

Embecosm Application Notes 1 and 2 and MacOS 10.4

Robert Guenzel
Dept. Integrated Circuit Design
Technical University of Braunschweig
Germany
guenzel@eis.cs.tu-bs.de

March 8, 2010

1 Preface

This document describes the necessary additional actions that have to be performed when following the steps described in the Embecosm Application Note 2, Issue 3, Nov. 2008 on MacOS 10.4.

There may be better solutions for one or the other thing described in this document. My primary goal was to get it working, and having achieved this, I am always open for discussions and improvements.

2 Building the tool chain

I followed EAN2. binutils-2.16.1, gcc-3.4.4, uClibc-0.9.28.3 and or1ksim-0.3.0 install smoothly on MacOS 10.4. I did not (yet) try to build the gdb. The only tricky part is the linux-2.6.23.

The first problem arises when trying to do `make menuconfig`. The script will crash complaining about undefined symbols when it comes to linking the executable for the graphical menu. The problem is¹ that on MacOS 10.4 `libintl` is not linked by default. Adding `-lintl` at the very end of line 90 of the Makefile located at `linux-2.6.23/scripts/kconfig/Makefile` fixed the problem for me. After the change the line should look like this:

```
HOST_LOADLIBES = $(shell $(CONFIG_SHELL) $(check-lddialog) \ -ldflags  
$(HOSTCC) -lintl
```

Afterwards the configuration works and you can start compiling the actual linux kernel. This will crash complaining about a missing header named `elf.h`. The workaround here is to take `uClibc-0.9.28.3/include/elf.h` and copy it into `/usr/include`. In case `/usr/include` does not exist, just create it (as root of course). In the copy, remove or comment the line `#include <features.h>`. After this change linux compiled successfully².

¹that means: I *believe* the problem could be...

²The `elf.h` trick was found at <http://codesnippets.joy.net.com/user/incognito>, and was posted by Guido Sohne

To make reasonable use of the linux kernel you need to be able to modify the ram disk located in file `linux-2.6.23/arch/or32/support/initrd-fb-03.ext2`. Since this has an ext2 filesystem it cannot be natively mounted on MacOS 10.4. First I tried the *Mac OS X Ext2 Filesystem* which is a sourceforge project. I was able to install it, but the thing can only mount block devices, and since there is no loop-device on MacOS that can convert a character device (the file) into a block device, I was unable to mount the ram disk. The next try was `fuse-ext2-0.0.7` (another sourceforge project). I tried `fuse-ext2-0.0.7` with `MacFUSE-2.0.3` and it worked. With write support. Afterwards I was able to use the cross compile tool chain to create a small app. I then used `fuse-ext2` to put the app into the ram drive and rebuild the uclinux kernel (just put the ram drive image into the kernel binary). Then I put it into an orlksim and it worked.

3 The example platform and TLM-2.0 connections

After having the tool chain and the simulator running, I wanted to try out the LT example platform described in EAN1. That was (to my surprise) more difficult than getting the tool chain running.

3.1 The XTerm

Under MacOS 10.4 there are three fundamental differences to linux: First, we do not have X11 running by default, secondly we do not have a pseudo terminal multiplexer (`ptmx`) and finally, the OS does not allow us use the `F_SETOWN` command on the file descriptor of a pseudo terminal slave so that we cannot receive `SIGIO`, which means we simply do not know when the user is typing inside the XTerm.

So we need to set up X11 (in a MacOS style) from within the simulation, then we need to emulate the `ptmx` (afaik MacOS 10.5 *has* a `ptmx` so this step can be skipped on Mac 10.5) and finally we need to find an alternative for `SIGIO`. This section will talk about that

Note:

I did not use `TermSyncSC`. I only used and modified `TermSC`. Changes to `TermSyncSC` might become necessary in case the proposed modifications to `TermSC` are applied.

3.1.1 Setting up X11

To allow the code to be used both on Linux and MacOS (my development platform is MacOS 10.4, while my benchmarking platform is CentOS 5) I introduced a macro `LINUX_MODE`, if defined the code assumes to be compiled for linux, while without the macro being defined the code assumes to be compiled for MacOS.

The plan is to bring up Xquartz and the quartz window manager before starting `xterm` when using Mac OS. If X11 is already running both programs will refuse to start (given they attempt to use the same display), but still the subsequent start of `xterm` will work. So we can just try to bring quartz to life and ignore errors.

So I just added the following code directly at the beginning of `xtermInit()`:

```

int
TermSC::xtermInit()
{
#ifdef LINUX_MODE
    //if not on linux (assuming MacOSX) we first bring up Xquartz
    system("/usr/X11R6/bin/Xquartz :99 -swapAltMeta 2>/dev/null"
           "& sleep .25; echo $!>pid.txt");
    quartz_pid=get_pid_from_pid_txt();
    system("/usr/X11R6/bin/quartz-wm >/dev/null 2>&1 & sleep .25");
#endif
    ...
} // xtermInit()

```

As you can see, I start both `Xquartz` and `quartz-wm` as background jobs using the `system` function. Apparently I assume that `bash` is used as a shell, because I use `$!` to get the PID of the last created background job. This PID is put into a file called `pid.txt`. The helper function `get_pid_from_pid_txt` extracts the PID from the file. Also you can see that I added the class member `int quartz_pid` to the class `TermSC`.

I do not need to remember the PID of the `quartz-wm`, because it will die when I kill the `Xquartz`. If there was already an X running at the same display, both calls will have no effect. The quartz PID stored will be the number of the last created background job, which is somewhat dangerous, because it will be killed when the simulation is over, so I use display 99 to have a good chance there is no X running at the same display. Having said that, you need to set the environment variable `DISPLAY` to 99 before starting the simulation. On `bash`, that would be `export DISPLAY=:99.0`.

The `sleep` commands make sure that the programs are started in order. I must confess that it would have been much cleaner to fork off processes that use `exec` to first start `Xquartz` and then the window manager, but this way was implemented quicker.

For the reader's convenience I also show the aforementioned helper function:

```

unsigned int
TermSC::get_pid_from_pid_txt()
{
    char newLine='\n';
    std::ifstream pid_txt;
    std::stringstream tmp;
    pid_txt.open("./pid.txt");
    while (pid_txt.good()) //loop while extraction from file is possible
    {
        char c = pid_txt.get(); // get character from file
        if (pid_txt.good()){
            tmp << c;
            if (c==newLine) break;
        }
    }
    pid_txt.close();
    return atoi(tmp.str().c_str());
}

```

Of course we have to kill the whole X environment when our simulation is done on MacOS, so we add code to the `xtermKill` function:

```
    if( xtermPid > 0 ) {                                     // Kill the terminal
#ifdef LINUX_MODE
        kill (quartz_pid, SIGKILL);
#endif
        kill( xtermPid, SIGKILL );
        waitpid( xtermPid, NULL, 0 );
    }
```

Finally, we need to make a minor change to `xtermLaunch`:

```
    ...
    // Start up xterm. After this nothing should execute
    // (the image is replaced)
#ifdef LINUX_MODE
    execvp("xterm", argv );
#else
    execvp("xterm", "/usr/X11R6/bin/", argv);
#endif
} // xtermLaunch()
```

We need to use `execvp` instead of `execvp`, because (usually) the X-executables are not in the search path on MacOS. By the way: I always assume that X is installed at `/usr/X11R6/bin`, which is the default on MacOS. If that does not match your Mac, you need to change that.

3.1.2 The pseudo terminal multiplexer

Now we need to get rid of the `ptmx`. On MacOS 10.4 all the pseudo terminals are accessed by name, no mux available. So we will simply iterate over a set of pseudo terminals until we find an available one. The following code replaces the line

```
ptMaster = open("/dev/ptmx", O_RDWR);
```

in function `xtermInit`:

```
    unsigned int i=0;
    while (ptMaster<0 && i<16){
        std::stringstream s;
#ifdef LINUX_MODE
        s<<"/dev/ptmx";
#else
        s<<"/dev/ptyv"<<std::hex<<i<<std::dec;
#endif
        ptMaster = open(s.str().c_str(), O_RDWR);
        i++;
    }
```

As you can see, on MacOS I simply iterate over the pseudo terminals from the "v" block (named `ptyv0` to `ptyvf`) looking for a free pseudo terminal. For a complete mux emulation I'd need to iterate over all blocks from "p" through "w", but that is not really necessary here.

You can also see that I re-use the loop on linux. That is of course a waste (because if the `ptmx` fails to get a free terminal at the first try it will fail the next 15 times as well), but it reduces the places of conditional compilation which normally increases the readability of the code.

3.1.3 The SIGIO problem

The biggest and most annoying problem was that the call to `fcntl(ptSlave, F_SETOWN, getpid())` failed, telling me *SETOWN: Inappropriate ioctl for device*. Still I have the feeling that this can be fixed (in some way) but I totally failed there. So I needed to change the basic concept of reacting to a signal, to polling the `xterm` from a separate system thread.

Due to the fundamental change in the concept I was able to make the `ioEvent` a real event instance (not a pointer), and so I was able to use it for static sensitivities (and of course it must not be deleted in the destructor anymore). This allowed me to change `xtermThread` into an `SC_METHOD`³ ⁴ So instead of having `SC_THREAD(xtermThread)` in the constructor of the class, we now have:

```
SC_METHOD( xtermThread );
dont_initialize();
sensitive<<ioEvent; //this is now a REAL event, not a pointer!
```

What does this method do? Not much, frankly. See below:

```
void
TermSC::xtermThread()
{
    tx.write( buff);          // Get the char and send it on
}
```

Apparently it writes a char named `buff` to `tx`. `buff` is a new class member. How does `buff` get set? There is a thread that uses blocking reads on the pseudo terminal. It is created inside `xtermSetup`. Since we do not need to use any `fcntl` calls, and given that the thread will have a direct association to the current instance of `TermSC`, we can even get rid of the `instList`. This simplifies `xtermSetup` as shown below:

³This has the nice side effect of being more efficient, because the context switch overhead for a method is significantly smaller than for a thread

⁴Hint for `TermSyncSC`: Since this is now a method, `wait` does not work anymore. A simple way to emulate that is to use Boolean class member that is initially false. When the method starts it checks the bool. If it is false it sets it to true and uses `next_trigger` to restart after the delay and returns. When it starts with the bool being true, it writes to `tx`, sets the bool to false and returns.

```

int
TermSC::xtermSetup()
{
    int    res;
    char   ch;

    // The xTerm spits out some code,
    // followed by a newline which we swallow up
    do {
        res = read( ptSlave, &ch, 1 );
    } while( (res >= 0) && (ch != '\n') );

    if( res < 0 ) {
        xtermKill( NULL );
        return -1;
    }

    //create pthread, pass current instance as argument
    pthread_create(&thread,NULL, &my_pthread, this);
    return 0;          // Success
} // xtermSetup()

```

The start has not changed, but the end of the function collapsed to simply creating a thread (note the new class member `pthread_t thread`). The thread gets the current `this` pointer as an argument, so it knows what instance it is connected to, and (as mentioned before) we do not need a list or map.

Note that we do not need the `ioHandler` function anymore. Instead, we need a new thread as shown below:

```

void* my_pthread(void* arg)
{
    TermSC* that=(TermSC*)arg;
    bool run=true;
    while(run)
        if( !that->pop_from_xterm() ) run=false;
    return NULL;
}

```

The thread enters a loop and then tells its associated `TermSC` instance to pop from the xterm. If that fails, the loop ends and the thread terminates. Such a failure will occur when the simulation ends and the xterm is killed. Now the question is, how does `pop_from_xterm` look like:

```

bool
TermSC::pop_from_xterm()
{
    if( read( ptSlave, &buff, 1 ) != 1 ) {
        perror( "TermSC: Error on read" );
        return false;
    }
    ioEvent.notify(sc_core::SC_ZERO_TIME);
    return true;
}

```

This function blocks the calling (system) thread until data is available on the terminal or an error occurs. If an error occurs the function simply returns with false, so that the calling thread will terminate. If a character was read (into class member `buff`) the `ioEvent` will be fired, starting the method named `xtermThread`, which will then in turn put the character from `buff` into `tx`. Note that function `xtermRead` is not used anymore.

These changes have two effects: First, they allow me to compile the code both on MacOS and linux, and secondly, they make the code a little smaller.

With this final change, the whole system (without `TermSyncSC`) could be compiled and simulated. I used a slightly modified version of the linux platform. The modification was that I did not use `TermSyncSC`, but `TermSC`, because I implemented the delay at another point in the system (I simulated the send and receive delays both in the UART component, so that the terminal could always stay untimed. But that is just a personal preference).

3.2 Getting some more speed

Since I wanted to benchmark the impact of certain timing modeling styles on overall simulation times, I wanted the untimed reference system to be as fast as possible. In other words, the only thread left should be the one for the ISS.

3.2.1 The UART models

I started with `UartSC`. Compiling with `"-O3"` I got linker errors, because some functions were defined `inline`, but were placed in the `".cpp"` file (which means there are no symbols for the functions in the `".o"` file). I moved the functions into the header file to fix that.

Then I removed the `printf` and the line before (the access to `sc_time_stamp()`) in `rxMethod`, because that is some kind of debug output that slows down the simulation. I removed the same kind of output from the `rxMethod` in `TermSC`.

As another change I made the `rxMethod` virtual, because I will override it at a later point (but that is only because I do not use `TermSyncSC`, so this change is your own choice).

Then I made the `busThread` a method as shown in the next code snippet. You can see that it is statically sensitive to the `txReceived` event. Also note the new Boolean class member `init_method` that is initialized to true. It is used to identify the initializing call of the `busThread` method. In this case it only sets the registers and interrupts. Only at the next start it will write to `tx` (which happens when the event is triggered).

```

UartSC::UartSC( sc_core::sc_module_name name) :
    sc_module( name ),
    intrPending( 0 ),
    init_method(true)
{
    // Set up the thread for the bus side
    SC_METHOD( busThread );
    sensitive<<txReceived;

    ...

}          /* UartSC() */

...

void
UartSC::busThread()
{
    if (init_method) init_method=false; else tx.write(regs.thr);
    set( regs.lsr, UART_LSR_THRE );          // Indicate buffer empty
    set( regs.lsr, UART_LSR_TEMT );
    genIntr( UART_IER_TBEI );                // Interrupt if enabled
}          // busThread()

```

The next module I modified is `UartSyncSC`. In this file I override both `busThread` (which is now a method) and `rxMethod` (which is new). The code snippet at the next page shows the changes. Note that the new Boolean class members `rd_state` and `wr_state` are both initialized to false. Both method use a two-state state machine. In the initial state they will switch to the next state and restart after the delay. Awakening in the non-initial state they will either read `rx` or write to `tx` and then go back to the initial state. It can be seen that now `UartSyncSC` is dealing with both the send and receive delays, so that we do not need `TermSyncSC` anymore.

```

void
UartSyncSC::rxMethod()
{
    switch(rd_state){
    case false:
        next_trigger(charDelay);
        rd_state=true; return;
    case true:
        rd_state=false;
        regs.rbr = rx.read();
        set( regs.lsr, UART_LSR_DR );           // Mark data ready
        genIntr( UART_IER_RBFI );           // Interrupt if enabled
    }
} // rxMethod()

void
UartSyncSC::busThread()
{
    switch(wr_state){
    case false:
        if (init_method) init_method=false;
        else
        {
            next_trigger(charDelay);
            wr_state=true; return;
        }
    case true:
        wr_state=false;
        tx.write(regs.thr);
    }
    set( regs.lsr, UART_LSR_THRE );           // Indicate buffer empty
    set( regs.lsr, UART_LSR_TEMT );
    genIntr( UART_IER_TBEI );           // Interrupt if enabled
}
} // busThread()

```

The next to change was UartIntrSC. There was a thread that was waiting at a fifo's output to feed the stuff that arrives there into an sc_signal. I converted the thread into a method as shown below

```

UartIntrSC::UartIntrSC (sc_core::sc_module_name  name,
                        unsigned long int        _clockRate) :
    UartDecoupledSC (name, _clockRate),
    intrQueue (1)
{
    intr.initialize(false);
    SC_METHOD( intrThread );
    sensitive<<intrQueue.data_written_event();
    dont_initialize();
}      /* UartIntrSC() */

void
UartIntrSC::intrThread()
{
    bool val;
    intrQueue.nb_read(val); //no need to check success
    intr.write(val);
}      // intrThread()

```

No further changes to the Uart components are needed.

3.2.2 The ISS wrapper

In the ISS wrapper a (unfortunately) common mis-understanding took place. We tried to emphasize this issue when we wrote the new TLM-2.0 LRM, but it can be easily overseen:

Creating temporary generic payloads is really bad. Inside the generic payload there is potentially big array that is being new-ed and deleted with the payload being constructed or destructed. The read and write upcalls both create generic payloads on the function stack and that will kill your LT simulation performance. I did some benchmarks for the TLMWG a couple of months ago, and the speed loss when allocating and deallocating payloads for each `b_transport` call can be (depends on how much communication is going on) an order of magnitude.

Since there can never be two read or write upcalls at a time (they happen strictly sequential), a "pool" of payloads with depth 1 suffices. The drawback of pools is that when you take a payload from the pool, it is in an unknown state and you have to initialize everything. To work around that I suggest using a pool for read transactions and one for write transactions that limits the number of members to initialize when a payload comes from the pool.

In our case that means that `Or1ksimSC` will get two new members of type `tlm_generic_payload`: `rd_trans` and `wr_trans`. They are initialized in the constructor:

```

OrksimSC::OrksimSC ( sc_core::sc_module_name  name,
                    const char                *configFile,
                    const char                *imageFile ) :
    sc_module( name ),
    dataBus( "data_initiator" )
{
    orksim_init( configFile, imageFile, this, staticReadUpcall,
                staticWriteUpcall );

    SC_THREAD( run );                // Thread to run the ISS

    rd_trans.set_read();
    rd_trans.set_data_length(4);
    rd_trans.set_streaming_width(4);
    rd_trans.set_byte_enable_length(4);

    wr_trans.set_write();
    wr_trans.set_data_length(4);
    wr_trans.set_streaming_width(4);
    wr_trans.set_byte_enable_length(4);

} // OrksimSC()

```

As you can see, we just set the members that can never change. The rest is set when the upcalls happen:

```

uint32_t
OrlksimSC::readUpcall( sc_dt::uint64  addr,
                      uint32_t      mask )
{
    uint32_t      rdata;          // For the result

    // Set up the payload fields. Assume everything is 4 bytes.
    rd_trans.set_address( addr );
    rd_trans.set_data_ptr( (unsigned char *)&rdata );
    rd_trans.set_byte_enable_ptr( (unsigned char *)&mask );
    rd_trans.set_response_status(tlm::TLM_INCOMPLETE_RESPONSE);
    rd_trans.set_dmi_allowed(false);

    // Transport. Then return the result

    doTrans( rd_trans );
    return rdata;
}      // readUpcall()

void
OrlksimSC::writeUpcall( sc_dt::uint64  addr,
                      uint32_t      mask,
                      uint32_t      wdata )
{
    // Set up the payload fields. Assume everything is 4 bytes.
    wr_trans.set_address( addr );
    wr_trans.set_data_ptr( (unsigned char *)&wdata );
    wr_trans.set_byte_enable_ptr( (unsigned char *)&mask );
    wr_trans.set_response_status(tlm::TLM_INCOMPLETE_RESPONSE);
    wr_trans.set_dmi_allowed(false);
    // Transport.
    doTrans( wr_trans );
}      // writeUpcall()

```

Here we set the addresses, the data pointer the byte enables and we reset the response to incomplete and the DMI hint to false. The former three members are different for each access, so setting them is obvious. The latter two members must be set, because they are still at the values they got from the last upcall.

Finally, in the `doTrans` function I propose to make the dummy delay a class member to avoid frequent creation and destruction of the time object. However, the impact is not as big as it is for the generic payloads.

As another speed improvement it might be good to inline the one or other function, but I did not want to change the code too much, so I refrained from doing that.

3.2.3 Performance gain

To see if the changes made sense I tried to benchmark them. To this end, I compared two system models: One in which I only did the changes to get the system running on MacOS, and one in which I applied the performance improvement changes as described in this section as well.

The time to boot the linux kernel did not measurably change: On my machine it took 28.2 seconds for the system which was not performance improved, and 28.1 seconds for the one that was. Did I fail?... No. The reason is that the system with the ISS booting the kernel spends 90+% in the ISS. To really measure the performance of the SystemC part, I disabled the ISS and added a sequence of UART accesses directly into the thread that normally executes the ISS.

The sequence first initializes the UART and then pumps one million characters through the UART to the xterm. Now the system which was not performance improved runs for 7.2 seconds, while the improved system runs for 5.7 seconds. That means the improved system saves about 21% of runtime. That is not too bad, given that the xterm was really printing the characters. Now I could also try to find out which change was the most important one (removing the threads or avoiding stack allocation of payloads), but I did not have time for that.

So as soon as your system spends much time in the SystemC part (e.g. when the quantum is small, or when more parts of the ISS, like memories, PIC, etc. are moved into SystemC), the improvements pay off.