mspgcc

A port of the GNU tools to the Texas Instruments MSP430 microcontrollers

Steve Underwood

mspgccA port of the GNU tools to the Texas Instruments MSP430 microcontrollers
by Steve Underwood

Table of Contents

Chapter 1. What is mspgcc?

mspgcc is a port of the GNU C and assembly language toolchain to the Texas Instruments MSP430 family of low-power microcontrollers. It is currently being used for production programs in C and assembly language. It is being be used on:

- Windows 98SE, Me, NT, 2000 and XP.
- Linux (A 2.4.x or later kernel is required for full debugger support).
- BSD Unix (??? required for full debugger support).

Parts of mspgcc have been merged into the official GNU versions of the toolchain. It is planned to merge the remainder at a suitable date.

The source code for the unmerged parts of mspgcc, and binary installers, may be obtained from the mspgcc web-site (http://mspgcc.sourceforge.net). Source code for the standard GNU tools may be obtained from the GNU (http://www.gnu.org) website, or one their mirrors.

There is an active mailing list available for mspgcc users at the mspgcc web-site (http://mspgcc.sourceforge.net) where users can get help with any problems they may find using mspgcc.

The GNU Binutils

The GNU assembler, linker and various support utilities are collectively known as 'binutils'. Beginning with version 2.14, the official releases of binutils contain support for the MSP430 processors.

The GNU GCC C Compiler

The mspgcc port of the GNU C compiler is currently based on version 3.2.3 of GNU GCC. It supports all the current variants of the MSP430 processor, and comes with a full set of header files for the processors, and a basic 'libc' library. Signed and unsigned integers of 8, 16, 32, and 64 bit lengths are supported. Floating point is supported, but only for single precision floating values - no double precision. Currently only C is supported. However, support for C++, Fortran and other languages supported by *GCC* might be added. Currently the mspgcc port fo the C compiler is stable, and suitable for production use. At the time of writing it is necessary to download a set of patches for the official GNU C compiler from the mspgcc web-site. When practical, these will be merged into the official GNU GCC releases.

The GNU GDB and Insight debuggers

The mspgcc port of the GNU GDB debugger is currently based on version 5.1.1. This can be used with the Texas Instruments JTAG interface when used with an additional program called msp430-gdbproxy, and a TI FET tool. GDB is a command line tool. Various graphical front ends are available for it. A merged graphical front end, called Insight, is also available.

Generally any of the available GUI front ends for GDB will work with *msp430-gdb* on Linux, or other Unix like platforms. However, many of these front ends do not function, or do not function well, on Windows machines. Some require (e.g. GVD) require a Windows NT based machine (i.e. Windows NT, 2000 or XP machines), and

will not work on a Windows 98 or Me machine. Choosing and installing front ends other than Insight is left as an excercise for the reader.

Extras

In addition to the GNU tools, a few extra programs are available from the mspgcc web-site. These include software to program the flash memory of an MSP430 using the bootstrap loader (BSL) built into the MSP430 flash devices.

Chapter 2. Installing mspgcc

Windows installation

If you are running Windows 98, Me, NT, 2000 or XP you can download a one step installer program for mspgcc from the mspgcc web-site download page (http://sourceforge.net/project/showfiles.php?group_id=42303). Download it. Run it. Choose the default installation directory, unless you have a very good reason to do otherwise. In most cases the default option to install everything is the right choice. Once component - giveio - will automatically not install on Windows 98 or Me machines, as it is not needed. It provides raw access to the parallel port on machines using Windows NT, 2000 or XP. If you are running Windows 98 or Me you will be prompted to add the installation path for the mspgcc programs to your PATH environment variable. For Windows NT, 2000 or XP this will be done automatically for you.

The Windows mspgcc installer uses the library cygwin1.dll, which is part of the Cygwin package. You may only have one copy of this on your machine. The installer checks if this file exists, and will not install its own copy if it does. If you already have Cygwin installed, you may need to check it is up to date. Older versions of cygwin1.dll may not function correctly with mspgcc.

If you wish to debug MSP430 programs using the JTAG interface you will use a parallel port on your computer to communicate with the JTAG tool. If other software is using this interface (e.g. a print spooler) you may have trouble. If you have any diffculty communicating with the JTAG interface, check for conflicting uses of the parallel port.

RedHat Linux installation

If you are running RedHat Linux (7.1 or later) RPMs for some parts of mspgcc are currently available from the mspgcc web-site, and a complete set should be available soon.

If you wish to debug MSP430 programs using the JTAG interface you will use a parallel port on your computer to communicate with the JTAG tool. If other software is using this interface (e.g. a print spooler) you may have trouble. If you have any diffculty communicating with the JTAG interface, check for conflicting uses of the parallel port.

Installation on other platforms

You can build mspgcc from source code for many other platforms. Most installations of Linux with kernels greater than 2.4.0 can fully support mspgcc. Some versions of BSD Unix can too. On other Unix like platforms everything except the JTAG interface should work OK. The JTAG interface requires raw access to a parallel port. Drivers for this do not currently exist for these platforms.

If you are running a version of the Linux kernel earlier than 2.4.0 you will not be able to drive the JTAG interface. The driver needed for raw access to the parallel port does not exist for these kernels (although someone is currently working on this). However, you can still use the rest of the mspgcc tools to write and compile code, and use the bootstrap loader (BSL) to program devices. The bootstrap loader requires nothing more than a standard serial port.

On platforms without parallel port, it may be possible to use the serial bootstrap loader (BSL). One implementation of a downloader can be found on the mspgcc web-site (http://mspgcc.sourceforge.net) look for pyBSL.

Chapter 3. An introduction to the TI MSP430 low-power microcontrollers

Overview

The MSP430 is a very clean 16-bit byte-addressed processor with a 64K unified address space, and memory-mapped peripherals. The current family includes a variety of on-chip peripherals, and ranges from a 20-pin package with 1K of ROM and 128 bytes of RAM to 100-pin packages with 60K of ROM and 2K of RAM. Devices with greater RAM and ROM, and additional peripheral blocks are in development.

The MSP430 excels where low power consumption is important. Many applications, such as water meters, are currently achieving more than 10 years operation from a single button cell battery. If low power is not critical, well, the MSP430 is a nice elegant device to use, anyway. It programs very well in C, making assembly language programming unnecessary. There is no memory bank switching to make the compiler's life difficult; it uses normal RAM for its stack; it has a clean 16 bit instruction set. In fact, it is somewhat like an ordinary desktop RISC processor, but requires very little power.

The memory map

All current MSP430s share a common memory map. The amount of each type of memory varies with the device, but the overall layout is common.

The main ROM is always at the highest addresses. In the 60K version it extends from address 0x1100 to 0xFFFF (see below for what happens between 0x1000 and 0x10FF). Some devices use mask programmed ROM or EPROM. All the more recent parts are available with flash (electrically erasable) memory, and have a mask programmed option for high volume users. If the device has flash memory, it is erasable in 512 byte pages. The device can self-program its own flash memory, although this imposes some constraints on the supply voltage.

At the low end of memory is a 512 byte space for the memory-mapped peripherals. The first 256 bytes of this are on an 8-bit bus, and can only be accessed 8 bits at a time. The second 256 bytes are on a 16-bit bus, and can only be accessed 16 bits at a time.

RAM begins at address 0x200. If there is 2K of RAM, it extends from address 0x0200 to 0x9FF.

Processors with flash memory have a 1K serial bootloader ROM at addresses 0x0C00 to 0x0FFF. This is unalterable, masked, ROM. It contains a factory set program to erase and reprogram the on board flash memory. (see later for other programming and debug options).

Processors with flash memory also have an additional 128 or 256 bytes of flash memory between addresses 0x1000 and 0x107F or 0x10FF. The only real difference between this and the main flash memory is that this is erasable in 128 byte pages. This makes it more suitable for efficiently handling configuration data.

The register set

The processor has 16 16-bit registers, although only 12 of them are truly general purpose. The first four have dedicated uses:

- r0 (aka PC) is the program counter. You can jump by assigning to r0, and immediate constants are fetched

from the instruction stream using the post-increment addressing mode on r0. The PC is always even.

- r1 (aka SP) is the stack pointer. This is used by call and push instructions, and by interrupt handling. There is only one stack pointer; the MSP430 doesn't have anything resembling a supervisor mode. The stack pointer is always even; It is unclear if the LSB is even implemented.
- r2 (aka SR) is the status register. Its bits are assigned as follows:

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Reserved							V	SCG1	SCG0	OS-COFF	CPUOFF	GIE	N	Z	GC

SCG (system clock generator), OSCOFF (oscillator off), and CPUOFF are used to control the various low-power modes.

GIE is the global interrupt enable. Turning off this bit masks interrupts. (NOTE: it may be delayed by 1 cycle, so an interrupt may be taken after the instruction after GIE is cleared. Add a NOP or clear GIE one instruction earlier than your real "critical section".)

N, Z, C and V are the usual processor status bits, set as a side effect to instruction execution. If r2 is specified as a destination, the explicitly written bits override the side effects. An instruction sets all 4 bits, or none of them. Logical instructions set C to the opposite of Z (C is set if the result is NOT zero), and clear V to 0.

C is a "carry" bit as opposed to a "borrow" bit when subtracting. That is, subtract with carry of A-B computes A + ~B + Carry. (~ is the C "not" or "bitwise invert" operator.)

Note that the basic move instruction does NOT set these bits (unless it's a move to r2).

- r3 is hardwired to 0. If specified as a source, its value is 0. If specified as a destination, the value is discarded.

r2 and r3 have no use as pointers. When specified in the context of a pointer they provide an alternate function - common constant values. This is one of the important features of the MSP430 instruction set, allowing it to achieve a high level of code density, and a flexible instruction set. These constant registers can provide the numbers -1, 1, 2, 4 or 8. So, for example, the "clr x" is actually emulated by the instruction "mov #0,x". The constant "0" is taken from the constant register r3. The assembler understands both "clr x" and "mov #0,x", and produces the same code for either. Many RISC and RISC like architectures suffer poor code density. The constant registers allow the MSP430 to achieve a very competitive code density. They also make the code faster, as less program memory read cycles are needed. See below for the actual encoding used to select a particular constant.

Note that some assemblers for the MSP430 allow the use of the alternate names "PC" for "r0", "SP" for "r1", and "SR" for "r2". GNU msp430 binutils does not understand these alternate names. You must use "r0", "r1" or "r2".

The available addressing modes

MSP430 instructions have at most two operands, a source and a destination.

All instructions are 16 bits long, followed by at most two optional offsets words, one for each of the source and the destination.

The source operand (or the only operand of one-operand instructions) is specified with 2 addressing mode bits and 4 register select bits:

00 nnnn	Rn	Register direct
01 nnnn	offset(Rn)	Register indexed

10 nnnn	@Rn	Register indirect
11 nnnn	@Rn+	Register indirect with post-increment

The only addressing mode that uses an extension word is the indexed mode. A 16-bit offset can reach anywhere in the address space.

The destination operand in a two-operand instruction has only one addressing mode bit, which selects either register direct or indexed. Register indirect can obviously be faked up with a zero index.

Operand addresses are computed in a simple, sequential way. The C statement

```
*p++ *= 2;
```

can be implemented as

```
add @Rn+,-2(Rn)
```

because the source operand is computed completely (including the register post-increment) before the destination is computed.

When r0 (the program counter) is used as a base address, indexed mode provides PC-relative addressing. This is, in fact, the usual way that TI's MSP430 assembler accesses operands when a label is referred to.

@r0 just specifies the following instruction word in ROM, but @r0+ specifies that word and skips over it. In other word, an immediate constant! You can just write #1234 and the assembler will specify the addressing mode properly.

r1, the stack pointer, can be used with any addressing mode, but @r1+ always increments by 2 bytes, even on a byte access.

When r2 (the status register) or r3 (the zero register) are specified, the addressing mode bits are decoded specially:

00 0010	r2	Normal access
01 0010	&<location>	Absolute addressing. The extension word is used as the address directly. The leading & is TI's way of indicating that the usual PC-relative addressing should not be used.
10 0010	#4	This encoding specifies the immediate constant 4.
11 0010	#8	This encoding specifies the immediate constant 8.
00 0011	#0	This encoding specifies the immediate constant 0.
01 0011	#1	This encoding specifies the immediate constant 1.
10 0011	#2	This encoding specifies the immediate constant 2.
11 0011	#-1	This specifies the all-bits-set constant, -1.

Byte and word issues

The MSP430 is byte-addressed, and little-endian. Word operands must be located at even addresses. Most instructions have a byte/word bit, which selects the operand size. Appending ".b" to an instruction makes it a byte operation. Appending ".w" to an instruction, to make it a word operation, is also legal. However, since it is also the default behaviour, if you add nothing, it is generally omitted. A byte instruction with a register destination clears the high 8 bits of the register to 0. Thus, the following would clear the top byte of the register, leaving the lower byte unchanged:

```
mov.b Rn,Rn
```

The on-chip peripherals are divided into an 8-bit bank and a 16-bit bank. The 8-bit peripherals must only be accessed using 8-bit instructions; using a 16-bit access produces garbage in the high byte. The 16-bit peripherals must only be accessed at even addresses. Byte accesses to even addresses are legal, but not usually useful.

The processor's behaviour when a word is accessed at an odd location is poorly documented. In all current processors the lower bit is just silently ignored. The effect is, therefore, the same as specifying an address which is one less.

It should be noted that the the byte and word addressing behaviour of the MSP430 prevents the processor supporting strict compliance with the standard C language. In standard C everything should be copiable, by copying at the byte level. This usually has little impact on the types of embedded program for which the MSP430 is typically used. However, it can sometimes catch you out!

The instruction set

All instructions are 16 bits long, and there are only three instruction formats:

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	0	0	1	0	0	Opcode			B/W	Ad		Dest reg			
0	0	1	Condition			PC offset (10 bit)									
Opcode				Source reg				Ad	B/W	As		Dest reg			

As and *Ad* are the source and destination addressing modes. *B/W* is a bit that is set to 1 for byte instructions. 2-operand opcodes begin at 0100 = 4.

As you can see, there are at most 8+8+12 = 28 instructions to keep track of, which is nice and simple.

One-operand instructions:

000	RRC(.B)	9-bit rotate right through carry. C->msbit->...->lsbit->C. Clear the carry bit beforehand to do a logical right shift.
001	SWPB	Swap 8-bit register halves. No byte form.
010	RRA(.B)	Badly named, this is an 8-bit arithmetic right shift.

011	SXT	Sign extend 8 bits to 16. No byte form.
100	PUSH(.B)	Push operand on stack. Push byte decrements SP by 2. CPU BUG: PUSH #4 and PUSH #8 do not work when the short encoding using @r2 and @r2+ is used. The workaround, to use a 16-bit immediate, is trivial, so TI do not plan to fix this bug.
101	CALL	Fetch operand, push PC, then assign operand value to PC. Note the immediate form is the most commonly used. There is no easy way to perform a PC-relative call; the PC-relative addressing mode fetches a word and uses it as an absolute address. This has no byte form.
110	RETI	Pop SP, then pop PC. Note that because flags like CPUOFF are in the stored status register, the CPU will normally return to the low-power mode it was previously in. This can be changed by adjusting the SR value stored on the stack before invoking RETI (see below). The operand field is unused.
111	Not used	The MSP430 actually only has 27 instructions.

The status flags are set by RRA, RRC, SXT, and RETI.

The status flags are NOT set by PUSH, SWPB, and CALL.

Relative jumps. These are all PC-relative jumps, adding twice the sign-extended offset to the PC, for a jump range of -1024 to +1022.

000	JNE/JNZ	Jump if Z==0 (if !=)
001	JEQ/Z	Jump if Z==1 (if ==)
010	JNC/JLO	Jump if C==0 (if unsigned <)
011	JC/JHS	Jump if C==1 (if unsigned >=)
100	JN	Jump if N==1 Note there is no "JP" if N==0!
101	JGE	Jump if N==V (if signed >=)
110	JL	Jump if N!=V (if signed <)
111	JMP	Jump unconditionally

Two-operand instructions. These basically perform *dest = src op dest* operations. However, MOV doesn't fetch the destination, and CMP and BIT do not write to the destination. All are valid in their 8 and 16 bit forms.

Operands are written in the order *src,dest*.

0100	MOV src,dest	dest = src	The status flags are NOT set.
0101	ADD src,dest	dest += src	
0110	ADDC src,dest	dest += src + C	
0111	SUBC src,dest	dest += ~src + C	
1001	SUB src,dest	dest -= src	Implemented as dest += ~src + 1.
1001	CMP src,dest	dest - src	Sets status only; the destination is not written.
1010	DADD src,dest	dest += src + C, BCD.	
1011	BIT src,dest	dest & src	Sets status only; the destination is not written.
1100	BIC src,dest	dest &= ~src	The status flags are NOT set.
1101	BIS src,dest	dest |= src	The status flags are NOT set.
1110	XOR src,dest	dest ^= src	
1111	AND src,dest	dest &=- src	

There are a number of zero- and one-operand pseudo-operations that can be built from these two-operand forms. These are usually referred to as "emulated" instructions:

NOP	MOV r3,r3	Any register from r3 to r15 would do the same thing.
POP dst	MOV @SP+,dst	

Note that other forms of a NOP instruction can be constructed as emulated instructions, which take different numbers of cycles to execute. These can sometimes be useful in constructing accurate timing patterns in software.

Branch and return can be done by moving to PC (r0):

BR dst	MOV dst,PC
RET	MOV @SP+,PC

The constants were chosen to make status register (r2) twiddling efficient:

CLRC	BIC #1,SR
SETC	BIS #1,SR

CLRZ	BIC #2,SR
SETZ	BIS #2,SR
CLRN	BIC #4,SR
SETN	BIS #4,SR
DINT	BIC #8,SR
EINT	BIC #8,SR

Shift and rotate left is done with add:

RLA(.B) dst	ADD(.B) dst,dst
RLC(.B) dst	ADDC(.B) dst,dst

Some common one-operand instructions:

INV(.B) dst	XOR(.B) #-1,dst
CLR(.B) dst	MOV(.B) #0,dst
TST(.B) dst	CMP(.B) #0,dst

Increment and decrement (by one or two):

DEC(.B) dst	SUB(.B) #1,dst
DECD(.B) dst	SUB(.B) #2,dst
INC(.B) dst	ADD(.B) #1,dst
INCD(.B) dst	ADD(.B) #2,dst

Adding and subtracting only the carry bit:

ADC(.B) dst	ADDC(.B) #0,dst
DADC(.B) dst	DADD(.B) #0,dst
SBC(.B) dst	SUBC(.B) #0,dst

Instruction timing

Generally, instructions take 1 cycle per word of memory accessed.

Thus, start with 1 cycle for the instruction itself. Then add 1 cycle for a memory source, 2 cycles for a memory destination, and one additional cycle per offset word.

Note that in two-operand instructions, memory destinations require an offset word, so they cost a total of 3

cycles.

This holds even for instructions (MOV, CMP and BIT) that only access the destination once.

Short immediate constants (using r2 or r3) count as register operands for instruction timing purposes.

Exceptions to this rule are:

- A 2-operand instruction which writes to PC (r0) takes an extra cycle if it's only one word long (i.e. source not indexed).
- Jumps take 2 cycles, whether taken or not.
- PUSH, CALL and RETI are special:

PUSH Rn	3 cycles
PUSH @Rn, @Rn+, #x	4 cycles
PUSH offset(Rn)	5 cycles
CALL Rn	4 cycles
CALL @Rn	4 cycles
CALL @Rn+, #x	5 cycles
CALL offset(Rn)	5 cycles
RETI	5 cycles

Other CPU operations take following times to execute:

Interrupt	6 cycles
Reset	4 cycles

Interrupts

The MSP430 supports 16 exception vectors, from 0xFFE0 to 0xFFFF. There are 14 maskable interrupts which are assigned to peripherals in a model-dependent way. The first 14 can be masked by clearing the GIE bit in the status register. The last two are non-maskable: 0xFFFC is the NMI vector, and 0xFFFE is the reset vector.

Actually, all of the "non-maskable" interrupt sources are maskable, just not with the GIE bit. They are:

- The RST/NMI pin can be configured to send an NMI rather than reset the processor when pulled low.
- Flash access violation.
- An oscillator fault occurs. The more recent MSP430 devices use a on chip system clock called the FLL - frequency locked loop. This can be programmed to provide a range of core clock frequencies which are phase locked to an external crystal (usually a 32kHz watch type crystal). If the frequency adjustment reaches the extreme limits, and the loop cannot lock, an oscillator fault is declared.

  Other MSP430 devices use a different oscillator module. Here the oscillator fault flag is set when one of the oscillators does not oscillate. The CPU should be using an alternate oscillator if this happens.

Handling an interrupt (other than RESET) consists of:

- Push PC on stack.
- Push SR on stack.
- Choose the highest priority interrupt to service.
- If the interrupt has only one source, reset the interrupt request bit. If there are multiple possible sources, leave them for software to poll.
- If this is an NMI, clear enable bits for the three NMI sources mentioned above.
- Clear the SR (except for SCG0), disabling interrupts and power-saving.
- Fetch the interrupt vector into the PC
- Start executing the interrupt handler

A reset is similar, but doesn't save any state.

You can nest interrupt handlers by disabling the current source and setting the GIE bit back to 1.

Note that there are no exceptions internal to the processor such as divide by zero or address error. You can cause exceptions or reset by writing to peripherals.

The hardware multiplier

Some MSP430 processors have, as a memory-mapped peripheral, a hardware 16x16->32 multiply/accumulate unit. This is accessed via eight 16-bit registers from 0x0130 to 0x013F.

Writing the first operand specifies the operation type depending on the address used:

- 0x0130 - MPY unsigned multiply.
- 0x0132 - MPYS signed multiply.
- 0x0134 - MAC unsigned multiply-accumulate.
- 0x0136 - MACS signed multiply-accumulate.

Writing the second operand to 0x0138 starts the operation. The product is available in 0x013A(SumLo), 0x013C(SumHi) and 0x013E(SumExt) with only 2 cycles of latency. Thus, you can fetch the result with the next instruction if it's an indexed or absolute addressing mode.

If you use a register indirect or post-increment mode, you need to insert a nop (or something) between writing the second operand and reading the results.

The accumulator (SumLo and SumHi) is only 32 bits. SumExt is set to the carry (0 or 1) of the 32+32-bit sum in a MAC operation, but the old value of SumExt is not used.

In MPYS and MACS, SumExt is just the sign-extension of SumHi (0 or -1), which is not tremendously useful.

While all registers can be read back, the operation specified by the first operand's address is not recoverable by an interrupt handler. Thus, it is not possible to context-switch the multiplier unless you add some sort of wrapper software (locking or shadow registers) around it.

All registers are read/write except:

- The first four are actually aliases for one register, so they always read the same value.
- SumExt is not writable.

The multiplier is one 16-bit peripheral where a byte write might make sense. A byte write is zero-extended to 16 bits, which allows 8-bit unsigned operands to be used naturally.

Once the first operand has been written, multiple second operands can be written without changing it. For example, when evaluating a polynomial by Horner's rule

```
a + b*x + c*x^2 + d*x^3 = (((d * x + c) * x) + b) * x + a
```

Then x can be written to the first operand register just once.

Low power modes

Low power operation is a key feature of the MSP430. Its design gives very low leakage, and it operates from a single supply rail. This gives an extremely low current drain when the processor is in standby mode. Several low power modes are supported, which balance the needs of different applications. As the number of the LPM mode number rises, the number of things disabled on the chip also rises:

- LPM0 - The CPU is disabled.
- LPM1 - The loop control for the fast clock (MCLK) is also disabled.
- LPM2 - The fast clock (MCLK) is also disabled.
- LPM3 - The DCO oscillator and its DC generator are also disabled.
- LPM4 - The crystal oscillator is also disabled.

As the LPM mode rises power consumption decreases, but the time needed to wake up increases. Note, however, that the MSP430's design keeps even the worst case wakeup time fairly low. For example, the parts which use the FLL system clock module need only a few microseconds to get the FLL locked after waking up.

The MSP430 is switched into a low power mode by altering bits in the status register. Typically processing within an interrupt routine will determine when the processor needs to change from a low power mode to normal operation, and alters those same status register bits to achieve that. It does this by directly modifying the memory location where the processor's status register was pushed onto the stack at the start of the interrupt. When the interrupt routine returns, using the RETI instruction, the altered status register value is loaded into the processor status register, and the processor continues operation in the newly selected mode. The C language tools support an easy method to handle this.

Programming the flash memory

An MSP430s using flash ROM can program themselves using software, but there is an initial chicken-and-egg problem getting the programming software into the chip in the first instance.

Fortunately, there are two ways you can do this on a "bare" MSP430:

- Via the JTAG interface

- Via the bootstrap loader

JTAG is a JEDEC-standard in-circuit testing interface. It uses 4 pins: mode, clock, data in and data out. It's basically a big shift register. You can chain devices together by connecting the in and out pins to make one giant whole-board shift register, and take over the I/O pins for various sorts of board testing.

There are also a few opcodes reserved to for manufacturer extensions, which TI uses for remote access and debugging purposes.

All MSP430 devices have a JTAG port, although on the 20- and 28-pin parts, the pins are multiplexed with normal I/O pins and only a dedicated "test" pin is needed, to enable the JTAG functionality.

The full capabilities of TI's extensions to the JTAG port are rather extensive, and include stopping and single-stepping the processor. See TI's app. note slaa149 for details.

But, in particular, you can perform arbitrary memory accesses, and thereby program the flash ROM.

This is quick, if you can do all the complex wiggling of the JTAG control lines fast enough, but that's rather complex piece of work.

An alternative is a bootstrap loader that is included on all flash MSP430 processors and uses standard 9600 baud asynchronous communications. For those parts with 2k of RAM you can download a replacement BSL and use 38400 baud. Downloading this at 9600 baud, and then flashing at 38400 baud is faster for programs larger than about 10k bytes.

This is also invoked by special wiggling of the TEST input while RESET is active. (For parts with a dedicated JTAG interface, and thus no TEST pin, TCK is used instead.)

All this requires is some level-shifters and a serial port. See TI application notes slaa089a and slaa096b for details.

Decoding part numbers

What does something like MSP430F1121 mean?

The letter indicates the type of ROM on board:

- C - Mask ROM. This is programmed at manufacturing time.
- E - UV-EPROM. This comes in a windowed package and is erasable with UV. This requires a special high voltage supply to program.
- F - Flash ROM. This is electrically erasable, and can be programmed with normal operating voltages.
- P - One-time programmable. This is an E part in a cheaper windowless package. Once programmed, it cannot be erased.

Note that only the original 3xx series parts use UV-EPROM. Everything after that uses Flash ROM. Programming the EPROM parts is done over the JTAG port.

The first digit after the letter is the overall family: 1, 3 or 4. They are roughly in increasing order of capability, but there's a lot of range. Parts are generally upward-compatible within a family.

Basically, 1xx parts don't have an LCD controller, while all the 3xx parts do. Both families have models with ADCs; the 11x2 parts have a 10-bit ADC, while others have a 12-bit ADC. Some parts have hardware UARTS, although any of them can bit-bang it.

Initially, the 1xx parts were small 20- and 28-pin parts, and the 3xx parts were 56 or 64 pins. However, the 1xx parts have grown up and the 13x and higher packages come in 64-pin packages, while the 3xx parts range from a 48-pin 31xS subset to 100 pins.

The second digit is the device within a family. Again, generally higher numbers are more capable, but it varies a lot. As a general overview:

- 11x: 20-pin parts. 11x1 adds comparator, 11x2 adds ADC10.
- 12x: 28-pin parts, like 11x1 but with more I/O and USART.
- 13x: 64-pin parts, adding lots more I/O, another timer, USART, and ADC12.
- 14x: 64-pin parts, like 13x but with a hardware multiplier and a second USART.
- 15x: Like 13x, but adding 2xDAC12, 3xDMA, brownout reset and I2C.
- 16x: Like 14x, but adding 2xDAC12, 3xDMA, brownout reset and I2C.
- 31x: 56 pins, basic device with I/O, timers, comparator/timer, LCD driver.
- 32x: 64 pins, like 31x but with ADC12+2 (can be kludged to do 14 bits).
- 33x: 100 pins, like 31x with more I/O and LCD, multiplier, and USART.
- 41x: 64 pins, basic device with I/O, timers, comparator, LCD driver.
- 43x: 80 or 100 pins, more LCD, ADC12, second timer, USART, second crystal oscillator.
- 44x: 100 pins, like 43x but with a hardware multiplier, second USART and expanded timer.

A fourth digit, if present, is a sub-version number. A ’1121 is a ’112 with a little bit extra (an analog comparator for software ADC). A ’1122 is a ’112 with a 10-bit ADC. (Exception: the 161x parts.)

The third digit encodes the amount of memory on the chip:

- xx0: 1K ROM, 128 RAM
- xx1: 2K ROM, 128 RAM
- xx2: 4K ROM, 256 RAM
- xx3: 8K ROM, 256 RAM
- xx4: 12K ROM, 512 RAM
- xx5: 16K ROM, 512 RAM
- xx6: 24K ROM, 1024 RAM
- xx7: 32K ROM, 1024 RAM
- xx8: 48K ROM, 2048 RAM
- xx9: 60K ROM, 2048 RAM

The 16x series adds:

- xx10: 32K ROM, 5K RAM
- xx11: 48K ROM, 10K RAM

Note the 161x numbers are an exception to the usual 4th digit rule.

Chapter 4. MSP430 specific extensions to the GNU toolchain

This section describes the MSP430-specific extensions to the GNU toolset. You should refer to the GNU documentation for information about the standard features of the GNU tools.

Compiler options

The compiler recognises the following MSP430 specific command line parameters:

-mmcu=	Specify the MCU name
-mno-volatile-workaround	Do not perform a volatile workaround for bitwise operations.
-mno-stack-init	Do not initialise the stack as *main()* starts.
-minit-stack=	Specify the initial stack address.
-mendup-at=	Jump to the specified routine at the end of *main()*.
-mforce-hwmul	Force use of a hardware multiplier.
-mdisable-hwmul	Do not use the hardware multiplier.
-minline-hwmul	Issue inline code for 32-bit integer operations for devices with a hardware multiplier.
-mnoint-hwmul	Do not disable and enable interrupts around hardware multiplier operations. This makes multiplication faster when you are certain no hardware multiplier operations will occur at deeper interrupt levels.
-mcall-shifts	Use subroutine calls for shift operations. This may save some space for shift intensive applications.

The following MCU names are currently recognised for the “-mmcu” parameter:

msp1	msp2			
msp430x110	msp430x112			
msp430x1101				
msp430x1111	msp430x1121			
msp430x122	msp430x123			
msp430x1222	msp430x1232			
msp430x133	msp430x135			
msp430x1331	msp430x1351			
msp430x147	msp430x148	msp430x149		
msp430x1471	msp430x1481	msp430x1491		
msp430x155	msp430x156	msp430x157		
msp430x167	msp430x168	msp430x169	msp430x1610	msp430x1611
msp430x311	msp430x312	msp430x313	msp430x314	msp430x315
msp430x323	msp430x325	msp430x336	msp430x337	

msp430x412	msp430x413			
msp430xE423	msp430xE425	msp430xE427		
msp430xW423	msp430xW425	msp430xW427		
msp430x435	msp430x436	msp430x437		
msp430x447	msp430x448	msp430x449		

“msp1” means an MCU without a hardware multiplier. “msp2” means an MCU with a hardware multiplier. These can be useful to make the compiler generate the correct code for a new device, before it is fully supported.

Compiler defined symbols

The compiler defines some symbols, so the header files and source code can easily behave in an MCU dependant manner. These are:

- MSP430
- __MSP430__
- __MSP430_xxx__, where xxx is replaced by the number of the MCU variant (e.g. __MSP430_149__ is defined for the msp430x149).

The mspgcc header files

The include path for the standard header files is automatically defined by the compiler. The header file "<io.h>" is usually included at the start of all mspgcc source files. This defines all TI’s standard definitions for the MCU variant being used, along with some mspgcc specific extensions.

If you have used other software tools for the MSP430, you will find mspgcc’s header file handling a little different and a little simpler to use. There is a header file for each peripheral module type. Where variants of a module exist (e.g. the UART exists in versions with and without I2C facilities), switches are used to select the appropriate defines. There is a customised header file for each MCU group (e.g. msp430x44x.h for the msp430x447, msp430x448 and msp430x449). <io.h> inlcudes the appropriate header file, based on the command line "-mmcu" parameter. If you program for a number of different MSP430 parts, nothing needs to be changed in your source code to rebuild it for a different chip.

Function attributes

A number of special function attributes are supported, to provide access to the special features of the MSP430, and the special needs of embedded programming.

reserve(x)	When applied to main(), this reserves “x” bytes of RAM above the stack. This cannot be used with C++ (if C++ is supported later on).

interrupt(x)	Make the function an interrupt service routine for interrupt “x”.
signal	Make an interrupt service routine allow further nested interrupts.
wakeup	When applied to an interrupt service routine, wake the processor from any low power state as the routine exits. When applied to other routines, this attribute is silently ignored.
naked	Do not generate a prologue or epilogue for the function.
critical	Disable interrupts on entry, and restore the previous interrupt state on exit.
reentrant	Disable interrupts on entry, and always enable them on exit.
saveprologue	Use a subroutine for the function prologue, to save memory.
noint_hwmul	Supress the generation of disable and enable interrupt instructions around hardware multiplier code.

The syntax for using these attributes is “__attribute__((attribute_name(x)))”, where “attribute_name” is replaced by the required attribute name, and “x” is replaced by a parameter to that attribute. For attributes which do not accept parameters, “(x)” should be omitted.

The header file “signal.h” defines a number of alternative names for some of these attributes. In the sections below, some of these alternative names are used in more detailed discussions of the use of these attributes.

Writing interrupt service routines

mspgcc allows interrupt service routines to be written efficiently in C. To access the interrupt features of mspgcc the header file

#include <signal.h>

should be included in any source files where interrupt service routines are defined.

To make a routine an interrupt service routine, define it as follows:

```
interrupt (INTERRUPT_VECTOR) IntServiceRoutine(void)
{
    /* Any normal C code */
}
```

where “INTERRUPT_VECTOR” is replaced with the name of the actual vector to be serviced. Definitions for these may be found in the header files. The generated code will save any registers used within the interrupt routine, and use the "RETI", rather than the usual "RET" instruction, to exit from the routine. The vector table will automatically point to the routine. Further interrupt related attributes are also recognised:

```
interrupt (INTERRUPT_VECTOR) [wakeup, enablenested] IntServiceRoutine(void)
{
    /* Any normal C code */
}
```

The “wakeup” attribute makes the compiler alway force exit from any low power modes that may be in force at exit from the routine. See later for ways to gain greater control of the lower power modes. “enablenested” causes an interrupt enable instruction to be inserted before the function prologue. This allows other higher priority interrupts to be serviced while handling the current one. Use this feature with care if you use it in conjunction with “wakeup”!

Although interrupt service routines normally accept no arguments, it is possible to define a function with the “interrupt” attribute and an argument list. The compiler will correctly allow for the extra register (r2) pushed on the stack, when accessing parameters on the stack. Parameters passed in registers are, obviously, unaffected by this. The ability to define functions in this way is provided for completeness. Their usefulness may be limited.

It should be noted that there is a performance hit associated with any function calls within an interrupt service routine (unless the function is of the “inline” type, which does not result in an actual function call instruction). Any call requires the compiler save register r12, r13, r14 and r15 on the stack during the function call. For example, something as simple as:

```
uint32_t localtime;

void incloctime()
{
    localtime++;
}

interrupt(BASICTIMER_VECTOR) isr()
{
    incloctime();
}
```

will cause the overhead of saving these registers to occur, even though none of them are using within the called function. In this case, declaring “incloctime” as “static inline” will make things much more efficient.

For every device, the macros “NOVECTOR” and “RESET_VECTOR” are defined. If an interrupt service routine is declared as

```
interrupt (NOVECTOR) [wakeup, enablenested] IntServiceRoutine(void)
{
    /* Any normal C code */
}
```

GCC will not assign an interrupt vector for this routine. The code generated for the routine itself will be just the same as for any real interrupt vector. Similarly the macro “RESET_VECTOR()” can be used as the vector name when the standard reset start-up routine needs to be replaced with a customised one.

Customising the interrupt vector table

The interrupt vectors table is defined in the startup file for each device - “crtXXX”. These files are automatically linked when a project is built. The tables are customised for the specific interrupt vectors present in each device. Undefined interrupts will result in a call to “_unexpected_1_”, which branches to “_unexpected_”. “_unexpected_” can be redefined in your code.

If you wish, you can completely customise the interrupt vector table, by defining your own, like this:

```
/* Define interrupt vector table */

INTERRUPT_VECTORS =
{
```

```
        zero_vector,
        zero_vector,
        zero_vector,
        zero_vector,
        zero_vector,
        zero_vector,
        wakeup_vector,
        zero_vector,
        zero_vector,
        zero_vector,
        zero_vector,
        zero_vector,
        zero_vector,
        zero_vector,
        zero_vector,
        zero_vector,
        reset_vector
};
```

For this to work you must give the coomand line argument “’-nostartfiles’” to the gcc.

Controlling interrupt processing

There are some function definitions in “signal.h” to make interrupt control easier.

```
#include <signal.h>
void eint(void);
```

Enable interrupts by setting the global interrupt enable bit. This function actually compiles to a single instruction, so there is no function call overhead. “_EINT()” is defined as an alternative name for this function, for compatibility with other MSP430 tools.

```
#include <signal.h>
void dint(void);
```

Disable interrupts by clearing the global interrupt enable bit. This function actually compiles to a single instruction, so there is no function call overhead. “_DINT()” is defined as an alternative name for this function, for compatibility with other MSP430 tools.

```
#include <signal.h>
void _RESET(void);
```

You may declare your own version of the “_RESET()” function to override the default reset vector handler.

```
#include <signal.h>
void UNEXPECTED(void);
```

You may declare your own version of the “UNEXPECTED()” function to override the default handling of unexpected interrupts (i.e. ones for which no specific interrupt service routine has been defined).

Data types and memory handling

MSP430 architecture processors use a single address space to map data and code. The registers and memory are 16 bits wide, and the CPU can only read and write 16 bit data at even addresses. If you attempt to read or write a 16 bit value at an odd address, the CPU behaves as if the LSB is not set. The processor has no exception handling. The MSP430 can read and write 8 bit data at any address.

The C compiler supports the following basic data types

- char - 1 byte (8 bits)
- int - 2 bytes (16 bits)
- long - 4 bytes (32 bits)
- long long - 8 bytes (64 bits)
- float - 4 bytes

All the integer types are supported in signed and unsigned forms. Pointers are always 2 bytes wide.

All global variables with the “const” attribute are allocated in the main ROM space. They are normally placed in the .text section. Accessing “const” variable is no different than accessing to any other type of variable. If the device uses flash memory and the flash memory is enabled for writing, you can write to the flash. You can place “const” variables to RAM, using the attribute “section(".data"))” as follows:

```
const char __attribute__ ((section(".data"))) foo = 1;
```

Please note that if you declare variables r0 - r15, the assembler will prepend ’_’ in order to allow the assembler to distinguish them from the registers names.

Variables larger than one byte are always located at an even address. Single byte variables can be located at any address.

Accessing the MSP430’s peripheral registers - the SFRs

The MSP430’s memory mapped peripheral registers are termed special function registers (SFRs). The SFRs valid for the device you are using are declared when you include “io.h” in your source code, and specify the MCU architecture in the *GCC* command line.

You can consider any SFR as a normal variable, which is simply mapped to the specific memory location. You do not have to care about exactly which one.

SFRs are read from and written to using normal C assignments, so:

```
SFR = value;
```

will write from to and SFR, and

```
variable = SFR;
```

will read from one.

All SFRs are declared as “volatile”. This implies a-la Harvard architecture *GCC* behaviour for read-modify-write: mov SFR’s value to a register, modify the register, write back to the SFR. However, the GCC port for the MSP430 takes into account the possibility that this can often be reduced to something like:

```
and.b #1, &0x0120
```

and the appropriate code is produced. This optimisation can be switched off with the "-mno-volatile-workaround" compiler flag.

Reserving space above the stack

Declaring the "main" routine in the form

```
int RESERVE_RAM(10) main()
{
    ...
}
```

will reserve 10 bytes of memory, which resides at the top of RAM and will not be used by your C code. This is useful for things like PUC restarts, or data which should persist across resets.

Handling the status register

Several routines are available to assist in handling the status register. These should be used with care. Altering the status register bits in an uncontrolled way could badly affect the operation of your program.

```
#include <signal.h>
void WRITE_SR(const uint16_tx);
```

Set the value of the status register (r2).

```
#include <signal.h>
uint16_t READ_SR(void);
```

Read the value of the status register (r2).

```
#include <signal.h>
void BIS_SR(const uint16_t x);
```

Set bits in the status register (r2), using the "bis" instruction.

```
#include <signal.h>
void BIC_SR(const uint16_t x);
```

Clear bits in the status register (r2), using the "bic" instruction.

```
#include <signal.h>
SFR_CMD(cmd, (typeof SFR) sfr, (typeof SFR) val);
```

Perform an operation on an SFR, which is neither optimized nor modified by the compiler. For example:

```
SFR_CMD(bis.b, IE1,WDTIE); /* Enable the watchdog interrupt. */
```

does the same thing as

```
IE1 |= WDTIE;
```

The main reason for the user to directly access the status register in a C program is to switch between the MSP430's low power modes. Although non-interrupt code typically put the CPU into a low power mode, it is usually inside an interrupt service routine that the decision to switch back to normal operation occurs. If the interrupt service routine simply changed the status register bits, these would simply change back at as the routine exists, and the CPU would return to a low power state. To avoid this, the status register stored on the stack must be altered, so the change of processor mode occurs as the interrupt service routine exits, and the stack is popped. Two routines are defined to assist in this task.

```
#include <signal.h>
void _BIS_SR_IRQ(int16_t x);
```

Set bits in the copy of the status register stored on the statck.

```
#include <signal.h>
void _BIC_SR_IRQ(int16_t x);
```

Clear bits in the copy of the status register stored on the statck.

These functions should only be used within interrupt service routines. At present *GCC* issues a warning when these functions are used, but the correct code is produced. To make these functions (and the BIS_SR and BIC_SR) easier to use for switching LPM modes, the following values are defined when "io.h" is included in your source code

- "LPM0_bits" - the combination of status register bit which selects LPM 0.
- "LPM1_bits" - the combination of status register bit which selects LPM 1.
- "LPM2_bits" - the combination of status register bit which selects LPM 2.
- "LPM3_bits" - the combination of status register bit which selects LPM 3.
- "LPM4_bits" - the combination of status register bit which selects LPM 4.

You can also use these names without the suffix "_bits".

The standard library functions

The MSP430 version of libc contains a subset of the standard C library functions. These are:

abs() bsearch() exit() malloc()
rand() strtoul() abort() atoi()
labs() setjmp() abort() atol()
errno() ldiv() qsort() strtol()
ffs() memcpy() strcat() strdup()
strncmp() strspn() atol() memmove()
strchr() strlcat() strncpy() strstr()
bcmp() memccpy() memset() strcmp()
strlcpy() strpbrk() strtok() bcopy()
memchr() rindex() strcpy() strlen()
strrchr() swab() bzero() memcmp()
strcasecmp() strcspn() strncat() strsep()
snprintf() sprintf()

The full definition of these functions can be found in any C manual, including the on line documentation for the GNU tools. They will not be detailed here.

Take care with format conversions in sprintf():

- %d, %x, etc. convert 16 bit variables
- %lx, %ld, etc. convert 32 bit long variables.

The function

```
uprintf(void (*func)(char c), const char *fmt,...);
```

is similar to “sprintf()”, except that caller provides an output function for printing, rather than an output buffer. This function must accept a single “char” parameter, and return “void” (for example to send a character to a UART). The user function is responsible for mutexes, slow interfaces, etc. “uprintf()” will not return until all characters have been printed.

Starting from reset

The standard library includes a start-up module that prepares the environment for running applications written in C. Several versions of the start-up script are available because each processor has different set-up requirements. The *msp430-gcc* compiler selects the appropriate module based on the processor specified in the command line options.

The start-up module is responsible for the following tasks

- Providing a default vector table.
- Providing default interrupt handlers.
- Initializing the watchdog timer.
- Initializing the .data segment.
- Zeroing the .bss segment.
- Jumping to main(). (A jump is used, rather than a call, to save space on the stack. main() is not expected to return.)

The start-up module contains a default interrupt vector table. The contents of the table are filled with predefined function names which can be overridden by the programmer. The last entry in the table, however, is the address of the reset vector. The “_reset_vector__” is defined as a weak symbol. This means that if the application doesn’t define it, the linker will use the version in the library (or module). However, a user defined version will take precedence.

Look at the disassembled code produced by

```
$ msp430-objdump -DS a.out
```

```
a.out:     file format elf32-msp430

Disassembly of section .text:

0000fc00 <_reset_vector__>:
   fc00: b2 40 80 5a  mov #23168, &0x0120 ;  #0x5a80
```

```
   fc04: 20 01
   fc06: 3f 40 50 fc  mov #-944, r15 ;  #0xfc50
   fc0a: 3e 40 00 02  mov #512, r14 ;  #0x0200
   fc0e: 3d 40 00 02  mov #512, r13 ;  #0x0200
   fc12: 0d 9e        cmp r14, r13
   fc14: 06 24        jz $+14      ;  abs dst addr 0xfc22
   fc16: 1d 53        inc r13
   fc18: fe 4f 00 00  mov.b @r15+, 0(r14)
   fc1c: 1e 53        inc r14
   fc1e: 0f 9d        cmp r13, r15
   fc20: fb 2b        jnc $-8      ;  abs dst addr 0xfc18
   fc22: 3f 40 00 02  mov #512, r15 ;  #0x0200
   fc26: 3d 40 00 02  mov #512, r13 ;  #0x0200
   fc2a: 0d 9f        cmp r15, r13
   fc2c: 06 24        jz $+14      ;  abs dst addr 0xfc3a
   fc2e: 1d 53        inc r13
   fc30: cf 43 00 00  mov.b #0, 0(r15) ;  subst r3 with As==00
   fc34: 1f 53        inc r15
   fc36: 0f 9d        cmp r13, r15
   fc38: fb 2b        jnc $-8      ;  abs dst addr 0xfc30
   fc3a: 30 40 44 fc  br #0xfc44

0000fc3e <_unexpected_1_>:
   fc3e: 30 40 42 fc  br #0xfc42

0000fc42 <_unexpected_>:
   fc42: 00 13        reti

0000fc44 <main>:
   fc44: 31 40 80 02  mov #640, SP ;  #0x0280
   fc48: 30 40 4c fc  br #0xfc4c

0000fc4c <__stop_progExec__>:
   fc4c: 02 43        clr SR
   fc4e: fe 3f        jmp $-2      ;  abs dst addr 0xfc4c
Disassembly of section .data:
Disassembly of section .vectors:

0000ffe0 <InterruptVectors>:
   ffe0: 3e fc       interrupt service routine at 0xfc3e
   ffe2: 3e fc       interrupt service routine at 0xfc3e
   ffe4: 3e fc       interrupt service routine at 0xfc3e
   ffe6: 3e fc       interrupt service routine at 0xfc3e
   ffe8: 3e fc       interrupt service routine at 0xfc3e
   ffea: 3e fc       interrupt service routine at 0xfc3e
   ffec: 3e fc       interrupt service routine at 0xfc3e
   ffee: 3e fc       interrupt service routine at 0xfc3e
   fff0: 3e fc       interrupt service routine at 0xfc3e
   fff2: 3e fc       interrupt service routine at 0xfc3e
   fff4: 3e fc       interrupt service routine at 0xfc3e
   fff6: 3e fc       interrupt service routine at 0xfc3e
   fff8: 3e fc       interrupt service routine at 0xfc3e
```

```
fffa: 3e fc       interrupt service routine at 0xfc3e
fffc: 3e fc       interrupt service routine at 0xfc3e
fffe: 00 fc       interrupt service routine at 0xfc00
```

OK. Lets start from the end. Every MSP430 device has interrupt vectors table located at 0xffe0. So, here we can see, that as execution begins, the PC is loaded with the address 0xfc00. “_reset_vector__” is located at this address.

The first thing that happens is the watchdog timer is initialized. Then the program copies the initialized global variables to RAM (0xfc0a - 0xfc20). After this, the uninitialized globals are cleared (0xfc22 - 0xfc3a). After this, we jump to ’main’, which is located at 0xfc44.

In main, we copy the value 0x0280 to r1. This initializes the stack pointer, taking into account the space required for local variables.

Next, as long as main does nothing, it jumps to “__stop_progExec__”. At this point the SR is zeroed. The next instruction is jump to “__stop_progExec__”. This is the end of program execution.

In this module, the application uses:

- “_etext” - end of the “.text” section. The place where the initial values of global variables are stored.
- “__data_start” - the start of RAM.
- “_edata” - the end of data RAM (“_edata - __data_start” is the size of the “.data” segment).
- “__bss_start” - the place where uninitialized variables resides in RAM.
- “__bss_end” - the end of this segment.
- “__stack” - the stack.

All of these variables can be overridden with -Wl,--defsym=[symname]=(value) For example, to set the initial stack point to0x280, use **`-Wl,--defsym=__stack=0x280`**.

In most cases it is not necessary to redefine these values. They can be obtained from the user application as follows

```
...
extern int __stack;
int m;

(int *) m = &__stack;

/* now m contains the address of the stack */
...
```

Please note that these values do not change once they have been initialized.

The startup code adds a litter overhead to the application. The size of the startup code is 80 bytes without interrupt vector table. If you do not like this approach, you can define your own startup code.

Redefining the startup procedure

By defining _reset_vector__ in the user application, the linker will not link standard startup code. For example:

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

```
NAKED(_reset_vector__)
{
 /* place your startup code here */

 /* Make shure, the branch to main (or to your start
    routine) is the last line in the function */
 __asm__ __volatile__("br #main"::);
}
```

produces the following code

```
a.out:     file format elf32-msp430

Disassembly of section .text:

0000fc00 <__zero_vector>:
   fc00:       30 40 04 fc     br      #0xfc04

0000fc04 <_unexpected_>:
   fc04:       00 13           reti

0000fc06 <_reset_vector__>:
   fc06:       00 3c           jmp     $+2             ;  abs dst addr 0xfc08

0000fc08 <main>:
   fc08:       31 40 80 02     mov     #640,   SP      ;  #0x0280
   fc0c:       30 40 10 fc     br      #0xfc10

0000fc10 <__stop_progExec__>:
   fc10:       02 43           clr     SR
   fc12:       fe 3f           jmp     $-2             ;  abs dst addr 0xfc10
Disassembly of section .data:
Disassembly of section .vectors:
[skip]
```

Please note that if you declare your own startup, you must take care about initialising the values of global variables.

Another way to define the reset routine is to use the _RESET() macro:

```
_RESET()
{
 /* place your startup code here */
 __asm__ __volatile__("br #main"::);
}
```

Redefining the end up procedure

From the example above you can see, that main jumps to “__stop_progExec__”, which can be redefined the same way in the user code. However, it is possible to save some space, by specifying the return point as “main()”. If you compile with

```
$ msp430-gcc -mendup-at=main ...
```

you will get

```
...
0000fc08 <main>:
   fc08:      31 40 80 02     mov     #640,   SP      ;  #0x0280
   fc0c:      30 40 10 fc     br      #0xfc08
...
```

and “__stop_progExec__” will not be linked.

Initializing the stack

Stack initialization is performed in the prologue of “main()”. This suits most cases. However, you redefine startup, and needs some space allocated on the stack, the stack has to be explicitly initialized. For example:

```
#define STACKINITIALVALUE 0x0280

NAKED(_reset_vector__)
{
    /* Initialise the stack */
    __asm__ __volatile__("mov %0, r1"::"i" (STACKINITIALVALUE));

    /* Your startup code goes here */

    __asm__ __volatile__("br #main"::);
}
```

*** NOTE *** DO NOT USE the register definitions PC, SP and SR, which some other MSP430 tools (e.g. IAR) recognise. The *as* assembler in *binutils* does not recognise these names. Instead use r0, r1 and r2.

The stack finally will be initialized in main(). Normally, the initial stack pointer address is at top of RAM. If you want to reserve some RAM space, which is not accessible by the compiler, you may specify -mno-stack-init flag and then define startup as follows:

```
#define STACKINITIALVALUE 0x0280

NAKED(_reset_vector__)
{
    char a[100]; /* Will be allocated on the stack */

    __asm__ __volatile__("mov       #__data_start ,r1"::);

    /* Your startup code goes here */

    /* Initialise the stack */
    __asm__ __volatile__("mov %0, r1"::"i" (STACKINITIALVALUE));
```

```
    /* Jump to main */
    __asm__ __volatile__("br #main"::);
}
```

If you do not make the function declaration NAKED, note that on function exit, the frame pointer value will be added to r1. Therefore, the stack initialisation should be as follows:

```
...
__asm__ __volatile__("mov %0, r1" :: "i" (STACKINITIALVLUE) );
__asm__ __volatile__("sub #.L__FrameSize__reset_vector__, r1"::);
...
```

*** NOTE *** The frame pointer register (r4) and arguments pointer register (r5) values are compiled in a mysterious way. Use them with care.

The variable ".L__FrameSize_[function name]" is defined by the compiler, and has a value of the stack space required by the function. For example:

```
#define STACKINITIALVALUE 0x280

void set(char *a) {} // dummy
void reset(char *a) {}  // dummy

void _reset_vector__()
{
    __asm__ __volatile__("mov       #__data_start ,r1"::);
    {
        char a[100];
        set(a);
        reset(a);
    }
    __asm__ __volatile__("mov %0, r1"::"i" (STACKINITIALVALUE));
    __asm__ __volatile__("sub #.L__FrameSize__reset_vector__, r1"::);
    __asm__ __volatile__("br #main"::);
}

int main()
{
    ...
}
```

compiled with

```
$ msp430-gcc -O m.c -mendup-at=main -mno-stack-init
```

will result in:

```
---
a.out:     file format elf32-msp430

Disassembly of section .text:

0000fc00 <__zero_vector>:
   fc00:       30 40 04 fc     br      #0xfc04

0000fc04 <_unexpected_>:
   fc04:       00 13           reti
```

```
0000fc06 <set>:
   fc06:       30 41           ret

0000fc08 <reset>:
   fc08:       30 41           ret

0000fc0a <_reset_vector__>:
   fc0a:       31 80 64 00     sub     #100,   SP      ; #0x0064
   fc0e:       31 40 00 02     mov     #512,   SP      ; #0x0200
   fc12:       0f 41           mov     r1,     r15
   fc14:       b0 12 06 fc     call    #-1018          ; #0xfc06
   fc18:       0f 41           mov     r1,     r15
   fc1a:       b0 12 08 fc     call    #-1016          ; #0xfc08
   fc1e:       31 40 80 02     mov     #640,   SP      ; #0x0280
   fc22:       30 40 30 fc     br      #0xfc30
   fc26:       31 80 64 00     sub     #100,   SP      ; #0x0064
   fc2a:       31 50 64 00     add     #100,   SP      ; #0x0064
   fc2e:       30 41           ret

0000fc30 <main>:
   fc30:       30 40 30 fc     br      #0xfc30
....
---
```

In this case, the “RESERVE_RAM” attribute to the “main()” function would be a simpler way to achieve the same effect.

Chapter 5. mspgcc’s ABI

Register usage

If you intend to interface assembly routines with your C code, you need to know how *GCC* uses the registers. This section describes how registers are allocated and used by the compiler. (You can override GCC’s settings by issuing -ffixed-regs=...)

r0, r2, and r3 - are fixed registers and not used by the compiler in any way. They cannot be used for temporary register arguments either.

r1 - is the stack pointer. The compiler modifies it only in the function prologues and epilogues, and when a function call with a long argument list occurs. Do not modify it yourself under any circumstances!!!

r4 - is the frame pointer. This can be used by the compiler, when va_args is used. When va_args is not used, and optimization is switched on, this register is eliminated by the stack pointer.

r5 - argument pointer. This can be used by the compiler, when a function call with a long argument list is performed. It refers to the stack position before the function call. Normally, when optimization is turned on, this register usage is eliminated and the argument list is accessed via the stack pointer.

Use the last two with care. If *GCC* uses them as these pointers, their values, after the function’s prologue are:

- r5 = r1 + [size of registers pushed in the prologue] + 2 bytes;
- r4 = r1 - [frame size];

where r1 is its value on function entry, minus the size of registers pushed in the prologue.

r12, r13, r14, and r15 - are call cloberred (in general) registers. If you are interfacing C with assembler language, you do not have to save these registers, except in interrupt service routines.

*** NOTE *** some library calls (such as divmodM, mulM) clobber some registers (r8 - r11). *GCC* allows for this during code generation, and will save clobbered registers on the stack in the calling function.

All other registers are caller used registers. If you are interfacing C with assembler language, you must save them on the stack and restore them before exit.

Registers are allocated in the order r12 to r15, r11 to r0. Please use this order if you plug in assembly language functions.

char, int and pointer variables take one register. long and float variables take two registers, in little-endian order (i.e. the LSB goes in lower numbered register). long long int variables take 4 registers, in little-endian order.

Function calling conventions

Fixed argument lists

Function arguments are allocated left to right. They are assigned from r15 to r12. If more parameters are passed than will fit in the registers, the rest are passed on the stack. This should be avoided since the code takes a performance hit when using variables residing on the stack.

Variable argument lists

Parameters passed to functions that have a variable argument list (printf, scanf, etc.) are all passed on the stack. Any char parameters are extended to ints.

Return values

The various functions types return the results as follows:

- char, int and pointer functions return their values r15
- long and float functions return their values in r15:r14
- long long functions return their values r15:r14:r13:r12

If the returned value wider than 64 bits, it is returned in memory. The first 'hidden' argument to such a function call will be a memory address. All other arguments will be allocated in the usual way, from r14.

Call definitions

Variables names are not transformed in any way, except those with the names r0 to r15. For these a '_' is prepended to the name. Aliasing a declaration (int A asm("M");) will not change the alias. If "M "is specified as a direct address (see the SFR definitions for example), *GCC* will not 'globalize' the symbol. So, you can include these 'address' definitions in any header file.

Each function call starts with a function prologue. The prologue pushes caller used registers (r4-r11, or r4-r15 for interrupt functions) onto the stack. The stack pointer is then adjusted by subtracting the frame size from r1. Naked functions do not issue a prologue. The 'interrupt' attribute forces all registers used within the function to be saved on the stack.

In every function definition

```
.L__FrameSize_[function name]=[frame size]
```

is defined. For example:

```
main:
.L__FrameSize_main=0x12
```

It can be used as an immediate variable in in line assembly code. For example

```
__asm__ __volatile__("bic #0xf0, .L__FrameSize_main(r1)"::);
```

is similar to using the _BIC_SR_IRQ function.

Every function call ends with a function epilogue. Here, the stack pointer is adjusted by adding the frame size to r1. The registers save registers are the popped from the stack. The function then returns. Normal functions just issue a "ret" instruction. Functions with the interrupt attribute issue a "reti" instruction. For wake-up interrupts the sequence:

```
bic #0xf0,0(r1)
reti
```

is used.

The function "main" is handled specially:

- main() sets the stack pointer, unless -mno-stack-init is specified on the *GCC* command line.
- main() does not save registers.
- main() does not return.
- main() jumps to "__stop_progExec__" at the end unless "-mendup-at=" is specified on the *GCC* command line.

Assembler extensions

GNU msp430-as supports TI style assembler syntax. Some extensions are:

- @Rn as destination treated as 0(Rn)
- 0(Rn) as source treated as @Rn
- jmp +N skips next N bytes (use with care).
- jmp $+-N advances/rewinds PC N bytes from current location.
- jmp -N rewinds PC N bytes from current location.

Chapter 6. Using inline assembly language in C programs with mspgcc

mspgcc tries to be largely compatible with the other C language toolchains for the MSP430. Inline assembly language is one area where this is impractical. mspgcc uses the usual *GCC* syntax for inline assembly language, with a few extensions to deal with MSP430 specific issues. At first sight GCC's way of handling inline assembly language may seem a little more difficult to use than some of the alternatives. It is, however, generally more efficient and powerful than those alternatives.

Inline assembly language syntax

mspgcc supports the standard GNU inline assembler feature 'asm'. In an assembler instruction using 'asm', you can specify the operands of the instruction using C expressions. This means you need not guess which registers or memory locations will contain the data you want to use.

You must specify an assembler instruction template much like what appears in an assembler language, plus an operand constraint string for each operand. For example:

```
asm("mov %1, %0": "=r" (result): "m" (source));
```

This could also be written:

```
asm("mov %src,%res": [res] "=r" (result): [src] "m" (source));
```

which may be clearer. Here 'source' is the C expression for the input operand while 'result' is that of the output operand. '=' indicates, that the operand is an output. m and r are constraints and inicate which types of addressing mode *GCC* has to use in the operand. These constraints are fully documented in the GNU GCC documentation.

Each asm statement is divided into four parts, by colons:

1. The assembler instructions, defined as a single string constant:

   "mov %src, %res"

2. A list of output operands, separated by commas. Our example uses just one, and defines the identifier "res" for it:

   [res] "=r" (result)

3. A comma separated list of input operands. Again, our example uses just one operand, and defines the identifier "src" for it:

   [src] "m" (source)

4. The clobbered registers. This is left empty in our example, as nothing is clobbered.

So, the complete pattern is:

```
asm((string asm statement) : [outputs]:[inputs]:[clobbers]);
```

Each input and output operand is described by a constraint string followed by a C expression in parantheses. msp430-gcc recognises the following constraint characters:

- m - memory operand.
- I - integer operand.
- r - register operand.
- i - immediate operand (int constants in most cases).
- P - constants, generated by r2 and r3.

and some other constraints which are common to all processors supported by GCC. These constraints cause the compiler to automatically generate preamble and postamble code, allocate registers, and save and restore anything necessary to ensure the assembly language is efficiently and compatibly handled. For example

```
    asm("add %[bar],%[foo]"
        : [foo] "=r" (foo)
        : "[foo]" (foo), [bar] "m" (bar));
```

is equivalent to

```
foo += bar;
```

and will result in the following generated assembly language (assuming "foo" is a global variable)

```
    mov &foo, r12
/* #APP */
    add &bar, r12
/* #NOAPP */
    mov r12, &foo
```

If there are only unused output operands, you will also need to specify ’volatile’ for the ’asm’ construct. If you are writing a header file that will be included in ANSI C programs, use ’__asm__’ instead of ’asm’ and ’__volatile__’ instead of ’volatile’.

A percent ’%’ sign followed by a digit or defined tag forces *GCC* to substitute the relevant operand. For 4 and 8 byte operands use the A, B, C, and D modifiers to select the appropriate 16 bit chunk of the operand. For example:

```
#define LONGVAL 0x12345678l

{
    long a,b;
    ...
    asm("mov %A2, %A0 \n\t"
        "mov %B2, %B0 \n\t"
        "mov %A2, %A1 \n\t"
        "mov %B2, %B1 \n\t"
        : "=r"((long)a),"=m"((long)b)
        : "i"((long)LONGVAL) );
    ...
}
```

or

```
#define LONGVAL 0x12345678l
```

```
{
    long a,b;
    ...
    asm("mov %A[longval], %A[a] \n\t"
        "mov %B[longval], %B[a] \n\t"
        "mov %A[longval], %A[b] \n\t"
        "mov %B[longval], %B[b] \n\t"
        : [a] "=r" ((long) a), [b] "=m" ((long) b)
        : [longval] "i"((long) LONGVAL));
    ...
}
```

This will result in something like the following generated assembly language (assuming 'a' is declared within the block, and 'b' is declared globally):

```
    ...
/* #APP */
    mov #llo(305419896), r12
    mov #lhi(305419896), r13
    mov #llo(305419896), 4(r1) ; mov #llo(305419896), &b
    mov #lhi(305419896), 6(r1) ; mov #lhi(305419896), &b+2
/* #NOAPP*/
    mov r12, 0(r1)
    mov r13, 2(r1)
    ...
```

So,

- %A[tag] acts as %[tag] for a register or address constant operand, or wraps an integer value as #llo(). #llo is an assembler macro, which extracts the lower 16 bits of the value.
- %B[tag] adds 1 to a register number, or 2 to an address constant, or substitutes #lhi() for an integer constant.
- %C[tag] adds 2 to a register number, or 4 to an address constant, or substitutes #hlo() for an integer constant.
- %D[tag] adds 3 to a register number, 6 to an address constant, or substitutes #hhi() for an integer constant.

The I, J, K and L modifiers are similar, except they add 1 to an address or register. They should only be used in zero_extendMN operations.

There is also a %E modifier, which substitutes Rn from (mem:xx (reg:xx n)) as @Rn. This is a useful modifier for the first element on the stack or for pointers. !!! Do not use this unless you know exactly what are you doing !!!

Registers, variables and labels

Since *GCC* cannot check the assembler syntax, you can do anything within an assembler asm() statement. However, please note, that *GCC* does not use r0, r2, or r3. Therefore, if you mention one of these registers as an output parameter in an asm() statement, or as an alias for register variable, *GCC* will substitute some another register instead. Using r0, r2, or r3 as input parameters will result in the error "'asm' operand requires impossible reload".

Variables can be used in any normal way within asm() statement (mind name conversion for [Rr][0-15] names)

GCC defines labels with the following patterns:

".Lfe%="	A function end label
".L__Frame_size_%s"	See above
".L%="	A local label for almost all purposes :)

where the %= modifier stands for a unique number within the file.

The following labels are defined for some expanded operands:

```
.Lsren%=
.Lsrcl%=
.Lsre%=
.Lae%=
.Lmsn%=
.Lcsn%=
.Lsend%=
.Lsst%=
.Leaq%=
.LcmpSIe%=
```

so, do not use these patterns. You may use any other label as you wish. Please note, that if label starts from .L it means, that the label is local and cannot be seen from another file as well as in disassembled output with *msp430-objdump*.

Library calls

There are some library functions used by GCC, during code generation. These use non-standard argument passing schemes, which do not follow the mspgcc ABI. So, do not use them in your assembly code unless you are absolutely sure what is going on. Namely:

Multiplication:

```
__mul{qi,hi,si}3
__umul{qi,hi,si}3
__umulsi3hw
```

For devices without a hardware multiplier, multiply routines are called to perform multiplication. If the destination is in HI mode (16 bit) or QI mode (8 bit), the first argument is passed in r10, and the second in r12. The returned value is in r14. In SI mode (32 bit), the first argument is passed in r11:r10, and the second in r13:r12. The result in r14:r15. In both cases the input arguments are clobbered after the function returns.

Division:

```
__divmod{qi,hi,si}4
__udivmod{qi,hi,si}4
```

If destination is in HI mode (16 bit) or QI mode (8 bit), the numerator is passed in r12, and the denominator in r10. The result of the calculation r12/r10 is returned with the quotient in r12 and the remainder in r14. Registers r10, r11 and r13 are clobbered.

In SI mode (32 bit), the numerator is passed in r13:r12 and denominator in r11:r10. The quotient is returned in r13:r12, and the remainder in r15:r14. Registers r8, r9, r10, and r11 are cloberred.

All floating point library calls can be used the in the usual way, as these obey the mspgcc ABI rules (when applicable - just help me to force this process :).

Chapter 7. Tips and trick for efficient programming

As well as the usual things a good programmer will do, there are some issues specific to mspgcc which you should be aware of.

1. If you are sure your main routine will never exit, you can use the “-mendup-at=main” flag when compiling. This will save 6 bytes of ROM.
2. Avoid passing long argument list to functions. Avoid returning long values from functions. The best functions types to use are void, int, or pointer.
3. Avoid the initialization of global variables within a small function call. Instead, assign a value during variable definition.
4. Avoid converting chars to another type. char variables can be located anywhere in RAM, while word variables can only be at even addresses. Because of this, the following code:

   ```
   const char *a = "1234";
   int k;
   k = *((int *)((char *a) + 3));
   ```

   will result in unpredictable CPU behaviour.
5. Avoid using global variables of small size - it is a waste of RAM.
6. Avoid using volatiles, unless they are really necessary.
7. Use int instead of char or unsigned char if you want an 8 bit integer.
8. Inspect assembler code (-S compiler flag). The compiler cannot eliminate dead code in some cases. Do not write dead code :)
9. Do not declare your own SFRs. They are all declared in include files in the right way to achieve maximum code performance.
10. Try to minimise the use of addition and subtraction with floating point numbers. These are slow operations.
11. Use shift instead of multiplication by constants which are 2^N (actually, the compiler may to do this for you when optimization is switched on).
12. Use unsigned int for indices - the compiler will snip _lots_ of code.
13. Use ’switch/case’ constructs rather than a chain of ’if/else’ constructs.
14. Use logical "or" (’|’) instead of ’+’ for bitmasks.
15. When defining bit fields, try to use signed integers. This produces more compact code that bit fields of unsigned integers.
16. Use ’alloca’ instead of ’malloc’ for locals. In embedded applications trying to avoid any dynamic memory allocation is usually even better ;).
17. Apart from C++ recomendations ;), it would be better to use:

    ```
    #define SMTS 1234567890l
    ```

    instead of declaring

    ```
    const long smts = 1234567890l;
    ```

18. If you execute

    ```
    while ((long) a & 0x80000l);
    ```

the program will hang, unless ’a’ is declared volatile. So, do it!

19. Delay loops are very sophisticated routines. Normally, users do something like:

    ```
    int i = 1234;

    while (i--);
    ```

    or

    ```
    int i;

    for (i = 0;  i < 1234;  i++);
    ```

    NEITHER WILL WORK AS YOU EXPECT when optimisation is switched on!!! The optimizer will detect dead code in both examples and will eliminate it. It might even eliminate the loop completely. Adding the volatile attribute to the definition of ’i’ might help, but don’t count on it if ’i’ is a local variable. The compiler can still detect the calculations are wasteful, and eliminate them.

    Regardless of these optimisation issues, this type of delay loop is poor programming style - you have no idea how long or short a delay it might produce (although there is an obvious minimum bound!). It would be better, and more reliable to define something like:

    ```
    static void __inline__ brief_pause(register unsigned int n)
    {
        __asm__ __volatile__ (
      "1: \n"
      " dec %[n] \n"
      " jne 1b \n"
            : [n] "+r"(n));
    }
    ```

    and call this routine where needed. This is simple, compact, and predictable.

Do not do anything unless you know what you’re doing :)

Chapter 8. Hardware tools

What is available?

Texas Instruments produce fairly inexpensive development kits for the MSP430 range of processors. These consist of a small prototyping board holding the processor, and a JTAG-to-parallel-port adapter suitable for any standard PC type machine. Evaluation boards and TI compatible JTAG-to-parallel port tools are available from several suppliers, including Olimex, SoftBaugh and Lierda. Any of these tools may be used with the debug facilities which mspgcc provides.

Setting up the JTAG interface

You might be able to just plug the 25-way D-type from the JTAG FET tool into the parallel port of your PC, and the 14-way IDC connector into the receptacle on the prototyping card and have no trouble. On the other hand.....read on!

Parallel port issue with Windows

If you are running Windows 98 or Windows Me the parallel port should basically just work, as long as you ensure no other drivers (such as printer drivers) are configured to use the port. If you are running Windows NT, 2000, or XP you need a driver to provide transparent access to the parallel port, so it can be used directly by the mspgcc software. The driver for this is called *giveio*, and is normally installed automatically by the Windows installer program. If you find you cannot access the parallel port with msp430-gdbproxy, check that *giveio* is installed and running.

If all is well, you should be able to run *msp430-gdbproxy*. Whilst gaining confidence in the tools, it might be a good idea to run *msp430-gdbproxy* with debugging on in a window of its own, so it can be monitored for any signs of trouble. The following command

```
$ msp430-gdbproxy --debug msp430
```

should do this. The --help argument will make it list its options.

Parallel port issues with Linux

If you are running a version of Linux based on a 2.0.x or 2.2.x kernel you will not be able to use the JTAG debug tools. The other parts of mspgcc - C compiler, assembler, linker, etc. - can be used with any version of Linux. However Linux kernels prior to 2.4 did not support the ppdev raw parallel port access device driver needed by msp430-gdbproxy.

If a printer daemon is configured to use the parallel port you wish to use for mspgcc you will need to stop that printer daemon before *msp430-gdbproxy* can access the port.

Check that your Linux kernel actually has raw parallel-port support. If you are running a vanilla Red Hat or Debian distribution, this should be the case already. Type

```
$ cat /proc/devices
```

and look for an entry referring to “ppdev”. If there isn’t one you will need to recompile your kernel. This is only likely of you have a custom kernel.

If a printer daemon is using the parallel port, stop it. A command like:

```
$ /etc/rc.d/init.d/lpd stop
```

will do this on any Red Hat Linux machine, and many other distributions.

Check that permissions of the raw parallel-port device(/dev/parport0). It should be readable and writable by whichever user will run ’msp430-gdbproxy’. *Don’t run ’msp430-gdbproxy’ as root just to get around this. Set the permissions properly, and maximise the security of your machine!*

For use on your own desktop machine, it is OK to set the permissions for “/dev/parport0” to 666 (read and write allowed for everybody) and then run ’msp430-gdbproxy’ from your own shell.

For shared machines, set the permissions for “/dev/parport0” to 660, and make it a member of a suitable group. ’lp’ will probably do, but it might be tidier to generate a new group for this purpose (say, mspgcc or jtag) and use that. Make ’msp430-gdbproxy’ a member of that group, and set its SGID bit.

You should now be able to run *msp430-gdbproxy*. Whilst gaining confidence in the tools, it might be a good idea to run *msp430-gdbproxy* with debugging on in a window of its own, so it can be monitored for any signs of trouble. The following command

```
$ msp430-gdbproxy --debug msp430
```

should do this. The --help argument will make it list its options.

MSP430 evaluation and prototyping cards

If are using the evaluation kits from TI, there are a couple of points to beware of.

If you are running your MSP430 chip from less than 3.6V, make sure you’ve removed the zero ohm link r8, and refitted it as r9. If you do not do this, all sorts of things can go wrong. The JTAG interface will try to drive signals referenced to the parallel-port’s 5V signals. It seems this then messes up the MSP430 whose core is running on whatever rails you’ve decided upon externally.

More details on this are provided by Texas’s own document on the CDROM that came with the FET kit. The CDROM is nearly un-navigable, but the file you want is "./Literature/Literature - MSP 430/User’s Guide/FET Users Guide/MSP-FET430P140 Users Guide.pdf" assuming you’re starting from the root directory of the CDROM. Page 28 has a circuit diagram. Beware of the minor version differences between the prototyping cards.

Chapter 9. Compiling and linking MSP430 programs

Getting started

Remember to use msp430-gcc (or msp430-as) and msp430-ld when developing code. The easiest thing to do is to put lines saying:

```
CC=msp430-gcc
LD=msp430-ld
```

at the top of your Makefiles. It is also a very good idea to specify

```
CFLAGS=-O2
```

or

```
CFLAGS=-g -O2
```

You will almost certainly want “-O2” for any real production code. This turns on the C compiler’s optimisation. Without it, the code can be quite large and slow.

During development specify at least the -g flag during compilation. This will put all the necessary information into the compiled binary file to allow symbolic debugging in GDB. It makes the binary files rather big, but the actual downloaded code is not affected.

Assembling assembly language programs

You can use GCC to build assembly language programs, as well as C language ones. If the extension of your assembly language file is ’.s’, the following command will assemble it:

```
$ msp430-gcc -D_GNU_ASSEMBLER_ -x assembler-with-cpp -c f.s -o f.o [options]
```

If the extension of your assembly language file is ’.S’, following command will assembl it:

```
$ msp430-gcc -D_GNU_ASSEMBLER_ -c file.s -o file.o [options]
```

Chapter 10. Programming and debugging MSP430s

Using the JTAG FET tool with gdbproxy

Before using *msp430-gdb*, make sure *msp430-gdbproxy* is running. By default it uses TCP port 2000 to communicate with the debugger. You can explicitly set a port on the command line. The command

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

will start “msp430-gdbproxy”

Put the following lines into your GDB startup file. For Unix and Linux this is the file “.gdbinit” in your home directory. For Windows users it is “gdb.ini”:

```
set remoteaddresssize 64
set remotetimeout 999999
target remote localhost:2000
```

or whatever well known port you have *msp430-gdbproxy* listening at.

msp430-gdbproxy understands several MSP430 specific commands, as well as the standard GDB ones.

help		This help text
erase	blank, ’info’, ’main’, or ’all’,	Erase target Flash
puc		Reset target over JTAG, using PUC
reset		Reset target over JTAG, using hardware reset
identify		Identify what target is connected to the JTAG port
jtag	blank, ’release’, or "hold’	Define/report how JTAG is to be handled when a program is running
vcc	<voltage>	Define/report the VCC of the MSP430
dump		[for debug] Read out target registers

Downloading code to a target processor

Let’s assume you have a fully-linked executable called ’foo’. Type:

```
$ msp430-gdb foo
```

Assuming that you set up your “/.gdbinit” file as suggested above, then nothing special has to be done. Otherwise type the three commands listed above at the *gdb* command prompt. It you have *msp430-gdbproxy* running in the foreground in its own window you should see evidence of it starting to work.

Type:

```
$ monitor erase all
```

and the JTAG interface will clear the flash memory of the MSP430. Type:

```
load foo
```

and after a few seconds you should get confirmation that the flash memory has been reprogrammed. You need to erase the flash before you can load new code into it.

Running code

Logic might tell you to type 'run' in order to run the program. *Don't*. msp430-gdb is built upon the standard GNU GDB, and inherits all its behaviour. The 'run' command is appropriate for starting a program being debugged on the host. For an embedded target processor the correct command is 'continue', or just 'c'.

If the code hits a breakpoint it will stop. Otherwise you need to type ^C (control-C) to interrupt the target processor. Assuming you compiled 'foo' with the -g flag, *msp430-gdb* will tell you exactly where it was interrupted, and let you inspect the state of registers, memory, variables, and so on. In fact, all the normal things you would expect from any other implementation of GDB.

Additional tools

There are some additional tools available for mspgcc.

pyBSL

Software for the bootstrap loader. Works with Flash devices (MSP430F1xx and F4xx): erase and download new software or upload RAM or Flash data from the device back to the PC.

Features:

- Load TI-Text, Intel-hex and ELF files into a device.
- Download and verify to Flash and RAM.
- Erase Flash.
- Reset and wait for keypress (to run a device directly from the port power).
- Load an addres into R0/PC and run.
- Password file can be any datafile, e.g. the one used to program the device in an earlier session.
- Upload a memory block MSP->PC (output as binary data or hex dump).
- Download a program, execute it, resynchronize and uplaod results. (for testing and calibration).
- Written in Python, runs on Win32, BSD, Linux (and other POSIX compatible systems) (and Jython).
- Use per command line, or in a Python script.
- Downloadable replacement MSP430-BSLs, which also allows higher baudrates.

For more a complete description, including installation notes and usage examples, look at the readme.txt (http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/*checkout*/mspgcc/pybsl/readme.txt?rev=HEAD&content-type=text/plain)

msp430simu

An MSP430 simulator, written in Python is under development. Although a simulator is built into msp430-gdb, it fulfills a somewhat different requirement. The simulator is under development. Currently it allows single stepping through programs, butno active peripherals are yet supported (values can be written at any address, but reading has the same effect as reading from RAM).

It has a simple GUI with memory and disassembler views, logging output, as well as a file open dialog to select intel hex files. Or it can be embedded in Python scripts e.g. for automatic testing, etc.

Requirements:

- Python 2.1 or newer.
- wxPython for the GUI only.

pyJTAG

pyjtag is a program to use the MSP430 parallel JTAG adapters as simple programming tools. Just like msp430-gdbproxy, it works with the devices from TI, Olimex, Softbaugh, and others. It works with Flash devices (MSP430F1xx and F4xx), and can erase Flash and download or upload both RAM and Flash data in an attached device.

Features:

- loads TI-Text, Intel-hex and ELF files.
- download to Flash and/or RAM, erase, verify.
- reset and wait for keypress.
- upload a memory block MSP->PC (output as binary data or hex dump).
- download a program, execute it. (limited/funclets)
- written in Python, runs on Win32, Linux, BSD (other platforms possible if parallel port module is ported)
- use per command line, or in a Python script.

For more a complete description, including installation notes and usage examples, look at the readme.txt (http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/*checkout*/mspgcc/pyjtag/readme.txt?rev=HEAD&content-type=text/plain)

Requirements:

- MSP430 JTAG library, MSP430mspgcc.dll/so,. See "jtag/msp430" in the CVS.
- hardware interface library, HIL.dll/so,. See "jtag/hardware_access" in the CVS.
- Python extension. See "jtag/python" in the CVS.
- Python 2.2

Unpack these files to the pyjtag directory.

pySerJTAG and the serial-JTAG adapter

pySerJTAG is the PC side software for the Serial-JTAG adapter. The PC software is open and written in Python. It should run on the same platforms as pyBSL (Win32, Linux, BSD and more POSIX compatible systems). The command line options are compatible with pyBSL and pyJTAG.

The hardware design is open. The schematics can be found in the CVS module "hardware/serialJTAG" (here’s a PDF (jtagF14x.pdf) 62kB). A binary of the firmware for this design is here serjtag.zip (<10k, beta).

Why would you want this? The parallel port has some drawbacks such as missing OS support for bit banging. We cannot access every platform that mspgcc users use. Therefore it’s impossible to distribute binaries for all. The Serial-JTAG adapter moves the proprietary code out to that box and the user software on the PC can be open source. It will allow us to provide debug features in the near future.

Chapter 11. Building mspgcc from source code

Binary installers are provided at the mspgcc web-site for some platforms. For others you can build the tools yourself. The source code for everything except *msp430-gdbproxy* is available for download. Even the majority of the code for msp430-gdbproxy can be downloaded from the mspgcc web-site, where it may be of value to other projects. The only source code which is not available is the library which interfaces to the JTAG port of an MSP430, via the FET tool. This is TI's proprietary code. They have made it available for use in mspgcc, provided the source code is not released.

Shopping list

To build mspgcc from source code you will need the standard GNU source code, and the MSP430 specific source code.

The basic GNU packages

The source code for binutils may be obtained from the official binutils web-site (http://sources.redhat.com/binutils/). Versions of binutils beginning with 2.14 contain support for the MSP430. This can be found at "ftp://sources.redhat.com/pub/binutils/releases/binutils-2.14.tar.bz2" (ftp://sources.redhat.com/pub/binutils/releases/binutils-2.14.tar.bz2) (about 10.5MB).

Only the core package for *GCC* is required to build the C compiler. This can be found as the file "/releases/gcc-3.2.3/gcc-core-3.2.3.tar.bz2" (about 10MB) at any of the GNU mirror sites (http://gcc.gnu.org/mirrors.html).

The source code for GDB and Insight (GDB + a GUI) may be obtained from the official GDB web-site (http://sources.redhat.com/gdb/). The source code for GDB 5.1.1 may be found at "ftp://sources.redhat.com/pub/gdb/old-releases/gdb-5.1.1.tar.bz2" (ftp://sources.redhat.com/pub/gdb/old-releases/gdb-5.1.1.tar.bz2) (about 10.5MB). The source for Insight 5.1.1 may be found at "ftp://sources.redhat.com/pub/gdb/old-releases/insight-5.1.1.tar.bz2" (ftp://sources.redhat.com/pub/gdb/old-releases/insight-5.1.1.tar.bz2) (about 15.5MB).

The mspgcc specific code

At present, the best way to obtain the MSP430 specific source code is directly from the CVS repository at the mspgcc project web-site (http://sourceforge.net/cvs/?group_id=42303), as follows:

```
export CVSROOT=:pserver:anonymous@cvs.mspgcc.sourceforge.net:/cvsroot/mspgcc
export CVS_RSH=ssh
cvs login
```

Just press enter when prompted for the password. Then continue with:

```
cvs checkout gcc
cvs checkout gdb
cvs checkout msp430-libc
cvs checkout jtag
cvs checkout packaging
```

Do not bother getting the *binutils* files from CVS. Now the mspgcc *binutils* code has been merged into the official binutils source tree, these CVS files are unnecessary, and no longer maintained.

Tools required to build mspgcc

If you are using Linux or BSD Unix you probably have all the tools you need already installed on you machine. You just need the basic GNU toolchain - GNU make, GCC, binutils, and basic utilities like tar and bzip2.

If you are using Windows you will need to install cygwin (http://cygwin.com/) on your machine. Just go to the Cygwin site, and follow the installation instructions (it is really quite simple). Make sure you have at least the following packages installed:

- GNU make
- GCC (host installation)
- binutils (host installation)
- bzip2 and tar

The build procedure

The build instructions apply to Linux installations. You may need to modify them a little for other systems. For example, you might need to use “gmake” rather than “make”. On Windows machines with Cygwin omit the “su” steps. You can unpack the code in your home directory, and compile the tools as a normal user. Only the installation need be performed as a superuser.

The files used to build the Windows installer, and Linux RPMs may be found in the “packaging” directory at the mspgcc web-site.

First configure, build and install *binutils*. The following commands will unpack the source code, configure binutils as a cross assembly package, build and install it:

```
$ tar --bzip2 -xf binutils-2.14.tar.bz2
$ cd binutils-2.14
$ ./configure --target=msp430 --prefix=/usr/local/msp430
$ make
$ su
$ make install
```

You may wish to change the prefix, to install the software in a directory other than “/usr/local/msp430”. Common alternatives would be “/usr” or your home directory.

Next, ensure the directory in which you installed the *binutils* binary files is included in your “PATH” variable. The next stage will require the MSP430 binutils to be functional, when the MSP430 library is compiled.

Next, configure, build and install *GCC*. Make sure you specify the same “--prefix” and “--target” that you specified for *binutils*. Unpack the source code, as follows:

```
$ tar --bzip2 -xf gcc-core-3.2.3.tar.bz2
```

Copy the files from the “gcc/gcc-3.3” directory in the CVS repository at the mspgcc web-site into the unpacked *GCC* source tree. The mismatch between the numbering of *GCC* and this directory is an historical accident. You really do want the “gcc/gcc-3.3” to go with *GCC* version 3.2.3. To copy these files, and build and install *GCC*, use the following commands:

```
$ cp -a gcc/gcc-3.3/* gcc-3.2.3
$ cd gcc-3.2.3
```

```
$ ./configure --target=msp430 --prefix=/usr/local/msp430
$ make
$ su
$ make install
```

Download *msp430-libc* as either a tarball or from the CVS repository at the mspgcc web-site.

```
$ cd msp430-libc/src
```

If you specified something other than “/usr/local/msp430” as the prefix, when building *binutils* and *GCC*, you will need to edit the *Makefile*. Change “/usr/local/msp430” to the installation directory you are actually using. The use the following commands to build and install the library:

```
$ make
$ su
$ make install
```

Now build and install *GDB*. This procedure works equally well for *insight-5.1.1*. Just replace "*gdb*" with "*insight*" in the following steps. Make sure that you specify the same “--prefix” and “--target” as for *binutils* and *GCC*:

```
$ tar --bzip2 -xf gdb-5.1.1.tar.bz2
```

Copy the *GDB* files from the CVS repository at the mspgcc web-site into the unpacked *GDB* source tree. Now build and install *GDB* with the following commands:

```
$ cd gdb-5.1.1
$ ./configure --target=msp430 --prefix=/usr/local/msp430
$ make
$ su
$ make install
```

The source code for the generic *gdbproxy* program may be downloaded from the mspgcc web-site. However, the MSP430 specific source code is not available. If you want to use the generic source code for another project, you can. If you want to run *msp430-gdbproxy* you will need to download a binary version. msp430-gdbproxy requires a library - *libHIL.so* - to allow access to the parallel port. If there is a binary file for *msp430-gdbproxy* available for your machine, you should now build and install *libHIL.so*.

Copy the *libHIL* source code from the “jtag/hardware_access/HILppdev” directory in the CVS repository at the mspgcc web-site. Then, build and install it with the following commands:

```
$ cd jtag/hardware_access/HILppdev
$ make libHIL.so
$ su
$ mv libHIL.so /usr/local/lib
$ ldconfig
```

You could actually put the library almost anywhere, but make sure its directory is listed in your “LD_LIBRARY_PATH” environment variable.

You can now test the installed tools. Try building an example program, as follows:

```
$ msp430-gcc -mmcu=msp430x148 -o test -O test.c
```

You could then try producing disassembled text, with

```
$ msp430-objdump -DS test
```

Or you could generate Intel format hex output (e.g. for a programmer) with:

```
$ msp430-objcopy -O ihex test test.ihex
```

If you are able to use *msp430-gdbproxy*, and you have a JTAG FET tool (or one of the available compatible devices) you can try debugging a program in a target MSP430, using *GDB* and *msp430-gdbproxy*.

If all this works OK, you should now have a fully working mspgcc. We hope you enjoy using it. If you have any problems, try consulting the archives of the mailing lists at the mspgcc web-site (http://mspgcc.sourceforge.net). If that doesn't help, trying asking questions using the mailing list.