8 A Tourist's Phrasebook for Reversing MSP430

by Ryan Speers and Travis Goodspeed

Howdy, y'all!

Welcome to another installment of our series of quick-start guides for reverse engineering embedded systems. Our goal here is to get you situated with the MSP430 architecture as quickly as possible, with a minimum of fuss and formality.

Those of you who have already used an MSP430 might find this to be a useful reference, while those of you new to the architecture will find that it isn't really all that strange. If you've already reverse engineered binaries for any platform, even x86, we hope that you'll soon feel right at home.

8.1 The Landscape

Architecture

Von Neumann
16-bit words

Registers

R0: Program Counter
R1: Stack Pointer
R2: Status Register
R3: Constant Generator
R4-R15: General Use

Address Space

16-bit (MSP430)
20-bit (MSP430X, X2)

8.2 Memory Map

Unlike other embedded platforms, which like to put the interrupt vector table (IVT) at the beginning of memory, the MSP430 places it at the very end of the 16-bit address space, in Flash. (On smaller chips, this is the very end of Flash.)

Early on, Low RAM at `0x0200` would be the only RAM location, but as that region proved too small, a High RAM area was created at `0x1100`. For firmware compatibility reasons, the Low RAM area is mapped on top of the High RAM area.

Note that Flash grows down from the top of memory, while the RAM grows up. On chips with a 20-bit address space, an Extended Flash region sometimes grows upward from `0x10000`.

Additionally, there is an Info Flash area at `0x1000`. While there is nothing to stop an engineer from using this for code, the region is generally used for configuration settings. In many devices, chips arrive with this region pre-programmed to contain calibration settings for the internal clock.

In most devices, the BSL ROM at `0x0C00` contains a serial bootloader that allows the chip to be reprogrammed even after the JTAG fuse has been blown, and if you know the contents of the last 32 bytes of Flash—the Interrupt Vector Table—you can also read out the contents of memory.

8.3 Loading into a Disassembler

Back in the old days, reverse engineering MSP430 code meant using GNU `objdump` and annotating on pen and paper. Some folks would wrap these tools in Perl, or fill paper notebooks with cross-referencing, but thankfully that's no longer necessary.

Nowadays, IDA Pro has excellent support for the platform. If you have a legit license, just open the Intel Hex image of your target and specify MSP430 as the architecture. Memory locations can be had from the appropriate datasheets.

Radare2's MSP430 support is a bit less mature, and you should make sure to sanity check the disassembly wherever it looks suspect. Luckily, the Radare2 developers are frighteningly quick about fixing bugs, so both bugs that bothered us in the writing this article will likely be patched by the time you read this. For best results, always run Radare2 built from the latest Git repository,[10]—and rebuild it often.

One last tool, which is fast becoming obsolete with Radare2's support, is the MSPGCC project's single-line assembler.[11] It is particularly handy, though, when sanity-checking your own implementation of an assembler or disassembler.

There are no known decompilers for the MSP430, but with small code sizes and rather legible assembly we don't expect one to be necessary.

[10] `git clone https://github.com/radare/radare2`
[11] `http://mspgcc.sourceforge.net/assemble.html`

Start	End	Size	Use
0x0000	0x000F	16	Interrupt Control Registers
0x0010	0x00FF	240	8-bit Peripherals
0x0100	0x01FF	255	16-bit Peripherals
0x0200	0x09FF		Low RAM (Mirrored at 0x1100)
0x0C00	0x0FFF	1024	BootStrap Loader (BSL ROM)
0x1000	0x10FF	256	Info Flash
0x1100			High RAM
	0xFFFF		Flash
0x10000			Extended Flash

Table 1 – MSP430 and MSP430X Address Space

8.4 Basics of the Instruction Set

The language is relatively simple, but there are a few dialects that the locals speak. There are 27 action words (instructions), and then some additional emulated instructions which are assembled to one of the 27. Most of these 27 instructions have two forms—`.B` when they are working on an 8-bit byte, or `.W` if they want to tackle a 16-bit word. If someone tells you something and doesn't specify it, you can assume it's a word. If you're doing a byte operation in a register, be warned that the most-significant byte is cleared.

The three main types of core words are single-operand arithmetic, two-operand arithmetic, and jumps.

Our simple single-operands are RRC (1-bit rotate right and carry), SWPB (swap the bytes of the word), RRA (1-bit rotate right as arithmetic), SXT (sign-extend a byte into a word), PUSH (onto the stack), CALL (a subroutine, by pushing PC and then moving the new address to PC), and RETI (return from interrupt, restoring the Status Register SR and PC from stack).

Although these are all simple folk, they can, of course, be addressed in many different ways. If our register is *n*, then we see a few major types of addressing, all based off of the 'As' (for source) and 'Ad' (limited options for destination) fields:

R*n* Operate on the contents of register *n*.

@R*n* Operate on what is in memory at the address held in R*n*.

@R*n*+ Same as above, then increment the register by 1 or 2.[12]

x(R*n*) Operate on what is in memory at the address R*n* + x.

Wait, we just told you about an 'x'. Where did that come from?! In this case, it's an *extension word*, where the next 16-bit word after the extension defines x. In other words, it's an index off the base address held in R*n*.

If the register is `r0` (PC, the program counter), `r2` (SR, the status register), or `r3` (the *constant generator*), special cases apply. A common special case is to give you a constant, either -1, 0, 1, 2, 4, or 8.

Now we tackle two-operand arithmetic operations, most of which you should recognize from any other instruction set. The `mov`, `add`, `addc` (add with carry), `sub`, and `subc` instructions are all as you'd expect. `cmp` pretends to subtract the source from the destination to set status flags. `dadd` does a decimal addition with carry. `xor` and `and` are bitwise operations as usual. We have three that are a little unique: `bis` (logical OR), `bic` (dest = dest AND src), and `bit` (test bits of src AND dest).

Even with these instructions, though, we're still missing many favorite mnemonics that you'll see in disassembly. These are *emulated* instructions, actually implemented using other instruction(s).

For example, `br dst` (branch) is an emulated instruction. There is no branch opcode, but instead the `br` instructions are assembled as `mov dst, pc`. Similarly, `pop dst` is really `mov @SP+, dst`, and `ret` is really `mov @sp+, pc`. If these mappings make sense, you're all set to continue your travels!

Thus, when we need to get around this land of MSP430, we look not to the many jump types of x86, but instead to simpler patterns, where the only kind of jump operands are relative, and that's that.

[12]Here are the rules: Increment by two if registers `r0` or `r1`, or if `r4-r15` are used with a .W (2-byte) operand. Increment by 1 if `r4` to `r15` are used with a `.B` operand.

So `jmp`, the instruction says, but where to? The first three bits (`001`) mean jump, the next three specify the conditional, and the remaining ten are a signed offset. To get there, the ten bits are multiplied by two (left shifted) and then are added to the program counter, `r0`. Why multiply by two? Well, we have 16-bit word alignment, in the MSP430 land, unlike with those pesky x86 instructions you might be thinking of. Ordnung muß sein!

You might have noticed in your disassembly that even though we told you this was a fixed-width instruction set, some instructions are longer than one 16-bit word! One way this can happen is when using immediate values, which—much like those of the glorious PDP-11 of old—are implemented by dereferencing and incrementing the program counter. This way, the CPU will skip over the immediate value in its code fetch path just as it's fetching that same value as data.

And, finally, there are prefix instructions that have been added in MSP430X, the 20-bit extension of the MSP430. These prefix instructions go before the normal instruction, and you'll most commonly see them setting the upper four bits of the pointer in a 20-bit function call.

8.5 What's a Function, Anyways?

In x86 assembly, we're used to looking for function preambles to pick out the functions—but what do we look for in MSP430 code? We've already discussed finding the entry point of the program and those of other ISRs by looking at the vectors in the IVT. What about other functions?

In MSP430, all functions that are not ISRs will end with a RET instruction—which, as you recall, is actually a `MOV @SP+, PC`.

Compilers vary greatly in the calling conventions—as there is actually no fixed ABI. Usually, arguments get passed in `r12`, `r13`, `r14`, and `r15`. This, however, is by no means a requirement. MSP430 GCC uses `r15` for the first parameter and for most return value types, and `r14`, `r13`, and `r12` for the other parameters. Texas Instruments' Code Composer and the IAR compiler (after EW430 4.10A release) use `r12`, `r13`, `r14`, and `r15` and return in `r12`.

We recommend using an additional heuristic instead of looking for a function preamble format. In this heuristic, we assume that indirect calls are rare, and look for `br #addr` and `call #addr` instructions. Both of these consist of two 16-bit words, and whatever the `#addr` we extract from that second word, there's a good chance that it's the start of a function.

Using this logic, you should be able to find functions even in stripped images disassembled with `msp430-objdump`. A short script, or a good disassembler, should help automate the marking of these functions.

8.6 Making Sense of Interrupts

As with your (other) favorite microcontroller, our exploration of the code can be preempted by an interrupt.

If you don't like these getting in the way of your travels, they can be globally or individually disabled—well, except for the non-maskable interrupts (NMI).[13]

The MSP430 handles any interrupts set in priority order, and goes through the interrupt vector table to find the right interrupt service routine's (ISR) starting address. It hides away the current PC and SR on the stack, and runs the ISR. The ISR then returns, and normal execution continues.

If one thing is for certain, it's that `0xFFFE` is the system's reset ISR address (used on power-up, external reset, etc.), and that it has the highest priority.

If you have an elf32-msp430 formatted dump,[14] use `msp430-objdump dump.msp430 -DS` to get disassembly. Then locate the interrupt table at the end of memory:

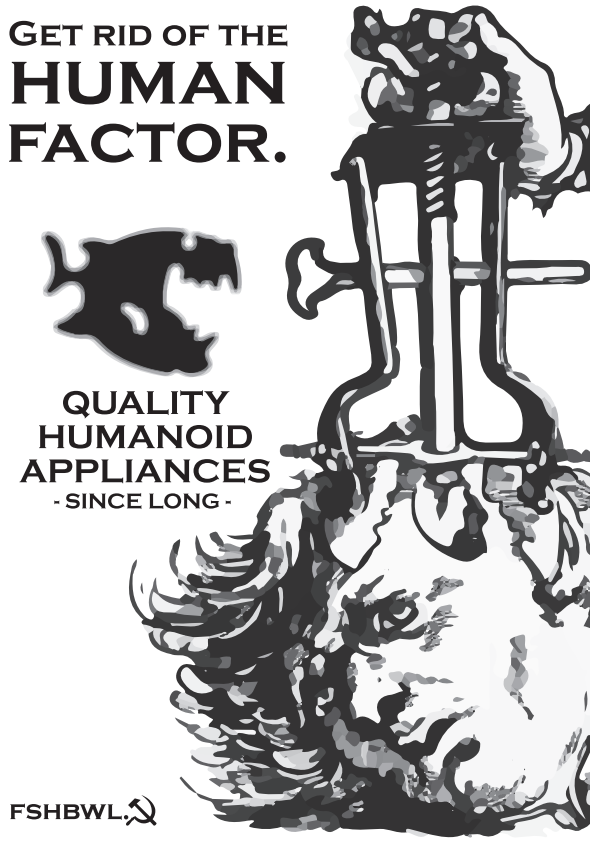
```
0000ffc0 <.sec2>:
 ffc0:   26 32   jn $-946 ;abs 0xfc0e
 ...
 fffc:   26 32   jn $-946 ;abs 0xfc4a
 fffe:   00 31   jn $+514 ;abs 0x200
```

We look at `0xFFFE` for the reset interrupt address, which is `0x3100` in this image. That's our entry point into the program, and you can see how it nicely lines up in the disassembly:

```
00003100 <.sec1>:
 3100: 31 40 00 31   mov    #12544, r1
 3104: 15 42 20 01   mov    &0x0120, r5
 3108: 75 f3         and.b  #-1,     r5
```

[13] Global disable is done by clearing the 'GIE' bit of the status register, `r2`.

[14] If not, use a command like `msp430-objcopy -I ihex -O elf32-msp430 dump.hex dump.msp430` to convert into one.



Maybe we want to look at some specific functionality that is triggered by an interrupt, for example incoming serial data. Looking in the MSP430F1611 data sheet, we find that `USART1` receive is a maskable interrupt at `0xFFE6`. If we look at the notated IVT in an example program (e.g., TinyOS's Printf program compiled for TelosB), we see addresses (in little endian) as shown here:

```
0000ffe0 <__ivtbl_16>:
    ffe0:       52 44   dac/dma
    ffe2:       52 44   i/o p2
    ffe4:       56 56   usart 1 tx
    ffe6:       d0 55   usart 1 rx
    ffe8:       52 44   i/o p1
    ffea:       94 4f   timer a3
    ffec:       76 4f   timer a3
    ffee:       52 44   adc12
    fff0:       52 44   usart 0 tx
    fff2:       52 44   usart 0 rx
    fff4:       52 44   watchdog timer
    fff6:       52 44   compartor a
    fff8:       d8 4f   timer b7
    fffa:       ba 4f   timer b7
    fffc:       52 44   nmi/etc
    fffe:       00 40   reset
```

We note that `0x4452` is used often. A quick look at this address shows that it is an empty IVT noting unused interrupts. Since we're interested in the `USART1` receive path, we follow `0x55d0` and see a large function that in turn calls another function—both nicely annotated, as we were working from an image with debug symbols:

```
000055d0 <sig_UART1RX_VECTOR>:
...
    563a:  b0 12 98 46   call #0x4698
...

00004698 <SerialP__rx_state_machine>:
...
```

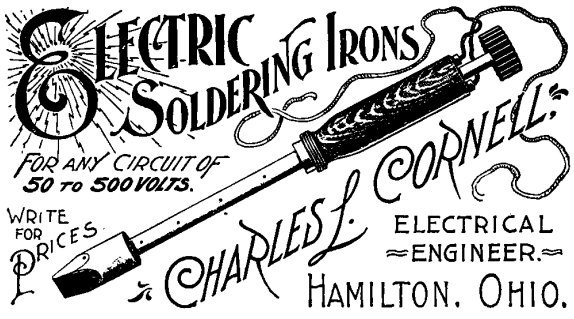
This technique of looking up your IVT entries and then working backwards to reverse engineer any handlers that correspond to the functionality you are interested in can help you avoid getting lost in reversing unimportant pieces of the code.

8.7 Sorting out Peripherals

If we're reversing some firmware, hopefully we have a target—often this can be data lines going to a radio or some peripheral that carry sensitive data.

Some peripherals are dealt with via interrupts, as shown above, but some are also either partially or totally handled via touching memory defined by the peripheral file map.

In particular, as an alternative to using interrupts, a program could simply poll for incoming data or a change in a pin's state. Likewise, setting up configurations for items such as the `USART` discussed above is done in the peripheral file map.



[15] Page 23 of `http://www.ti.com/lit/ds/symlink/msp430f1611.pdf`

Let us take the same file we used above, and look in the MSP430F1611 guide for the `USART1` in the peripheral file map.[15] Here we see the registers in the range from `0x0078` to `0x007F`. Let us search for a few of these in the image to demonstrate the applicability of this technique.

First, we look for `0x0078` (USART control), `0x0079` (transmit control), and `0x007A` (receive control). We find them all together in a function that is responsible for configuring the USART resource. A reader referencing the documentation will see the other control registers also updated:

```
4e8e <Msp430Uart...Configure...>:
...
4eb4:  c2 4e 78 00  mov.b r14,     &0x0078
4eb8:  d2 42 04 11  mov.b &0x1104,&0x0079
4ebc:  79 00
4ebe:  d2 42 05 11  mov.b &0x1105,&0x007a
4ec2:  7a 00
4ec4:  1e 42 00 11  mov   &0x1100,r14
4ec8:  c2 4e 7c 00  mov.b r14,     &0x007c
4ecc:  8e 10        swpb  r14
4ece:  4e 4e        mov.b r14,     r14
4ed0:  c2 4e 7d 00  mov.b r14,     &0x007d
4ed4:  d2 42 02 11  mov.b &0x1102,&0x007b
...
```

Whereas this approach can help you understand the settings to better sniff the serial bus physically, often you'd rather want to understand the actual data being written out. For this, we look for the peripheral holding the transmit buffer pointer—in our case at `0x007F`, according to the chip documentation. Searching for this in the disassembly leads us to a few interesting functions. Firstly, there's one that disables the UART, which fills this address with null bytes. This helps us confirm we're looking at the right address. We also see this address written to in the interrupt handler that we located in the previous section—and in a large function that ends up being a form of `printf` for writing out to this serial line.

As you can see, working backwards from the addresses located in the peripheral file map can help you quickly find functions of interest.

– — — – — — — — — — –

This guide is neither complete nor perfectly accurate. We told a few lies-to-children as all teachers do, and we omitted a dozen nifty examples that would've fit. Still, we hope that this will whet your appetite for working with the MSP430 architecture, and that, when you begin to work on the '430s, you can get your bearings quickly, jumping into the fun part of the journey with less hassle.

Also, for more MSP430 exploitation tricks, check out PoC||GTFO 2:5!