UCSD Embedded Linux online
Week 4

Slide 1

This week we're going to get down to the nitty gritty. We'll build a couple of applications using the ARM cross tool chain and then debug those applications on the target using Eclipse. We'll also look at Posix threads as a standard approach to true multi-tasking. It turns out that just because of the way the class material breaks down, this will be a fairly intense week.

Slide 2

Let's review our environment. We have an ARM9-based target board connected to a Linux workstation via both RS232 and Ethernet. The target boots Linux from its NAND flash memory and also mounts a root file system from NAND flash. The /home directory is mounted from the workstation over NFS. We communicate with the Linux shell running on the target over the serial port using the minicom terminal emulator.

On the workstation we have a cross tool chain that allows us to build executables for the ARM9

Slide 3

Like many contemporary architectures, the ARM places I/O registers in the same space as memory so that registers are accessible with any instruction that can access memory. But before you can access those registers from Linux, you must "map" them into user process space by opening a connection to the mem device and using the mmap() function.

Slide 4

So let's move on to something a little more challenging. The target board incorporates a multi-channel analog to digital converter with a pot attached to channel 0. We'll use that as the basis of a simple data acquisition example that will serve throughout much of the remainder of this class.

The measure program simply reads ADC channel 0 at a specified interval in seconds and prints the value on stdout. If no interval is entered, the default is two seconds. There's also code that reads the pushbuttons and transfers them to the LEDs.

Rather than build access to the I/O ports directly into the application code, as we did with the LED program two weeks ago, measure utilizes a simple "device driver" interface and places the driver functions in a separate file. The idea here is to provide a set of higher level, application-oriented APIs that abstract out the details of dealing with real hardware. This is just good practice anyway.

Another motivation for this division will become apparent later when we look at using our host workstation for debugging.

Slide 5

We need to create a new Eclipse project for measure, but in this case, and in most subsequent examples, the makefile already exists. So we'll create what Eclipse calls a Makefile project. I must emphasize this because it's a step that many students overlook. You must create an empty Makefile project

Also, we can't use the default workspace. The executable must be visible in the target's file system and so we have to put it somewhere within the target_fs/ directory. The location specified here is in fact where the project files are.

Slide 6

Open measure.c. After setting up a simple signal handler to terminate the program gracefully, it calls a couple of functions to initialize the I/O system. Then it enters a loop where it sleeps for awhile, reads the ADC, and prints the current value. It reads the pushbuttons and sends a pattern to the LEDs for each pushbutton that's pressed.

When the program terminates, by typing Ctrl-C, a couple of functions are called to do whatever is necessary to shut down the I/O system cleanly.

Now open trgdrive.c. Let's start with the initDigIO() function at line 54 Oh, and you'll probably want to show line numbers if you don't already have them turned on. initDigIO() should be the first function called by main because it opens a file descriptor to /dev/mem that will also be needed by initAD(). Then it sets up the LED bits for output and the switch bits for input.

closeDigIO(), just below initDigIO(), simply unmaps the GPIO peripheral space.

The three digital output functions at lines 87 through 114 work specifically on port B and more specifically, the LED bits. Digital input at line 116 is fairly trivial. These four functions could reasonably be inlined.

Go back up to initAD() at line 20. It maps the ADC register space, closes the /dev/mem file descriptor and does some initialization.

Again, closeAD() at line 46 simply unmaps ADC peripheral space. readAD() at line 37 sets the channel number, starts the conversion and reads the value when conversion is finished.

Finally, have a look at the makefile and note that it defines multiple targets. Note also that for a couple of these targets, the compiler is named arm-linux-gcc. There's a 3-part convention for naming cross tool chains. The first part represents the architecture, in this case ARM. The second part is the operating system and the third part is the tool itself. If you look back in the cross tool chain at /usr/local/arm/4.3.2/bin, you'll find a whole set of entries with the arm-linux prefix. These are in fact links to another set of tools that uses a more extensive naming convention.

Slide 7

There's a minor problem here. At the top of measure.c where the header files are included, each of the #include directives has a question mark symbol in the marker bar. If you hover over that symbol, you get the message "Unresolved inclusion". This means that Eclipse couldn't find the header files and thus can't itself resolve the symbols in those files. Also, you can't directly open header files from the Outline view. This anomaly only seems to affect Makefile projects.

This isn't really a "problem", it's more of an annoyance. Despite any errors about unresolved symbols that Eclipse might throw up, the project will in fact build.

This slide and the next one have instructions for fixing this issue.

Slide 8

This resolves the include problem for the local header file s3c2410-regs.h that defines peripheral registers. When you click Apply, you're asked if you would like to rebuild the index. Yes, you would.

We've solved the problem for the measure project, but every subsequent project we create will have the same problem. But don't worry you won't have to type that whole path again. There's a slicker way to do it once we've entered the information.

Slide 9

Remember that the only target Eclipse knows how to make is "all." In the case of the measure project all builds something called a "simulation version." We'll talk about that in a little while. We have to tell Eclipse about the other targets. Follow the instructions in this slide to create a new target called measure. Then build it. Note that measure is built with the ARM cross compiler.

Slide 10

If you haven't done so already, start up minicom in a shell window on your workstation and then boot the target board. Run the measure program as shown in this slide. The "1" argument means it will report temperature every second instead of the default 2 seconds. Turn the pot with a small screwdriver and you should see the value change.

Slide 11

OK, it's easy enough to run programs on the target. How do we use Eclipse to debug them? In a typical desktop environment, the target program runs on the same machine as the debugger. But in our embedded environment Eclipse with gdb runs on the host and the program being debugged runs on the ARM target. Fortunately, gdb implements a serial protocol that allows it to work with remote targets either over an RS-232 link or Ethernet.

You can think of GDB as having both a client and a server side. The client is the user interface and the server is the actual interaction with the program under test. As this slide illustrates, the server in this case implements the remote protocol and talks to a piece of code running on the target that interacts with the program under test.

Slide 12

There are actually two approaches to interfacing the target to the gdb serial protocol:

- gdb stubs is a set of functions linked to the target program. One consequence of this is that the program builds differently in a debug vs. a release mode. The other drawback is that gdb stubs is limited to RS-232.
- gdbserver, on the other hand, is a stand-alone program running on the target that, in turn, runs the program to be debugged. So gdbserver is totally independent of the target program. In other words, the target builds the same regardless of remote debugging. The other major advantage of gdbserver is that it runs over Ethernet.

Eclipse uses gdbserver. In the target file system gdbserver is located in the /usr/bin directory, which is part of the path.

The arguments to gdbserver are:

- A network port number preceded by a colon. The normal format is "host:port", where "host" is either a host name or an IP address in the form of a dotted quad, but gdbserver ignores the host portion. The port number is arbitrary as long as it doesn't conflict with another port number in use. Generally, port numbers below 1024 are reserved for established network protocols, so it's best to pick a number above 1023. Port number 10000 happens to be the default for the Eclipse debugger.
- Target program name.
- Arguments to the target program, if any.

gdbserver responds that it started a process for the target program and that it's listening on port 10000.

Slide 13

Now we need to set up remote debugging in Eclipse. When you bring up the Debug Configurations dialog, note first of all that even though we're debugging remotely, Eclipse treats our new configuration as a C/C++ Local Application. Feel free to discard the "Default" in the configuration name.

With a makefile project, the Search Project button doesn't work for specifying the application name in the Main tab. You need to click Browse and then go to target_fs/home/src/measure and select measure.

Down at the bottom of the Main tab, you'll probably see something that says "Using GDB (DSF)", something, something, followed by a link called "Select other". Click that link and select Standard Create Process Launcher. Click OK.

The rest of the setup for remote debugging occurs in the Debugger tab. In the Debugger drop-down, select gdbserver Debugger. In accordance with the 3-part naming convention we discussed earlier, the name of the debugger is arm-linux-gdb. In the Connection tab, select TCP from the Type drop-down and enter 192.168.1.50 as the IP address replacing the default name localhost. Be sure that Stop on startup at main is checked.

Click Apply, then Debug. Eclipse switches to the Debug perspective, looking essentially identical to what we saw when debugging record_sort on the workstation. gdbserver tells us that it's debugging from host 192.168.1.2. All of the debugging tools you used last week are available to debug programs on the target.

Slide 14

Often in my career as an embedded developer I have found it useful to be able to do initial debugging in a simulation environment on the host workstation, independent of the target. In the early days of embedded computing, debugging tools for target boards tended to be either somewhat primitive or very expensive whereas decent workstation tools have been around for a long time. Eclipse and related open source tools have substantially mitigated this particular problem.

Another motivation for testing on the workstation is that the target hardware may not be ready, or may not be completely functional, when you're ready to start testing application code. And the host has a file system that can be used to create test scripts and document test results.

Of course, in order to use this technique, you must have both target and host versions of your operating system. With Linux this is not a problem since we have the source code by definition and can build it any way we want.

Slide 15

This slide illustrates one reason that simulation can be a practical approach to testing application code. When you characterize the content of most embedded system software, you'll usually find that something like 5% of the code deals directly with the hardware. "Everything else" is independent of the hardware and therefore shouldn't need hardware to test it, provided that you can supply the appropriate stimulus to exercise it.

Unfortunately, all too often that 5% of hardware-dependent code is scattered randomly throughout the entire system. In this case you're virtually forced to use the target for testing because it is so difficult to "abstract out" the hardware dependencies.

That's why I made such a big point earlier about separating and isolating hardware dependent code from hardware independent application code.

Slide 16

If your hardware-dependent code is carefully isolated behind a well-defined set of APIs and confined to one or two source code modules, you can substitute a *simulation* of the hardware-dependent code that uses the host's keyboard and screen for I/O. In effect, the simulated driver "fools" the application into thinking the hardware is really there.

Now you can exercise the application by providing stimulus through the keyboard and noting the output on the screen. In later stages of testing you may want to substitute a file-based "script driver" for the screen and keyboard to create reproducible test cases.

Think of this as a "test scaffold" that allows you to exercise the application code in a well-behaved, controlled environment. Among other things, you can simulate limit conditions that might be very difficult to create, and even harder to reproduce, on the target hardware.

Let me be clear here. I'm not talking about any kind of instruction level simulator. That's not the point. The idea is to simply replace the target driver code with equivalent code that uses the screen and keyboard, or the file system, for I/O.

Slide 17

OK, it's time to do some real programming. Let's turn the measure program into a thermostat that turns on a cooler if the temperature rises above a setpoint and then turns it off again when the temperature drops below the setpoint. An alarm indicator will flash if the temperature exceeds a limit. The cooler and alarm are two of the three LEDs.

In practice, real thermostats incorporate hysteresis that prevents the cooler from rapidly switching on and off when the temperature is right at the setpoint. We implement that in the form of a "deadband" such that the cooler turns on when the temperature rises above the setpoint + deadband and doesn't turn off until the temperature reaches setpoint - deadband. It turns out that a deadband of 1 is quite sufficient.

Slide 18

Conceptually, our thermostat is a state machine with three states: OK, high, and limit. This slide illustrates in pseudo code form how the state machine works. It's really just a big switch statement.

Slide 19

Make a copy of measure.c and call it thermostat.c. Modify thermostat.c to incorporate the functionality outlined in the previous slide. Note by the way, that the header file driver.h includes #defines for the cooler and alarm bits. I suggest dividing the A/D input by 10 to get a sensible temperature and also to make the device less sensitive.

I'm going to throw one extra complication into this. We want to retain this program argument that sets the sample period in number of seconds. But we want the limit alarm to flash at a one second rate, that is, one second on and one second off, regardless of the specified sample period.

Slide 20

The thermostat program is probably complex enough that we would like to test it thoroughly in a simulation environment before moving it to the target. The file simdrive.c contains a set of driver functions that substitute for corresponding functions in trgdrive.c. These functions read A/D inputs from and write digital output to a shared memory region. Another program, called devices, gets A/D values from the keyboard and displays the digital outputs on the screen.

Have a look at the source file simdrive.c. It's really pretty simple. Note that we use the kill function to send a signal to the devices program when digital output changes.

Also take a look at devices.c. It too is pretty simple. It uses the curses library to create a pseudo graphical user interface.

Slide 21

At this point we need to add a couple more make targets to the measure project. Note that the thermostat simulation is the default all target that was already built when the project was created. Go ahead and build the devices target.

Slide 22

Fundamentally, we just need two debug launch configurations: one for the target and one for the host workstation. Then we just change to project and application to match what we're working on. So change the name of the record_sort configuration to host and point it at the thermostat_s application in the measure project. Change the name of the measure configuration to target. We'll change the application later after we build it.

Slide 23

So now just follow the instructions in this slide to debug your thermostat simulation. Enter different values for A/D in in the devices program and watch how the thermostat state machine responds.

This is a perfect use for the breakpoint condition feature. Set a breakpoint at the top of the switch statement and set the condition as current value > setpoint + deadband. Then after you've tested that state transition, change the condition to current value > limit.

Then of course, test the state transitions as the temperature is coming back down.

By the way, the devices program is terminated by entering a non-numeric value for A/D in. I usually type "q."

Slide 24

When you feel that the thermostat is running correctly in the simulation, follow these steps to build and debug it on the target. Be sure to delete thermostat.o before building the target version. Otherwise, the build fails because the object file doesn't get rebuilt and is in the wrong format.

Slide 25

OK, so now you have the thermostat running on the target. You might raise an objection that our thermostat isn't terribly useful in its current form because the setpoint and limit are "hardwired" into the program. You're right. In any real-world situation, we would want to be able to change the setpoint and limit, and maybe even the deadband.

So how might we approach the problem of making these parameters settable? If we were building this under DOS, we would probably use the function kbhit() to poll for console input within the main processing loop in thermostat. But Linux is a multi-tasking system. Polling is tacky! The proper solution is to create an independent thread of execution whose sole job is to wait for console input, then interpret it and act on it.

As shown here there are several possible ways to create this independent thread of execution. We might, for example, fork another process from within thermostat.c. Or we could arrange to start another process from the init script that we'll look at in a couple of weeks. In either of these cases we would have to use shared memory to communicate between the processes just as we did in the simulation version of the thermostat.

The other option, and the one I'm leading up to here, is to use threads, specifically Posix threads. But before we dive into the threads solution, let's look at why the alternatives aren't particularly useful.

Slide 26

Linux starts life with one process, the init process, created at boot time. Every other process in the system is created by invoking the fork() system call. The process calling fork() is termed the *parent* and the newly created process is termed the *child*. So every process has ancestors and may have descendants depending on who created who.

When I first started playing around with Linux, I found the fork concept to be totally bizarre.

fork() creates a copy of the parent process—code, data, file descriptors and any other resources the parent may currently hold. This could easily add up to megabytes of memory space to be copied. So to avoid copying a lot of stuff that may be overwritten anyway, Linux employs a *copy-on-write* strategy. All that's initially copied is the page table. All entries in both tables are set to read-only. Then when either process tries to write something, that causes a page fault and the kernel allocates a new page.

Since both processes are executing the same code, they both continue from the return from fork(). This is what I found so bizarre about it. In order to distinguish parent from child, fork() returns a function value of 0 to the child process but returns the PID, the process ID, of the child to the parent process.

The concept behind fork() is that *both processes continue in parallel*. So in the case of our thermostat, we could create a child process that simply monitors stdin for commands to alter the operating parameters. The next question of course is, how do we communicate the changed parameter value back to the parent process? In the memory-protected Linux environment, each process has its own private address space that no other process has direct access to. So the only way for two processes to share data is through shared memory as we've already seen with the simulation.

Slide 27

The file pseudo_code.pdf in the Week 4 Blackboard folder has some pseudo-code examples to illustrate the fork concept. These just don't fit well on a PowerPoint slide. Figure 1 shows how we might implement our thermostat using fork to create a second process. The usual way of handling fork is with a switch statement. fork returns a -1 if it fails, which it almost never does. A return value of 0 means this is the child process and a non-zero return means it's the parent process. In this case the child process waits for terminal input and the parent process handles the thermostat state machine.

Figure 2 illustrates what happens in most cases. After the fork, the child process loads a new executable image using one of the execve family of system calls. This, by the way, is how the shell works. When you type a command, the shell forks a new process and loads the executable image with that command name.

Note that execve() doesn't return if it succeeds.

Slide 28

As we've seen, the basic structural element in Linux is a *process* consisting of executable code and a collection of *resources* like data, file descriptors and so on. These resources are fully protected such that one process can't directly access the resources of another. In order for two processes to communicate with each other, they must use the inter-process communication mechanisms defined by Linux such as shared memory regions as we've already seen.

There is an alternative. We can create multiple threads of execution within one process. So for the thermostat, one thread carries out the state machine algorithm and the other waits for input at stdin. The operational parameters, setpoint, limit and deadband, reside in memory that is accessible to both threads. So the stdin monitoring thread modifies parameters as required and the thermostat algorithm uses these values without caring that they may have been changed.

Another common term for a thread is "task." And in fact the pthreads model looks very much like many commercial off-the-shelf multi-tasking operating systems.

Slide 29

Specifically, what we're going to use here is Posix threads. The acronym POSIX stands for Portable Operating System Interface with an X thrown in for good measure. It's actually a whole collection of specifications defining various parts of an operating system. Our interest here is the Posix Threads interface known as 1003.1c.

The mechanism for creating and managing a thread is analogous to creating and managing a process. The pthread_create() function is like fork() except that the new thread doesn't return from pthread_create() but rather begins execution at start_routine(), which takes one void * argument and returns void * as its value. The arguments to pthread_create() are:

- pthread_t – A *thread object* that represents or identifies the thread. pthread_create() initializes this as necessary..
- Pointer to a thread *attribute* object. More on that later.
- Pointer to the start routine.
- Argument to be passed to the start routine when it is called.

A thread may terminate by calling pthread_exit(). The argument to pthread_exit() is the start function's return value.

In much the same way that a parent process can wait for a child to complete by calling waitpid(), a thread can wait for another thread to complete by calling pthread_join(). The arguments to pthread_join() are the ID of the thread to wait on and a place to store the thread's return value. The calling thread is blocked until the target thread terminates.

A thread can determine its own ID by calling pthread_self(). Finally, a thread can voluntarily yield the processor by calling sched_yield().

Note that most of the functions above return an int value. This reflects the threads approach to error handling. Rather than reporting errors in the global variable errno, threads functions report errors through their return value.

Figure 3 in the pseudo_code document shows a trivial example of thread operation.

Slide 30

A thread may be terminated either voluntarily or involuntarily. A thread terminates itself either by simply returning or by calling pthread_exit(). In the latter case, all *cleanup handlers* that the thread registered by calls to pthread_cleanup_push() are called prior to termination.

A thread may be involuntarily terminated if another thread *cancels* it. The cleanup handlers are also called in this case.

Threads have an *attribute* called *detach state*. The detach state determines whether or not a thread can be joined when it terminates. The default detach state is PTHREAD_CREATE_JOINABLE meaning that the thread can be joined on termination. The alternative is PTHREAD_CREATE_DETACHED, which means the thread can't be joined.

Joining is useful if you need the thread's return value or if you need to synchronize several threads shutting down. Otherwise it's better to create the thread detached. The resources of a joinable thread can't be recovered until another thread joins it whereas a detached thread's resources can be recovered as soon as it terminates. It's worth noting that most multitasking kernels have no concept of a joinable task. All tasks are detached.

As a personal note, when I first encountered the Posix threads APIs, I thought to myself, "this looks like it was invented by a committee." Well, it was of course. But after reading David Butenhof's book on threads (see the resource list) I gained more appreciation for why it seems so complex. That's because I think the Posix folks addressed some subtle issues of multi-tasking programming that a lot of people had overlooked.

Slide 31

One thread can terminate another by calling pthread_cancel(). Thread cancellation is asynchronous, meaning it must be done with care. You don't just arbitrarily stop a thread at any point in its execution. Suppose it has a mutex locked. If you terminate a thread that has locked a mutex, it can never be unlocked. The thread may have allocated one or more dynamic memory blocks. How does that memory get returned if the thread is terminated?

The key to proper cancellation is the "cleanup handler." A thread can register one or more cleanup handlers that are called when the thread is cancelled or when it calls pthread_exit(). A cleanup handler is registered by calling pthread_cleanup_push(). Multiple handlers are invoked in the reverse order in which they were registered. The last one pushed is the first one executed.

A cleanup handler may only be useful during part of the thread's execution. For example, its job may be to release a mutex that the thread locked. Once the thread releases the mutex, that functionality is no longer relevant. A handler can be removed with pthread_cleanup_pop() when it's no longer needed. pthread_cleanup_pop() can only be called from the same function that invoked phtread_cleanup_push().

The default, and really the only safe, mode of cancellation is called deferred cancellation. This means that cancellation only occurs at defined cancellation points. To make a long story short, any system call that blocks is a cancellation point. If a thread is cancelled while it is blocked, the wait is interrupted and the cleanup handlers are called. You can also create your own cancellation points by calling pthread_test_cancel().

Slide 32

POSIX provides an open-ended mechanism for extending the API through the use of *attribute objects*. For each thread object there is a corresponding attribute object, which is effectively an extended argument list to the related object create function. A pointer to an attribute object is always the second argument to a create function. If this argument is NULL the create function uses appropriate default values.

An important philosophical point is that all pthread objects are considered to be “opaque”. This means that you never directly access members of the object itself. All access is through API functions that get and set the member arguments of the object.

This allows new arguments to be added to a pthread object by simply defining a corresponding pair of get and set functions for the API.

Slide 33

The pthreads specification only defines one required attribute for a thread object and that’s the detach state that we talked about a few slides back.

Before an attribute object can be used, it must be initialized. Then any of the attributes defined for that object may be set or retrieved with the appropriate functions. This must be done before the attribute object is used in a call to pthread_create(). If necessary, an attribute object can also be “destroyed”. Note that a single attribute object can be used in the creation of multiple threads.

There are a number of optional thread attributes, two of which are shown here. I can see the rationale for a stack size attribute, but I wouldn’t want to try setting a stack address. Surely that can’t be very portable.

Slide 34

Of course, as soon as we introduce a second independent thread of execution that accesses a common resource like shared memory, we have the potential for resource conflicts. Here’s a trivial and somewhat contrived example. Here are two threads, each of which wants to print the message “I am Thread n”. In the absence of any kind of synchronizing mechanism, the result could be something like what’s shown here.

What is needed then is some way to regulate access to the printer so that only one task can use it at a time.

Slide 35

A *mutex*, that’s short for “mutual exclusion,” acts like a key to control access to a resource. Only the thread that has the key can use the resource. In order to use the resource (in this case a printer) a thread must first *acquire* the key, the mutex, by calling an appropriate pthreads function. If the key is available, that is the printer is not currently in use by someone else, the thread is allowed to proceed. Following its use of the printer, the thread releases the mutex so another thread can use it.

If however, the printer is in use, the thread is blocked until the thread that currently has the mutex releases it. Any number of threads may try to acquire the mutex while it is in use. All of them will be blocked. The waiting threads are queued in priority order.

Slide 36

The Pthreads mutex API follows much the same pattern as the thread API. There is a pair of functions to initialize and destroy mutex objects and a set of functions to act on the mutex objects. This slide also shows an alternate way to initialize statically allocated mutex objects. PTHREAD_MUTEX_INITIALIZER provides the same default values as pthread_mutex_init().

Two operations may be performed on a mutex: lock and unlock. The lock operation causes the calling thread to block if the mutex is not available. trylock allows you to test the state of a mutex without blocking. If the mutex is available trylock returns success and locks the mutex. If the mutex is not available it returns EBUSY.

Slide 37

OK, we now have enough background to add a thread to our thermostat to monitor the serial port for changes to the operating parameters. We'll use a very simple protocol to set parameters consisting of a letter followed by a number. "s" represents the setpoint "l" the limit, and "d" the deadband.

cd to the Posix/ directory and open monitor.c. The first thing to notice is #include <pthread.h>. This header file prototypes the Pthreads API. Note the declarations of paramMutex and monitorT. The latter will become the handle for our monitor thread.

Because the parameters, setpoint, limit, deadband, are accessed independently by the two threads, the thermostat and the monitor, we need a mutex to guarantee exclusive access to one thread or the other. In reality, this particular application is so simple it probably doesn't matter, but it does serve to illustrate the point.

Note incidentally that setpoint, limit, and deadband are declared extern in thermostat.h. It is assumed that these variables are allocated in thermostat.c. If you happened to use different names for your parameters, you'll need to change them to match.

Take a look at the monitor() function. It's just a simple loop that gets and parses a string from stdin. Note that before monitor() changes any of the parameters, it locks paramMutex. At the end of the switch statement it unlocks paramMutex. This same approach will be required in your thermostat state machine.

Move down to the function createThread(). This will be called from main() in thermostat.c to initialize paramMutex and create the monitor. createThread() returns a non-zero value if it fails.

There's also a terminateThread() function that cancels and joins the monitor thread. This is the last function called before main exits.

Slide 38

Perhaps not surprisingly, we need to create a new project for the Posix version of the thermostat. This project, like the last one requires an additional make target to build the ARM version. And you'll need to point the debug launch configurations at the posix project.

Slide 39

Here are the changes required in thermostat.c to accommodate the monitor. Pretty simple really. The makefile uses the driver files from the measure/ directory.

Build the simulation and try it out. Be aware that the command input shares the same terminal as the program output.

Slide 40

Multi-threading does introduce some complications to the debugging process but, fortunately, GDB has facilities for dealing with those. Start the Eclipse debugger on thermostat_s and set a breakpoint just before the call to createThread(). When it stops at the breakpoint the Debug view shows one thread currently executing in the main function just as we've seen before.

Now step over the createThread() call. A second thread shows up. If you expand its call stack, you find that it's inside the clone() system function. Clone is similar to fork but gives you more control over what elements get copied.

Set a breakpoint just after the sleep call. When this breakpoint is hit, thread 1 indicates it's suspended because of a breakpoint and is still in the main function, obviously. Thread 2 is just suspended and it is way deep inside the fgets function. Click on the monitor entry in the call stack. Monitor.c opens in the editor with the fgets call highlighted. Set a breakpoint after fgets and resume the program.

Now enter a line of text. It doesn't really matter if it's a valid command or not. All we want to do at this point is get the program to stop at the breakpoint.

Play around with setting breakpoints in various places in both main() and monitor() to verify that parameters get updated properly and the thermostat responds correctly to the new values. When you're satisfied with the program running under the simulation, rebuild it for the target and run it there.

Slide 41

We've covered quite a bit of ground in this lesson. We began by looking at how to access hardware from Linux. I should point out, by the way, that accessing hardware directly from user space applications is not necessarily good practice. Toward the end of the class we'll look at Linux device drivers, which is the preferred mechanism.

We learned about Eclipse Makefile projects as a way to bring existing projects into Eclipse. We also saw how to tell Eclipse about multiple make targets. Next we looked at high level simulation as a way to do some initial hardware-independent testing on the workstation. Finally we considered mechanisms for creating multiple threads of execution including the Linux fork() function and Posix threads.

Next week we'll dive into network programming.