Table of Contents

mspgcc and C for the MSP430 FAQ

FAQ

Steve Underwood

This FAQ discusses a range of common issues related to programming the MSP430. More specifically, to programming the MSP430 in C. Most specifically to programming the MSP430 using the mspgcc toolchain.

Using C to program a small MCU produces slow bulky programs, doesn’t it?

You’ve been programming 8051s for too long. :-)

Seriously, an MCU like the MSP430 programs well in C. You can always beat the efficiency of any compiler if you try really hard in assembly language. However, you do have to make a real effort to get substantially better performance on good modern MCUs, not just the MSP430, when using a good C compiler.

I still have resort to some assembly language for things like interrupt routines, and controlling the lower power modes, right?

Wrong! mspgcc has simple extensions to the standard C language are provided to make interrupt service routines easy to write, without a single line of assembly language. Facilities are provided for controlling the MSP430’s lower power modes, too. See the mspgcc manual for details.

I will still need to use some assembly language somewhere, won’t I?

This is no more or less true for an MSP430 C program, than for a C program anywhere else. For example, a "Where is the first 1 in this word?" instruction exists on many large modern processors (x86, Alpha, etc.). You don’t have to use it, but it can really speed up certain types of code. You can only make use of it with assembly language programming. However, that can be as simple as a single line of embedded assembly language in the middle of a C routine. This is no different with the MSP430.

As an example, an area where you might choose to use assembly language in this way is to access the BCD facilities of the MSP430. Many people like to use BCD for a variety of things, in small embedded projects. Like most other processors, the MSP430 has decimal adjust instructions to make this straightforward. You can only gain access to those through assembly language. However, the assembly language need only be a few simple routines you create once and reuse. They can be simple functions, like:

```
void bin2bcd16(register uint8_t bcd[3], register uint16_t bin)
{
    register int i;
    register uint16_t decimal_0;
```

```
    register uint16_t decimal_1;

    i = 16;
    decimal_0 =
    decimal_1 = 0;
    __asm__ __volatile__(
        "1: \n"
        " rla   %[bin] \n"
        " dadd  %[decimal_0],%[decimal_0] \n"
        " dadd  %[decimal_1],%[decimal_1] \n"
        " dec   %[i] \n"
        " jnz   1b \n"
        " mov.b %[decimal_1],0(%[bcd]) \n"
        " mov.b %[decimal_0],2(%[bcd]) \n"
        " swpb  %[decimal_0] \n"
        " mov.b %[decimal_0],1(%[bcd]) \n"
        : [bcd] "+r"(bcd), [decimal_0] "+r"(decimal_0), [decimal_1] "+r"(decimal_1)
        : [bin] "r"(bin), [i] "r"(i));
}
```

which will convert a 16 bit binary value to a BCD string stored in a byte array. They could be simple inlined routines, for very arithmetic operations. Being inlined means there will be no call overhead, so it is clean, compact, and fast. For example:

```
static __inline__ uint16_t bcd_add16(register uint16_t bcd_a, register uint16_t bcd_b)
{
    __asm__(
 " clrc \n"
        " dadd %[bcd_b],%[bcd_a] \n"
        : [bcd_a] "+ro" (bcd_a)
        : [bcd_b] "g" (bcd_b));
    return bcd_a;
}
```

will add two 4 digit BCD values, stored as unsigned integers, and return the 4 digit BCD result, as an unsigned integer. Create a few routines like this, and you need not touch any assembly language for the rest of your project.

This thing is free, so it’s second rate. Right?

The GNU tools are used extensively in commercial work. It is naive to equate cost with value. The mspgcc port is currently stable, and being used to produce good quality commercial code. Other tools provide a more integrated GUI appearance, but do not offer much extra that is of real substance.

Where’s the IDE? Development tools are worthless without an IDE.

Whatever gave you that strange idea? Most IDEs offer a poor project manager, a poor editor and a poor debugger - poor tools, but really well integrated!

Excellent freely available editors, like SciTE (http://www.scintilla.org/SciTE.html), allow you to edit, compile, jump straight to the lines with syntax errors, and so on. GUI debugger front ends, such as Insight (http://sources.redhat.com/insight), GVD (http://libre.act-europe.fr/gvd/main.html), DDD

(http://www.gnu.org/software/ddd), etc., provide source level debug features as good or better than most other offerings.

Without a IDE project manager it is a pain to compile code. What do I do?

Learn about make. Most IDEs just provide a front end to a program called make. Versions of make exist on most platforms. The Windows installer for mspgcc includes a copy of the GNU version of make. On most other platforms, you probably already have a make program installed.

It takes little time to get used to the basics of make, and the make files for most projects are just a few lines long. It is a simple scripting tool, to define how your software is to be built. When you edit your source code, and run make it will rebuilt only what is necessary, on the basis of the files which have changed. The GNU documentation can be found at the GNU make home page (http://www.gnu.org/software/make/make.html).

But I know from the past that the GNU GCC compiler is not the world's most efficient. Right?

If you have used older versions of GCC you may well have formed that opinion. GCC has always been good for development, but older versions where not so efficient for runtime use. That changed a lot with the release of version, 3.0. Version 3.2.3 (the current basis for mspgcc) produces code that is very competitive in both size and speed. The mspgcc uses the facilities optimisation provided by GCC well, and produces very competitive code.

The mspgcc manual doesn't really describe the GNU tools. Why?

Extensive documentation, in multiple formats and languages, exists for the GNU tools. mspgcc is a port of those tools, and has all their standard features, plus a few special ones. Documenting all the standard things would be a waste of time, and probably confusing to those familiar with the GNU tools for other targets. The mspgcc documentation only seeks to document things which are special to mspgcc.

I've used the IAR compiler to program an MSP430, and it has a few nice features to access the special features of the MSP430. What does mspgcc have?

In general, mspgcc has similar facilities to the IAR tools. The syntax is a little different in some cases. For example, declaring a routine as an interrupt service routine requires round brackets, instead of square ones, around the vector name. However, just a little editing, and a few "#if defined(__GNUC__)"s will allow a program to compile with either the IAR or the mspgcc tools. In fact, just a little more work should allow programs to compile with the Quadravox or Rowley compilers, too.

The way embedded assembly language is handled is the biggest difference between source code for these compilers. This is likely to be the greatest source of code porting work, assuming you have resorted to using assembly language.

IAR provide a function to get a frame address of a function, allowing you more control. Does mspgcc have anything similar?

GCC provides the function "__builtin_frame_address(LEVEL)", but this will not help you much, unless R4 is being used as a frame pointer. With optimisation switched on, it will not. Of course, mspgcc has a real solution to this problem. Look in the Calls Definition section of the mspgcc manual for information on ".L__FrameSize". Of course, mspgcc also has things like "_BIC_SR_IRQ()" and "_BIS_SR_IRQ()", which simplify the most common reasons to work directly with the frame address.

When I link my files with msp430-ld it keeps saying "undefined reference to ‘__stop_progExec__’". What’s up?

You forgot to link a couple of things - startup code, interrupt vectors and libraries. The proper linker command looks something like:

```
$ msp430-ld -m msp430x[arch]
/usr/local/msp430/bin/../lib/gcc-lib/msp430/3.2.3/../../../../msp430/lib/crt430x[arch].o
-L/usr/local/msp430/bin/../lib/gcc-lib/msp430/3.2.3/msp1
-L/usr/local/msp430/bin/../lib/gcc-lib/msp430/3.2.3
-L/usr/local/msp430/bin/../lib/gcc-lib
-L/usr/local/msp430//lib/gcc-lib/msp430/3.2.3/msp1
-L/usr/local/msp430//lib/gcc-lib/msp430/3.2.3
-L/usr/local/msp430/bin/../lib/gcc-lib/msp430/3.2.3/../../../../msp430/lib/msp1
-L/usr/local/msp430/bin/../lib/gcc-lib/msp430/3.2.3/../../../../msp430/lib
-L/usr/local/msp430//lib/gcc-lib/msp430/3.2.3/../../../../msp430/lib/msp1
-L/usr/local/msp430//lib/gcc-lib/msp430/3.2.3/../../../../msp430/lib
-L/usr/local/msp430/bin/../lib/gcc-lib/msp430/3.2.3/../../..
[object file].o -lgcc -lc -lgcc
```

Of course, this is a pain to type, so don’t! The simple way is to use msp430-gcc as a front end to the linker. Then you can just type:

```
$ msp430-gcc -mmcu=[arch] -L[extra lib path] -l[extra lib] obj1.o obj2.o ... -o outfile
```

I need to place a chunk of data in its own page in flash memory, so I can erase and reprogram that page without

erasing anything else. How can I do that?

This is a typical requirement for an application with large calibration tables. There are two ways to deal with this. You can use a custom linker script, or special command line arguments to the linker. In either case you will want to tag the items to be placed in your special area, so the linker can identify them. the following illustrates how to do this:

```
#define __special_area__ __attribute__((section(".specialarea")))

void __special_area__ special_routine(int x, float y);

const __special_area__ int my_const = MY_VALUE;
```

The attribute "__special_area__" may be used with any other attributes in defining routines and data. At link time, the linker will associate all these with the section ".specialarea". Either your custom linker file, or command line parameters must define this section.

The easiest way to make a custom linker script is to start from the standard one for the device you are using. You can adjust the boundaries of the existing sections, and add new ones to lay out your code and data any way you want in memory. The linker script language is defined in the documentation for binutils. A compiler command such as:

```
$ msp430-gcc -Wl,--section-start -Wl,-T linker_script.x [other options and files]
```

Specifies that the customer linker script "linker_script.x" should be used in place of the default one.

For the simple definition of one special section, it may be easier to just define that section on the command line with a command such as:

```
$ msp430-gcc -Wl,--section-start -Wl,.newsection=0xd400 [other options and files]
```

which will place .newsection at address 0xd4000. Although this will place objects where you want them, it may not reliably fulfill your requirement. it will not ensure that nothing else goes in the relevant page of the flash memory, which would cause trouble if you need to erase that page. Defining your data to be the exact size of one or more flash memory pages, and forcing that data to start at the required page boundary will be reliable. The following example uses a union to ensure the overall size of your data block is well known:

```
#define __special_area__ __attribute__((section(".specialarea")))

const __special_area__ union
{
    char force[512];
    struct
    {
        int a;
        int b;
        char c[10];
        float d;
    };
} my_erasable_data_area;
```

Bear in mind that startup code (if linked) will start from the ROM’s first address. Its size is 0x40 (including _reset_vector__(), _unexpected_() and _unexpected_1_(), providing last two are not redefined in your code).

I want to jump from some arbitrary places to my exception handling code, to deal with fatal error conditions. Is it possible to do this with mspgcc?

Any standards compliant implementation of C, including mspgcc, supports this feature, with the routines setjmp() and longjmp(). The following example illustrates their use, with two destinations to jump to. You may define any number of destinations. You may jumps from anywhere, even from an interrupt service routine, and the stack will be properly cleaned up:

```
#include <setjmp.h>

jmp_buf __jmp_a;    /* make this global */
jmp_buf __jmp_b;    /* make this global */

void first_place_to_jump_from(void)
{
    ...
    if (something1)
        longjmp(__jmp_a, 1);
    if (something2)
        longjmp(__jmp_b, 123);
    ...
}

type second_place_to_jump_from(...)
{
    ...
    if (something3)
        longjmp(__jmp_a, 2);
    ...
}

void the_place_to_jump_to(void)
{
    int res;
    ...

    res = setjmp(__jmp_a);
    switch (res)
    {
    case 0:
        /* Not a jump. Just normal program flow. */
        res = setjmp(__jmp_b);
        switch (res)
        {
        case 0:
            /* Not a jump. Just normal program flow. */
            break;
        case 123:
            /* We jumped here from the first subroutine */
            break;
        }
        break;
    case 1:
        /* We jumped here from the first subroutine */
```

```
        break;
    case 2:
        /* We jumped here from second routine */
        break;
    }
    for (;;)
    {
        first_place_to_jump_from();
        second_place_to_jump_from();
    }
}
```

I keep loosing communication with the target processor, when using msp430-gdbproxy and a FET tool. Is mspgcc buggy?

No doubt there are bugs in mspgcc, but don't blame all your FET tools problems on them! People seem to have various problems with FET tools, for a variety of reasons.

- Communication between your PC and your target processor can be affected by ground noise. Ground noise explains some of the problems people see. When the target is self-powered (rather than powered from the FET tool), and you switch the target off and on, the FET tool sometimes seems unable to resynchronise with the target. How troublesome this is seems to depend on the particular board design.
- If you have an early FET TI tool, which has not been modified as per the TI note on this subject, modify it. That causes trouble for some people with fast PC (i.e. anything fairly recent).
- Do not extend the cable between the FET tool and your target. This is asking for trouble.
- Some people have reported that if their parallel port is set to anything other than SPP mode in the computer's BIOS, they get problems. Possibly working in EPP, or other enhanced modes, adversely affects the port's timing on these machines.
- Some people have trouble with the FET tools when using some models of IBM ThinkPad. The signal voltages seem lower than usual on these machines, and do not drive the FET tool reliably.

You may notice that none of these issues is specific to mspgcc. They tend to affect users of any of the MSP430 tools, under Windows, Linux or BSD.

Some assembly language instructions seems to keep changing. Whats up?

Many instructions can be looked at in more that one way. "add x to x" is really the same as an "arithmetic left shift x". The assembly language nmemonics supported by mspgcc follow TI's original assembly language. There are alternative names for some instructions, to allow assembly language code to be written in a more expressive manner. You can write

```
add R15,R15
```

or

```
rla R15
```

and the same binary code will be produced. However, when you look at a listing from, say, the debugger or the C compiler the same source representation is always used for any particular binary instruction. This often confuses people, and results in a number of false bug reports. It is actually the expected behaviour.

There is more information about these alternative names for the instuctions in the mspgcc manual.

I think I found a bug in mspgcc. What do I do?

You are in the right place to get started. Look through this FAQ first, and see if your issue is covered here. A large number of reported bugs turn out to be misunderstandings, and not bugs at all. If you feel sure that what you have found is a genuine bug, the right place to report it is the mspgcc mailing list at mspgcc-users@lists.sourceforge.net (mailto:mspgcc-users@lists.sourceforge.net). You can join this mailing list, or look through the message archive, by going to http://sourceforge.net/mail/?group_id=42303 (http://sourceforge.net/mail/?group_id=42303).

Remember mspgcc is built upon the standard GNU toolchain. If an internal error occurs many of the programs, you will receive a message telling you to report the problem to an e-mail address at gnu.org. Don't do this for now. When mspgcc is fully merged with the generic GNU toolchain those addresses will be the correct ones for bug reports. Right now reporting problems to those addresses will get you nowhere.

When submitting a bug report make sure you include:

- The OS you are using - e.g. Win XP, or RedHat 8.0. Just saying Windows is less helpful, as many issues are specific to particular versions of Windows.
- If you are using a prebuilt installer, the version of the installer you are using.
- If you built the mspgcc software yourself, the versions of the generic GNU tools you used, and the date on which you checked out code from the mspgcc CVS archive.
- Which MCU you are working with - e.g. MSP430F149
- The command line you used. A number of false bug reports are due to people using incorrect command line parameters. Tell us what you used and you may get a much quicker answer to your problem.
- If you are reporting a problem with the C compiler, a full preprocessed source listing. This can be created with a command like

  ```
  $ msp430-gcc [options] -E buggyfile.c > buggyfile.i
  ```

- If you are reporting a problem with the code generated by the C compiler, a full assembly language listing. This can be created with a command like

  ```
  $ msp430-gcc [options] -S buggyfile.c -dp
  ```

I am getting link errors about missing multiply routines. What is wrong?

The usual cause for this problem is that you specified a different "-mmcu=" option at the compile and link stages. Both the compiler and the linker need to know which device you are using. The compiler needs to know, so it can use hardware or software multiplies. The linker needs to know so it can choose the library for software

multiplication or hardware multiplication, as appropriate. Unless you have explicitly defined the memory map, the linker also needs to know the target device, so it can choose the appropriate default memory definitions.