
}
```

Figure 14. The `_link_r` and `_unlink_r` stubs.

For `_lseek_r`, pretend that the request is always successful. See Figure 15.

```
off_t
_lseek_r( struct _reent *_r, int fd,
                off_t pos, int whence )
{
   return 0;
}
```

Figure 15. The `_lseek_r` stub.

`getpid`

This function returns the context's process id, which we can emulate using uC/OS's `OSTaskQuery()` function, as we did for `__malloc_lock`. The code is in Figure 16.

```
int
getpid ( void )
{
   OS_TCB tcb;
   INT8U err;
   int id;


   /* use our priority as a task id */
   OSTaskQuery( OS_PRIO_SELF, &tcb );
   id = tcb.OSTCBPrio;
```

```
   return id;
}
```

Figure 16. The `getpid` stub.

`_times_r`

This stub returns various time measurements for the current context. uC/OS doesn't keep statistics on a task's run time, so leave this unimplemented as shown in Figure 17.

```
int
_times_r ( struct _reent *r, struct tms *tmsbuf )
 {
   return -1;
}
```

Figure 17. The `_times_r` stub.

Onward!

The code I provide in this article is just the minimum set needed to get newlib up and running on your system. As you grow into newlib, you are likely to find places where it makes sense to replace newlib's implementations with your own, as I often do for `malloc()`. I won't claim that newlib was designed with this in mind, but its clean implementation makes this and many other kinds of modifications simple and easy.

I hope that this document encourages you to get started on a newlib-based project of your own.

Resources

For more information about newlib, see the newlib project's home page, at http://sources.redhat.com/newlib/[7].

About the Author

Bill Gatliff is an independent consultant with almost ten years of embedded development and training experience. He specializes GNU-based embedded development, and in using and adapting GNU tools to meet the needs of difficult development problems. He welcomes the opportunity to participate in projects of all types.

Bill is a Contributing Editor for Embedded Systems Programming Magazine[8], a member of the Advisory Panel for the Embedded Systems Conference[9], maintainer of the Crossgcc FAQ, creator of the gdbstubs[10] project, and a noted author and speaker.

Bill welcomes feedback and suggestions. Contact information is on his website, at http://www.billgatliff.com[11].

Notes

1. http://sources.redhat.com/glibc
2. http://www.redhat.com/
3. newlib/libc/stdio/vfprintf.c:144
4. Actually, the `errno` macro calls the function `__errno()`, which in turn references `__impure_ptr` [errno.c:11]. This behavior is permitted by ANSI.
5. This location moves with the `--prefix` option.
6. http://www.ucos-ii.com/
7. http://sources.redhat.com/newlib
8. http://www.embedded.com/
9. http://www.esconline.com/
10. http://sourceforge.net/projects/gdbstubs
11. http://www.billgatliff.com