MSP430 Microcontroller Program Development

This document provides hints on program development for the msp430 microcontroller. It is assumed that free development tools are to be used. All the below applies equally well to Linux and Windows, except where explicitly stated. For convenience at Durham, all the relevant files mentioned below can be found at http://www.dur.ac.uk/peter.baxendale/stuff/msp430 .

Documents

The following 3 documents are essential reading.

- ***User manual for microcontroller*** type (eg slau49d.pdf for msp430x1xx). See http://www.ti.com (follow the link for Microcontrollers). This gives full programming details for all modules present in some or all of a particular group of microcontrollers. This manual is **big**, and I suggest you only read the bits you want when you need them.
- ***Data sheet for microcontroller*** concerned. This shows which modules are fitted in a particular microcontroller and what the chip's pins are connected to.
- ***GCC (C/C++ compiler) manual*** (mspgcc-manual.pdf). Included in msp-gcc distribution (in docs directory, see below for installation) and available on-line at http://mspgcc.sourceforge.net . As well as documentation on the compiler this gives a nice introduction to the MSP430 microcontroller hardware. Since the compiler is a port of the Gnu gcc (Gnu Compiler Collection) compiler, if you want detailed information about gcc (eg standard compiler options other than the special msp430 ones) you have to refer to the standard gcc manual (on-line on Linux systems, or from http://www.gcc.gnu.org (although in practice, you may well never need to refer to this).

Tools

C compiler, mspgcc from http://mspgcc.sourceforge.net. A useful page with instructions on installation of the compiler is at http://www.mikrocontroller.net/en/index . After installation you probably want to modify your PATH environment variable to include the various compiler executables. In Linux, edit ~/.bashrc to include this. In Windows XP, right click on My Computer, select properties, select advanced, select environment variables. Edit PATH to append ";\mspgcc\bin" (assuming this is where you installed the compiler).

Jedit, free editor which knows about C and C++ and has lots of nice features to make program development easier. Written in java, it requires a java run-time environment. Full details and downloads at www.jedit.org. Of course, you can use any editor you are familiar with.

Using Jedit

Installation

Once Jedit is installed, set the proxy by selecting utilities/global options/proxy servers. In the dialogue, click “Use HTTP proxy server” and enter wwwcache.dur.ac.uk as HTTP proxy host, and 8080 as HTTP proxy port. Leave other fields blank (This all assumes you are working in the University. If you are outside the University you probably don't need a proxy.)

Install the console plugin by selecting plugins/plugin manager. Click on the install tab and the list of available plugins should be downloaded from the net. (If it doesn't get downloaded, check your network connection is ok and that you've set your proxy correctly.) Select the console and error list plugins and click on install. They should be downloaded and installed.

The console plugin allows you to run system commands from within jedit, eg the compiler. The error list plugin parses the output generated by the system command you run. It understands gcc error messages and directs you automatically to the corresponding line in the editor.

Make the console and error list windows docked at the bottom of the screen by selecting utilities/global options/docking. By default, the windows that will be opened by the “console” and “error list” plugins float. Select “bottom” from the drop down menu next to console and error list (note: do something else if you like – this is just what I find convenient).

Copy the file make.xml into .jedit/console/commando. The .jedit (note the leading dot) directory can be found in your home directory in Linux, or in Documents and Settings\your_user_name in windows. (Note: the contents of make.xml are appended to this document in case you can't download it.)

Using the console plugin

The console plugin can be used to open a window which allows shell commands to be typed in. The output of these is interpreted by the error list plugin. To just run such a shell command, select plugins/Console/Console. Also listed in the plugins/Console menu are various “commander” scripts, which effectively enter the command for you. This command can be based on the file currently opened in the editor, or various other aspects of the jedit editor.

If you select one of these commands you get a dialogue tailored to that command, with typically (possibly always) 2 tabs: settings and commands. The Commands tab shows you the system command that will actually be run. Most of these commands are not useful for msp430 work but come built in with the console plugin.

The make command has been added by copying the file make.xml to the correct place, as above. This command (as you can see if you invoke it) runs “make” after changing directory to that corresponding to the current buffer (ie the file currently displayed in the editor). Thus it assumes you have a Makefile (see below for information about make) in that directory.

The two buttons at the bottom left hand corner of the jedit window (assuming

you have docked them as described above) allow you to open the “console” and “error list” windows. You can close them again by clicking on the X. The “console” window shows you the output produced by running a command, including error messages. Compiler error messages are also displayed in the “error list” window.

The first time you use the “make” command, check the console window to make sure that “make” ran as you expected. Then check the error list window for compiler errors. Clicking on the error (what? you haven't any?) will take you to the relevant line in the editor. Remember, compiler error messages are sometimes caused by an earlier error in the file, so it's usually a good idea to start with the first error. Sometimes it's a good idea to correct the first 2 or 3 and then re-run make to see which errors remain.

Once you've run “make” this way, the console is left in the right directory for further makes. So if you want, next time you can just click at the end of the console window and type “make” instead of using plugins/console/make from the menu. Both will have the same effect.

It's easy to write your own “commander” scripts like make.xml. Take a look at make.xml and also gcc.xml to see how it's done. You can use jedit to open and edit them – jedit understands xml files and gives you a nice coloured display.

Using Make

First, read the document “Using Make” for a basic introduction to make.

Take a copy of Makefile.msp430 and use this as a template for your own makefiles. For work on the CC2420 radio transceiver boards use Makefile.cc2420 instead (you will also need my libraries). If you use one of these files, rename it to "Makefile" after you copy it - this is the file name that "make" looks for by default.

The line near the top containing “MCU =” is used to specify the microcontroller you are using. In the body of the Makefile, items like $(CC) are replaced by the setting of the variable in the brackets (eg, CC = msp430-gcc).

The first line (excluding rules and variable definitions) is the one that is built by default if you just type “make”. You can build something else defined in the makefile by typing, for example, “make clean”. This example will build what the “clean:” line specifies, which in this case deletes all the files generated by the compiler to leave you a “clean” directory with just the source files in it.

In my template Makefile, “Make program” will invoke the JTAG programmer to program your built file into the microcontroller flash memory.

To write or modify your Makefile, **don't** use windows editors that put spaces instead of tabs into text files (eg notepad often does this) – “make” **insists** on having tab characters.

The Makefile automatically builds the dependency list for each source file. Note that this doesn't include (in the current version) libraries. If you change a library it's probably best to do a "make clean" followed by "make" to force a complete rebuild.

The C Compiler

The compiler is well documented in its manual, mentioned above. Particularly read how to access microcontroller registers and how to write interrupt service routines. Look at the example files appended to this document to see the kind of thing you can do. The compiler manual also contains useful hints on how to write efficient code. Remember to always #include <io.h>.

Usually you will have plenty of program memory (flash) so you don't need to worry too much about reducing program size. But the RAM (for variables) is **very** limited in size, so you need to avoid using large arrays etc. Floating point arithmetic is **very** slow, so only do this if you really have to and you can afford the performance hit.

The compiler comes with a C library; see the manual for a list of the functions included. The sample files below include bits and pieces which allow you to use normal(ish) "printf" and other C i/o functions, assuming you want output on a microcontroller UART (it's easy to change that). Notice that the printf library formatting facilities are limited (eg no floats supported) and remember that including a single printf or sscanf in your program will hugely inflate your code size (but they are very handy for debugging).

On the subject of debugging, there are 2 utilities that I know of: "insight" and "ddd". I gather that insight is more appropriate for Windows and ddd for Linux. You're pretty much on your own here, because I've only very briefly looked at ddd and don't use it regularly (I get by with printf and an oscilloscope). Look on the web for more information.

The template Makefile includes the option to generate an assembler source listing, intermixed with the C source. If you include a source file such as "foo.c" in your Makefile, then type "make foo.lst" in the console window. It's sometimes useful to look at this to check you're doing the right thing, or to see how efficient the gcc code generator is. I usually find that the code is very good, leaving room for only marginal improvements by hand coding in assembler.

JTAG Programmer

The template Makefile incorporates the necessary commands to run the JTAG programmer in order to erase and program the MSP430 flash and to force a reset. You could add other lines to, for example, erase the information memory.

If you find flash programming refuses to work, don't panic (not yet, anyway). Usually this is due to forgetting to turn the power on or connecting the JTAG lead the wrong way round (or not at all). But sometimes it just refuses to play. In these circumstances I have found that using gdb with gdbproxy (see appendix for more information) allows me to erase the flash, after which the jtag programmer works again. Just one of those mysteries that make life interesting.

PRB 12 Oct 2006

Appendix

make.xml

```
<?xml version="1.0" ?>

<!-- build by invoking make from directory of current buffer -->
<!-- adapted from gcc.xml by Stefan Beckert (becki at web dot de), 2004-10-25 --
>

<!DOCTYPE COMMANDO SYSTEM "commando.dtd">

<COMMANDO>
  <UI>
        <CAPTION LABEL="Make">
        <FILE_ENTRY LABEL="Source file(s)" VARNAME="source"
            EVAL="buffer.getName()" />
        </CAPTION>
        <CAPTION LABEL="Path to Make">
          <ENTRY LABEL="(no spaces)" VARNAME="make" DEFAULT="make" />
      </CAPTION>
  </UI>
  <COMMANDS>
      <COMMAND SHELL="System" CONFIRM="FALSE"><!-- cd to working dir -->
        buf = new StringBuffer("cd \"");
        buf.append(MiscUtilities.getParentOfPath(buffer.getPath()));
        buf.append("\"");
        buf.toString();
      </COMMAND>
      <COMMAND SHELL="System" CONFIRM="FALSE">
          buf = new StringBuffer(make);
        buf.toString();
      </COMMAND>
  </COMMANDS>
</COMMANDO>
```

SPI

Here's an initialisation routine for the SPI interface on the MSP430, plus some functions to access it. The functions are simple and declared "inline" - this means they generate code in line without the overhead of a function call. Alternatively you could make them ordinary functions, or use #define to declare macros to do the same job. Inline is supported by C99 compatible compilers (like gcc) but not basic ANSI standard ones.

```
#include <io.h>
#include <spi.h>

/** \file spi.c
SPI functions are either on here or in spi.h.
*/

/** Initialise spi interface on uart0.
Set for 8 bits data, master, 3 pin mode.
Uses SMCLK with divide set to 2 (ie maximum speed). Interrupts disabled.
*/

void spiInit(void) {
      U0CTL = 0x17;     // SPI, 8 bit data, no loopback, SPI mode, master, reset
      U0TCTL = 0xa3;    // CKPH=1, CKPL=0, SMCLK, 3 pin mode, Tx empty

      U0BR0 = 2;  // smclk divider
      U0BR1 = 0;
      U0MCTL = 0; // not used, but should be 0

      IE1 &= ~0x80;     // disable interrupts
      IE1 &= ~0x40;

      ME1 |= 0x40;      // enable spi module
      U0CTL &= ~1;      // release reset

      P3SEL |= 0xe;     // P3 bits 1,2,3 used for spi
      P3DIR |= 0xa;     // P3 bits 1,3 used as output
      P3DIR &= ~4;      // P3 bit 2 input
}
```

Include file spi.h. Notice the #ifndef at the beginning. This with the #define and #endif at the end of the file make sure that this file can be included only once. It avoids problems caused when you may include another file which itself also includes this file. It's a good idea to get into the habit of always putting this kind of thing at the start of every .h file.

```
#ifndef SPI_INCL
#define SPI_INCL
/** \file spi.h
Function prototypes and in-line functions for spi handling.
*/

/*
There's no need for an Rx function, since have to send
something on SPI in order to get something back.
*/
```

```
void spiInit(void);                // initialises spi port

/** Send a byte to spi and return byte received on spi.
\param c - byte to send to SPI.
*/

static inline unsigned int spiTx(unsigned char c) {
      while(!(IFG1 & UTXIFG0));    // wait for tx buffer empty
      U0TXBUF = c;

      while(!(IFG1 & URXIFG0));    // wait for response
      return U0RXBUF;
}

/** Disable the spi module (for power saving).
*/

static inline void spiOff(void) {
      ME1 &= ~0x40;     // disable spi module
}

/** Send a byte to the spi but don't return anything.
This is quicker than spiTx() but leaves
a character in the spi rx buffer. Also, the transmit may not have completed
before the
function returns, so make sure to check it's all been sent before disabling or
deselecting the spi.
*/
static inline void spiTxQuick(unsigned char c) {
      while(!(IFG1 & UTXIFG0));    // wait for tx buffer empty
      U0TXBUF = c;
}

/** Macro to test if spi transmission has finished.
\return non-zero if tx buffer is empty (ie transmission finished), 0 otherwise.
*/
#define SPI_TXDONE (U0TCTL & TXEPT)

/** Macro to clear byte ready flag in spi receive register.
Useful after a series of spiTxQuick calls,
when there will be a character in the buffer which you probably don't want.
*/
#define SPI_RXCLEAR IFG1 &= ~URXIFG0

#endif
```

uart.h

```
#ifndef UART_INCL
#define UART_INCL

#include <io.h>

/** \file uart.h
In-line functions and prototypes for UART1 communications.
*/

void uartInit(void);
void uartTxString(char *);
unsigned int uartGets(char *s, unsigned int n);
int pchar1(int);  // function to send a char to uart 1 (note difference from gcc
manual)

// printf functions
/** Printf calls msp430-gcc library function uprintf.
uprintf calls pchar1 to print characters to uart1.
uprintf limitations include can't print floats.
\sa pchar1
*/
#define printf(format, ...) uprintf(pchar1, format, ## __VA_ARGS__)

/** Send null terminated string to UART.
Returns when last character written to UART.
Expands cr to cr lf.
\sa uartTxString */
#define puts    uartTxString

/** Send a single character to UART.
Returns when character written to UART.
\sa uartTx
*/
#define putc    uartTx

/** Get a character from UART.
Waits until character available.
\sa uartRx
*/
#define getc     uartRx


/** Inline function to send a character to uart1.
\param c - character to send
*/
static inline void uartTx(char c) {
      while(!(IFG2 & UTXIFG1));     // wait for tx buf empty
      U1TXBUF = c;
}

/** Inline function to receive a character from uart.
Waits for a character to be available in receive buffer of uart1, then
reads and returns the character.
*/
static inline unsigned int uartRx(void) {
      while(!(IFG2 & URXIFG1));     // wait for rx buf full
      return U1RXBUF;
}
```

```
/** Check receive buffer status. */

/** Inline function to check uart1 receive status.
\return true (non-zero) if char is ready.
*/
static inline unsigned int inkey(void) {
      return (IFG2 & URXIFG1);
}

#endif
```

uart.c

```
#include <io.h>
#include <uart.h>

/** \file uart.c
Functions to use uart1 for transmitting and receiving characters.
Simplest functions are inline, defined in uart.h
*/

/** Transmit a character to uart1 (function call version).
Expands newline to carriage return, newline. Returns 1 as required by msp430-gcc
library routine.
Provided for use by gcc uprintf funtion. Use uartTx otherwise.
\param c - character to send
\return 1
\sa uartTx
*/

int pchar1(int c) {
      if(c == '\n') uartTx('\r');
      uartTx(c);
      return 1;
}

/** Initialise uart1.
Uses SMCLK, assumed to be set at 8MHz.
Baud rate = 9600, 1 stop bit, 8 bits data.
*/
void uartInit(void) {
      U1CTL = SWRST + CHAR;
//    U1TCTL = SSEL_ACLK;     // use aclk (32khz)
      U1TCTL = SSEL_SMCLK;    // use smclk
      U1RCTL = 0;

//    U1BR1 = 0;        // 2400 baud, 32Khz MCLK
//    U1BR0 = 0xd;
//    U1MCTL = 0x6b;

//    U1BR1 = 01; // 9600 baud, 3.06MHz dco clock
//    U1BR0 = 0x3e;
//    U1MCTL = 0xbb;

      U1BR0 = 0x41;     // 9600 baud, 8MHz clock
      U1BR1 = 0x3;
      U1MCTL = 0x9;

      ME2 |= UTXE1;     // enable transmitter
      ME2 |= URXE1;     // enable receiver

      U1CTL &= ~SWRST;

      P3SEL |= (1 << 6) | (1 << 7); // P3 bits 6,7 used by uart
      P3DIR |= (1 << 6);            // P3 bit 6 tx
      P3DIR &= ~(1 << 7);           // P3 bit 7 rx
}

/** Send a null terminated string to uart1.
 Expands newline to carriage return, newline. Assumes uart1 already initialised.
\param p - Pointer to null terminated string.
*/
```

```
void uartTxString(char *p) {
	while (*p) {
		if(*p == '\n') uartTx('\r');
		uartTx(*p++);
	}
}


/** Get a line of characters from uart1.
Echoes chars, cr echoed as cr lf.
Up to n characters received, returns on carriage return, replacing carriage
return with 0.
\return Number of characters received, excluding terminating 0.
\param s - Pointer to char array to store characters received.
\param n - Maximum number of characters to receive, including terminating 0 (eg
size of array).
*/
unsigned int uartGets(char *s, unsigned int n) {
	unsigned int len=0;
	 while(n--){
		*s = uartRx();    // get a char
		uartTx(*s); // echo
		if(*s == '\r') {
			uartTx('\n');
			*s = 0;
			break;
		}

		else if(*s == '\b') {   // backspace
			if(len != 0) uartTxString(" \b");   // delete char on screen
			len--;
			s--;  // remove from buffer
		}

		else {
			len++;
			s++;
		}
	}
	return len;
}
```

Using gdb to erase flash

Try this:

In a command window (or shell):

```
msp430-gdbproxy --port=2000 msp430
```

This sets up gdbproxy to chat to the JTAG programmer. It should identify the microcontroller correctly. If it can't, and you've checked the obvious things (eg power, jtag lead) than maybe your hardware is broken. The following won't then work.

In another command window:

```
msp430-gdb
(gdb) target remote localhost:2000
(gdb) monitor erase all
(gdb) quit
```

On Windows XP, if you get a message about blocking a connection, choose unblock (gdb talks to gdbproxy over tcp, Windows offers to block unknown tcp connections).

I usually find this works when the stand alone jtag programmer software refuses to cooperate. Going back to the stand alone software after running through this, I find everything is ok again.

You may also be able to do the above using a gui like insight or ddd. I leave that as a user exercise. Once you've executed the above "target" command successfully in gdb then the gdb debugger is connected to your microcontroller through the JTAG port. Try "monitor help" to see what else you can do (programming flash etc). You can also use gdb in all its glory to debug your program (set break points, examine variables etc etc), but that's another story...