The Hunt for the Red Bootcode

Author : Andras Tantos

In the previous story I've detailed my odyssey to get reasonable quality data back from the Cray boot-disk that Chris have recovered. Now with the bits at hand, I can finally start poking around. So what's on it? I knew that the disk contained OS code for the mainframe. Chris said so. But what else is there? Or more importantly, what **should** be there?

In order to get a functional X-MP system, I need code for both the mainframe CPUs and the various IOPs (processors in the I/O subsystem). And there's a lot of them! The largest X-MPs (model 48 for example) have up to 8 processors to worry about: four IOPs and four main CPUs. While the boot-code we have on the disk belongs to a much smaller configuration – a model 14 with only 3 IOPs and a single main CPU – that's a lot of code to find.

The boot process

So how did these monsters boot? Here's what I could gather from the documentation I had:

1. Using a maintenance computer (which already got magically booted somehow) you loaded the boot image of the first IOP from tape into something called the 'Buffer memory'; a large, shared piece of memory in the IO subsystem.
2. After loading the boot code you released the first IOP from reset. Coming out of reset, one of the IOPs I/O channels is configured to DMA the first 128kByte from the Buffer memory to I/O memory (which is the main memory for the IOPs). When that was done, an interrupt was raised which started the execution of instructions on the IOP.
3. The first booted IOP was responsible for loading the boot-image into the other (up to three) IOPs and starting them.
4. Once the full IO subsystem was operational, you could finally load the boot-code into the mainframe, release one of it's CPUs from reset, which then was responsible for system initialization and starting up of the rest of the mainframe CPUs.

First glimpse at the disk

So, what I need was the following:

1. Boot code for the first IOP
2. Boot code for the rest of the IOPs
3. Boot code for the mainframe

When I've first looked into the image with a hex editor, the (almost) first ASCII string I saw (at offset 0x0001b6) was this:

IOP-0 KERNEL, VERSION 4.2.2, Sn302/25, * Leading Edge * 06/06/89 12:35:38

Wow!! This was easy, I though.

I didn’t have any documentation for the X-MP IOPs but I did for the Cray-1S version of them. I could only hope the two versions were similar. I quickly wrote up an IOP dis-assembler to be able to look at the content of the file.

The first attempts were rather discouraging. You see, what I expected at the beginning of the boot image was some executable code (the IOP started execution at address 0 after reset), doing some startup initialization, like clearing the registers, doing memory tests, setting up interrupts, the usual stuff.

Instead, the first few bytes looked like this:

0000000000: 01 BB 00 00 01 2C 00 00 ? 41 4D 41 50 00 00 00 00 ?» ?, AMAP
0000000010: 80 0B 00 00 00 03 02 00 ? 08 00 00 00 08 00 00 11 €? ?? ? ? ?
0000000020: 00 3F 00 00 00 9B 00 C9 ? 00 30 00 40 00 06 00 20 ? › É 0 @ ?
0000000030: 30 00 00 20 00 01 00 1C ? 00 D6 00 00 00 FF 01 10 0 ? ? Ö ÿ??
0000000040: 00 27 80 19 80 19 00 00 ? 00 00 80 1B 80 1B 00 00 '€?€? €?€?
0000000050: 00 00 80 09 80 08 80 1C ? 80 1C 00 00 00 00 86 1F €?€?€?€? †?
0000000060: 86 1F 00 00 00 00 00 00 ? 00 00 00 00 00 00 00 00 †?
0000000070: 00 00 00 00 00 00 80 13 ? 80 13 80 13 80 13 80 13 €?€?€?€?€?
0000000080: 80 13 80 13 80 13 00 D0 ? 00 14 00 20 12 00 00 40 €?€?€? Ð ¶ ? @
0000000090: 00 01 00 4A 00 D6 00 00 ? 01 10 01 19 00 27 80 18 ? J Ö ???? '€?
00000000A0: 80 18 00 00 00 00 80 1B ? 80 1B 80 1D 80 1D 00 00 €? €?€?€?€?
00000000B0: 00 00 80 02 80 02 80 02 ? 00 00 80 2A 80 2A 80 2A €?€?€? €*€*€*
00000000C0: 00 00 80 02 80 02 80 02 ? 00 00 00 00 00 00 00 00 €?€?€?
00000000D0: 00 00 00 00 00 00 80 13 ? 80 13 00 00 00 00 00 00 €?€?
00000000E0: 00 00 00 20 00 08 00 20 ? 10 00 00 60 00 01 00 78 ? ? ` ? x
00000000F0: 00 D6 00 00 00 00 00 00 ? 00 27 80 18 80 18 80 19 Ö '€?€?€?
0000000100: 80 19 00 00 00 00 00 00 ? 00 00 00 00 00 00 00 00 €?
0000000110: 00 00 00 00 00 00 00 00 ? 00 00 00 00 00 00 00 00
0000000120: 00 00 00 00 00 00 00 00 ? 00 00 00 00 00 00 00 00
0000000130: 00 00 80 13 80 13 00 00 ? 00 00 00 00 00 00 13 88 €?€? ?ˆ
0000000140: 00 10 00 20 28 00 00 20 ? 00 01 00 A6 00 D6 00 00 ? (? ¦ Ö
0000000150: 01 1C 01 25 00 27 80 18 ? 80 18 80 19 80 19 00 00 ???% '€?€?€?€?
0000000160: 00 00 00 00 00 00 00 00 ? 00 00 80 21 80 21 80 21 €!€!€!
0000000170: 80 21 00 00 00 00 00 00 ? 00 00 00 00 00 00 00 00 €!
0000000180: 00 00 00 00 00 00 00 00 ? 00 00 00 00 00 00 80 13 €?
0000000190: 80 13 00 00 00 00 00 00 ? 00 00 00 01 00 10 00 50 €? ? ? P
00000001A0: 00 00 00 02 00 11 00 0A ? 00 12 00 54 00 55 00 56 ? ? ? ? T U V
00000001B0: 00 00 FF FF 0A 0D 49 4F ? 50 2D 30 20 4B 45 52 4E ÿÿ??IOP-0 KERN
00000001C0: 45 4C 2C 20 56 45 52 53 ? 49 4F 4E 20 34 2E 32 2E EL, VERSION 4.2.
00000001D0: 32 2C 20 20 53 6E 33 30 ? 32 2F 32 35 2C 20 2A 20 2, Sn302/25, *
00000001E0: 4C 65 61 64 69 6E 67 20 ? 45 64 67 65 20 2A 20 20 Leading Edge *

00000001F0: 30 36 2F 30 36 2F 38 39 ? 20 20 31 32 3A 33 35 3A 06/06/89 12:35:
0000000200: 33 38 00 00 00 00 00 00 ? 43 4F 4E 46 49 47 00 00 38 CONFIG
0000000210: 00 00 00 00 41 55 54 4F ? 44 4D 50 20 4F 4E 00 00 AUTODMP ON
0000000220: 00 00 00 00 20 00 00 00 ? 00 00 43 4F 4E 46 49 47 CONFIG

The disassembly didn't show anything meaningful either:

0x0000 (0x000000) 0x01BB- f:00000 d: 443 | PASS
0x0001 (0x000002) 0x0000- f:00000 d: 0 | PASS
0x0002 (0x000004) 0x012C- f:00000 d: 300 | PASS
0x0003 (0x000006) 0x0000- f:00000 d: 0 | PASS
0x0004 (0x000008) 0x414D- f:00040 d: 333 | C = 1, io 0515 = DN
0x0005 (0x00000A) 0x4150- f:00040 d: 336 | C = 1, io 0520 = DN
0x0006 (0x00000C) 0x0000- f:00000 d: 0 | PASS
0x0007 (0x00000E) 0x0000- f:00000 d: 0 | PASS
0x0008 (0x000010) 0x800B- f:00100 d: 11 | P = P + 11 (0x0013), C = 0
0x0009 (0x000012) 0x0000- f:00000 d: 0 | PASS
0x000A (0x000014) 0x0003- f:00000 d: 3 | PASS
0x000B (0x000016) 0x0200- f:00001 d: 0 | EXIT
0x000C (0x000018) 0x0800- f:00004 d: 0 | A = A > 0 (0x0000)
0x000D (0x00001A) 0x0000- f:00000 d: 0 | PASS
0x000E (0x00001C) 0x0800- f:00004 d: 0 | A = A > 0 (0x0000)
0x000F (0x00001E) 0x0011- f:00000 d: 17 | PASS
0x0010 (0x000020) 0x003F- f:00000 d: 63 | PASS
0x0011 (0x000022) 0x0000- f:00000 d: 0 | PASS
0x0012 (0x000024) 0x009B- f:00000 d: 155 | PASS
0x0013 (0x000026) 0x00C9- f:00000 d: 201 | PASS
0x0014 (0x000028) 0x0030- f:00000 d: 48 | PASS
0x0015 (0x00002A) 0x0040- f:00000 d: 64 | PASS
0x0016 (0x00002C) 0x0006- f:00000 d: 6 | PASS

This is gibberish. The first four instructions are 'PASS', which stands for no operation. The next two set the 'C' (carry bit) depending on the 'done' bit of certain I/O channels, but the channel numbers 0515 and 0520 are invalid.

OK, so things aren't as simple. Maybe there's some header before the boot-code? Maybe the first few bytes describe file size, file name, attributes of sorts, and the real code starts afterwards?

Still no dice, the code further down in the disassembly is just as invalid:

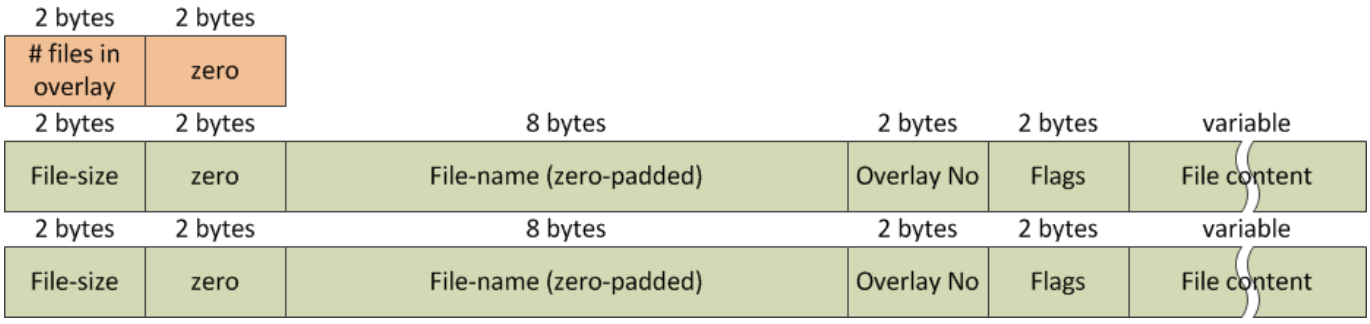
0x0070 (0x0000E0) 0x0000- f:00000 d: 0 | PASS
0x0071 (0x0000E2) 0x0020- f:00000 d: 32 | PASS
0x0072 (0x0000E4) 0x0008- f:00000 d: 8 | PASS
0x0073 (0x0000E6) 0x0020- f:00000 d: 32 | PASS
0x0074 (0x0000E8) 0x1000- f:00010 d: 0 | A = 0 (0x0000)

```
0x0075 (0x0000EA) 0x0060- f:00000 d: 96  | PASS
0x0076 (0x0000EC) 0x0001- f:00000 d: 1   | PASS
0x0077 (0x0000EE) 0x0078- f:00000 d: 120 | PASS
0x0078 (0x0000F0) 0x00D6- f:00000 d: 214 | PASS
0x0079 (0x0000F2) 0x0000- f:00000 d: 0   | PASS
0x007A (0x0000F4) 0x0000- f:00000 d: 0   | PASS
0x007B (0x0000F6) 0x0000- f:00000 d: 0   | PASS
0x007C (0x0000F8) 0x0027- f:00000 d: 39  | PASS
0x007D (0x0000FA) 0x8018- f:00100 d: 24  | P = P + 24 (0x0095), C = 0
0x007E (0x0000FC) 0x8018- f:00100 d: 24  | P = P + 24 (0x0096), C = 0
0x007F (0x0000FE) 0x8019- f:00100 d: 25  | P = P + 25 (0x0098), C = 0
0x0080 (0x000100) 0x8019- f:00100 d: 25  | P = P + 25 (0x0099), C = 0
0x0081 (0x000102) 0x0000- f:00000 d: 0   | PASS
0x0082 (0x000104) 0x0000- f:00000 d: 0   | PASS
0x0083 (0x000106) 0x0000- f:00000 d: 0   | PASS
0x0084 (0x000108) 0x0000- f:00000 d: 0   | PASS
```

There are two options here: either this is in fact the boot-code, but the IOPs are significantly different between the Cray-1S and the X-MP systems, or this is not the boot code I'm looking at.

The Overlay

Still! It says so in the string that it is the IOP Kernel. So, what could be it? Both Chris and I have noticed that there is some structure to the data at the beginning of the disk. So I started re-constructing it to figure out what this mass of data could be. Later – in fact much later – I've figured out the structure to be the following:



The Overlay Number field contains the index of the overlay and it's MSB bit set if the overlay contains data. If it's code, the MSB bit is cleared. I don't know the meaning of the Flags field at the moment. The file-size field includes the size of the header as well as the size of the file content and is measured in 16-bit quantities.

Now we're getting somewhere! I can walk this chain, split the files into individual units and try to find the real boot code! Well, it turns out, the structure contains more than 400 files, none of

them bigger than a few kilobytes. Nothing that's even close to the expected 128k in size.

One file caught my eye though. It's called OVLNUM. It contains a set of strings, which seem to describe overlay files. So, I've written a program that paired up the descriptions with the file names. You can look at the fill list here, but here's an excerpt:

> SDMPA: Dump executing exchange packages
> SDMPB: Dump Cray registers
> SDMPC: Dump cluster registers
> SDMPD: Dump SSD from a Cray 1/S
> SDMPE: Dump SSD from a Cray X-MP
> SDMPYA: Dump exchange packages, Y-MP
> SDMPYB: Dump Cray registers, Y-MP
> SDMPYC: Dump cluster registers, Y-MP
> SDMPYD: Dump SSD from a Cray Y-MP
> MFCHK: Mainframe confidence test
> MFCHKY: Mainframe confidence test, Y-MP
> AMAP: Configuration map
> ADEM: Cray interface demon
> AMSG: Inter-IOP communications demon
> BEGIN: IOS initialization
> BTD: Binary to decimal ASCII conversion
> BTH: Binary to hex ASCII conversion
> BTO: Binary to octal ASCII conversion
> CALL: Kernel command processor
> CDEM: Cray interface demon
> CLKSNC: Synchronize day clock
> CLOCK: Clock demon
> CONFIG: Configuration display
> CRAY: Cray interface initialization

It all started making sense. This is in fact an overlay file for the IOPs. It contains both code and data in many small files. The files containing code – looking at the disassembly – started to be reasonable as well:

```
0x0000 (0x000000) 0x1000-      f:00010 d: 0   | A = 0 (0x0000)
0x0001 (0x000002) 0x2922-      f:00024 d: 290 | OR[290] = A
0x0002 (0x000004) 0x2118-      f:00020 d: 280 | A = OR[280]
0x0003 (0x000006) 0x1602-      f:00013 d: 2   | A = A - 2 (0x0002)
0x0004 (0x000008) 0x8003-      f:00100 d: 3   | P = P + 3 (0x0007), C = 0
0x0005 (0x00000A) 0x8402-      f:00102 d: 2   | P = P + 2 (0x0007), A = 0
0x0006 (0x00000C) 0x71DE-      f:00070 d: 478 | P = P + 478 (0x01E4)
0x0007 (0x00000E) 0x2118-      f:00020 d: 280 | A = OR[280]
0x0008 (0x000010) 0x85DC-      f:00102 d: 476 | P = P + 476 (0x01E4), A = 0
0x0009 (0x000012) 0x2119-      f:00020 d: 281 | A = OR[281]
```

```
0x000A (0x000014) 0x85DA-       f:00102 d: 474 | P = P + 474 (0x01E4), A = 0
0x000B (0x000016) 0x2119-       f:00020 d: 281 | A = OR[281]
0x000C (0x000018) 0x1603-       f:00013 d: 3   | A = A - 3 (0x0003)
0x000D (0x00001A) 0x8003-       f:00100 d: 3   | P = P + 3 (0x0010), C = 0
0x000E (0x00001C) 0x8402-       f:00102 d: 2   | P = P + 2 (0x0010), A = 0
0x000F (0x00001E) 0x71D5-       f:00070 d: 469 | P = P + 469 (0x01E4)
0x0010 (0x000020) 0x211A-       f:00020 d: 282 | A = OR[282]
0x0011 (0x000022) 0x1610-       f:00013 d: 16  | A = A - 16 (0x0010)
0x0012 (0x000024) 0x81D2-       f:00100 d: 466 | P = P + 466 (0x01E4), C = 0
0x0013 (0x000026) 0x211A-       f:00020 d: 282 | A = OR[282]
0x0014 (0x000028) 0x161F-       f:00013 d: 31  | A = A - 31 (0x001F)
0x0015 (0x00002A) 0x8003-       f:00100 d: 3   | P = P + 3 (0x0018), C = 0
0x0016 (0x00002C) 0x8402-       f:00102 d: 2   | P = P + 2 (0x0018), A = 0
0x0017 (0x00002E) 0x71CD-       f:00070 d: 461 | P = P + 461 (0x01E4)
0x0018 (0x000030) 0x211F-       f:00020 d: 287 | A = OR[287]
0x0019 (0x000032) 0x1E00-0x0200 f:00017 d: 0   | A = A - 512 (0x0200)
0x001B (0x000036) 0x8003-       f:00100 d: 3   | P = P + 3 (0x001E), C = 0
0x001C (0x000038) 0x8402-       f:00102 d: 2   | P = P + 2 (0x001E), A = 0
```

It didn’t make too much sense at first what was actually going on in the code, but it at least looked like a reasonable instruction stream, with no undefined instruction codes, obviously invalid parameters or jump targets.

By the way, I’ve found a second copy of the overlay structure on the disk. It’s not identical to the first one, but pretty close. It seems that some parameters got changed and the image got re-compiled at some point. It started at 0x4fb000.

So far so good. It’s nice to have two sets of overlays, but it doesn’t help much unless I can get the rest of the image. So I kept looking.

The IOP boot image

Just before the second overlay (at offset 0x4f1000 for the curious) I stumbled upon a binary segment, this time without any obvious header structure. It is only about 40k long, much shorter than what is expected from the IOP boot image (should be 128k), but let’s take a look! I’ve extracted the section of the image with the binary data and loaded it to the disassembler. This is what I’ve seen:

```
0x0000 (0x000000) 0x7003-       f:00070 d: 3   | P = P + 3 (0x0003)
0x0001 (0x000002) 0x7C34-       f:00076 d: 52  | R = OR[52]
0x0002 (0x000004) 0x0007-       f:00000 d: 7   | PASS
0x0003 (0x000006) 0x1000-       f:00010 d: 0   | A = 0 (0x0000)
0x0004 (0x000008) 0x2800-       f:00024 d: 0   | OR[0] = A
0x0005 (0x00000A) 0x7A00-0x41E4 f:00075 d: 0   | P = OR[0]+16868 (0x41E4)
```

Let’s see what it does line by line: the first instruction is a jump, targeting address 3. So it jumps over the next two instructions, which would rely on the value of the operand register 52 (OR[52]). After that, it zeros the accumulator (A = 0), loads this zero value into operand register 0 (OR[0] = A) and jumps to address 0x41e4 (P = OR[0]+16868).

This is very promising. I’m not sure why it jumps over a rather large portion at the very end, but at least it makes sense as a boot code. For example, it doesn’t make assumptions about the value of the registers (OR[x] is one of the 512 operational registers).

Here’s where it jumps to:

```
0x41E4 (0x0083C8) 0x100A- f:00010 d: 10  | A = 10 (0x000A)
0x41E5 (0x0083CA) 0x2801- f:00024 d: 1   | OR[1] = A
0x41E6 (0x0083CC) 0x21FF- f:00020 d: 511 | A = OR[511]
0x41E7 (0x0083CE) 0x3801- f:00034 d: 1   | (OR[1]) = A
0x41E8 (0x0083D0) 0x1001- f:00010 d: 1   | A = 1 (0x0001)
0x41E9 (0x0083D2) 0x5800- f:00054 d: 0   | B = A
0x41EA (0x0083D4) 0x5000- f:00050 d: 0   | A = B
0x41EB (0x0083D6) 0x8405- f:00102 d: 5   | P = P + 5 (0x41F0), A = 0
0x41EC (0x0083D8) 0x1000- f:00010 d: 0   | A = 0 (0x0000)
0x41ED (0x0083DA) 0x6800- f:00064 d: 0   | OR[B] = A
0x41EE (0x0083DC) 0x5C00- f:00056 d: 0   | B = B + 1
0x41EF (0x0083DE) 0x7205- f:00071 d: 5   | P = P - 5 (0x41EA)
0x41F0 (0x0083E0) 0x1001- f:00010 d: 1   | A = 1 (0x0001)
```

What happens here is the following: first, the value 10 is stored in OR[1]. Than, the value of OR[511] (the last operand register) is stored at the address pointed to by OR[1], which is 10, and is consequently in the memory region we’ve jumped over. After that, we set up a loop that goes from 1 to 511 (you’ll see in a minute why) with the loop counter in the ‘B’ register. In the body of the loop, we simply set OR[B] to zero. The reason the loop terminates after B reaches 511 is that B is a 9-bit register, so it rolls over to 0. After the jump happens at address 0x41EF for this last time to 0x41EA where we see A = B. So A gets zeroed out as well. The jump at address 0x41EB is a conditional one, it only executes if A = 0. The condition is now true, so it jumps to 0x41F0, past the end of the loop.

Things start to make sense. Apart from the strange saving of OR[511] to address 10 all the code did so far was to set the processor into a known state. This in fact feels very much like the boot image!!

Great. So we have one IOP boot image. Two more to go (remember, this Cray configuration had three IOPs). Except they are nowhere to be found. I’ve searched and searched the disk over and over again, but I couldn’t find anything.

Well, at least I can boot one of the IOPs, and see where the boot process takes me.

So, what else is on the disk?

The COS boot images

There are two of them. One at 0x09a000 and another one at 0x2d0000. These are two ‘INIT’ images and the actual boot image is preceded by a header. This file format is described in the CAL (Cray Assembler) reference manual as ‘absolute binary output’. It contains a PDT header followed by a single TXT block:

PDT header

Word	Bit	Description
0	63-60	Table code (15)
	59-36	Word count (7)
	35-22	Number of external names (0)
	21-8	Number of entry names * 2 (2)
	7-0	Number of blocks referenced * 2 (2)
1	63-0	Program name (left adjusted, zero-filled)
2	23-0	Program length
3	63-0	Entry point name (left adjusted, zero-filled)
4	63-0	Entry point address
5	63-0	Date (DD/MM/YY)
6	63-0	Time (HH:MM:SS)

TXT header

Word	Bit	Description
0	63-60	Table code (14)
	59-36	Word count (program length + 1)
1-N	63-0	Binary machine code

The boot process for the mainframe CPUs starts with an ‘exchange sequence’ with the XA (exchange address) register set to 0. The exchange sequence is the Cray term for a partial context switch: much of the processor state is read from memory (into temporary storage), then the old state is dumped into the same memory location. Finally, the newly fetched state is made active, completing the context switch. Of course on reset the processor state is largely non-deterministic, but the new state that’s picked up from memory should clean things up. This means that the beginning of the boot image should contain a valid ‘exchange packet’. The exchange packet layout is the following:

And the dump of the first few bytes of the image (after the header) looks like this:

```
0x000000 00 00 00 7f f0 00 00 00 ; P = 1FFC.p0         A0 = 0
0x000001 00 00 00 00 00 00 00 00 ; IBA = 0x0           A1 = 0
0x000002 00 00 00 3e 01 00 00 00 ; ILA = 0x3e00        A2 = 0
0x000003 00 00 00 00 00 00 00 00 ; XA = 0 VL = 0 F = 0 A3 = 0
0x000004 00 00 00 00 00 00 00 00 ; DBA = 0             A4 = 0
```

```
0x000005 00 00 3f ff e0 00 00 00 ; DLA = 0x3fffe0      A5 = 0
0x000006 00 00 00 00 00 00 00 00 ;                     A6 = 0
0x000007 00 00 00 00 00 00 00 00 ;                     A7 = 0
0x000008 00 00 00 00 00 00 00 00 ;                     S0 = 0
0x000009 00 00 00 00 00 00 00 00 ;                     S1 = 0
0x00000a 00 00 00 00 00 00 00 00 ;                     S2 = 0
0x00000b 00 00 00 00 00 00 00 00 ;                     S3 = 0
0x00000c 00 00 00 00 00 00 00 00 ;                     S4 = 0
0x00000d 00 00 00 00 00 00 3e 00 ;                     S5 = 0x3e00
0x00000e 00 00 00 00 00 00 00 00 ;                     S6 = 0
0x00000f 00 00 00 00 00 00 00 00 ;                     S7 = 0
```

This looks very reasonable. The ILA and DLA registers are set to the expected size (these are the size of the code and data segments) and the program counter (P) is set to somewhere in the middle of the code segment. Almost all other registers are set to 0 (A and S registers). It even shows the reset vector being the same as the entry point indicated in the PDT header. Here's what the reset code looks like:

```
0x001FFC:p0 RT 0            ; Zero out the real-time-clock
0x001FFC:p1 0x0008C5 S6     ; Save restart reason
0x001FFC:p3 IP 0            ; Clear received inter-processor interrupt
0x001FFD:p0 A2 0x000008     ; Use channel 8. That's the MIOP input channel

; Store the channel-address register for CH8 at location 0x8c7
0x001FFD:p1 A6 CA,A2        ; A6 = channel address register for channel 8
0x001FFD:p2 A0 A6
0x001FFD:p3 JAZ 0x001FFF:p1 ; Jump if A0 == 0
0x001FFE:p1 0x0008C7 A6
0x001FFE:p3 J 0x001FFF:p3
0x001FFF:p1 0x0008C7 S7

; Set up CH8 to receive 30 bytes and store them at address 0x8DA.
; Also store the channel buffer setup in
; 0x920 (CAT, Channel Address Table) and
; 0x982 (CLT, Channel Limit Table) tables respectively.
0x001FFF:p3 A2 0x000008
0x002000:p0 A6 0x0008DA
0x002000:p2 A7 0x0008E0
0x002001:p0 CL,A2 A7
0x002001:p1 0x000982,A2 A7
0x002001:p3 CA,A2 A6
0x002002:p0 0x000920,A2 A6

; Update entry 8 in the CHT (CHannel Table) table
0x002002:p2 A7 0x00095C
0x002003:p0 A7 A7+A2
```

0x002003:p1 S5 0x00008917
0x002003:p3 S6 0x000000,A7
0x002004:p1 S7