Embed With GNU

Porting The GNU Tools To Embedded Systems

Spring 1995
Very *Rough* Draft

Rob Savoye - Cygnus Support

Rough Draft

The goal of this document is to gather all the information needed to port the GNU tools to a new embedded target in one place. This will duplicate some info found in the other manual for the GNU tools, but this should be all you'll need.

1 Libgloss

Libgloss is a library for all the details that usually get glossed over. This library refers to things like startup code, and usually I/O support for `gcc` and `C library`. The C library used through out this manual is `newlib`. Newlib is a ANSI conforming C library developed by Cygnus Support. Libgloss could easily be made to support other C libraries, and it can be used standalone as well. The standalone configuration is typically used when bringing up new hardware, or on small systems.

For a long time, these details were part of newlib. This approach worked well when a complete tool chain only had to support one system. A tool chain refers to the series of compiler passes required to produce a binary file that will run on an embedded system. For C, the passes are cpp, gcc, gas, ld. Cpp is the preprocessor, which process all the header files and macros. Gcc is the compiler, which produces assembler from the processed C files. Gas assembles the code into object files, and then ld combines the object files and binds the code to addresses and produces the final executable image.

Most of the time a tool chain does only have to support one target execution environment. An example of this would be a tool chain for the AMD 29k processor family. All of the execution environments for this processor have the same interface, the same memory map, and the same I/O code. In this case all of the support code is under newlib/libc/sys. Libgloss's creation was forced initially because of the `cpu32` processor family. There are many different execution environments for this line, and they vary wildly. newlib itself has only a few dependencies that it needs for each target. These are explained later in this doc. The hardware dependent part of newlib was reorganized into a separate directory structure within newlib called the stub dirs. It was initially called this because most of the routines newlib needs for a target were simple stubs that do nothing, but return a value to the application. They only exist so the linker can produce a final executable image. This work was done during the early part of 1993.

After a while it became apparent that this approach of isolating the hardware and systems files together made sense. Around this same time the stub dirs were made to run standalone, mostly so it could also be used to support GDB's remote debugging needs. At this time it was decided to move the stub dirs out of newlib and into it's own separate library so it could be used standalone, and be included in various other GNU tools without having to bring in all of newlib, which is large. The new library is called Libgloss, for Gnu Low-level OS support.

1.1 Supported Targets

Currently libgloss is being used for the following targets:

1.1.1 Sparclite Targets Supported

This is for the Fujitsu Sparclite family of processors. Currently this covers the ex930, ex931, ex932, ex933, and the ex934. In addition to the I/O code a startup file, this has a GDB debug-stub that gets linked into your application. This is an exception handler style debug stub. For more info, see the section on Porting GDB. Chapter 4 [Porting GDB], page 15.

The Fujitsu eval boards use a host based terminal program to load and execute programs on the target. This program, `pciuh` is relatively new (in 1994) and it replaced the previous ROM monitor which had the shell in the ROM. GDB uses the the GDB remote protocol, the relevant source files from the gdb sources are remote-sparcl.c. The debug stub is part of libgloss and is called sparcl-stub.c.

1.1.2 Motorola CPU32 Targets supported

This refers to Motorola's m68k based CPU32 processor family. The crt0.S startup file should be usable with any target environment, and it's mostly just the I/O code and linker scripts that vary. Currently there is support for the Motorola MVME line of 6U VME boards and IDP line of eval boards. All of the Motorola VME boards run `Bug`, a ROM based debug monitor. This monitor has the feature of using user level traps to do I/O, so this code should be portable to other MVME boards with little if any change. The startup file also can remain unchanged. About the only thing that varies is the address for where the text section begins. This can be accomplished either in the linker script, or on the command line using the '`-Ttext [address]`'.

There is also support for the `rom68k` monitor as shipped on Motorola's IDP eval board line. This code should be portable across the range of CPU's the board supports. There is also GDB support for this target environment in the GDB source tree. The relevant files are gdb/monitor.c, monitor.h, and rom58k-rom.c. The usage of these files is discussed in the GDB section.

1.1.3 Mips core Targets Supported

The Crt0 startup file should run on any mips target that doesn't require additional hardware initialization. The I/O code so far only supports a custom LSI33k based RAID disk controller board. It should easy to change to support the IDT line of eval boards. Currently the two debugging protocols supported by GDB for mips targets is IDT's mips debug protocol, and a customized hybrid of the standard GDB remote protocol and GDB's standard ROM monitor support. Included here is the debug stub for the hybrid monitor. This supports the LSI33k processor, and only has support for the GDB protocol commands `g`, `G`, `m`, `M`, which basically only supports the register and memory reading and writing commands. This is part of libgloss and is called lsi33k-stub.c.

The crt0.S should also work on the IDT line of eval boards, but has only been run on the LSI33k for now. There is no I/O support for the IDT eval board at this time. The current I/O code is for a customized version of LSI's `pmon` ROM monitor. This uses entry points into the monitor, and should easily port to other versions of the pmon monitor. Pmon is distributed in source by LSI.

1.1.4 PA-RISC Targets Supported

This supports the various boards manufactured by the HP-PRO consortium. This is a group of companies all making variations on the PA-RISC processor. Currently supported are ports to the WinBond '`Cougar`' board based around their w89k version of the PA. Also supported is the Oki op50n processor.

There is also included, but never built an unfinished port to the HP 743 board. This board is the main CPU board for the HP700 line of industrial computers. This target isn't exactly an embedded system, in fact it's really only designed to load and run HP-UX. Still, the crt0.S and I/O code are fully working. It is included mostly because their is a barely functioning exception handler GDB debug stub, and I hope somebody could use it. The other PRO targets all use GDB's ability to talk to ROM monitors directly, so it doesn't need a debug stub. There is also a utility that will produce a bootable file by HP's ROM monitor. This is all included in the hopes somebody else will finish it. :-)

Both the WinBond board and the Oki board download srecords. The WinBond board also has support for loading the SOM files as produced by the native compiler on HP-UX. WinBond supplies a set of DOS programs that will allow the loading of files via a bidirectional parallel port. This has never been tested with the output of GNU SOM, as this manual is mostly for Unix based systems.

1.2 Configuring and building libgloss.

Libgloss uses an autoconf based script to configure. Autoconf scripts are portable shell scripts that are generated from a configure.in file. Configure input scripts are based themselves on m4. Most configure scripts run a series of tests to determine features the various supported features of the target. For features that can't be determined by a feature test, a makefile fragment is merged in. The configure process leaves creates a Makefile in the build directory. For libgloss, there are only a few configure options of importance. These are –target and –srcdir.

Typically libgloss is built in a separate tree just for objects. In this manner, it's possible to have a single source tree, and multiple object trees. If you only need to configure for a single target environment, then you can configure in the source tree. The argument for –target is a config string. It's usually safest to use the full canonical opposed to the target alias. So, to configure for a CPU32 (m68k) with a separate source tree, use:

```
../src/libgloss/configure --verbose --target m68k-coff
```

The configure script is in the source tree. When configure is invoked it will determine it's own source tree, so the –srcdir is would be redundant here.

Once libgloss is configured, `make` is sufficient to build it. The default values for `Makefiles` are typically correct for all supported systems. The test cases in the testsuite will also built automatically as opposed to a `make check`, where test binaries aren't built till test time. This is mostly cause the libgloss testsuites are the last thing built when building the entire GNU source tree, so it's a good test of all the other compilation passes.

The default values for the Makefiles are set in the Makefile fragment merged in during configuration. This fragment typically has rules like

```
CC_FOR_TARGET = `if [ -f $${OBJROOT}/gcc/xgcc ] ; \
then echo ${OBJROOT}/gcc/xgcc -B${OBJROOT}/gcc/ ; \
else t='${program_transform_name}'; echo gcc | sed -e '' $$t ; fi`
```

Basically this is a runtime test to determine whether there are freshly built executables for the other main passes of the GNU tools. If there isn't an executable built in the same object

tree, then *transformed*the generic tool name (like gcc) is transformed to the name typically used in GNU cross compilers. The names are typically based on the target's canonical name, so if you've configured for `m68k-coff` the transformed name is `m68k-coff-gcc` in this case. If you install with aliases or rename the tools, this won't work, and it will always look for tools in the path. You can force the a different name to work by reconfiguring with the `--program-transform-name` option to configure. This option takes a sed script like this `-e s,^,m68k-coff-,` which produces tools using the standard names (at least here at Cygnus).

The search for the other GNU development tools is exactly the same idea. This technique gets messier when build options like `-msoft-float` support are used. The Makefile fragments set the `MUTILIB` variable, and if it is set, the search path is modified. If the linking is done with an installed cross compiler, then none of this needs to be used. This is done so libgloss will build automatically with a fresh, and uninstalled object tree. It also makes it easier to debug the other tools using libgloss's test suites.

1.3 Adding Support for a New Board

This section explains how to add support for a new board to libgloss. In order to add support for a board, you must already have developed a toolchain for the target architecture.

All of the changes you will make will be in the subdirectory named after the architecture used by your board. For example, if you are developing support for a new ColdFire board, you will modify files in the '`m68k`' subdirectory, as that subdirectory contains support for all 68K devices, including architecture variants like ColdFire.

In general, you will be adding three components: a '`crt0.S`' file (see Section 3.1 [Crt0], page 6), a linker script (see Section 3.2 [Linker Scripts], page 9), and a hardware support library. Each should be prefixed with the name of your board. For example, if you ard adding support for a new Surf board, then you will be adding the assembly '`surf-crt0.S`' (which will be assembled into '`surf-crt0.o`'), the linker script '`surf.ld`', and other C and assembly files which will be combined into the hardware support library '`libsurf.a`'.

You should modify '`Makefile.in`' to define new variables corresponding to your board. Although there is some variation between architectures, the general convention is to use the following format:

```
# The name of the crt0.o file.
SURF_CRT0    = surf-crt0.o
# The name of the linker script.
SURF_SCRIPTS = surf.ld
# The name of the hardware support library.
SURF_BSP     = libsurf.a
# The object files that make up the hardware support library.
SURF_OBJS    = surf-file1.o surf-file2.o
# The name of the Makefile target to use for installation.
SURF_INSTALL = install-surf
```

Then, you should create the `${SURF_BSP}` and `${SURF_INSTALL}` make targets. Add `${SURF_CRT0}` to the dependencies for the `all` target and add `${SURF_INSTALL}` to the dependencies for the `install` target. Now, when libgloss is built and installed, support for your BSP will be installed as well.

2 Porting GCC

Porting GCC requires two things, neither of which has anything to do with GCC. If GCC already supports a processor type, then all the work in porting GCC is really a linker issue. All GCC has to do is produce assembler output in the proper syntax. Most of the work is done by the linker, which is described elsewhere.

Mostly all GCC does is format the command line for the linker pass. The command line for GCC is set in the various config subdirectories of gcc. The options of interest to us are `CPP_SPEC` and `STARTFILE_SPEC`. CPP_SPEC sets the builtin defines for your environment. If you support multiple environments with the same processor, then OS specific defines will need to be elsewhere.

`STARTFILE_SPEC`

Once you have linker support, GCC will be able to produce a fully linked executable image. The only *part* of GCC that the linker wants is a crt0.o, and a memory map. If you plan on running any programs that do I/O of any kind, you'll need to write support for the C library, which is described elsewhere.

2.1 Compilation passes

GCC by itself only compiles the C or C++ code into assembler. Typically GCC invokes all the passes required for you. These passes are cpp, cc1, gas, ld. `cpp` is the C preprocessor. This will merge in the include files, expand all macros definitions, and process all the `#ifdef` sections. To see the output of ccp, invoke gcc with the `-E` option, and the preprocessed file will be printed on the stdout. cc1 is the actual compiler pass that produces the assembler for the processed file. GCC is actually only a driver program for all the compiler passes. It will format command line options for the other passes. The usual command line GCC uses for the final link phase will have LD link in the startup code and additional libraries by default.

GNU AS started it's life to only function as a compiler pass, but these days it can also be used as a source level assembler. When used as a source level assembler, it has a companion assembler preprocessor called `gasp`. This has a syntax similar to most other assembler macros packages. GAS emits a relocatable object file from the assembler source. The object file contains the executable part of the application, and debug symbols.

LD is responsible for resolving the addresses and symbols to something that will be fully self-contained. Some RTOS's use relocatable object file formats like `a.out`, but more commonly the final image will only use absolute addresses for symbols. This enables code to be burned into PROMS as well. Although LD can produce an executable image, there is usually a hidden object file called `crt0.o` that is required as startup code. With this startup code and a memory map, the executable image will actually run on the target environment. Section 3.1 [Startup Files], page 6.

The startup code usually defines a special symbol like `_start` that is the default base address for the application, and the first symbol in the executable image. If you plan to use any routines from the standard C library, you'll also need to implement the functions that this library is dependent on. Chapter 3 [Porting Newlib], page 6.

Options for the various development tools are covered in more detail elsewhere. Still, the amount of options can be an overwhelming amount of stuff, so the options most suited to embedded systems are summarized here. If you use GCC as the main driver for all the

passes, most of the linker options can be passed directly to the compiler. There are also GCC options that control how the GCC driver formats the command line arguments for the linker.

Most of the GCC options that we're interested control how the GCC driver formats the options for the linker pass.

`-nostartfiles`
`-nostdlib`
`-Xlinker` Pass the next option directly to the linker.

`-v`

`-fpic`

3 Porting newlib

3.1 Crt0, the main startup file

To make a program that has been compiled with GCC to run, you need to write some startup code. The initial piece of startup code is called a crt0. (C RunTime 0) This is usually written in assembler, and it's object gets linked in first, and bootstraps the rest of the application when executed. This file needs to do the following things.

1. Initialize anything that needs it. This init section varies. If you are developing an application that gets download to a ROM monitor, then there is usually no need for any special initialization. The ROM monitor handles it for you.

   If you plan to burn your code in a ROM, then the crt0 typically has to do all the hardware initialization that is required to run an application. This can include things like initializing serial ports or run a memory check. It all depends on the hardware.
2. Zero the BSS section. This is for uninitialized data. All the addresses in this section need to be initialized to zero so that programs that forget to check new variables default value will get unpredictable results.
3. Call main() This is what basically starts things running. If your ROM monitor supports it, then first setup argc and argv for command line arguments and an environment pointer. Then branch to main(). For G++ the the main routine gets a branch to __main inserted by the code generator at the very top. __main() is used by G++ to initialize it's internal tables. __main() then returns back to your original main() and your code gets executed.
4. Call exit() After main() has returned, you need to cleanup things and return control of the hardware from the application. On some hardware, there is nothing to return to, especially if your program is in ROM. Sometimes the best thing to do in this case is do a hardware reset, or branch back to the start address all over again.

   When there is a ROM monitor present, usually a user trap can be called and then the ROM takes over. Pick a safe vector with no side effects. Some ROMs have a builtin trap handler just for this case.

portable between all the m68k based boards we have here. Section A.2 [Example Crt0.S], page 20.

```
/* ANSI concatenation macros.  */

#define CONCAT1(a, b) CONCAT2(a, b)
#define CONCAT2(a, b) a ## b
```

These we'll use later.

```
/* These are predefined by new versions of GNU cpp.  */

#ifndef __USER_LABEL_PREFIX__
#define __USER_LABEL_PREFIX__ _
#endif

/* Use the right prefix for global labels.  */
#define SYM(x) CONCAT1 (__USER_LABEL_PREFIX__, x)
```

These macros are to make this code portable between both *COFF* and *a.out*. *COFF* always has an _ *(underline)* prepended on the front of all global symbol names. *a.out* has none.

```
#ifndef __REGISTER_PREFIX__
#define __REGISTER_PREFIX__
#endif

/* Use the right prefix for registers.  */
#define REG(x) CONCAT1 (__REGISTER_PREFIX__, x)

#define d0 REG (d0)
#define d1 REG (d1)
#define d2 REG (d2)
#define d3 REG (d3)
#define d4 REG (d4)
#define d5 REG (d5)
#define d6 REG (d6)
#define d7 REG (d7)
#define a0 REG (a0)
#define a1 REG (a1)
#define a2 REG (a2)
#define a3 REG (a3)
#define a4 REG (a4)
#define a5 REG (a5)
#define a6 REG (a6)
#define fp REG (fp)
#define sp REG (sp)
```

This is for portability between assemblers. Some register names have a % or $ prepended to the register name.

```
/*
 * Set up some room for a stack. We just grab a chunk of memory.
 */
.set stack_size, 0x2000
.comm SYM (stack), stack_size
```

Set up space for the stack. This can also be done in the linker script, but it typically gets done here.

```
/*
 * Define an empty environment.
 */
        .data
        .align 2
SYM (environ):
        .long 0
```

Set up an empty space for the environment. This is bogus on any most ROM monitor, but we setup a valid address for it, and pass it to main. At least that way if an application checks for it, it won't crash.

```
  .align 2
.text
.global SYM (stack)

.global SYM (main)
.global SYM (exit)
/*
 * This really should be __bss_start, not SYM (__bss_start).
 */
.global __bss_start
```

Setup a few global symbols that get used elsewhere. *__bss_start* needs to be unchanged, as it's setup by the linker script.

```
/*
 * start -- set things up so the application will run.
 */
SYM (start):
link a6, #-8
moveal #SYM (stack) + stack_size, sp

/*
 * zerobss -- zero out the bss section
 */
moveal #__bss_start, a0
moveal #SYM (end), a1
1:
movel #0, (a0)
leal 4(a0), a0
cmpal a0, a1
bne 1b
```

The global symbol `start` is used by the linker as the default address to use for the `.text` section. then it zeros the `.bss` section so the uninitialized data will all be cleared. Some programs have wild side effects from having the .bss section let uncleared. Particularly it causes problems with some implementations of `malloc`.

```
/*
 * Call the main routine from the application to get it going.
 * main (argc, argv, environ)
 * We pass argv as a pointer to NULL.
 */
       pea     0
       pea     SYM (environ)
       pea     sp@(4)
       pea     0
jsr SYM (main)
movel d0, sp@-
```

Setup the environment pointer and jump to `main()`. When `main()` returns, it drops down to the `exit` routine below.

```
/*
 * _exit -- Exit from the application. Normally we cause a user trap
 *          to return to the ROM monitor for another run.
 */
SYM (exit):
trap #0
```

Implementing `exit` here is easy. Both the `rom68k` and `bug` can handle a user caused exception of `zero` with no side effects. Although the `bug` monitor has a user caused trap that will return control to the ROM monitor, this solution has been more portable.

3.2 Linker scripts for memory management

The linker script sets up the memory map of an application. It also sets up default values for variables used elsewhere by sbrk() and the crt0. These default variables are typically called `_bss_start` and `_end`.

For G++, the constructor and destructor tables must also be setup here. The actual section names vary depending on the object file format. For `a.out` and `coff`, the three main sections are `.text`, `.data`, and `.bss`.

Now that you have an image, you can test to make sure it got the memory map right. You can do this by having the linker create a memory map (by using the `-Map` option), or afterwards by using `nm` to check a few critical addresses like `start`, `bss_end`, and `_etext`.

Here's a breakdown of a linker script for a m68k based target board. See the file `libgloss/m68k/idp.ld`, or go to the appendixes in the end of the manual. Section A.1 [Example Linker Script], page 19.

```
STARTUP(crt0.o)
OUTPUT_ARCH(m68k)
INPUT(idp.o)
SEARCH_DIR(.)
__DYNAMIC  =  0;
```

The `STARTUP` command loads the file specified so that it's first. In this case it also doubles to load the file as well, because the m68k-coff configuration defaults to not linking in the crt0.o by default. It assumes that the developer probably has their own crt0.o. This behavior is controlled in the config file for each architecture. It's a macro called `STARTFILE_SPEC`, and if it's set to `null`, then when `gcc` formats it's command line, it doesn't add `crto.o`. Any file name can be specified here, but the default is always `crt0.o`.

Course if you only use `ld` to link, then the control of whether or not to link in `crt0.o` is done on the command line. If you have multiple crto files, then you can leave this out all together, and link in the `crt0.o` in the makefile, or by having different linker scripts. Sometimes this is done for initializing floating point optionally, or to add device support.

The `OUTPUT_ARCH` sets architecture the output file is for.

`INPUT` loads in the file specified. In this case, it's a relocated library that contains the definitions for the low-level functions need by libc.a. This could have also been specified on the command line, but as it's always needed, it might as well be here as a default. `SEARCH_DIR` specifies the path to look for files, and `_DYNAMIC` means in this case there are no shared libraries.

```
/*
 * Setup the memory map of the MC68ec0x0 Board (IDP)
 * stack grows up towards high memory. This works for
 * both the rom68k and the mon68k monitors.
 */
MEMORY
{
  ram     : ORIGIN = 0x10000, LENGTH = 2M
}
```

This specifies a name for a section that can be referred to later in the script. In this case, it's only a pointer to the beginning of free RAM space, with an upper limit at 2M. If the output file exceeds the upper limit, it will produce an error message.

```
/*
 * stick everything in ram (of course)
 */
SECTIONS
{
  .text :
  {
    CREATE_OBJECT_SYMBOLS
    *(.text)
     etext  =  .;
     __CTOR_LIST__ = .;
     LONG((__CTOR_END__ - __CTOR_LIST__) / 4 - 2)
    *(.ctors)
     LONG(0)
     __CTOR_END__ = .;
     __DTOR_LIST__ = .;
     LONG((__DTOR_END__ - __DTOR_LIST__) / 4 - 2)
    *(.dtors)
     LONG(0)
     __DTOR_END__ = .;
    *(.lit)
    *(.shdata)
  }  > ram
  .shbss SIZEOF(.text) + ADDR(.text) :  {
    *(.shbss)
  }
```

Set up the `.text` section. In a `COFF` file, .text is where all the actual instructions are. This also sets up the *CONTRUCTOR* and the *DESTRUCTOR* tables for `G++`. Notice that the section description redirects itself to the *ram* variable setup earlier.

```
  .talias :   { }  > ram
  .data  : {
    *(.data)
    CONSTRUCTORS
    _edata  =  .;
  } > ram
```

Setup the `.data` section. In a `coff` file, this is where all he initialized data goes. `CONSTRUCTORS` is a special command used by `ld`.

```
  .bss SIZEOF(.data) + ADDR(.data) :
  {
   __bss_start = ALIGN(0x8);
   *(.bss)
   *(COMMON)
      end = ALIGN(0x8);
      _end = ALIGN(0x8);
      __end = ALIGN(0x8);
  }
  .mstack  : { }  > ram
  .rstack  : { }  > ram
  .stab  . (NOLOAD) :
  {
    [ .stab ]
  }
  .stabstr  . (NOLOAD) :
  {
    [ .stabstr ]
```

```
  }
}
```

Setup the `.bss` section. In a `COFF` file, this is where unitialized data goes. The symbols `_bss_start` and `_end` are setup here for use by the `crt0.o` when it zero's the `.bss` section.

3.3 What to do when you have a binary image

A few ROM monitors load binary images, typically `a.out`, but most all will load an `srecord`. An srecord is an ASCII representation of a binary image. At it's simplest, an srecord is an address, followed by a byte count, followed by the bytes, and a 2's compliment checksum. A whole srecord file has an optional *start* record, and a required *end* record. To make an srecord from a binary image, the GNU `objcopy` program is used. This will read the image and make an srecord from it. To do this, invoke objcopy like this: `objcopy -O srec infile outfile`. Most PROM burners also read srecords or a similar format. Use `objdump -i` to get a list of support object files types for your architecture.

3.4 Libraries

This describes `newlib`, a freely available libc replacement. Most applications use calls in the standard C library. When initially linking in libc.a, several I/O functions are undefined. If you don't plan on doing any I/O, then you're OK, otherwise they need to be created. These routines are read, write, open, close. sbrk, and kill. Open & close don't need to be fully supported unless you have a filesystems, so typically they are stubbed out. Kill is also a stub, since you can't do process control on an embedded system.

Sbrk() is only needed by applications that do dynamic memory allocation. It's uses the symbol `_end` that is setup in the linker script. It also requires a compile time option to set the upper size limit on the heap space. This leaves us with read and write, which are required for serial I/O. Usually these two routines are written in C, and call a lower level function for the actual I/O operation. These two lowest level I/O primitives are inbyte() and outbyte(), and are also used by GDB back ends if you've written an exception handler. Some systems also implement a havebyte() for input as well.

Other commonly included functions are routines for manipulating LED's on the target (if they exist) or low level debug help. Typically a putnum() for printing words and bytes as a hex number is helpful, as well as a low-level print() to output simple strings.

As libg++ uses the I/O routines in libc.a, if read and write work, then libg++ will also work with no additional changes.

3.4.1 Making I/O work

3.4.2 Routines for dynamic memory allocation

To support using any of the memory functions, you need to implement sbrk(). `malloc()`, `calloc()`, and `realloc()` all call `sbrk()` at there lowest level. `caddr_t` is defined elsewhere as `char *`. `RAMSIZE` is presently a compile time option. All this does is move a pointer to heap memory and check for the upper limit. Section A.3 [Example libc support code], page 22. `sbrk()` returns a pointer to the previous value before more memory was allocated.

```
/* _end is set in the linker command file *
extern caddr_t _end;/

/* just in case, most boards have at least some memory */
#ifndef RAMSIZE
#  define RAMSIZE             (caddr_t)0x100000
#endif

/*
 * sbrk -- changes heap size size. Get nbytes more
 *         RAM. We just increment a pointer in what's
 *         left of memory on the board.
 */
caddr_t
sbrk(nbytes)
     int nbytes;
{
  static caddr_t heap_ptr = NULL;
  caddr_t        base;

  if (heap_ptr == NULL) {
    heap_ptr = (caddr_t)&_end;
  }

  if ((RAMSIZE - heap_ptr) >= 0) {
    base = heap_ptr;
    heap_ptr += nbytes;
    return (base);
  } else {
    errno = ENOMEM;
    return ((caddr_t)-1);
  }
}
```

3.4.3 Misc support routines

These are called by `newlib` but don't apply to the embedded environment. `isatty()` is self explanatory. `kill()` doesn't apply either in an environment withno process control, so it justs exits, which is a similar enough behavior. `getpid()` can safely return any value greater than 1. The value doesn't effect anything in `newlib` because once again there is no process control.

```
/*
 * isatty -- returns 1 if connected to a terminal device,
 *           returns 0 if not. Since we're hooked up to a
 *           serial port, we'll say yes and return a 1.
 */
int
isatty(fd)
     int fd;
{
  return (1);
}

/*
 * getpid -- only one process, so just return 1.
 */
#define __MYPID 1
int
getpid()
{
```

```
  return __MYPID;
}

/*
 * kill -- go out via exit...
 */
int
kill(pid, sig)
     int pid;
     int sig;
{
  if(pid == __MYPID)
    _exit(sig);
  return 0;
}
```

3.4.4 Useful debugging functions

There are always a few useful functions for debugging your project in progress. I typically implement a simple `print()` routine that runs standalone in liblgoss, with no `newlib` support. The I/O function `outbyte()` can also be used for low level debugging. Many times print will work when there are problems that cause `printf()` to cause an exception. `putnum()` is just to print out values in hex so they are easier to read.

```
/*
 * print -- do a raw print of a string
 */
int
print(ptr)
char *ptr;
{
  while (*ptr) {
    outbyte (*ptr++);
  }
}

/*
 * putnum -- print a 32 bit number in hex
 */
int
putnum (num)
unsigned int num;
{
  char  buffer[9];
  int   count;
  char  *bufptr = buffer;
  int   digit;

  for (count = 7 ; count >= 0 ; count--) {
    digit = (num >> (count * 4)) & 0xf;

    if (digit <= 9)
      *bufptr++ = (char) ('0' + digit);
    else
      *bufptr++ = (char) ('a' - 10 + digit);
  }

  *bufptr = (char) 0;
  print (buffer);
  return;
}
```

If there are LEDs on the board, they can also be put to use for debugging when the serial I/O code is being written. I usually implement a `zylons()` function, which strobes the LEDS (if there is more than one) in sequence, creating a rotating effect. This is convenient between I/O to see if the target is still alive. Another useful LED function is `led_putnum()`, which takes a digit and displays it as a bit pattern or number. These usually have to be written in assembler for each target board. Here are a number of C based routines that may be useful.

`led_putnum()` puts a number on a single digit segmented LED display. This LED is set by setting a bit mask to an address, where 1 turns the segment off, and 0 turns it on. There is also a little decimal point on the LED display, so it gets the leftmost bit. The other bits specify the segment location. The bits look like:

```
[d.p | g | f | e | d | c | b | a ] is the byte.
```

The locations are set up as:

```
    a
   -----
f |     | b
  |  g  |
   -----
  |     |
e |     | c
   -----
    d
```

This takes a number that's already been converted to a string, and prints it.

```
#define LED_ADDR 0xd00003

void
led_putnum ( num )
char num;
{
    static unsigned char *leds = (unsigned char *)LED_ADDR;
    static unsigned char num_bits [18] = {
      0xff, /* clear all */
      0xc0, 0xf9, 0xa4, 0xb0, 0x99, 0x92, 0x82, 0xf8, 0x80, 0x98, /* numbers 0-9 */
      0x98, 0x20, 0x3, 0x27, 0x21, 0x4, 0xe  /* letters a-f */
    };

    if (num >= '0' && num <= '9')
      num = (num - '0') + 1;

    if (num >= 'a' && num <= 'f')
      num = (num - 'a') + 12;

    if (num == ' ')
      num = 0;

    *leds = num_bits[num];
}

/*
 * zylons -- draw a rotating pattern. NOTE: this function never returns.
 */
void
zylons()
{
  unsigned char *leds  = (unsigned char *)LED_ADDR;
  unsigned char curled = 0xfe;
```

```
  while (1)
    {
      *leds = curled;
      curled = (curled >> 1) | (curled << 7);
      delay ( 200 );
    }
}
```

4 Writing a new GDB backend

Typically, either the low-level I/O routines are used for debugging, or LEDs, if present. It is much easier to use GDb for debugging an application. There are several different techniques used to have GDB work remotely. Commonly more than one kind of GDB interface is used to cober a wide variety of development needs.

The most common style of GDB backend is an exception handler for breakpoints. This is also called a *gdb stub*, and is requires the two additional lines of init code in your `main()` routine. The GDB stubs all use the GDB *remote protocol*. When the application gets a breakpoint exception, it communicates to GDB on the host.

Another common style of interfacing GDB to a target is by using an existing ROM monitor. These break down into two main kinds, a similar protocol to the GDB remote protocol, and an interface that uses the ROM monitor directly. This kind has GDB simulating a human operator, and all GDB does is work as a command formatter and parser.

4.1 The standard remote protocol

The standard remote protocol is a simple, packet based scheme. A debug packet whose contents are <*data*> is encapsulated for transmission in the form:

```
$ <data> # CSUM1 CSUM2
```

<*data*> must be ASCII alphanumeric and cannot include characters `$` or `#`. If <*data*> starts with two characters followed by `:`, then the existing stubs interpret this as a sequence number. For example, the command `g` is used to read the values of the registers. So, a packet to do this would look like

```
$g#67
```

CSUM1 and *CSUM2* are an ascii representation in hex of an 8-bit checksum of <*data*>, the most significant nibble is sent first. the hex digits 0-9,a-f are used.

A simple protocol is used when communicating with the target. This is mainly to give a degree of error handling over the serial cable. For each packet transmitted successfully, the target responds with a `+` (`ACK`). If there was a transmission error, then the target responds with a `-` (`NAK`). An error is determined when the checksum doesn't match the calculated checksum for that data record. Upon reciept of the `ACK`, `GDB` can then transmit the next packet.

Here is a list of the main functions that need to be supported. Each data packet is a command with a set number of bytes in the command packet. Most commands either return data, or respond with a `NAK`. Commands that don't return data respond with an `ACK`. All data values are ascii hex digits. Every byte needs two hex digits to represent t. This means that a byte with the value '`7`' becomes '`07`'. On a 32 bit machine this works out to 8 characters per word. All of the bytes in a word are stored in the target byte order.

When writing the host side of the GDB protocol, be careful of byte order, and make sure that the code will run on both big and little endian hosts and produce the same answers.

These functions are the minimum required to make a GDB backend work. All other commands are optional, and not supported by all GDB backends.

'`read registers g`'
: returns `XXXXXXXX...`

  Registers are in the internal order for GDB, and the bytes in a register are in the same order the machine uses. All values are in sequence starting with register 0. All registers are listed in the same packet. A sample packet would look like `$g#`.

'`write registers GXXXXXXXX...`'
: `XXXXXXXX` is the value to set the register to. Registers are in the internal order for GDB, and the bytes in a register are in the same order the machine uses. All values are in sequence starting with register 0. All registers values are listed in the same packet. A sample packet would look like `$G00000000111111122222222...#`

  returns `ACK` or `NAK`

'`read memory mAAAAAAAA,LLLL`'
: `AAAAAAAA` is address, `LLLL` is length. A sample packet would look like `$m00005556,0024#`. This would request 24 bytes starting at address *00005556*

  returns `XXXXXXXX... XXXXXXXX` is the memory contents. Fewer bytes than requested will be returned if only part of the data can be read. This can be determined by counting the values till the end of packet `#` is seen and comparing that with the total count of bytes that was requested.

'`write memory MAAAAAAAA,LLLL:XXXXXXXX`'
: `AAAAAAAA` is the starting address, `LLLL` is the number of bytes to be written, and `XXXXXXXX` is value to be written. A sample packet would look like `$M00005556,0024:10101010111111100000000...#`

  returns `ACK` or `NAK` for an error. `NAK` is also returned when only part of the data is written.

'`continue cAAAAAAAAA`'
: `AAAAAAAA` is address to resume execution at. If `AAAAAAAA` is omitted, resume at the curent address of the `pc` register.

  returns the same replay as `last signal`. There is no immediate replay to `cont` until the next breakpoint is reached, and the program stops executing.

'`step sAA..AA`'
: `AA..AA` is address to resume If `AA..AA` is omitted, resume at same address.

  returns the same replay as `last signal`. There is no immediate replay to `step` until the next breakpoint is reached, and the program stops executing.

'`last signal ?`'
: This returns one of the following:

  - `SAA` Where `AA` is the number of the last signal. Exceptions on the target are converted to the most similar Unix style signal number, like `SIGSEGV`. A sample response of this type would look like `$S05#`.

- TAAnn:XXXXXXXX;nn:XXXXXXXX;nn:XXXXXXXX; `AA` is the signal number. `nn` is the register number. `XXXXXXXX` is the register value.
- WAA The process exited, and `AA` is the exit status. This is only applicable for certains sorts of targets.

These are used in some GDB backends, but not all.

'`write reg Pnn=XXXXXXXX`'
Write register `nn` with value `XXXXXXXX`.

returns `ACK` or `NAK`

'`kill request k`'
'`toggle debug d`'
toggle debug flag (see 386 & 68k stubs)

'`reset r`' reset – see sparc stub.

'`reserved other`'
On other requests, the stub should ignore the request and send an empty response `$#<checksum>`. This way we can extend the protocol and GDB can tell whether the stub it is talking to uses the old or the new.

'`search tAA:PP,MM`'
Search backwards starting at address `AA` for a match with pattern PP and mask `MM`. `PP` and `MM` are 4 bytes.

'`general query qXXXX`'
Request info about XXXX.

'`general set QXXXX=yyyy`'
Set value of `XXXX` to `yyyy`.

'`query sect offs qOffsets`'
Get section offsets. Reply is `Text=xxx;Data=yyy;Bss=zzz`

'`console output Otext`'
Send text to stdout. The text gets display from the target side of the serial connection.

Responses can be run-length encoded to save space. A `*`means that the next character is an ASCII encoding giving a repeat count which stands for that many repetitions of the character preceding the `*`. The encoding is n+29, yielding a printable character where n >=3 (which is where run length encoding starts to win). You can't use a value of where n >126 because it's only a two byte value. An example would be a `0*03` means the same thing as `0000`.

4.2 A linked in exception handler

A *GDB stub* consists of two parts, support for the exception handler, and the exception handler itself. The exception handler needs to communicate to GDB on the host whenever there is a breakpoint exception. When GDB starts a program running on the target, it's polling the serial port during execution looking for any debug packets. So when a breakpoint occurs, the exception handler needs to save state, and send a GDB remote protocol packet to GDB on the host. GDB takes any output that isn't a debug command packet and displays it in the command window.

Support for the exception handler varies between processors, but the minimum supported functions are those needed by GDB. These are functions to support the reading and writing of registers, the reading and writing of memory, start execution at an address, single step, and last signal. Sometimes other functions for adjusting the baud rate, or resetting the hardware are implemented.

Once GDB gets the command packet from the breakpoint, it will read a few registers and memory locations an then wait for the user. When the user types `run` or `continue` a `continue` command is issued to the backend, and control returns from the breakpoint routine to the application.

4.3 Using a ROM monitor as a backend

GDB also can mimic a human user and use a ROM monitors normal debug commands as a backend. This consists mostly of sending and parsing `ASCII` strings. All the ROM monitor interfaces share a common set of routines in `gdb/monitor.c`. This supports adding new ROM monitor interfaces by filling in a structure with the common commands GDB needs. GDb already supports several command ROM monitors, including Motorola's `Bug` monitor for their VME boards, and the Rom68k monitor by Integrated Systems, Inc. for various m68k based boards. GDB also supports the custom ROM monitors on the WinBond and Oki PA based targets. There is builtin support for loading files to ROM monitors specifically. GDB can convert a binary into an srecord and then load it as an ascii file, or using `xmodem`.

4.4 Adding support for new protocols

Appendix A Code Listings

A.1 Linker script for the IDP board

This is the linker script script that is used on the Motorola IDP board.

```
STARTUP(crt0.o)
OUTPUT_ARCH(m68k)
INPUT(idp.o)
SEARCH_DIR(.)
__DYNAMIC  =  0;
/*
 * Setup the memory map of the MC68ec0x0 Board (IDP)
 * stack grows up towards high memory. This works for
 * both the rom68k and the mon68k monitors.
 */
MEMORY
{
  ram     : ORIGIN = 0x10000, LENGTH = 2M
}
/*
 * stick everything in ram (of course)
 */
SECTIONS
{
  .text :
  {
    CREATE_OBJECT_SYMBOLS
    *(.text)
     etext  =  .;
     __CTOR_LIST__ = .;
     LONG((__CTOR_END__ - __CTOR_LIST__) / 4 - 2)
    *(.ctors)
     LONG(0)
     __CTOR_END__ = .;
     __DTOR_LIST__ = .;
     LONG((__DTOR_END__ - __DTOR_LIST__) / 4 - 2)
    *(.dtors)
     LONG(0)
     __DTOR_END__ = .;
    *(.lit)
    *(.shdata)
  }  > ram
  .shbss SIZEOF(.text) + ADDR(.text) :  {
    *(.shbss)
  }
  .talias :   { }  > ram
  .data  : {
    *(.data)
```

```
    CONSTRUCTORS
    _edata  =  .;
  } > ram

  .bss SIZEOF(.data) + ADDR(.data) :
  {
   __bss_start = ALIGN(0x8);
   *(.bss)
   *(COMMON)
      end = ALIGN(0x8);
      _end = ALIGN(0x8);
      __end = ALIGN(0x8);
  }
  .mstack  : { }  > ram
  .rstack  : { }  > ram
  .stab  . (NOLOAD) :
  {
    [ .stab ]
  }
  .stabstr  . (NOLOAD) :
  {
    [ .stabstr ]
  }
}
```

A.2 crt0.S - The startup file

```
/*
 * crt0.S -- startup file for m68k-coff
 *
 */

.title "crt0.S for m68k-coff"

/* These are predefined by new versions of GNU cpp.  */

#ifndef __USER_LABEL_PREFIX__
#define __USER_LABEL_PREFIX__ _
#endif

#ifndef __REGISTER_PREFIX__
#define __REGISTER_PREFIX__
#endif

/* ANSI concatenation macros.  */

#define CONCAT1(a, b) CONCAT2(a, b)
#define CONCAT2(a, b) a ## b
```

```
/* Use the right prefix for global labels.  */

#define SYM(x) CONCAT1 (__USER_LABEL_PREFIX__, x)

/* Use the right prefix for registers.  */

#define REG(x) CONCAT1 (__REGISTER_PREFIX__, x)

#define d0 REG (d0)
#define d1 REG (d1)
#define d2 REG (d2)
#define d3 REG (d3)
#define d4 REG (d4)
#define d5 REG (d5)
#define d6 REG (d6)
#define d7 REG (d7)
#define a0 REG (a0)
#define a1 REG (a1)
#define a2 REG (a2)
#define a3 REG (a3)
#define a4 REG (a4)
#define a5 REG (a5)
#define a6 REG (a6)
#define fp REG (fp)
#define sp REG (sp)

/*
 * Set up some room for a stack. We just grab a chunk of memory.
 */
.set stack_size, 0x2000
.comm SYM (stack), stack_size

/*
 * Define an empty environment.
 */
        .data
        .align 2
SYM (environ):
        .long 0

  .align 2
.text
.global SYM (stack)

.global SYM (main)
.global SYM (exit)
/*
 * This really should be __bss_start, not SYM (__bss_start).
```

```
 */
.global __bss_start

/*
 * start -- set things up so the application will run.
 */
SYM (start):
link a6, #-8
moveal #SYM (stack) + stack_size, sp

/*
 * zerobss -- zero out the bss section
 */
moveal #__bss_start, a0
moveal #SYM (end), a1
1:
movel #0, (a0)
leal 4(a0), a0
cmpal a0, a1
bne 1b

/*
 * Call the main routine from the application to get it going.
 * main (argc, argv, environ)
 * We pass argv as a pointer to NULL.
 */
        pea     0
        pea     SYM (environ)
        pea     sp@(4)
        pea     0
jsr SYM (main)
movel d0, sp@-

/*
 * _exit -- Exit from the application. Normally we cause a user trap
 *          to return to the ROM monitor for another run.
 */
SYM (exit):
trap #0
```

A.3 C based "glue" code.

```
/*
 * glue.c -- all the code to make GCC and the libraries run on
 *           a bare target board. These should work with any
 *           target if inbyte() and outbyte() exist.
 */
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#ifndef NULL
#define NULL 0
#endif

/* FIXME: this is a hack till libc builds */
__main()
{
  return;
}

#undef errno
int errno;

extern caddr_t _end;                /* _end is set in the linker command file */
extern int outbyte();
extern unsigned char inbyte();
extern int havebyte();

/* just in case, most boards have at least some memory */
#ifndef RAMSIZE
#  define RAMSIZE             (caddr_t)0x100000
#endif

/*
 * read  -- read bytes from the serial port. Ignore fd, since
 *          we only have stdin.
 */
int
read(fd, buf, nbytes)
     int fd;
     char *buf;
     int nbytes;
{
  int i = 0;

  for (i = 0; i < nbytes; i++) {
    *(buf + i) = inbyte();
    if ((*(buf + i) == '\n') || (*(buf + i) == '\r')) {
      (*(buf + i)) = 0;
      break;
    }
  }
  return (i);
}
```

```
/*
 * write -- write bytes to the serial port. Ignore fd, since
 *          stdout and stderr are the same. Since we have no filesystem,
 *          open will only return an error.
 */
int
write(fd, buf, nbytes)
     int fd;
     char *buf;
     int nbytes;
{
  int i;

  for (i = 0; i < nbytes; i++) {
    if (*(buf + i) == '\n') {
      outbyte ('\r');
    }
    outbyte (*(buf + i));
  }
  return (nbytes);
}

/*
 * open -- open a file descriptor. We don't have a filesystem, so
 *         we return an error.
 */
int
open(buf, flags, mode)
     char *buf;
     int flags;
     int mode;
{
  errno = EIO;
  return (-1);
}

/*
 * close -- close a file descriptor. We don't need
 *          to do anything, but pretend we did.
 */
int
close(fd)
     int fd;
{
  return (0);
}

/*
```

```
 * sbrk -- changes heap size size. Get nbytes more
 *         RAM. We just increment a pointer in what's
 *         left of memory on the board.
 */
caddr_t
sbrk(nbytes)
     int nbytes;
{
  static caddr_t heap_ptr = NULL;
  caddr_t        base;

  if (heap_ptr == NULL) {
    heap_ptr = (caddr_t)&_end;
  }

  if ((RAMSIZE - heap_ptr) >= 0) {
    base = heap_ptr;
    heap_ptr += nbytes;
    return (base);
  } else {
    errno = ENOMEM;
    return ((caddr_t)-1);
  }
}

/*
 * isatty -- returns 1 if connected to a terminal device,
 *           returns 0 if not. Since we're hooked up to a
 *           serial port, we'll say yes and return a 1.
 */
int
isatty(fd)
     int fd;
{
  return (1);
}

/*
 * lseek -- move read/write pointer. Since a serial port
 *          is non-seekable, we return an error.
 */
off_t
lseek(fd,  offset, whence)
     int fd;
     off_t offset;
     int whence;
{
  errno = ESPIPE;
```

```
  return ((off_t)-1);
}

/*
 * fstat -- get status of a file. Since we have no file
 *          system, we just return an error.
 */
int
fstat(fd, buf)
     int fd;
     struct stat *buf;
{
  errno = EIO;
  return (-1);
}

/*
 * getpid -- only one process, so just return 1.
 */
#define __MYPID 1
int
getpid()
{
  return __MYPID;
}

/*
 * kill -- go out via exit...
 */
int
kill(pid, sig)
     int pid;
     int sig;
{
  if(pid == __MYPID)
    _exit(sig);
  return 0;
}

/*
 * print -- do a raw print of a string
 */
int
print(ptr)
char *ptr;
{
  while (*ptr) {
    outbyte (*ptr++);
```

```
  }
}

/*
 * putnum -- print a 32 bit number in hex
 */
int
putnum (num)
unsigned int num;
{
  char  buffer[9];
  int   count;
  char  *bufptr = buffer;
  int   digit;

  for (count = 7 ; count >= 0 ; count--) {
    digit = (num >> (count * 4)) & 0xf;

    if (digit <= 9)
      *bufptr++ = (char) ('0' + digit);
    else
      *bufptr++ = (char) ('a' - 10 + digit);
  }

  *bufptr = (char) 0;
  print (buffer);
  return;
}
```

A.4 I/O assembler code sample

```
/*
 * mvme.S -- board support for m68k
 */

.title "mvme.S for m68k-coff"

/* These are predefined by new versions of GNU cpp.  */

#ifndef __USER_LABEL_PREFIX__
#define __USER_LABEL_PREFIX__ _
#endif

#ifndef __REGISTER_PREFIX__
#define __REGISTER_PREFIX__
#endif

/* ANSI concatenation macros.  */
```

```
#define CONCAT1(a, b) CONCAT2(a, b)
#define CONCAT2(a, b) a ## b

/* Use the right prefix for global labels.  */

#define SYM(x) CONCAT1 (__USER_LABEL_PREFIX__, x)

/* Use the right prefix for registers.  */

#define REG(x) CONCAT1 (__REGISTER_PREFIX__, x)

#define d0 REG (d0)
#define d1 REG (d1)
#define d2 REG (d2)
#define d3 REG (d3)
#define d4 REG (d4)
#define d5 REG (d5)
#define d6 REG (d6)
#define d7 REG (d7)
#define a0 REG (a0)
#define a1 REG (a1)
#define a2 REG (a2)
#define a3 REG (a3)
#define a4 REG (a4)
#define a5 REG (a5)
#define a6 REG (a6)
#define fp REG (fp)
#define sp REG (sp)
#define vbr REG (vbr)

.align 2
.text
.global SYM (_exit)
.global SYM (outln)
.global SYM (outbyte)
.global SYM (putDebugChar)
.global SYM (inbyte)
.global SYM (getDebugChar)
.global SYM (havebyte)
.global SYM (exceptionHandler)

.set vbr_size, 0x400
.comm SYM (vbr_table), vbr_size

/*
 * inbyte -- get a byte from the serial port
 * d0 - contains the byte read in
 */
```

```
.align 2
SYM (getDebugChar):  /* symbol name used by m68k-stub */
SYM (inbyte):
link a6, #-8
trap  #15
.word inchr
moveb  sp@, d0
extbl d0
unlk a6
rts

/*
 * outbyte -- sends a byte out the serial port
 * d0 - contains the byte to be sent
 */
.align 2
SYM (putDebugChar):  /* symbol name used by m68k-stub */
SYM (outbyte):
link fp, #-4
  moveb fp@(11), sp@
trap  #15
.word outchr
unlk fp
rts

/*
 * outln -- sends a string of bytes out the serial port with a CR/LF
 * a0 - contains the address of the string's first byte
 * a1 - contains the address of the string's last byte
 */
.align 2
SYM (outln):
link a6, #-8
moveml a0/a1, sp@
trap  #15
.word  outln
unlk a6
rts

/*
 * outstr -- sends a string of bytes out the serial port without a CR/LF
 * a0 - contains the address of the string's first byte
 * a1 - contains the address of the string's last byte
 */
.align 2
SYM (outstr):
link a6, #-8
moveml a0/a1, sp@
```

```
trap  #15
.word  outstr
unlk a6
rts

/*
 * havebyte -- checks to see if there is a byte in the serial port,
 *      returns 1 if there is a byte, 0 otherwise.
 */
SYM (havebyte):
trap  #15
.word instat
beqs empty
movel  #1, d0
rts
empty:
movel #0, d0
rts

/*
 * These constants are for the MVME-135 board's boot monitor. They
 * are used with a TRAP #15 call to access the monitor's I/O routines.
 * they must be in the word following the trap call.
 */
.set inchr, 0x0
.set instat, 0x1
.set inln, 0x2
.set readstr, 0x3
.set readln, 0x4
.set chkbrk, 0x5

.set outchr, 0x20
.set outstr, 0x21
.set outln, 0x22
.set write, 0x23
.set writeln, 0x24
.set writdln, 0x25
.set pcrlf, 0x26
.set eraseln, 0x27
.set writd, 0x28
.set sndbrk, 0x29

.set tm_ini, 0x40
.set dt_ini, 0x42
.set tm_disp, 0x43
.set tm_rd, 0x44

.set redir, 0x60
```

```
.set redir_i, 0x61
.set redir_o, 0x62
.set return, 0x63
.set bindec, 0x64

.set changev, 0x67
.set strcmp, 0x68
.set mulu32, 0x69
.set divu32, 0x6A
.set chk_sum, 0x6B
```

A.5 I/O code sample

```
#include "w89k.h"

/*
 * outbyte -- shove a byte out the serial port. We wait till the byte
 */
int
outbyte(byte)
     unsigned char byte;
{
  while ((inp(RS232REG) & TRANSMIT) == 0x0) {  } ;
  return (outp(RS232PORT, byte));
}

/*
 * inbyte -- get a byte from the serial port
 */
unsigned char
inbyte()
{
  while ((inp(RS232REG) & RECEIVE) == 0x0) { };
  return (inp(RS232PORT));
}
```

A.6 Led control sample

```
/*
 * leds.h -- control the led's on a Motorola mc68ec0x0 board.
 */

#ifndef __LEDS_H__
#define __LEDS_H__

#define LED_ADDR 0xd00003
#define LED_0           ~0x1
#define LED_1           ~0x2
```

```
#define LED_2           ~0x4
#define LED_3           ~0x8
#define LED_4           ~0x10
#define LED_5           ~0x20
#define LED_6           ~0x40
#define LED_7           ~0x80
#define LEDS_OFF 0xff
#define LEDS_ON 0x0

#define FUDGE(x) ((x >= 0xa && x <= 0xf) ? (x + 'a') & 0x7f : (x + '0') & 0x7f)

extern void led_putnum( char );

#endif /* __LEDS_H__ */

/*
 * leds.c -- control the led's on a Motorola mc68ec0x0 (IDP)board.
 */
#include "leds.h"

void zylons();
void led_putnum();

/*
 * led_putnum -- print a hex number on the LED. the value of num must be a char with
 *               the ascii value. ie... number 0 is '0', a is 'a', ' ' (null) clears
 * the led display.
 * Setting the bit to 0 turns it on, 1 turns it off.
 *  the LED's are controlled by setting the right bit mask in the base
 *  address.
 * The bits are:
 * [d.p | g | f | e | d | c | b | a ] is the byte.
 *
 * The locations are:
 *
 *  a
 *        -----
 *     f |     | b
 *       |  g  |
 *        -----
 *                    |     |
 *                 e  |     | c
 *                    -----
 *                      d                . d.p (decimal point)
 */
void
led_putnum ( num )
char num;
```

```
{
    static unsigned char *leds = (unsigned char *)LED_ADDR;
    static unsigned char num_bits [18] = {
      0xff, /* clear all */
      0xc0, 0xf9, 0xa4, 0xb0, 0x99, 0x92, 0x82, 0xf8, 0x80, 0x98, /* numbers 0-9 */
      0x98, 0x20, 0x3, 0x27, 0x21, 0x4, 0xe  /* letters a-f */
    };

    if (num >= '0' && num <= '9')
      num = (num - '0') + 1;

    if (num >= 'a' && num <= 'f')
      num = (num - 'a') + 12;

    if (num == ' ')
      num = 0;

    *leds = num_bits[num];
}

/*
 * zylons -- draw a rotating pattern. NOTE: this function never returns.
 */
void
zylons()
{
  unsigned char *leds  = (unsigned char *)LED_ADDR;
  unsigned char curled = 0xfe;

  while (1)
    {
      *leds = curled;
      curled = (curled >> 1) | (curled << 7);
      delay ( 200 );
    }
}
```

Table of Contents