PHYS 319 Programming and Debugging with the MSP430 for Linux

This is a summary of how to get started writing assembly and C programs for the MSP430 Launchpad that we will be using in Phys 319. There are separate Windows and Mac versions of this document.

We will need several components: an assembler, a program to load our assembled programs onto the MSP430, a C compiler, and a debugger. There are a number of options available for several of these components. For the first couple of weeks we only need the assembler and loader, so we'll deal with those first. For the third and fourth weeks we need a C compiler and debugger, and for the following two weeks, we'll use some other software on the host computer to talk to the Launchpad via its serial port.

Installing and using an Assembler (Needed for Labs 1+2)

There are a number of assemblers available for the MSP430 line. To keep things simple and the same for everyone, we're going to use a small, bare-bones assembler.

1) Download the .tar.gz source from:
http://www.mikekohn.net/micro/naken430asm_msp430_assembler.php

At a command line, uncompress the source code with:

```
tar -zxvf ~/Downloads/naken430asm-2011-10-30.tar.gz
```

move into the source code directory:

```
cd naken430asm-2011-10-30
```

run the configure script:

```
./configure
```

build the assembler:

```
make
```

install it:

```
sudo make install
```

2) To load programs onto the msp430, we'll use a program called mspdebug. This program is in the software repositories for many Linux distributions. On Ubuntu, try:

```
sudo apt-get install mspdebug
```

3) We need to do set up so that when the Launchpad is plugged in, you have permission to use it. Again at the command line:

```
sudo sh -c "cat > /etc/udev/rules.d/93-msp430uif.rules"
ATTRS{idVendor}=="2047",ATTRS{idProduct}=="0010",MODE="0666", ENV{ID_MM_DEVICE_IGNORE}="1"
ATTRS{idVendor}=="0451",ATTRS{idProduct}=="f432",MODE="0666", ENV{ID_MM_DEVICE_IGNORE}="1"
^D
```

(^D is CTRL-d)

You've just created a file that tells the device manager that all users should be allowed to access the Launchpad. You will need to restart udev: `sudo service udev restart` on Ubuntu, or just reboot, before this will take effect.

4) Finally, we create a file that tells the driver that talks to the launchpad to ignore some features of the board that it doesn't have.

```
sudo sh -c "cat > /etc/modprobe.d/msp430uif.conf"
options usbhid quirks=0x0451:0xf432:0x20000000,0x2047:0x0010:0x20000000
^D [^D is CTRL-D]
```

Reboot at this point, and you should be ready to use the Launchpad.

To use the assembler:
`naken430asm -o <name>.hex <name>.asm`
(where <name> is the name of your program)

Grab: http://phas.ubc.ca/~michal/phys319/blink.asm
and ensure that you can assemble it without errors.

Then to load it into the Launchpad (this won't work if you don't have a Launchpad connected):
```
mspdebug rf2500
prog <name>.hex
^D    [^D is CTRL-D]
```
rf2500 is the name of the protocol used to talk to the Launchpad.

Installing a C compiler and debugger (Needed for Lab 3)

For labs after the first two weeks, we'll need a C compiler, and a debugger will likely be useful.
TI provides a feature-rich integrated development environment (IDE) called Code-Composer Studio (CCS). There is a Linux version of CCS, though it doesn't support our Launchpad. So, we'll use gcc/gdb/mspdebug.

Download the linux installer package from:
http://software-dl.ti.com/msp430/msp430_public_sw/mcu/msp430/MSPGCC/latest/index_FDS.html
(you'll need to register with TI and declare that you won't export the software to any hostile states).
To install it, do:
(these first three lines may be necessary on a 64bit computer)
```
sudo dpkg --add-architecture i386
sudo apt-get update
sudo apt-get install libc6:i386 libstdc++6:i386
chmod ugo+x msp430-gcc-full-linux-install-4.2.0.36.run
```
then:
`./msp430-gcc-full-linux-install-4.2.0.36.run`
When installing, let it install into the default folder ($HOME/ti).

Finally, in your home directory, edit your .profile file to add the compiler to the path. Try:
`nano ~/.profile`
at the bottom of the file add:
`export "PATH=$PATH:$HOME/ti/msp430_gcc/bin"`
You'll need to open a new terminal for this to take effect.

Using the Compiler:

You will have the best experience if you put your program into its own directory, along with a Makefile that automates some of the building.

Grab:http://www.phas.ubc.ca/~michal/phys319/cblink.tar.gz
uncompress it:
```
tar -zxvf cblink.tar.gz
```
move into the directory:
```
cd cblink
```
compile the source:
```
make
```
load it into the Launchpad:
```
mspdebug rf2500
prog main.elf
^D
```

Using the debugger [not needed immediately, for debugging programs later]
You need to ensure that your program is compiled with a `-g` option (the supplied Makefiles do this) so that the executable files contain information allowing the debugger to know which instructions came from which source code lines, and where variables reside in memory. Debugging is complicated by compiler optimizations, so you should check to make sure that there is no -O option (-Os, -O2 etc) in the compilation command.

Now, load your program to be debugged into the Launchpad with mspdebug:
```
mspdebug rf2500
prog main.elf
```
but now, instead of quitting mspdebug, start gdb:
```
gdb
```

mspdebug is now in a mode where it is awaiting debugger commands from another program. Open a second terminal window, and type:
```
msp430-elf-gdb main.elf
```

which tells the debugger which program you want to debug. This program must be identical to the one downloaded with mspdebug! **If** this gives an error about not being able to find libexpat.so.0, then try:
```
cd /usr/lib/i386-linux-gnu; sudo ln -s libexpatw.so.1 libexpat.so.0
```

Now tell gdb that the program is running on a remote target, accessible on port 2000:
```
target remote :2000
```
If all is well, you can now use gdb commands to execute the program and examine variables as they run.

You can find lots of documentation on gdb on the web, but some useful things to try are:

```
break <line>
```
sets a breakpoint at line number `<line>`.

```
c
```
'continue' until a breakpoint is hit. This will let the program execute until the breakpoint you just set. You can only set two different breakpoints. You can see which breakpoints are set with
```
info break
```
and remove a breakpoint with:

`clear <line>`

`print a` shows you the value of the variable a.
`list` will show you a few lines of source code just ahead of the current position
`info reg` will show you all the registers.
`condition 1 i == 3` sets a condition on breakpoint 1, if i is not 3, it will continue (note that this is very slow, so if you're waiting for i to get to 5000, be prepared to wait a long time.
`monitor reset` resets the program on the Launchpad to start over. Anything after `monitor` is assumed to be a raw mspdebug command.
`load prog.elf` will load the program from prog.elf into the msp430 (same as the prog command entered directly into mspdebug).

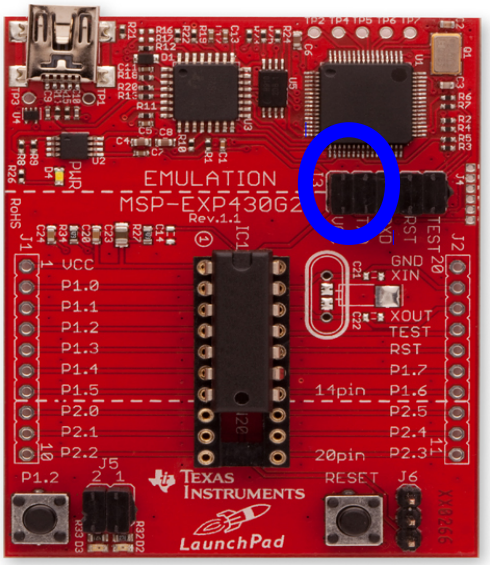
Serial Port Example (For lab 5).

Our next example is to set up a program that talks to the MSP430 while it is running. The Launchpad board presents a USB-Serial interface to the host computer, so that we can talk to the MSP430 as though it were connected by an old-fashioned serial port.

We will do this with a program in python. The python program uses a graphical user interface called gtk to draw windows on the screen, and a plotting library called matplotlib to make graphs. We will need to install these components.

You probably have python 2.7 installed already, but you will need some other bits. On Ubuntu, try:
`sudo apt-get install python-gtk2 python-serial python-matplotlib`

Now: we have two versions of the Launchpad board floating around and there are two versions of our demo program. One version of the program will work on either versio
complicated). The simpler version of the program will only work on t
board. Have a look at your board. Underneath the words “Emulation”
should be either Rev. 1.4 or Rev. 1.5. If your board says Rev. 1.5, then
a) Download
http://www.phas.ubc.ca/~michal/phys319/temperature_demo4.tar.gz
b) Ensure that the two left-most jumpers (circled in the photo)
are oriented horizontally (opposite to all the other jumpers).



If your board says Rev. 1.4 (or earlier), then
a) Download
http://www.phas.ubc.ca/~michal/phys319/temperature_demo3.tar.gz
b) Ensure that all the jumpers are oriented vertically.

Build the program in main.c and flash into the Launchpad.

You will need to edit the python program to contain the correct serial port name.

On Linux it is probably /dev/ttyACM0

Then you can start the python program (you can double click on it, or start it from the command line with `python2 python-serial-plot.py`)
After it has started, push button s2 on the Launchpad, and temperature measurements will be delivered from the Launchpad to the python program and plotted.

Older versions of the linux kernel had some trouble with the serial port on the Launchpad. If the Launchpad sent characters to the host while no program was listening for them, the driver would fail, and you could never receive characters from the board again, until you unplugged and replugged the board. This is fixed in recent versions of Linux (as of Ubuntu 14.10, or kernel-3.16). But if you are using an older version of linux, you won't want the Launchpad serial port to be sending characters to the PC when no program is listening to receive them. If that happens, you may need to unplug and replug the Launchpad USB connection. So in the case of this example, don't quit out of the python program while the Launchpad is sending data. Press the s1 reset button on the Launchpad before quitting the python program.

If you don't see data coming onto the plot immediately after pushing the button, check that the jumpers circled in the image are oriented correctly for the program version you're using.