Using Sourcery G++ to Develop and Debug a Linux Kernel Module

Sourcery G++ Application Note AN003

March 7, 2011

Abstract

This application note is a continuation of Sourcery G++ Application Note AN002, *Using Sourcery G++ to Debug the Linux Kernel*, that guides you through the Linux kernel module development process. It includes a short introduction to kernel modules and how they work, and shows how to use Sourcery G++ to build and debug a module.

Particular attention is paid to using a SEGGER J-Link to debug an example kernel module on a PHYTEC phyCORE-LPC3250 development board.

1. Introduction

A very useful part of the Linux kernel architecture is the support for loadable kernel modules. These modules allow the otherwise monolithic kernel to be split up into smaller components that can later be loaded as required, allowing the kernel to ship with support for a wide range features but only load those that are needed.

Kernel modules also ease the development of new features such as file systems or device drivers, as a new experimental modules can be quickly built, loaded into a basic kernel, exercised, and then unloaded. This is much faster then the build, flash, restart process that would otherwise be required.

This application note is a continuation of Sourcery G++ Application Note AN002, *Using Sourcery G++ to Debug the Linux Kernel*, that focuses on the procedures for building and debugging kernel modules.

Section 2, “Requirements” covers the required background knowledge and equipment needed to complete the note.

Section 3, “Anatomy of a Kernel Module” introduces the Linux kernel, modules system, and the life cycle of a module.

Section 4, “Developing Kernel Modules” introduces the example module used in the rest of the note, the build process, and using Sourcery G++ to build and navigate around.

Section 5, “Module Debugging” discusses the considerations unique to kernel modules, loading the module, the actual debugging, and automating the whole process.

2. Requirements

The procedures for building and debugging a kernel module in the subsequent sections of this note assume that you have already set up the PHYTEC phyCORE-LPC3250 development board, built and installed a debuggable Linux kernel, and set up the SEGGER J-Link device with the Sourcery G++ Debug Sprite, as documented in Sourcery G++ Application Note AN002.

3. Anatomy of a Kernel Module

This section covers the basics of the kernel, modules, and the module life cycle from a developer's perspective. There's a large body of good information on the Internet and in print about the Linux kernel and kernel modules such as those listed in Section 7, “Further Readings”.

Linux is a monolithic kernel, where all of the code and data that makes up the image are linked into one binary and loaded into memory. The kernel module system gives a way of splitting out the support for optional features so that the running kernel contains only what's needed for a particular configuration. Modules can be loaded on demand by the kernel itself, such as when a new device is plugged in or an unrecognized file system is mounted, or they can be loaded manually using command-line utilities. Developing new kernel functionality as a kernel module is convenient, as modules can be loaded into a running kernel and exercised without having to re-flash or restart.

Internally a module is a standard ELF executable file with a `.ko` extension and a few special sections such as `.modinfo` for the module metadata and `.init.text` for the module initialization code. A nice thing about modules being ELF files is that they can be generated and inspected by standard tools.

Modules are loaded and unloaded using the user-space `insmod` and `rmmod` commands. `modprobe` is a higher-level tool that also handles dependencies between modules. `insmod` is quite small as the actual work is performed by the kernel in `kernel/module.c`. This handles the loading from user space, dependencies, relocation, and initialization.

From a debugging point of view, the most interesting step is relocation as it affects the addresses of any breakpoints or global variables. Relocation is the process of fixing up any internal and undefined symbols to point to their final locations, meaning that a symbol that starts at certain address in the module file will have a different address when actually loaded into RAM. Later on in this note we will use relocation information supplied by the kernel to inform the debugger of this change.

Understanding the kernel memory layout can be helpful when debugging. In the current 2.6 series, the ARM kernel is laid out as follows:

Start	**End**	**Contents**
0xFF000000	0xFFFFFFFF	Vector page, DMA region, and others
`VMALLOC_END`	0xFEFFFFFF	free
`VMALLOC_START`	`VMALLOC_END`	`vmalloc()` / `ioremap()` space
`PAGE_OFFSET` 0xC0000000	`high_memory`	The Linux kernel
`TASK_SIZE` 0xBF000000	`PAGE_OFFSET-1` 0xBFFFFFFF	Kernel module space (16 MB)
0x00001000	`TASK_SIZE-1` 0xBEFFFFFF	User space (~3 GB)
0x00000000	0x00001000	Vector page / Null pointer trap

Note that these are virtual addresses, which are different than the physical address space of the board. Kernel modules are allocated into the Kernel module space, meaning that the relocated functions, static, and global variables will have addresses similar to `0xBF000000`. See the documentation on the ARM Linux Project website[1] for more information.

4. Developing Kernel Modules

This section introduces the `hello` kernel module that we will use for the rest of this note.

4.1. Module and Makefile

```
/* An example kernel module that shows the life cycle and provides a
 * function to user space over debugfs.
 */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/debugfs.h>
#include <linux/seq_file.h>

MODULE_LICENSE("Dual BSD/GPL");

/* debugfs root directory.  Used while cleaning up in exit */
static struct dentry *root_dir;
```

[1] http://www.arm.linux.org.uk/developer/memory.txt

```
/* Number of calls to hello_print */
static int calls;

/* Prints some text out to the open debugfs file */
static int hello_print(struct seq_file *s, void *p)
{
        seq_printf(s, "Called %d times\n", ++calls);
        return 0;
}

/* Called when the debugfs file is opened.  Pass off to the
 * sequential file helpers
 */
static int hello_open(struct inode *inode, struct file *file)
{
 return single_open(file, hello_print, inode->i_private);
}

/* File operations.  Most pass through to the sequential file
 * system
 */
static const struct file_operations hello_fops = {
 .open   = hello_open,
 .write  = NULL,
 .read   = seq_read,
 .llseek = seq_lseek,
 .owner  = THIS_MODULE,
};

/* Initializes the module.  Create the debugfs directory and
 * file
 */
static int hello_init(void)
{
        printk(KERN_ALERT "Hello world\n");

        root_dir = debugfs_create_dir("hello", NULL);
        debugfs_create_file("ping",
                            0444, /* mode */
                            root_dir, /* parent */
                            NULL, /* callback data */
                            &hello_fops);

        return 0;
}

/* Cleans up the module.  Delete the debugfs directory and all
 * files under it
 */
static void hello_exit(void)
{
        debugfs_remove_recursive(root_dir);

        printk(KERN_ALERT "Goodbye cruel world\n");
```

```
}

/* Hooks that tell the kernel which functions should be used at
 * initialization and exit
 */
module_init(hello_init);
module_exit(hello_exit);
```

`hello.c` is a simple, “Hello world”-style module that we will use in the rest of this note. It creates a new file at `/sys/kernel/debug/hello/ping` that can be opened and read from user space. Reading this file gives a short message that includes the number of times the file has been opened.

A basic module like this is a good introduction to the debugging methods used on much larger modules. In particular, we can see:

- The life cycle, from the initialisation `hello_init` through to finalization in `hello_exit`.
- Exposing data to help debugging through `debugfs`.
- Causing an event in user space and debugging in kernel space.
- Common debugging such as inspecting variables and the call stack, all while in kernel space.

The Makefile passes most of the work off to the Linux kernel build system:

```
# Makefile that builds the 'hello' kernel module.
#
# Derived from Linux Device Drivers, Third Edition
#

# If KERNELRELEASE is defined, we've been invoked from the
# kernel build system and can use its language.
ifneq ($(KERNELRELEASE),)
obj-m := hello.o

# Otherwise we were called directly from the command
# line; invoke the kernel build system.
else
PWD := $(shell pwd)
# Path to the full kernel tree
KERNELDIR ?= $(PWD)/../linux

# User and IP address used to connect to the board
USER = root@192.168.1.40
# SSH port number to connect to
PORT = 22

# Default rule that calls the kernel build system and builds this
# module
all: build

# Call the kernel build system
build:
 $(MAKE) -C $(KERNELDIR) M=$(PWD) O=build modules
```

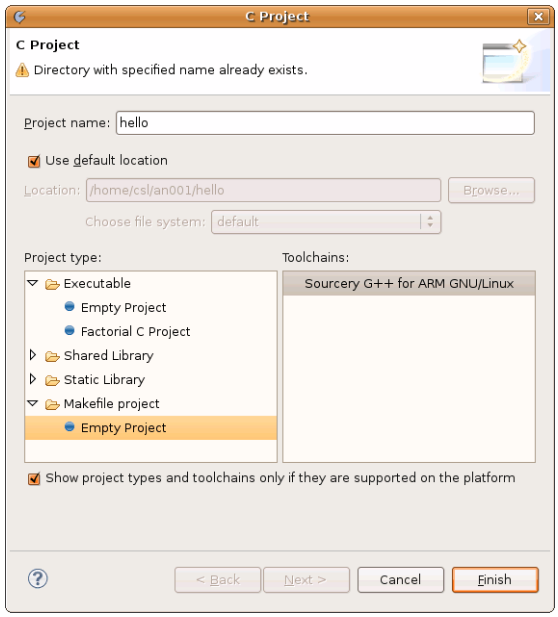
```
hello.ko: build

# Helper that removes, copies, loads, and then prints the base
# address of the module
load: hello.ko
 -ssh -p $(PORT) $(USER) rmmod $<
 scp -qP $(PORT) $< $(USER):~
 ssh -p $(PORT) $(USER) insmod $<
 ssh -p $(PORT) $(USER) \
  cat /sys/module/$(basename $<)/sections/.text

clean:
 rm -f *.ko *.o *.mod.c *.elf modules.order *.symvers *~

endif
```

This is an example of a “out-of-tree” module where the files live outside the main Linux kernel tree. Modules can be built directly from the command line, but it's easier to wrap the rules and variables up in a Makefile. Here we've used the standard idiom where running the Makefile with no arguments calls the real kernel build system with the correct arguments instead. This simplifies the build and integrates better with Sourcery G++.

Near the end of the file are rules that automate the loading of the module onto the board. These will be used later in the note.

4.2. Creating the Module Project

- Open up the workspace that you created in AN002.
- Switch to a terminal. Take a copy of `hello.c` and the `Makefile` and save them in the directory `~/an002/hello`. Your directory `~/an002` should now contain the kernel in `linux` and the module in `hello`.
- Switch back to the Sourcery G++ IDE window. The Project Explorer view should contain `linux` kernel project in it and nothing else.
- Select `File` → `New` → `C Project...`.
- Enter `hello` for the Project name.
- Expand `Makefile project`.
- Select `Empty project`.
- Click `Finish`.

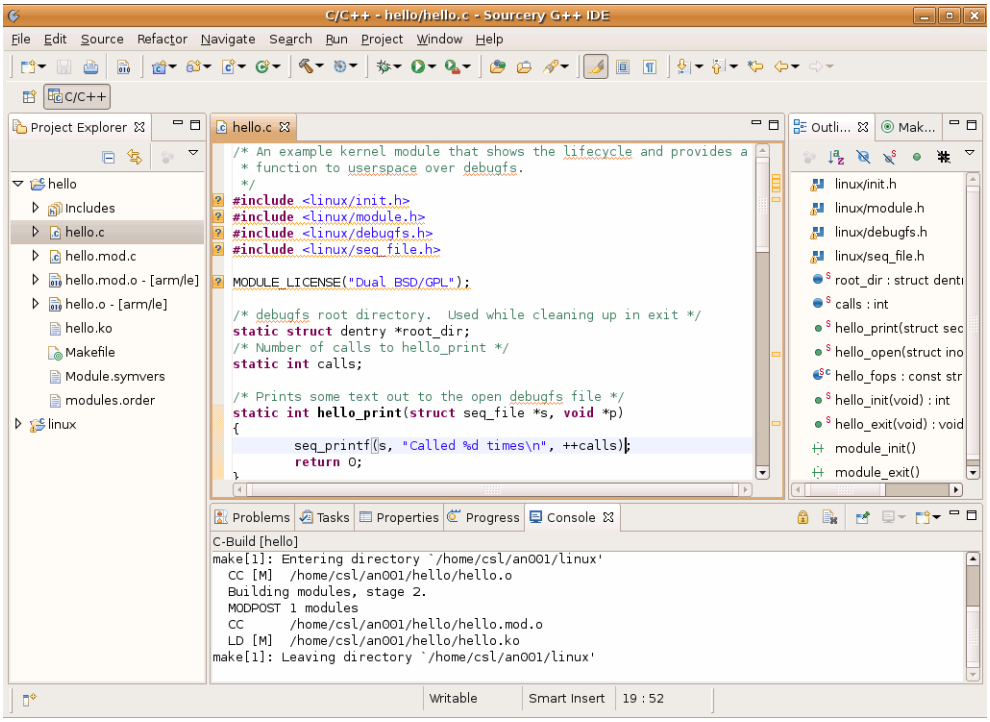


Creating a new project that builds the module

The newly copied files are automatically added to the project. Sourcery G++ will start building the module in the background.

4.3. Navigating Around

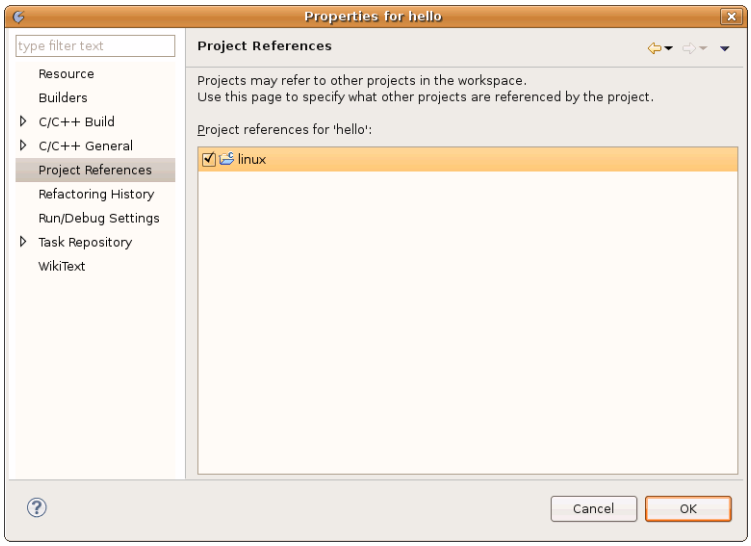
Click on the triangle next to the `hello` project in the Project Explorer view to expand the project and show the automatically found files. Note that you can double click on `hello.c` or the `Makefile` to open them up. Notice the question marks beside the `printk()` lines and others. At the moment the project doesn't have the appropriate include paths or defines set and is wrongly reporting an error.



The main window showing the Project Explorer and editor

Let's tell Sourcery G++ that the module and kernel are related:

- Right click on `hello`.
- Select `Properties`.
- Select `Project References`.
- Click on `linux` to check it.
- Click on `OK` to close the window.



Adding a project reference

Sourcery G++ now knows that the module uses the kernel, which enables the advanced features such as jumping to header files, jumping to definitions, and code completion. Let's try jumping to a header file:

- Make sure `hello.c` is open.
- Right click on the `#include <linux/debugfs.h>` line.
- Select `Open Declaration`.
- See a new editor containing `debugfs.h` pop up.

Next let's have a look at the implementation of `seq_printf()`.

- Switch back to the `hello.c` tab.
- Scroll down to line 19.
- Right click on `seq_printf()`.
- Select `Open Declaration`.
- See a new editor containing seq_file.c pop up showing the `seq_printf()` function.

Selecting `Open Declaration` or pressing **F3** again jumps to the definition in the corresponding header file. It's easy to get back to `hello.c` as well - try selecting `Navigate` → `Back` or pressing **Alt+Left Arrow** to work your way back to the top.

5. Module Debugging

Your environment, including the board, host, and Sourcery G++, are now set up for module development. In AN002 we exercised the debugger against the kernel. The next step is to load and debug the module itself.

5.1. Loading the Module

Modules are like user-space shared libraries in that they are linked at one address and then run at another. Every time you load a module, the kernel allocates memory from the kernel module space, copies the module in, and relocates it. Our concern is the dynamic aspect and the relocation: we need to tell the debugger that the module has been loaded and where each of its sections ended up.

Use the following steps to load a module:

- Copy `hello.ko` to the board using the RSE.
- Log into the board using either the Terminal or a RSE SSH shell.
- Change to the home root directory using `cd /root`.
- Load the module with `insmod hello.ko`.
- See `Hello world` over on the Terminal. Alternatively you can also see the most recent kernel messages using `dmesg | tail`.

We can see where a module ended up by looking under the `/sys/modules/hello` directory:

```
[root@nxp /root]# ls -a /sys/module/hello/sections/
.
..
.bss
.data
.gnu.linkonce.this_module
.note.gnu.build-id
.rodata
.rodata.str1.1
.strtab
.symtab
.text
[root@nxp /root]# cat /sys/module/hello/sections/.text
0xbf000000
[root@nxp /root]# cat /sys/module/hello/sections/.bss
0xbf000820
[root@nxp /root]# cat /sys/module/hello/sections/.data
0xbf0006c1
```

As this is the very first module loaded it has ended up right at the start of the kernel module region at `0xbf000000`. As it is the only module in this system, removing and re-adding it will also put it back at the same address. We will use this fact later to automate some of the process.

Back on the host, the Makefile has a target that automates the remove old, copy, load, get sections process. Run `make load` and you'll see the following:

```
michaelh@crucis:~/projects/cs/lpc32xx/run/hello$ make load
make -C ~/an002/hello/../linux M=~/an002/hello modules
make[1]: Entering directory `~/an002/linux'
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: Leaving directory `~/an002/linux'
ssh -p 22 root@192.168.1.40 rmmod hello.ko
```

```
scp -qP 22 hello.ko root@192.168.1.40:~
ssh -p 22 root@192.168.1.40 insmod hello.ko
ssh -p 22 root@192.168.1.40 cat /sys/module/hello/sections/.text
0xbf000000
```

Notice the `.text` load address right at the end.

> **Saving the login password**
>
> The Sourcery G++ RSE automatically saves the board login password for you. To get a similar effect from the command line, set up a password-less SSH key and add it to your SSH agent.

5.2. Debugging

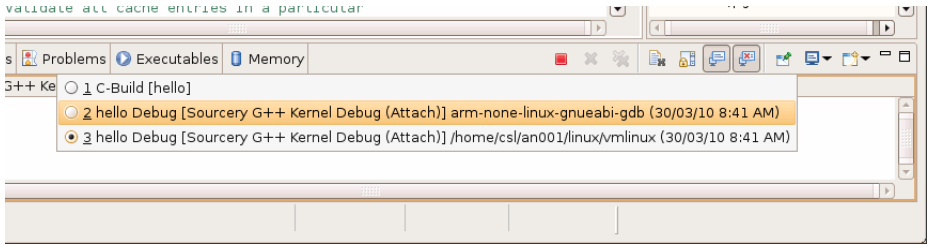
Debugging a kernel module is very similar to the kernel debugging done earlier. The extra step is to tell the debugger that the module has been loaded and where it has been loaded to. We'll do this manually the first time and then automate it.

Do the following:

- Switch to the IDE.
- Click on the `Debug` button to start debugging.
- See the Debug Perspective open.
- If the board is not already paused, click the `Suspend` button.

Sourcery G++ uses the GNU Debugger in the background to handle many of the debugging operations. While the board is suspended we can send commands directly to GDB to tell it about the module.

- Click on the down arrow next to the `Display Selected Console` button.
- Select the console with `arm-none-linux-gnueabi-gdb` in the name.
- See an empty console with a flashing cursor.



Selecting the GDB console

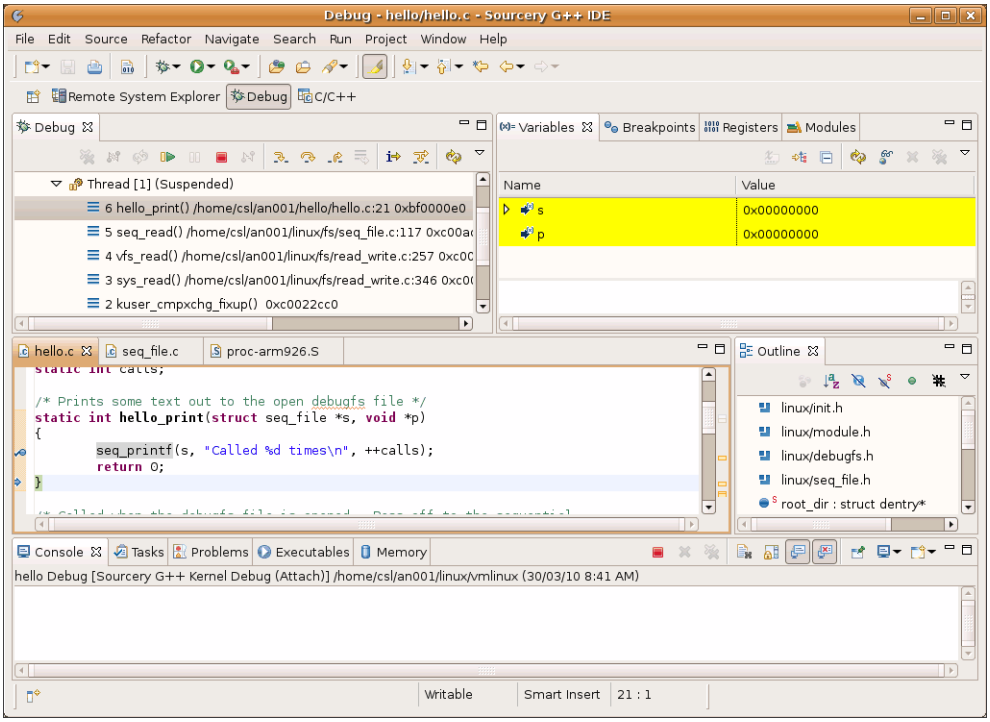
If you've used GDB before and are used to its command-line system then this may seem a bit unusual. GDB is running in the special machine interface mode which is much quieter than you may be used to.

- Tell the debugger about the module using `add-symbol-file hello.ko 0xbf000000`.
- See

```
add-symbol-file hello.ko 0xbf000000
add symbol table from file "hello.ko" at
 .text_addr = 0xbf000000
```

The debugger now knows about the module. Let's do some debugging:

- Switch to the `hello.c` editor tab.
- Scroll down to line 19.
- Select `Run` → `Toggle Breakpoint`.
- See a dot and a tick appear in the margin. This shows that you have successfully set a breakpoint on this line.
- Click `Resume` to start the board running again.
- Log into the board using the Terminal or a RSE SSH shell.
- Change to the modules `debugfs` directory using `cd /sys/kernel/debug/hello`.
- Print out the contents of the `ping` file using `cat ping`.
- Note that nothing happens and the board halts. The run LED stops flashing.
- Switch back to the IDE.
- In the Debug view see that we've suspended due to a breakpoint hit.
- In the `hello.c` editor, see that the line we've stopped at is highlighted.
- Try pressing **F6** to step over the call to `seq_printf()`.
- See that the highlight moves on to the end of function. Note that the compiler has optimized out the `return 0`.
- Click on `Resume` to hand control back to the board.
- Switch back to the console.
- See `Called 1 times` and the command prompt on the screen.

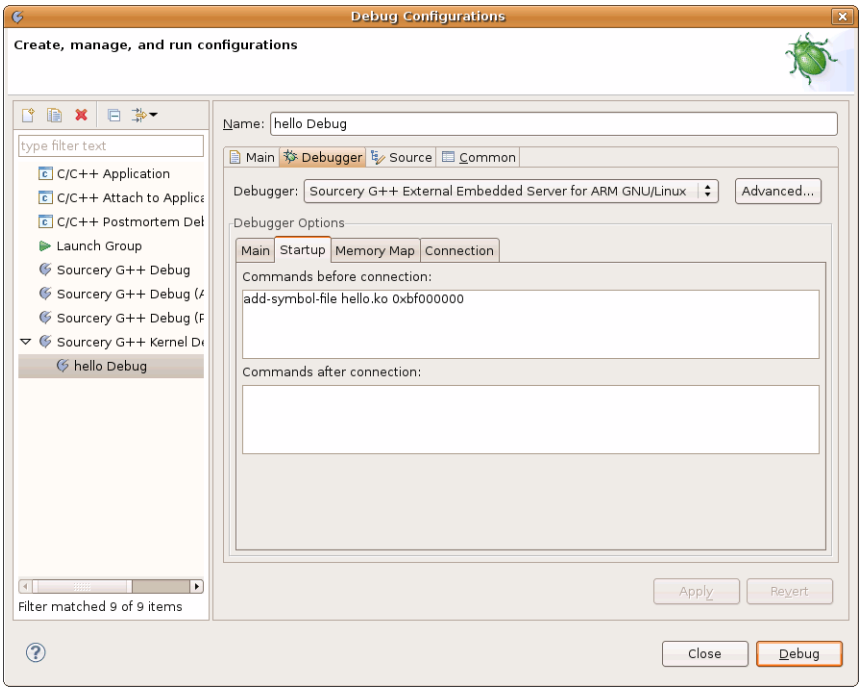


Debugging a kernel module

5.3. Automating

We can automate the GDB `add-symbol-file` command by assuming that the module is always loaded to the same address and telling the debugger about it at the start of debug. To do this:

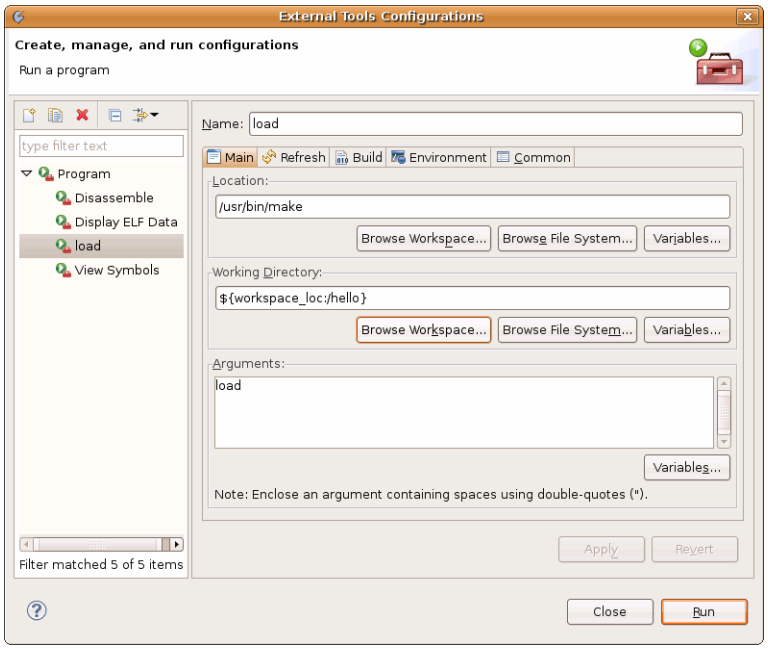
- Switch to the IDE.
- Select `Run` → `Debug Configurations...`.
- Click on `hello Debug` in the tree.
- Click on the `Debugger` tab.
- Click on the `Startup` subtab.
- Click on the `Commands before connection` box.
- Type in `add-symbol-file hello.ko 0xbf000000`.
- Click `Apply` to save these settings.
- Click `Close` to close the window.



Adding a before command to load the module symbols

Unfortunately we can't also load the module when starting a debug session as we don't know what state the board is in - it might be reset, halted, booting, or at the command prompt. We can however add a one-button shortcut by setting up the load as an external tool. To do this:

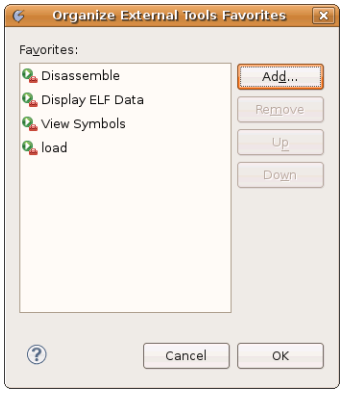
- Select `Run` → `External Tools` → `External Tools Configurations...`.
- See the `External Tools Configurations` window pop up.
- Click on the `New launch configuration` button.
- Enter `load` for the `Name`.
- Enter `/usr/bin/make` for the location. The unqualified name `make` will not work.
- Under `Working Directory`, click `Browse Workspace...`.
- Select `hello`.
- Click `OK`.
- Enter `load` for the arguments.
- Click on the `Build` tab.
- De-select `Build before launch`.
- Click `Apply` to save the new tool.
- Click `Close`.



Adding an external tool

We will now add this new tool as a favorite and make it run when you click on the External Tools button.

- Select `Run` → `External Tools` → `Organize Favorites...`.
- Click `Add...`.
- Select `load`, the new tool we just created.
- Click `OK`.
- Click `OK`.

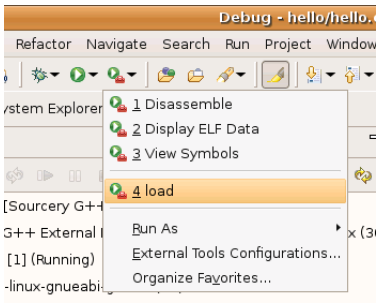


After organizing the external tools

Run the tool for the very first time:

- Click on the down arrow beside the External Tools button.

- Click `load`.



Running the new external tool to load the module in

Sourcery G++ will remember this selection and automatically use it next time you click on the main External Tools button.

5.4. Breakpoint Usage

Kernel modules are much closer to the hardware than the user-space applications you may be used to. Debugging a kernel module involves the on-chip debug hardware and puts limitations on the type and number of breakpoints you can have.

Software breakpoints work by modifying memory and are great due to being unlimited in number and automatically mapped with the virtual memory they apply to. They are only available if the memory is mapped in and writable. *Hardware breakpoints* use the on-chip debug support and can always be set but are limited in number and fixed to a processor address.

As you can see in the memory map above, the main kernel is at a fixed address and never gets mapped out. When the debugger has control this memory is also writable, meaning that software breakpoints can be used in the main kernel. Kernel modules do end up at a fixed address but are paged in on demand, meaning that hardware breakpoints must be used.

The LPC3250 uses the EmbeddedICE-RT block seen on most ARM9s and ARM7s. This block provides two watchpoints that can be used for either breakpoints or data watchpoints. The SEGGER J-Link will automatically pick between hardware and software breakpoints based on the address and available resources but be aware that you may have to manage your breakpoints. Note that single stepping sometimes requires a breakpoint on the next line to work. If you run out of hardware breakpoints then the system may fall back to a different type of breakpoint or run in a slow instruction-by-instruction mode.

6. Conclusion

This note has shown how to structure, build, and debug a simple out-of-tree kernel module using the Sourcery G++ IDE. You can apply the same techniques shown here to developing more complex target-specific modules such as device drivers.

7. Further Readings

1. Corbet, Rubini, Kroah-Hartman (2005) “Linux Device Drivers, Third Edition” O’Reilly Media, Inc.

Available in print or online[2]. Chapter 2 “Building and Running Modules” and chapter 4 “Debugging Techniques” are particularly useful.

2. Corbet, J. (2009) “An updated guide to debugfs”. Available at:

   - http://lwn.net/Articles/334546/

   A quick introduction to the Debug filesystem used in this example.

3. Henderson, B. (2006) “Linux Loadable Kernel Module HOWTO”. Available at:

   - http://tldp.org/HOWTO/Module-HOWTO/

4. Jones, T. (2008) “Anatomy of Linux loadable kernel modules”. Available at:

   - http://www.ibm.com/developerworks/linux/library/l-lkm/

   A good, technical overview of kernel modules and their life cycle.

[2] http://lwn.net/Kernel/LDD3/