



## PCM hacking 101 - The step by step approach - Third Generation F-Body Message Boards

By: **dimented24x7**  
Moderator

I've been getting some requests to do something like this, so here it goes:

Here at TGO, there has been a new trend to update to the later OBD-II based PCMs to gain additional features like MAF/MAP fueling, coil per cyl. Ignition, SFI, etc. One thing that is missing for the PCMs, though, is good DIY support like there is for the 7730/7749 ECMs. Part of the problem is a general lack of understanding of how to work with these. The later PCMs are not impossible to work with, but will require a different approach due to the 32-bit nature of these, as well as the enhanced vehicle communications functions that are mandated as part of the OBD-II package. The steps needed to start work with the PCM can be broken down as follows:

- Getting a good stock binary to work with
- Using TunerPRO 2D and 3D hex viewers to find calibration tables
- Documentation: What you need to read and know before starting
- Disassembling the binary into Moto 68332 Assembly
- Using the J1850 communication bus/Datalink Controller and SAE standards to unlock the basic engine parameters and hardware I/O
- Working with the code: Finding tables and constants using the assembly

The PCM I will be using as an example is a '279 Ebay clone that was used in the 98/99 vortec pickups. This is more simplified than the '411 as it still shares some of its routines and code with the \$0D/\$0E TBI PCMs, but the same approach should still apply to the LS1 based computers.

## **Getting a good binary to work with:**



Lets get started: So you bought your PCM and now you want to get a bin for it, but a bin isn't available, and you don't have any OBD scanners available to download the bin. All is not lost... Within most GM PCMs, there is a standard intel 28FXXX-BB flash chip that can be read in some commercially available EEPROM readers such as the PP-II. Unlike the earlier ECMs with EPROMs or MEMCALs, though, the flash chip is a SOP surface mount IC that is hard soldered to the PCMs PCB. You will need to take a different approach to remove the IC and get it into a reader.

The first step is to get the right tools. One tool that you will need is an SMD rework station. This is a device that has an air pump and closed loop controller in it that can produce controlled air temperatures for working with SMD ICs (Fig. 1). These are available as professional units, or as low priced hobby grade units sourced from Asia. The hobby grade units aren't too precise, but are good enough for our purposes. Some other tools to have are a solder pen, needle-nose pliers, wire cutters, hobby knife, and desoldering brade (Fig. 2).



The next step is to remove the PCB from the PCMs enclosure. First, locate a suitable work area, such as a table or large sized desk with adequate lighting. Also, if your work area is in a dry climate, a space with A/C or anywhere with a low relative humidity, it's a good idea to wear a grounded wrist strap to prevent static electricity build-up. Static discharges will not only damage the flash chip, but the other ICs as well, rendering the PCM useless.

You will next need to open the PCMs enclosure. There are typically torx head screws holding the PCMs covers together. Remove these, and carefully remove the outer cover + heat sink, setting it, the screws, and weather seal aside. Next, get a damp sponge and wet paper towel ready for removing the flash chip. The sponge should be placed underneath the PCB where the flash chip is. This is to keep the bottom of the PCB cool so the surface mount components on that side will not become dislodged by the heat. The paper towel should be placed over the top of the flash chip (a generic IC is shown as an example) to keep it cool (as shown in Fig. 3). Set your rework station for low to medium flow and for an output temp of 300 deg C. Carefully slip your hobby knife under one side of the flash chip and heat one set of the pins in a back and fourth motion to melt the solder and loosen that side while carefully prying the chip up with your hobby knife (Fig. 3). Once the pins get hot enough, that side of the chip will come up. Grasp the chip carefully with your pliers, and repeat the process for the other side of the chip. Once those pins have been loosened, the IC will come away completely (Fig. 4). After that, you will want to clean the solder pads off for either resoldering the chip, or installing a socket. This is accomplished by placing the desoldering braid under the heated soldering pen, and working your way across the solder pads to lift up the excess solder (Fig. 5).



Fig. 3





Once you have the chip desoldered, you can now place it in your reader. One popular reader that can be used for this purpose is the xtronic pocket programmer. For around \$350, the reader and needed 28FXXX adaptor can be purchased from xtronic.com (Fig. 6). Connect the reader to your PC and launch the reader's software. Select the appropriate chip from the device menu, in this case, a 28F400 flash chip. The programmer interface should also have an option for how the chip should be read. Select the 8/16 reads as Hi/Lo. Insert the flash chip into the reader, and click move device, or read device. DO NOT click program or erase device, as your reader will ERASE or ALTER the flash chips contents, rendering the binary useless. It's a good idea once the read is complete to verify the downloaded buffer to the flash chip. As a final check, save the binary and open it up in either hex workshop or hex editor neo, and search for 4E75 (this is the machine code for Return from Subroutine). This should come up hundreds of times. If not then either the chip didn't read correctly, or the Hi/Lo order is set incorrectly. Once you have your bin, then you can socket the PCM for development. There are a variety of sockets available, such as surface mount clamshell types, or spring loaded ZIF sockets. In Fig. 7, the PCM has been socketed with an external socket to allow for the use of AMD 29F400-BB flash chips. This allows one to edit the bin as needed without having to worry about the need for re-flashing the PCM. The stock flash chip can also be re-programmed, but it's a good idea to get multiple flash chips rather than continuously use just one.

Coming up next: Using TunerPRO 2D and 3D hex viewers to find calibration tables



Fig. 6





Fig. 7

## Using TunerPRO 2D and 3D hex viewers to find calibration tables

This section will be fairly short. The main reason is that only a minority of the tables can be conclusively found using just a hex viewer. Some tables like the composite cylinder volume/engine VE tables are relatively featureless in a 3D hex viewer, leaving some doubt as to what the table is without supporting information from the actual code. Never the less, some tables can be readily found using a 2D or 3D hex viewer present in programs like TunerPro and WinOLS. Before I did any hacking, I immediately opened the bin I was working with in TunerPro's hex viewers, and found a few tables right away. Because of this, it's worth mentioning this technique as it can pinpoint some items before you even need to do any hard work.

The two examples I have chosen are the MAF flowratre table and the spark tables. These tables have distinctive shapes, and can be easily ID'd. Because the flash chip is broken into sectors, you can narrow down your search for tables. The engine calibration section typically runs from \$8000 - \$F000. Shown below is a breakdown of the calibration sectors within my '279:

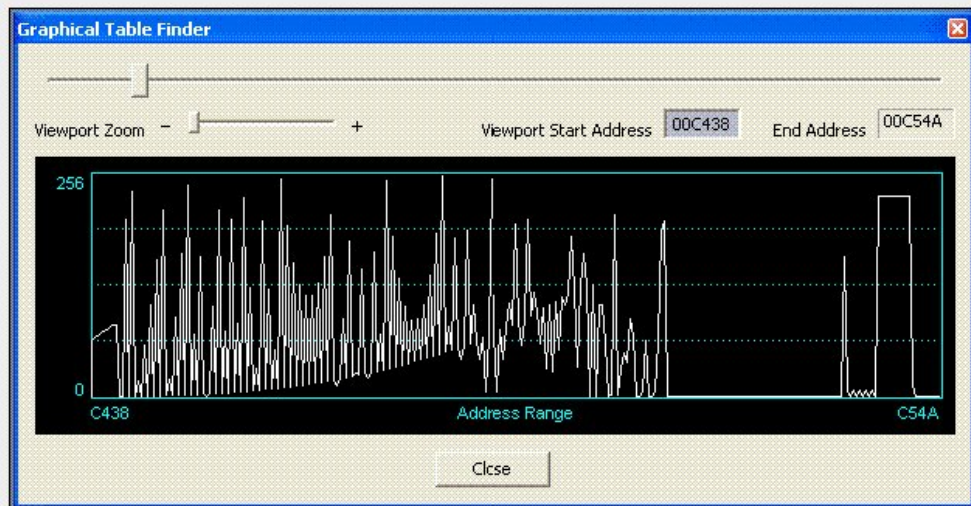
Code:

```
;
;~~~~~
;-ROM sectors to perform checksum on
;~~~~~
;
;-Sector #0: Operating System
;
LBL_$2000C:
  DC.L $00020000  ;Start of checksum area
LBL_$20010:
  DC.L $00066F63  ;End of checksum area
;
;-Sector #1: Engine
;
LBL_$20014:
  DC.L $00008000  ;Start of checksum area
LBL_$20018:
  DC.W $0000E651  ;End of checksum area
;
;-Sector #2: Fuel System
;
LBL_$2001C:
  DC.L $0000E652  ;Start of checksum area
LBL_$20020:
  DC.L $0000EF93  ;End of checksum area
;
;-Sector #3: System
;
LBL_$20024:
  DC.L $0000EF94  ;Start of checksum area
LBL_$20028:
  DC.L $0000F0D1  ;End of checksum area
;
;-Sector #4: Speedometer
```

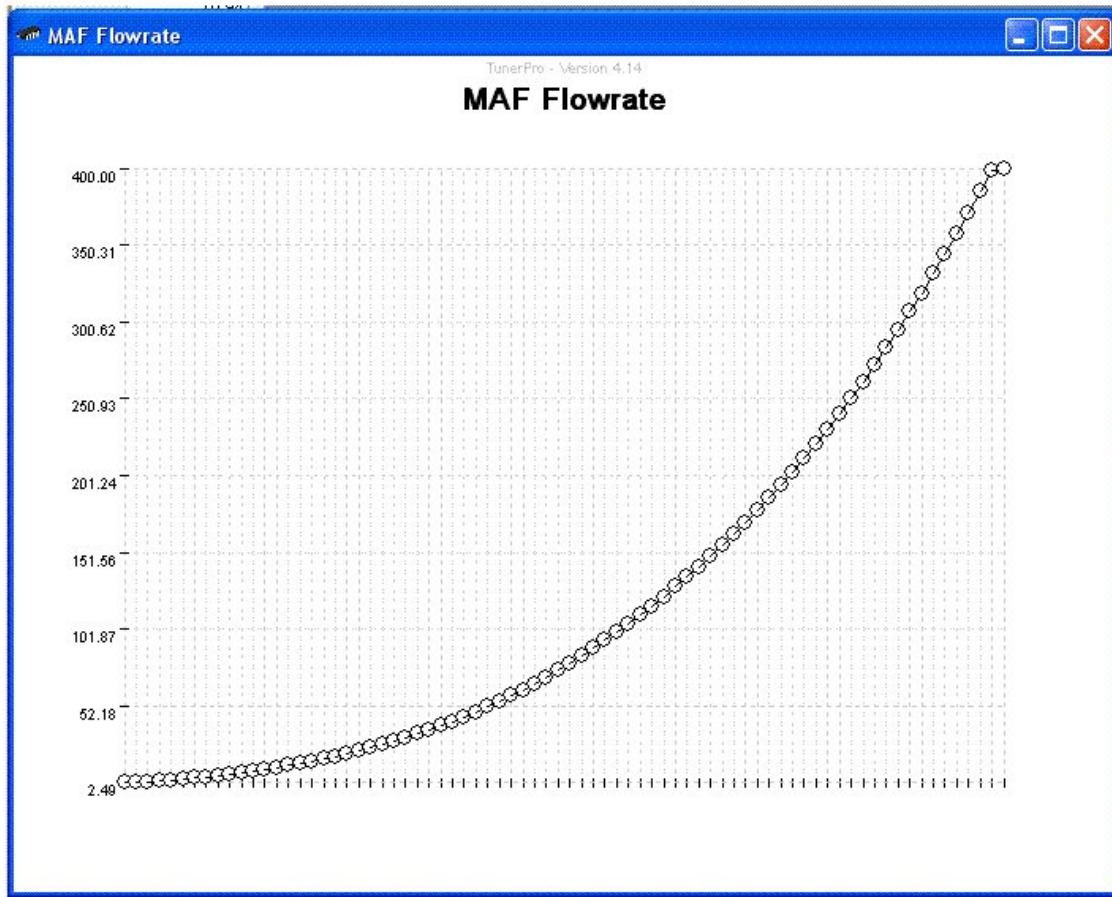
```
;
LBL_$2002C:
  DC.L $0000F0D2  ;Start of checksum area
LBL_$20030:
  DC.L $0000F109  ;End of checksum area
;
;-Sector #5: HVAC
;
LBL_$20034:
  DC.L $000148FE  ;Start of checksum area
LBL_$20038:
  DC.L $0001491D  ;End of checksum area
;
;-Sector #6: Transmission
;
LBL_$2003C:
  DC.L $0000F10A  ;Start of checksum area
LBL_$20040
  DC.W $000148FD  ;End of checksum area
```

As seen from these sections, this is where you should concentrate on.

Because most if not all GM PCMs use the hybrid SD/MAF fueling after 95 or so, the base MAF flowrate table is present in the calibration section of the bin. The MAF table has a very unique shape due to the fact that the flowrate increases in a parabolic fashion. This gives the MAF table a parabolic or exponential shape. Even though TunerPRO can only show the 2D tables as 8-bit, the MAF table still shows up when you scan for it as seen below.

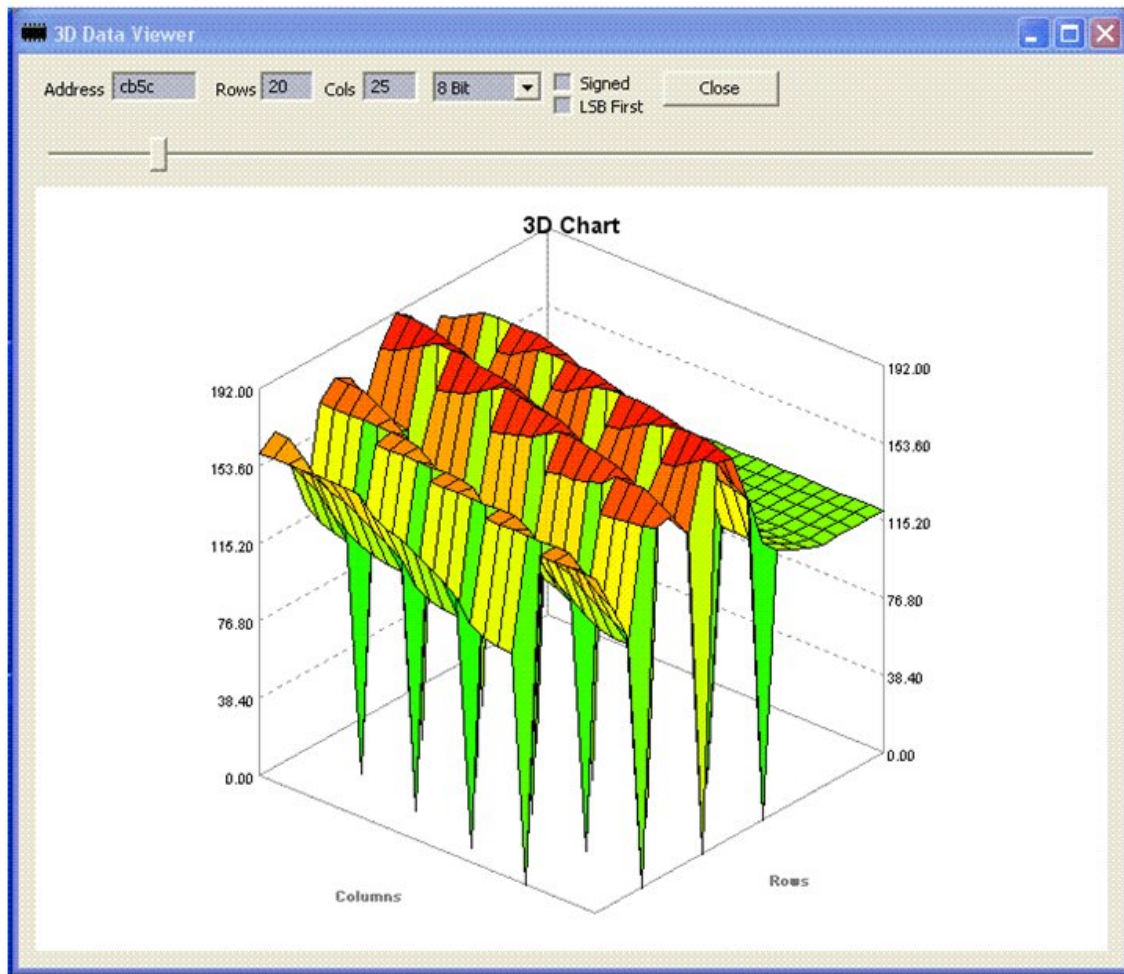


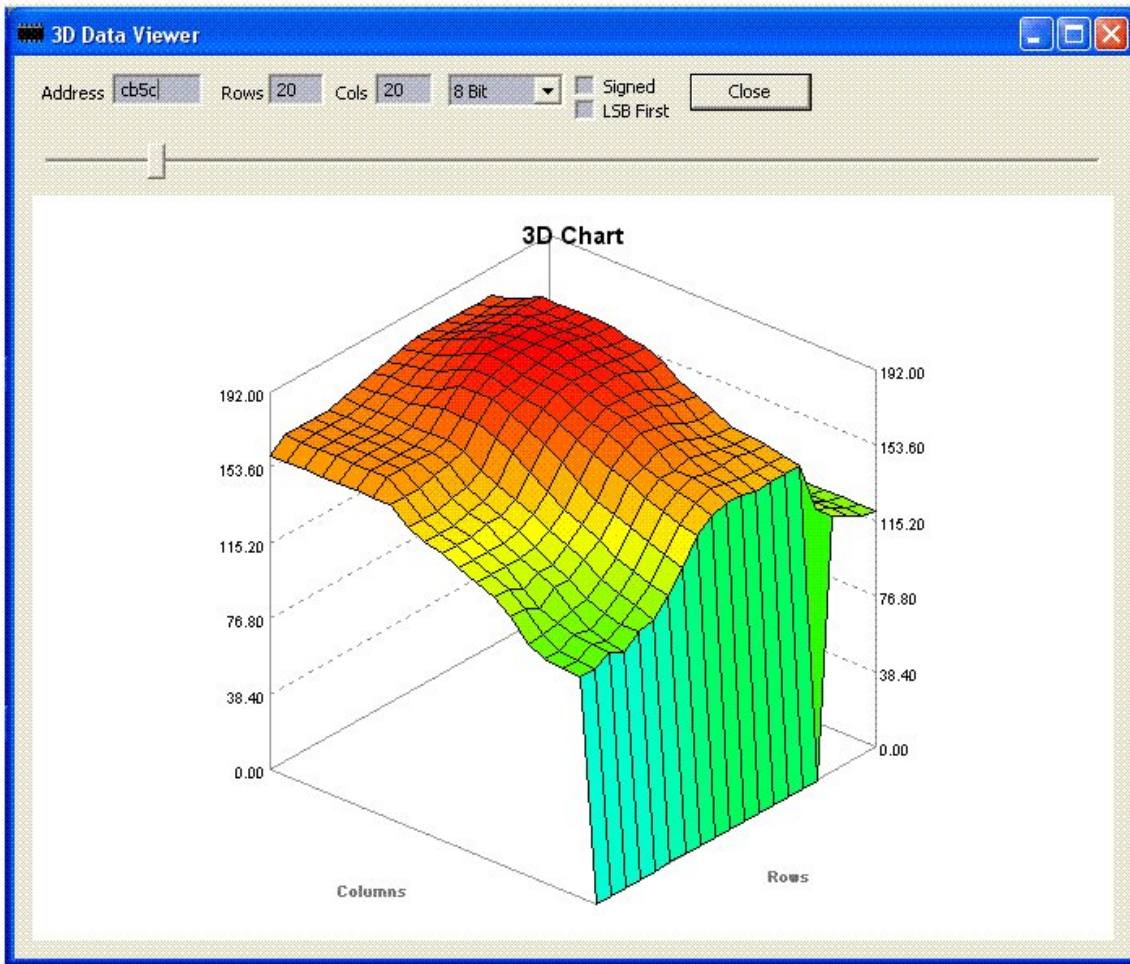
Even though it looks like noise, there is still a clear lower boundary to the table in the shape of a parabola. When I went into the bin and added a scratch XDF with the extents of the table defined, I ended up with the following graph, which is clearly the MAF table. This was confirmed later by tracing the MAF hardware input and flowrate lookup routines.

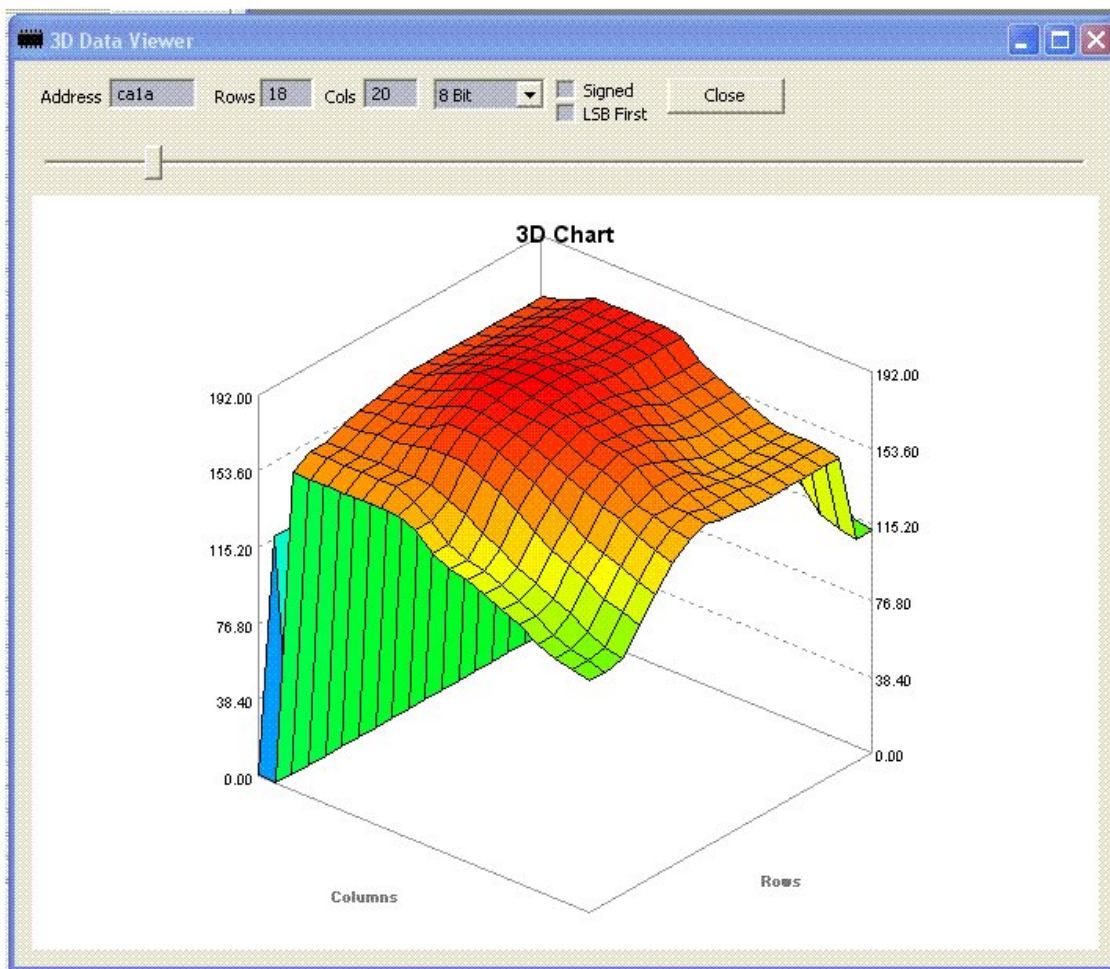




The next example is the spark tables. These are easily ID'd due to their shape. The table will have a high peak at low kPa values that slopes down to lower values at higher kPa's. One thing to keep in mind is that you will need to adjust the size of the row and columns shown within the window due to the varying sizes of the tables. If the table looks like a mountain range (or more technically, tearing), then you will need to tweak the row and column sizes until the table doesn't show any more tearing and looks the way it should. In the first picture, there is an example of tearing due to the incorrect size of row and column values selected. The table shown is the closed throttle spark advance vs. kPa and RPMs. In the two following examples, the closed and open throttle SA tables are shown with the correct row and column values selected. Note the difference and how the tables now look like spark advance curves.







So... While only some tables can be readily located using just the viewers, they are still quite useful as you can hit the ground running without even having to look at the code.

Coming up next: Documentation: What you need to read and know before starting

## **Documentation: What you need to read and know before starting**

Lets start with some background information on whats typically within a VPW based PCM. The main components are a custom to GM spec motorola 68332 32-bit MCU, Intel 28F400BX flash chip for code and calibration storage, and a motorola datalink controller. To really be able to understand the code and the operation of the PCM in general, you should get at least some familiarity with these three devices. Shown here is the '411 PCM with the major components

Before continuing, I packaged up all the most essential documentation into a zipfile titled 'M68332PCMDocs.zip' on moates.net. Its in the misc. uploads section on the file server. It contains documentation on the hardware discussed below.

At the heart of the PCM is the 68332 processor. This processor is geared towards automotive and robotics control. This means that there are some additional commands included such as the boxed table look up commands, and the math co-processor present in other 68XXX MPUs is instead replaced by the TPU, or timing processing unit. This is a somewhat nebulous piece of internal hardware that you wont have direct access to in the code. Its main purpose is to independently process timing based events such as the MAF frequency input, crank position reference pulses, etc. with the intent of reducing the load on the main processor. It has its own 'microcode' that it executes to interface with the hardware and read in the various inputs. The places where you will see it is mainly when the PCM handles timer updates and periodic input reads like the crank reference pulses and outputs, such as the injector pulse widths for each cylinder. For the most part, this can be assumed to be a black box that you send information to, and get information back from. For more info, see the TPU reference manual. It gives some general information as to how the TPU functions.

The processor itself is the 68332 core that uses the CPU32 instruction set. This is quite a bit different than the older 68HC11. The 68HC11 has a very neat, orthogonal instruction set, meaning that each command has an opposite command. For example, there is Branch if Equal and Branch if Not Equal. This makes it very easy to remember the instructions and apply them. The CPU32 instruction set is quite a bit different. As the processor is no longer 8-bit, there needs to be a way to handle 8-bit, 16-bit, and 32-bit data and instructions. This gives rise to .B(yte), .S(hort), .W(ord) and .L(ong) extensions to all instructions. For example, a short branch instruction might be BRA.S while a long branch to some distant part of the address map might be BRA.L. This tells the processor how many of the proceeding bytes or words it needs to use to construct the address to jump to.

The machine code is also much more complex. In the older processors, all the instructions were simply single bytes with each having a value corresponding to an instruction. In the CPU32 instruction set, the instructions are bit-field encoded, meaning that the instruction itself is a word long and the 16 individual bits encode the instruction type and addressing mode. Following the instruction are extension words that encode additional information such as the operand, base address index, etc. This gives rise to instructions that can be multiple words in length. To make things even MORE complex, the processor also supports many types of addressing modes to allow for the full flexibility needed to operate within the memory and MPU address map.

To add to your misery, the processor also has many more registers. There are eight 32-bit data registers D0-D7, eight 32-bit address registers A0-A6, with A7 typically assigned to the stack, and status and vector base registers. These are both a blessing and a curse. They are nice in that they drastically cut down on the need to use the stack. With the older MCUs, there was only registers A, B, X, and Y. This limits the kinds of operations that you can perform. With the plethora of address and index registers, many complex operations can be performed and data is readily accessible without the need to have to constantly push and pull stuff from the stack. The downside is that there is A LOT to keep track of. It's not uncommon to have a value stashed in a data register for no apparent reason, only to have it come back up hundreds of lines later in a separate operation within a routine. This means that you need to search around a bit and keep notes as to what is being used and stored where. The other curse is that when they get together and fornicate with the addressing modes, they can give rise to some very convoluted instructions. Below is an example of this:

```
MOVE.W (EXT_$FFA44E.W,D2.W*2),EXT_12A9.W      ;Load stored dynamic  
airflow from index
```

All in one shot, this instruction loads a value from the index in the RAM using register D2 as the index value and address \$FFA44E as the base offset, and stores it in the RAM address pointed to by the label 'EXT\_12A9.W'. The .W indicates to the MCU that this instruction is within the MCUs internal address map at the top of the addressing space, and not within the flash chip. It assumes that the address is preceded by \$FFFF.

All of this is not to deter you, but to give you fair warning that there is a bit of a learning curve with the CPU32 instruction set, to say the least. It's painful at first, but once you become familiar with it, it's not as bad as it first seems. More can be found on this in the CPU32/programmers reference manuals. Pay close attention to these, as they are the holy bible of hacking when your working with the 68332 based PCMs.

The next piece of hardware is the 28F400BX flash chip with 512K of NV storage space. This is sort of neat in that it's not only memory, but it also has its own little internal microprocessor. During normal operation, it operates as regular memory addressable from \$000000 - \$7FFFFFFF. During a reflash, though, the PCM places it in a special mode that allows the PCM to send instructions and data to erase and program the blocks of memory. You won't see much of this in the code, except early on when the PCM initializes the chip. No reflash commands or



routines are present in the code (for obvious reasons). Any reflash routines have to be loaded as memory resident programs sent thru the OBD-II bus (much like your PC loading a program from the HD) and executed by routines that require you to present a valid security key. More on the chips operation and internal instructions can be found in the 28F400BX information document.

The last major piece of hardware is the VPW datalink controller (DLC). The DLC is seen in the picture above next to the blue Viagra looking thing, which is its ceramic resonator that provides the clock signal for the DLC.

The datalink controller handles all the communications on the VPW bus with only top level control from the MCU. The reason for this is two-fold. First, the VPW, and later CAN buses aren't just for hooking up your laptop to datalog. These are actually local area networks, with many of the cars devices using the network to communicate with each other. For instance, in the later CAN based cars, the body control module, power train control module, dashboard, driver information center, radio, etc. aren't hooked with individual wires. Instead, each device sends out requests for data over the network. For instance, your speedometer doesn't get a pulse train from the PCM, instead, the cluster sends out requests for the relevant data from the BCM and PCM, and relays it via the position of the needles on the 'analog' gauges. This means that there is a good deal of data streaming around. The MCU would need to expend a tremendous amount of processing power to handle all the individual communications. The second reason is the requirements for the VPW and J1850 bus waveforms and timing. The DLC has special internal circuitry to provide the correct waveforms and timing for the 1x and 4x communications modes.

From the code standpoint, the DLC appears as two FIFO (first in, first out) hardware buffers. In my '279, the DLCs buffers are located at \$FF9000 and \$FF9001 (I will go into more detail on its operation later...). Basically, the DLC reads in frames of data off of the bus, and stores them internally. When it detects one or more complete frames of data, it issues an interrupt to the MCU. The MCU then spools the data from the receive buffer on the DLC, interprets it, and if needed, uploads a response frame to the DLCs transmit buffer along with instructions. The DLC then autonomously transmits the datafram from the MCU over the VPW network, and the whole process repeats again. The reason that the DLC is so important is that it not only is the link with the PCM, but it also gives you a way to find out where the basic things such as O2 volts, mass airflow rate, kPa MAP, etc. are stored within the PCMs address map. This is a crucial first step that you need to take to reverse a PCM (again, much more on this to follow). In the information packet, I included the MC68HC58 user's manual. The actual DLC used in each PCM varies, but the user manual will give you a good idea of how the DLC works and how the MCU interfaces with it and is a very good introduction.

### **Using the J1850 communication bus/Datalink Controller and SAE standards to unlock the basic engine parameters and hardware I/O**

I'll probably present this in multiple sections as it's a long topic. Before getting started, it's recommended that you get the SAE docs if possible. The two most

important ones are J1979 - Diagnostic Test Modes, and J2178 - Class B Communication Network Messages - Detailed Header Formats and Physical Address Assignments. They are essential to have. These are only available for purchase from the SAE and are licensed on a by user basis, so I can't post them up. I also cannot post any diagrams from them (copy write) so unfortunately there won't be too many diagrams. Also, it's a good idea to take a look at the OBD 1 and II hardware message board on HP Tuners. LOTS of great info available there. Also, don't forget to check out my hack in my other OBD-II thread here at TGO. That contains a lot of the OBD-II functional and physical routines with full comments.

In every VPW PCM there are routines that are mandated by SAE. One of them is basic diagnostics parameters so that all service stations will be able to at least have some way to perform diagnostics. Pardon my French, but IMO, the minimum mandates are \*\*\*\*. They only give you a bare bones amount of functionality. Never the less, even though they don't give you a lot, they give you enough to get started. The other half of the puzzle is the diagnostic trouble codes. Within the PCMs, there is a large index table that contains all the power train DTCs supported by the PCM. This is important because it can give you some hints as to what memory and hardware addresses are associated with what inputs and outputs.

Let's first take a look at how the OBD message structure works. On the VPW network, all messages have a three byte header. This header tells the nodes (the PCM, TCM, BCM, etc.) the information about the message, who it is addressed to, and who it originated from. The first byte in the three byte format contains the bitwise encoded information about the message. This information is the messages priority, whether or not an immediate response is required, the type of addressing (functional or physical), and specific message type.

The second header byte contains the target address of the message being sent. This lets the specific nodes know who the message is for. This is important as ALL the nodes on the network will get the message, even the node that's sending the message. For example the PCMs base address is \$10, while the address for all nodes on the network is \$FE. The third byte is the address of the sender that the message originated from. The rest of the message is composed of the data fields and finally the CRC bit, which is used for error checking.

Let's next take a look at the addressing and modes. Physical addressing is primarily used for sending actual data. Such data might be to request a block of data to be read from the ROM, secure data link address, or control of a specific device controlled by the PCM. The physical messages are important mainly from the standpoint of gaining access to the PCM for flashing, but have other purposes as well. The OBD-II mode for physical address requests can be found in the first data byte after the three byte header. Below is a list of the typical OBD-II modes, which are in this case the ones supported by my PCM.

```
-List of recognized OBD-II modes-  
;  
; Mode $12, ???  
; Mode $14, Clear Diagnostic Information  
; Mode $17, Request Status of Diagnostic Trouble Codes
```

```

; Mode $19, Request DTC Information by Status
; Mode $20, Return to Normal Mode
; Mode $22, Request Diagnostic Data by PID
; Mode $25, ???
; Mode $27, Data Link Security Access
; Mode $28, Disable Normal Message Transmission
; Mode $29, Enable Normal Message Transmission
; Mode $2A, Request Diagnostic Data Packets
; Mode $2C, Define Diagnostic Data Packet
; Mode $34, Request Download
; Mode $35, Request Upload
; Mode $36, Block Transfer Message
; Mode $3B, Request to Write Data Block
; Mode $3C, Request to Read Data Block
; Mode $3F, Test Device Present
; Mode $76, Response To Block Transfer Message
; Mode $A0, Request High Speed Mode
; Mode $A1, Begin High Speed Mode
; Mode $A8, ???
; Mode $AE, Request Device Control
;
;=====
====
;

```

The next type of address is functional addressing. This is the addressing used during such things as scan tool diagnostics. The header formatting is a little different for functional messages. The diagnostics message format for a request from a device such as a scan tool is as follows: \$68, \$6A, \$F1, [databyte #1: service request], [databyte #2], ... [databyte #N]. The \$F1 indicates that the request is originating from an off board device, such as a scan tool hooked to the OBD port. The service request indicates which service is requested, such as a PID value, or the vehicle information like the VIN. The most important are the PIDs. These are basically like the ALDL data sent from the earlier ECMs.

Lets take a closer look at the PIDs. They are the most important to you if you are hacking. Each PID is requested one at a time in a sequential fashion. Once the PCM receives a request, it takes the received frame and reads in the requested PID. Then, using an index of supported PIDs, it locates the internal routine associated with that PID, and executes it. The routine properly formats the data, loads it into the buffer, and then sends the response message with the requested PID to the scan tool.

Hers an example: Say the scan tool requests the Mass airflow rate. The frame from the scan tool would be: \$68 \$6A \$F1 \$01 \$10, with \$10 being the PID for the mass airflow rate. The PCM then responds with: \$48, \$6B, \$10, \$41, \$10, [MSB MAF], [LSB MAF]. The nice thing about this is that the SAE dictates what the scaling shall be for all the data sent thru the OBD-II port for the PIDs. The rational behind this is so each scan tool will display the data correctly regardless of what units are used within the PCM. For this example, the mass airflow rate is given with a scaling of .01 grams/second. Since we know what the units are of the data being sent, and we also know the location of the routine that sent the mass airflow, we can simply work the equations within the routine backwards. In this case, it turns out that the PCM stores the mass

airflow rate as grams/second x 81.92 in the RAM at an address of \$FF9CD4. This required no tracing through the PCM's hardware or detailed knowledge of the processor to locate it. All we had to do was simply look at the routine that handles the message. So, you can see how powerful a tool this can be.

Next, I'll go into more detail on how the OBD-II requests are handled within the PCM.

Now let's take a look at how the OBD communications are handled within the PCM. Below are the addresses of the two FIFO buffers within the DLC.

Code:

```
EXT_1060 EQU $FFFF9000 ;OBD VPW Rx FIFO hardware buffer
EXT_1061 EQU $FFFF9001 ;OBD VPW Tx FIFO hardware buffer
```

These can actually be located by tracing the address pins on the MCU itself. If you look that the address leads leading from the DLC, you can trace them back to the flash chip/MCU and find them within the address space.

In my PCM, the DLC's interrupt pin is mapped to the level 1 interrupt. This gives the DLC a pretty high interrupt priority, meaning that it will pretty much always be serviced right away. Once the DLC has detected that it has a frame or a buffer nearing its capacity, it will generate an interrupt. The PCM will then jump to that interrupt to service the DLC. The first byte read in is the status byte. This byte is read ahead of the actual OBD data frame, can be read at any time, and gives the status of the last transmission to the MCU. The status encoded includes whether there is a complete frame, if the data is valid or the buffer is empty, if a completion code was received, etc. A complete description can be found in the 68HC58 reference manual. Below is an excerpt of the routine that handles the interrupt and read-in of the status byte and data. It loads the address of the routine that handles the VPW serial communications, and then goes and executes it.

Code:

```
;
;~~~~~
;
; Level 1 interrupt vector
;
;~~~~~
;
LBL_$56BD4:
  ORI #$0700,SR ;Disable interrupts
  MOVEM.L D0-D7/A0-A6,-(A7) ;Move data and address regs. to stack
  CLR -(A7) ;Clear word on stack
  MOVEA.L #$00049BF2,A0 ;Address of VPW serial service routine
  BRA.S LAB_2815 ;Bra to continue

....

;
;~~~~~
;
; Routine to load received OBD VPW serial data
;
;~~~~~
;
```

```

LBL_$49BF2:
    LINK A6,#-6    ;Link and allocate space on stack
    MOVEM.L D0-D2/A0,-(A7) ;Move data/addr. regs to stack
    MOVE SR,-(A7)  ;Move status register to stack
    ORI #$0700,SR  ;Disable interrupts
    MOVE.B EXT_1060.W,D0 ;Load frame status byte from Rx FIFO
    ...

```

As we can see, from the code, the first thing the MCU does is read in the status bit from the DLC. This will tell the MCU if it has a valid data frame present. If the data frame is not valid, or the data is a long stream of bytes originating from an off board device, the PCM spools the contents of the buffer in to empty the buffer and takes any needed actions. If the PCM sees that there is a completion code and valid data available, it will then go and process that data as seen below

```

Code:
;
;-Check if completion code available
;
LSR.B #5,D3    ;Shift right to isolate upper 3 bits
MOVEQ #2,D1    ;Move 2 into D1
CMP.B D3,D1    ;Compare to recieve status
BEQ.S LAB_1F5B ;Bra if ==, buffer contains a completion code
...
;
LAB_1F5B:
    JSR EXT_0C93 ;Routine to read in OBD serial data

```

The PCM does not service the request right away; instead it loads the data into a cyclitic data buffer that holds several OBD data frames. The only time you will see the PCM actually service the data frame right away is in the boot kernel when the PCM is doing no other external processes other than servicing the OBD port. During normal operations, however, the PCM is controlling the engine and transmission in real time, so the PCM only has time to service the TX/RX buffers within the DLC. The end result of this is that you won't see the PCM immediately go and send a PID or some other request. Instead, it spools the data into the buffer as seen below:

```

Code:
;
;~~~~~
;
; Routine to upload OBD serial data frame to OBD data buffer
;
;~~~~~
;
LBL_$51FB6:
    LINK A6,#-2    ;Link and allocate space on stack
    MOVEM.L D0-D2/D5/A0-A1,-(A7) ;Move addr. and data regs. to stack
    ...

```



```

LAB_24F0:
    JSR EXT_0CF0    ;Routine to load OBD serial data from Rx buffer

    ...

;
;~~~~~
; No errors detected, advance OBD Rx data buffer
;~~~~~
;
LAB_24F4:
    TST.B EXT_1B5E.W    ;OBD data buffer read/write collision flag
    BEQ.S LAB_24F5    ;Bra if ==0, no overruns detected
;
;-Read/write collision flagged
;
    MOVEQ #1,D2    ;Load 1 to exit routine
    BRA LAB_24FD    ;Bra to return
;
;-Check to see if buffer pointer needs to be rewound
;
LAB_24F5:
    MOVEA.L EXT_1B5C.W,A0    ;Load start addr of OBD data frame
    MOVEA.L A0,A1    ;Move to A1
    MOVE.L A1,D0    ;Move to D0
    MOVE.L EXT_1B5D.W,D3    ;Load OBD Rx data buffer pointer
    SUB.B D0,D3    ;Subtract previous position from current position,
                    ;now # of bytes in previous frame
    MOVE.B D3,(A0)    ;Save size of last frame to beginning of frame in
                    ;buffer
    MOVE.L EXT_1B5D.W,D0    ;Load OBD data buffer write pointer
    ADDQ.L #1,D0    ;+1, advance to start of next frame
    MOVE.L D0,EXT_1B5D.W    ;Save pointer
    MOVEA.L #$00FFB318,A0    ;Load base addr. of OBD data buffer
    MOVEQ #47,D3    ;47 bytes, position of last OBD serial data frame
    ADD.L A0,D3    ;Add in to current data frame position
    CMP.L D0,D3    ;Compare current position to position of last data
                    ;frame
    BCC.S LAB_24F6    ;Bra if <=, enough space for an additional frame
;
;-End of buffer reached, rewind to beginning of buffer
;
    MOVE.L A0,EXT_1B5D.W    ;Load base address into write pointer
;
LAB_24F6:
    MOVE.L EXT_1B5D.W,D3    ;Load OBD data buffer write pointer
    CMP.L EXT_1B5B.W,D3    ;Compare to OBD data buffer frame pointer
    BNE.S LAB_24F7    ;Bra if !=, no collisions detected
;
;-Read/write collision detected
;
    MOVE.B #$01,EXT_1B5E.W    ;Load with 1, flag read/write collision
;
LAB_24F7:
    MOVE.L EXT_1B5D.W,EXT_1B5C.W    ;Save write pointer as start addr. of
current OBD
                    ;data frame

```

```

JSR EXT_0F51 ;520B6: 4EB900066128
BRA.S LAB_24FD ;Bra to continue

...

LAB_24FD:
TST.B D2 ;Test terminate read-in flag
BEQ LAB_24F0 ;Bra if ==0, continue reading in data

```

So, as long as there is data, the PCM keeps looping and streaming it to the OBD data buffer. Once it is done, it terminates the routine, and moves on. What this means from the standpoint of locating the PIDs is that there is not one single routine that you can go to and say 'Aha! found it!'. Instead, the routines that handle the OBD functional and physical messages are spread out through the code and the execution of them is handled by main scheduling routines.

I will go over how the PCM goes about reading in the frames from the buffer and carrying out the requests. I'll do some research to try and distill the essence of this and go through the routines that handle the read in of the PIDs and other important items.

Ok... This part was a bit confusing. I remember having a hard time with this early on when I was working on the OBD data transmission stuff. Hopefully, I can present it in a way that makes sense.

From the code above, there are two pointers used for the OBD data buffer, one for writing, and one for reading. Before, we were looking at the one used for writing to the buffer, which is EXT\_1B5C. We also know that there is one used for reading from the buffer as well. This one is EXT\_1B5B. It would be logical to assume that when this address is used, that we would likely find the other half of the OBD routines that process the requests. With some searching, this proved to be the case. Below is the header of the routine that starts the process of reading in the frame:

writing to the buffer, which is EXT\_1B5C. We also know that there is one used for reading from the buffer as well. This one is EXT\_1B5B. It would be logical to assume that when this address is used, that we would likely find the other half of the OBD routines that process the requests. With some searching, this proved to be the case. Below is the header of the routine that starts the process of reading in the frame:

Code:

```

;
;~~~~~
;
; Routine to process OBD VPW data frames in buffer
;
;~~~~~
;
LBL_$399E8:
LINK A6,#-14 ;Link and allocate space on stack
MOVEM.L D0-D2/D5-D7/A0-A1/A4,-(A7) ;Move data and addr. regs to stack
MOVE.L EXT_1B5B.W,D7 ;Load D7 with OBD serial data buffer frame
pointer

```

```

    CMP.L EXT_1B5C.W,D7 ;Compare to Start of current OBD serial data
frame
    ;being written to buffer
    BNE.S LAB_128B ;Bra if !=, no collision detected
;
;-Collision detected
;
    TST.B EXT_1B5E.W ;Test read/write collision flag
    BNE.S LAB_128B ;Bra if !=, continue anyway
;
;-Buffer has yet to be initialized
;
    MOVEA #$0000,A4 ;Clear addr., no valid OBD data frame available
    BRA.S LAB_128C ;Bra to continue
;
LAB_128B:
    MOVEA.L EXT_1B5B.W,A4 ;Load A4 with OBD serial data buffer frame
    ;pointer, now addr. of frame to be read
;
;-Transfer params to stack
;
LAB_128C:
    MOVE.L A4,(-14,A6) ;Move frame pointer to stack
    MOVE.L A4,(-14,A6) ;Move frame pointer to stack
    MOVE.L A4,D7 ;Move frame pointer to D7
    TST.L D7 ;Test frame pointer
    BRA LAB_12B6 ;Bra to continue

```

Now, before talking about this it's worth discussing how each frame is stored in the buffer. There are no pre-defined addresses for each frame, its just put into the buffer as they are received on a constant cycling fashion. At the start of each frame, the PCM places a single byte size of the frame so the PCM knows how much space the frame occupies. The buffer looks something like this:

|size of frame #1|OBD frame 1|size of frame #2|OBD frame 2| ...

This way, the PCM knows how far to read into the buffer to obtain the needed data. In the code above, the PCM first checks the read and write buffers to make sure they are not both pointing to the same address. If they are, the PCM checks if the collision flag has been set. If it has, the PCM attempts to process the request, anyway. If not, then the buffer is assumed to not be initialized or empty, and the PCM exits the routine.

To process the request, the PCM loads the needed data and jumps to the end of the routine as shown below:

```

Code:
...
;
LAB_12B6:
    BNE LAB_128D ;Bra if !=, valid OBD data frame available

```

```

;for use
;
;-No data frame available, return
;
MOVEM.L (A7)+,D0-D2/D5-D7/A0-A1/A4 ;Restore data and addr. regs
UNLK A6 ;Unallocate space on stack
RTS ;Return

```

If the address was not cleared (IOW, the buffer has valid data in it), the PCM will jump to process the request. If not, it simply exits. The code that does the initial process of the message is below

```

Code:
;
;~~~~~
; Here if new OBD data frame available, decode message
;~~~~~
;
LAB_128D:
CLR.B D5 ;Preclear D5
CLR.B (-10,A6) ;Preclear message type flag
CLR.L (-8,A6) ;39A20: 42AEFFF8
MOVEA.L (-14,A6),A4 ;Move start addr. of OBD data frame to A4
MOVE.B (1,A4),D1 ;Load D1 with first header byte
MOVE.L D1,D2 ;Move first header byte to D2
ANDI.B #$1F,D2 ;Mask lower 5 bytes
ANDI.B #$1C,D1 ;Mask b2-b4
MOVE.B (4,A4),D0 ;Load D0 with requested mode #
MOVE.B D0,D6 ;Move requested mode to D6 as well
ANDI.B #$40,D6 ;Mask out all but b6, set if response from PCM
;to external node
CMPI.B #$0C,D2 ;01100, physical addressing message recieved
BNE.S LAB_1291 ;Bra if !=, functional message recieved

```

The PCM starts off by checking if the message is functional (the type that send the mode requests) or physical (the type to send actual data). If you recall, the address of the current message frame was saved to the temporary space in the stack at position (-14,A6). The PCM again indexes this address into addr. Register A4 and looks one ahead to load in the first header byte (remember, the first byte is the size of the frame itself). From the discussion above, in the three byte header format, the first byte encodes what type of message it is. The PCM checks to see if the message is a functional or physical request, and jumps to the appropriate scheduling routine to process the request.

Now, the routine that is of interest is the one that processes the functional requests. These requests are typically grouped by modes. Below are the typical diagnostic modes used on OBD-II systems:

- Mode 01 Data stream (sensor readings and switch status)
- Mode 02 Freeze-frame data (if DTCs are present)
- Mode 03 Diagnostic Trouble Codes
- Mode 04 Clear codes and freeze-frame data
- Mode 05 Oxygen sensor monitor
- Mode 06 Non-continuous monitors (EVAP, catalyst, EGR, etc.)
- Mode 07 Continuous monitors
- Mode 08 Bidirectional communication (onboard tests)
- Mode 09 Vehicle VIN, PCM calibration, etc.

Mode 01 is the one we are most interested in. This one is used to transmit the scan tool data. Now, let's take a look at how the PCM continues to process a functional message:

```
Code:
;
LAB_1290:
    MOVE.B D2,(-10,A6) ;Move message type flag to stack
    BRA LAB_129B ;Bra to continue
;
;~~~~~
; Functional message recieved
;~~~~~
;
LAB_1291:
    MOVEQ #8,D6 ;Set b3 in D6
    CMP.B D2,D6 ;Compare to lower 5 bytes of first header byte
    BNE.S LAB_1292 ;Bra if !=, function command/status not recieved
;
;-Function command/status recieved
;
    CMPI.B #$6A,(2,A4) ;Compare value for diagnostic request to header
byte 2
    BNE.S LAB_1292 ;Bra if !=, diagnostic request not recieved
;
;-Diagnostic request recieved
;
    MOVEQ #2,D2 ;Move 2 to D2 to flag diagnostic request
    BRA.S LAB_1290 ;Bra to continue
```

The PCM checks first to see that it is a function command, and then it does something important. It compares the SECOND header byte to \$6A. If you recall, a scan tool request starts with the two bytes \$68, \$6A. The PCM is checking for that \$6A, and if it sees this, it sets D2 to 2 to signal a diagnostics request. Bingo, we hit pay dirt!

Now that we found the part of the routine that processes the scan tools request, let's see what it does next:

```
Code:
...
;
;~~~~~
; Here when functional or physical message recieved
;~~~~~
;
LAB_129B:
    MOVEA.L (-8,A6),A1 ;39C16: 226EFFF8
    MOVE.B (-10,A6),D1 ;Move message flag to D1
;
;-Test functional target request type
;
    MOVEA.L A4,A0 ;Move start addr. of OBD data buffer to A0
    BTST #0,(2,A0) ;Test functional target addr. type
```



```

SNE D3    ;Signal if !=0, status request recieved
NEG.B D3   ;Invert result
ASL.B #1,D3    ;Shift left 1, result in b1
;
;-Test message type
;
BTST #0,(1,A0) ;Test b0 of first header byte
SNE D0    ;Signal if !=0, physical message recieved
NEG.B D0   ;Invert result
ASL.B #2,D0    ;Shift left 2, result in b2
ADD.B D3,D0    ;Add in previous result
;
;-Check to make sure a physical or fuctional request has been recieved
;
BTST #6,(4,A0) ;Test b6 of first data byte
SNE D3    ;Signal if !=0,
NEG.B D3   ;Invert result
ADD.B D3,D0    ;Add in result
MOVE.L A1,(-4,A6) ;Save to stack
ANDI #$00FF,D1 ;Clear all but lower byte of message flag
CMPI #$0001,D1 ;Compare to value for physical message flag
BCS LAB_12B2 ;Bra if <, no physical or functional message recieved
;
;~~~~~
; Schedule appropriate routine to process request
;~~~~~
;
MOVE.L ($2330E,D1.W*4),A0 ;Index addr. of routine to process the
request
MOVE A6,-(A1) ;39C5C: 330E
JMP (A0) ;Jump to execute routine

```

This is the target jumped to in the code after the PCM has determined that it has a functional or physical message to process. Remember that the address (-10,A6) pointed to on the space allocated in the stack contains the message type flag. If the message was a functional diagnostics request, it was set to '2', otherwise, for a physical request, it is set to '1'. The message type flag is loaded from the temporary stack address at (-10,A6) into D1. We know that when a functional scantool request is received, this flag is set to '2'. We can find the routine that handles the request by looking at the following line of code from above:

Code:

```

MOVE.L ($2330E,D1.W*4),A0 ;Index addr. of routine to process the
request

```

The two addresses stored at \$2330E are \$39C60 and \$39C98. Since we know that the message flag was set to '2', the routine we need to next look at is \$39C98.

---

Here is that routine that processes the functional request:

Code:

```

;
;*****

```

```

;
; Routine to process functional diagnostic request
;
;*****
;
LBL_$39C98:
    CMPI.B #$03,(A4) ;Compare to OBD frame message length
    BLS LAB_12B2 ;Bra if message <= 3, message too short
;
    MOVEA.L EXT_1B5B.W,A0 ;Load OBD message frame pointer
    MOVE.B (1,A0),D0 ;Load first header byte
    LSR.B #5,D0 ;Isolate upper 3 bits, priority of message
    MOVE.B D0,EXT_1B5A.W ;Save OBD message priority
    MOVE.B (3,A4),EXT_1B58.W ;Load 3rd header byte into RAM
    MOVE.B (4,A4),D0 ;Load D0 with first data byte
    JSR EXT_0C77 ;Routine to process OBD functional mode request
...

```

Those of you who know assembly will notice the 'JSR EXT\_0C77' command. The PCM is yet *again* jumping to *another* routine! Following these routines is sort of like catching a connecting flight at Chicago O'Hare International Airport. There's a whole lot of running around in a short period of time to get there...

So, we have to continue with routine EXT\_0C77. Hopefully, we're getting a little closer. Below is the routine:

```

Code:
;
;*****
;
; Routine to process OBD functional mode request
;
;*****
;
LBL_$4F40A:
    MOVEM.L D1-D2/D5/A0,-(A7) ;Move addr. and data regs. to stack
    ANDI.B #$00FF,D0 ;Mask out lower byte of first data byte,
    ;requested diag. mode
    CMPI.B #$0001,D0 ;Compare to mode 1
    BCS LAB_232F ;Bra if <
;
    CMPI.B #$0008,D0 ;Compare to mode 8
    BHI LAB_232F ;Bra if >
;
; -Valid mode presented
;
    MOVE.L ($239DE,D0.W*4),A2 ;Index routine to process mode request
    JMP (A2) ;Jump to execute routine

```

Remember that in a scantool request, the first databyte after the three byte header contains the diag. mode requested. From the above code, the PCM checks to see if the mode falls between mode 1-8. If it does, the PCM then uses that mode # to index the routine to process the request. It then jumps to process that routine (yes, I know, *another routine*). If we look at addr. \$239DE, we see that the routine that is in question for mode 1 requests is located at \$4F42C. That routine is shown below:

Code:

```

;
;*****
;
; Routine to process mode 1 requests
;
;*****
;
LBL_$4F42C:
    MOVE.L EXT_1B5B.W,D2    ;Load OBD data frame pointer
    CMP.L EXT_1B5C.W,D2    ;Load start of OBD data frame being saved to
buffer
    BNE.S LAB_231E    ;Bra if !=
;
;-Check for possible read/write collision
;
    TST.B EXT_1B5E.W    ;Read/write collision flag
    BNE.S LAB_231E    ;Bra if !=, continue with read in
;
    MOVEA #$0000,A0    ;Clear addr., buffer not initialized
    BRA.S LAB_231F    ;Bra to continue
,
LAB_231E:
    MOVEA.L EXT_1B5B.W,A0    ;Load start of OBD data frame in buffer
;
LAB_231F:
    MOVE.L A0,EXT_199C.W    ;Move addr. to RAM
    CLR D0    ;Preclear D0
    MOVE.B (5,A0),D0    ;Move requested PID to D0
    MOVE D0,EXT_199F.W    ;Save requested PID to RAM
    MOVE.B (A0),D1    ;Load D1 with OBD data frame size
    MOVE.B D1,EXT_199E.W    ;Save frame size to RAM
    CMPI.B #$05,D1    ;Compare 5 to frame size
    BNE LAB_232F    ;Bra if !=, invalid frame size
;
;-Valid frame size recieved
;
    TST D0    ;Test requested PID
    BNE.S LAB_2320    ;Bra if !=0
;
;-Request for supported modes recieved
;
    MOVEQ #1,D0    ;Load D0 with 1
    BRA.S LAB_2327    ;Bra to continue
;
LAB_2320:
    CMPI #$0009,D0    ;OBD mode request 1-8
    BCC.S LAB_2322    ;Bra if mode requested >
Blah, blah, blah...

```

I dont know about you, but Im getting tired of all this jumping around. There must be a short cut to find the routine. By now, you'd even have trouble remembering what you where looking for in the first place (in case you really did forget, its the PIDs). There is indeed a short cut... Look at LAB\_231F. If you notice, the PCM is

indexing the OBD message frame. In a mode 1 diagnostics request, the requested PID # is always the 5th byte in. With the message size at the head of each frame, this is the 6th byte in the buffer, or current index + 5. Commands:

Code:

```
MOVE.B (5,A0),D0 ;Move requested PID to D0
MOVE D0,EXT_199F.W ;Save requested PID to RAM
```

shows us that the PCM saves the requested PID # to the RAM. This is something we can use to cut right to the chase. We no longer need to look at the endless indexed routines and endless jumps and branches to find our way there, we can simply search for address EXT\_199F to find the code that uses that PID to process the request.

Next, I will go into the actual routines that look up the PIDs and process them.

Ok, now that we know where the PID # is stored, lets see where it takes us... Some searching shows that the PID is only used in one other place, at the tail end of the routine where the transmission takes place. The routine where this resides is shown below:

Code:

```
;
;*****
;
; Routine to send requested OBD mode 1 PIDs
;
;*****
;
LBL_$431C8:
;
;-Echo PID request and requested PID # back to sender
;
MOVEQ #$41,D0 ;Load value for PID request
JSR EXT_0DCF ;Routine to transmit OBD serial data
;
MOVE EXT_199F.W,D2 ;Load PID # requested from RAM
MOVE.L D2,D0 ;Move to D0 as well
JSR EXT_0DCF ;Routine to transmit OBD serial data
;
;-Send value of PID
;
CLR.B D1 ;Prclear D1, mode #
MOVE.L D2,D0 ;Move requested PID to D0
LEA (-4,A6),A0 ;Load addr. of stack
JSR EXT_0E37 ;Routine to match and look up requested PID from
;index
;
;-Check size of transmission
;
CMPI.B #$01,D0 ;Compare # of bytes to transmit to 2
BHI.S LAB_1A4F ;Bra if size of xmission > 2 bytes
;
BEQ.S LAB_1A4E ;Bra if ==2 bytes
;
;-Transmit 1 byte
;
```

```

LAB_1A4D:
    MOVE.B (-4,A6),D0    ;Load D0 with byte #1
    BRA LAB_1A5A    ;Routine to transmit OBD serial data
;
; -Transmit 2 bytes
;
LAB_1A4E:
    MOVE.B (-4,A6),D0    ;Load D0 with byte #1
    JSR EXT_0DCF    ;Routine to transmit OBD serial data
;
    MOVE.B (-3,A6),D0    ;Load D0 with byte #2
    BRA LAB_1A5A    ;Routine to transmit OBD serial data

```

Now, look closely at what is sent thru the OBD port at label LBL\_\$431C8: . The value \$41 is sent thru the OBD port. Whats the significance? Well, recall what the PCM sends as a response to a mode 1 request, \$48, \$6B, \$10, \$41... The \$41 is the PCMs response to a mode 1 request. Without the other comments, seeing this might lead us to guess that this might be our routine that sends the mode 1 PIDs. Lets take a look at the routine that calls this one. It appears below:

```

Code:
;
; ~~~~~
;
; Routine to schedule mode request transmission
;
; ~~~~~
;
LBL_$431A2:
    LINK A6,#-4    ;Link and allocate 4 bytes on stack
    MOVEM.L D1-D2/D5/A0,-(A7) ;Move data and addr. regs to stack
    ANDI #$00FF,D0 ;Mask out lower byte
    CMPI #$0041,D0 ;Value to transmit OBD mode 1 request
    BCS LAB_1A64    ;Bra if <
;
    CMPI #$0048,D0 ;Value to transmit OBD mode 8 request
    BHI LAB_1A64    ;Bra if >
;
    MOVE.L ($237FE,D0.W*4),A2 ;Load addr. of applicable transmission
    routine
    JMP (A2)    ;Jump to routine

```

If we look at the value being loaded into D0, this looks like the PCM is checking the loaded response to modes 1-8. If we look at the routines pointed to in the index \$237FE, we can see that each routine starts of by sending \$41 thru \$48 via the OBD port, and then additional data, which we might assume to be the PIDs. Additional work on the routines proved this to be the case. There is a routine for each PID to transmit the requested information for each mode. For the mode 1 PIDs, as well as others, this is handled by routine EXT\_OE37. Lets have a look:

```

Code:
;
; ~~~~~
;

```

```

; Routine to match and look up requested PID from index
;
;~~~~~
;
LBL_$63516:
LINK A6,#-2    ;Link and allocate space on stack
MOVE.L A1,-(A7) ;Move addr. reg to stack
MOVEA.L A0,A1  ;Move target store addr. to A1
CLR.L (A1)     ;Clear contents of addr.
LEA (-2,A6),A0 ;Load addr. of stack
JSR EXT_0DA9   ;Routine to match requested PID # to those
               ;supported
;
TST.B D0       ;Test result
BEQ.S LAB_3085 ;Bra if ==0, match not found
;
;-Match found in index
;
MOVE.B (-2,A6),D0 ;Move index value into D0
CMPI.B #$1A,D0   ;Compare index position to end of base PID
               ;params
BCC.S LAB_3084   ;Bra if >=
;
;~~~~~
; Base PID requested
;~~~~~
;
MOVEA.L A1,A0    ;Move target addr. to store PIDs to A0
JSR EXT_0BE2     ;Routine to schedule xmission of OBD-II
               ;PID (up to $1E)
;
BRA.S LAB_3086   ;Bra to return

```

We enter into this routine with our desired PID in D0 and the address of the area in the stack to which we will store the ultimate data that comprises the PID. The supported PIDs are stored in an index. There are the base mandated PIDs spanning from PID \$00 - \$1E, as well as enhanced PIDs that are unique to each PCM. It would be nice to have the enhanced PIDs as they transmit parameters for the more sophisticated functions of the PCM, such as the idle airflow, IAC, e-throttle (if equipped), e-trans, etc. But, info on these is not widely available, so you will only really be able to work with the base PIDs.

The PIDs are looked up by a program that searches thru the index until it finds a match to the PID. It is shown below:

```

Code:
;
;~~~~~
;
; Routine to match and verify requested PID # to those supported
;
;~~~~~
;

```

```

LBL_$605A6:
    LINK A6,#0    ;Link to stack
    MOVE D0,-(A7) ;Move requested PID # to stack
    MOVEM.L D2/D5-D7,-(A7) ;Move data and addr. regs to stack
    CLR D0        ;Preclear D0
    MOVE #$00DE,D6 ;Init counter
    MOVE.L D0,D4   ;Clear D4 as well
;
;~~~~~
; Search index for match to requested PID #
;~~~~~
;
LAB_2EB3:
    MOVE.L D6,D5    ;Move counter to D5
    ADD D4,D5       ;Add lower index search bound to current
                    ;position
    ADDQ #1,D5      ;+1
    LSR #1,D5       ;/2 for index lookup
    MOVE.L D5,D7     ;Move current index position to D7
    CLR.L D3        ;Preclear D3
    MOVE.B D7,D3     ;Move offset for lookup to D3
    MOVE.W ($21676,D3.W*4),D2 ;Look up PID # from index
    MOVE (-2,A6),D3  ;Load requested PID # from stack
    CMP D3,D2       ;Compare requested PID # to PID # from index
    BNE.S LAB_2EB4  ;Bra if !=
;
;-Requested PID # found from index
;
    MOVEQ #1,D0     ;Load D0 with 1, match found
    BRA.S LAB_2EB6  ;Bra to continue
;
;~~~~~
; Here to advance search through index for desired PID #
;~~~~~
;
LAB_2EB4:
    CMP D2,D3       ;Compare PID # from index to requested PID #
    BCC.S LAB_2EB5  ;Bra if PID # from index <=
;
;-PID # from index > requested PID
;
    MOVE.L D5,D6     ;Update upper index search bound
    SUBQ #1,D6       ; -1
    CMP D6,D4       ;Compare to lower index search bound
    BLS.S LAB_2EB6  ;Bra if upper index counter still >
;
    MOVE D4,D6       ;Reset upper bound to lower bound value
    BRA.S LAB_2EB6  ;Bra to continue
;
;-PID # from index is <= requested PID
;
LAB_2EB5:
    MOVE.L D5,D4     ;Update lower index search bound
;
LAB_2EB6:
    CMP D6,D4       ;Check lower bound against upper bound
    BCC.S LAB_2EB7  ;Bra if lower index search bound >= upper index

```

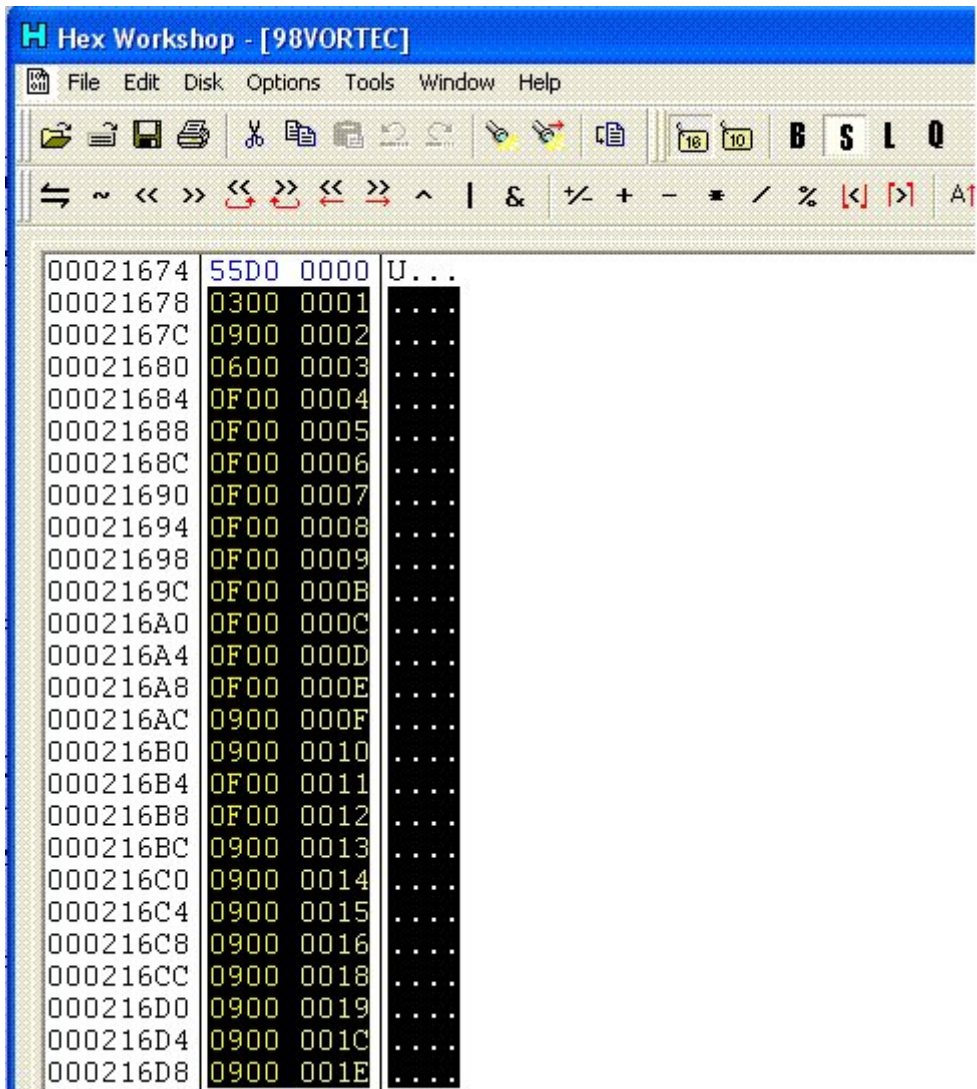


```

        ;search bound, match not found
;
TST.B D0    ;Test index search result
BEQ.S LAB_2EB3 ;Bra if ==0, result not yet found, continue to
        ;search
;
;~~~~~
; Here to terminate search
;~~~~~
;
LAB_2EB7:
CMP D6,D4    ;Compare lower index search bound to upper index
        ;search bound
BNE.S LAB_2EB8 ;Bra if !=, return with result
;
MOVE.B D4,D7    ;Move lower search bound into D7
ANDI #$00FF,D4 ;Mask out lower byte portion
CMP.W ($21676,D4.W*4),D3 ;Compare requested PID # to PID # from index
BNE.S LAB_2EB8 ;Bra if !=, match not found
;
;-Requested PID # found from index
;
MOVEQ #1,D0    ;Load D0 with 1, positive match found
;
LAB_2EB8:
MOVE.B D7,(A0)    ;Move result to target addr.
MOVEM.L (A7)+,D2/D5-D7 ;Restore data and addr. regs
UNLK A6    ;Unlink stack
RTS        ;Return

```

In a nutshell, what this routine does is start roughly in the middle of the index, and search through it exhaustively either forward or backward until it finds a match, or either reaches the end or the beginning of the index. If a match is found, it flags it and returns with the index that the PID was found at. This index # also corresponds with a list of addresses of routines that process the PID requests. Each PID has its own unique routine to handle it, so there are quite a few routines dedicated to this task. The index of supported base PIDs is shown in the screenshot below. The PID #'s appear in the right hand column.



Ok, now that we know which index #s correspond to which PID, we can make sense of all the routines in the index. The scheduler for the base mode 1 PIDs is EXT\_OBE2. This routine uses the index # found previously to look up the applicable routine in the index. A portion of the index is shown below:

Code:

```
[Index at addr. $231C4]

$00030A30 (PID $00 - Send supported PIDs)
$00030A82 (PID $01 - Xmit trouble codes and onboard test info)
$00030C2C (PID $02 - Return freeze-frame trouble code)
...
```

The routine that handles the scheduling of each routine is shown below:

Code:

```
;
;~~~~~
```

```

;
; Routine to schedule xmission of OBD-II PID (up to $1E)
;
;~~~~~
;
LBL_$30A18:
  LINK A6,#-4    ;Link and allocate space on stack
  MOVEM.L D2/D5/A1,-(A7)  ;Move data and addr. regs to stack
  MOVEA.L A0,A1    ;Move target addr. to store to to A1
  CLR D5    ;Clear D5
  MOVE.B D0,D5    ;Move index position to D5
  MOVE.L ($231C4,D5.W*4),A2 ;Load routine from index
  JMP (A2)    ;Jump to routine

```

Ok, now that we found these, how can they be useful? What will all these routines tell me? Well, lets say for example you want to know where the PCM stores the value for the TPS. We know from the SAE J1979 standard that for PID 11, the TPS is transmitted to the scantool in the format of 100/255, which means that a hex value of 255 will correspond to 100% open throttle. Lets now take a look at the routine that handles PID 11:

```

Code:
;
;*****
;
; PID 11, xmit TPS
;
;*****
;
LBL_$31088:

...

;
;-In mode 1
;
LAB_0C3F:
  MOVE.B EXT_177D.W,D0    ;Load %TPS for OBD diags.
  BRA.S LAB_0C43    ;Bra to xmit

...

```

From the above code, the PCM simply loads the 8-bit value for the TPS, and transmits it right away with no additional scaling. This tells us that the value stored in EXT\_177D is scaled in terms of 100/255%, which is the typical %TPS as used in many other GM ECMs and PCMs. If we follow the address that the TPS is stored in, it will lead us to the routine that loads the A/D TPS and formats it for use. Once we trace through the code, we will have not only found the scaling for the TPS, but also the other functional addresses that the TPS is stored in as well as where the A/D TPS resides. All of this stemming from using the known TPS value in address EXT\_177D.

Ok, we found the TPS, lets look for something else. Lets say that we want to know where the mass airflow is stored. From the SAE doc, this is PID 10. The mass airflow must be returned with the range of 0-655.35 g/sec with a scaling of .01 grams/second. If we take a look at the routine, we will see the math used to convert the MAF flowrate. The routine is shown below:

Code:

```
;
;*****
;
; PID 10, xmit mass airflow
;
;*****
;
LBL_$3103A:
    CMPI.B #$02,D1    ;Compare mode # to mode 3
    BHI.S LAB_0C39    ;Bra if in higher modes
;
    BEQ.S LAB_0C3B    ;Bra if ==, in mode 3
;
    TST.B D1          ;Test mode #
    BNE.S LAB_0C3A    ;Bra if !=0, in mode 2
;
;-Here if in mode 1 or other modes
;
LAB_0C39:
    MOVE EXT_12AA.W,D0 ;Load D0 with mass airflow rate
    MULU #625,D0       ;(g/sec x 81.92) x 625
    LSR.L #8,D0        ;/256
    LSR.L #1,D0        ;/2, now (g/sec x 81.92) x (625/512)
    BRA LAB_0C2E       ;Bra to transmit scaled airflow
```

From the commands, we can see what is done to the stored mass airflow rate to format it for output to the scantool. Remember that the mass airflow must be returned with a scaling of .01 g/sec. With some basic hand calculations, we find that the mass airflow is stored as grams/sec x 81.92. With some additional legwork, we will also find the address that the TPU stores the MAF pulses to as well as the other addresses that are used for storing the mass airflow rate and MAF based mass air per cylinder, which is a key component in the fuel calcs.

One thing to bear in mind is that the OBD routines are quite large, so you don't want to be a hero and reverse them all right away. Just get what you need at get out. I can tell you that up front, you will not have the information you need to map them out completely. This information can only be found two ways. 1) get it from a source that has the info from GM, or 2) come back at the end and complete the routines.

It's also worth noting that the OBD routines are some of the more complex routines in the PCM, so the work you do initially will be the hardest. Once you unlock these, though, it gets progressively easier to find the basic stuff like the VE, spark tables, fuel cut-off thresholds, etc. I will review finding and working with tables in the next and final section: Working with the code: Finding tables and constants using the assembly.

### **Working with the code: Finding tables and constants using the assembly**

Before delving into finding the actual tables, it's worthwhile looking at how the PCM actually performs lookups. In the 68332, Motorola actually implemented a machine level command to perform this due to the fact that these were intended specifically for automotive and robotics applications. The command is TBLU and TBLS. TBLU is for 2D unsigned table lookups while TBLS is for 2D

signed table lookups. The command takes the form of TBLU/S.[SZ] [Table Addr.],[Data Reg]. The SZ indicates the size of the table elements. They can be byte, word, or long. The data reg. is the register that contains the value to be used for the table lookup. The tables can be up to 257 elements in length with a spacing of 256 between each element. So, the value used for lookup can span from 0-65535. The looked up value is automatically scaled and returned in the data reg.

But, the 68332 is unique in this aspect. Most other processors used today in automotive applications do not have instruction sets as expansive as the CPU32 instruction set. As such, there is no built in routine in the CPU to perform a table lookup. The table lookups will not be a one line instruction, but actually a subroutine to perform the lookup. The closest analog that we can use to examine how these routines are structured is the 68HC11 used prior to the 68332. Lets first take a look at the standard 2D lookup routine. Its important that you learn to recognize these, as they will be some of the first routines that you encounter, and will almost always signal when a table lookup is taking place. Below is the standard routine to perform a 2D lookup:

Code:

```
;
;~~~~~
;
; 2D lookup Routine
;
; Enter with:
; A=value to use for lookup
; X=Address of table
;
; Exit with:
; A=looked up table value
; B=Previous value on entry
; X=Previous value on entry
;
;~~~~~
;
LF15E PSHX      ;Push table addr. to stack
PSHB          ;Push offset to stack
LDAB          #$10 ;Multiplier for lookup, 16 counts/row
;
;-ALternate entry point
;
LF162 MUL      ;Val x multiplier for lookup
PSHB          ;remainder to stack
TAB           ;Row offset into B
ABX           ;Add in offset to X
LDD           0,X ;Load in table value and value following that
;
;-Perform interpolate
;
SBA           ;Subtract following value to get delta table value
PULB          ;Get remainder
BCC           LF172 ;Bra if underflow, next table value is greater
;
;-Interpolate up
;
```

```

    NEGA      ;Make positive again
    MUL       ;remainder x delta table value
    ADCA      0,X    ;Add in table value to interpolated value and round if
needed
    BRA       LF178   ;Bra to return
;
;-Interpolate down
;
LF172 MUL     ;remainder x delta table value
    ADCA      #0     ;Round if needed
    NEGA      ;Switch sign
    ADDA      0,X    ;Table value + (-interpolated value)
LF178 PULB    ;Restore acc. B
    PULX      ;Restore acc. X
    RTS       ;Return
;~~~~~

```

This is the standard routine used in the ECMs and PCMs based on the 68HC11 MCU. Upon entry, the byte sized value used for the lookup is stored in the 8-bit reg. A and the address is stored in the 16-bit addr. reg. X. The standard spacing between table elements is decimal 16, or \$10 in hex. Since the value of A can be anything from 0-255, it will not always fall exactly right on a table element. Instead, the value for lookup may be somewhere between two elements. Due to this, the routine performs what's called interpolation. Interpolation is basically the construction of an intermediate value using the two known adjacent table entries and the interval determined by the value in A.

The first step in the lookup is to multiply the value in A by 16. This gives a 16 bit result stored in the 8-bit registers A and B. A will contain the desired element, and B will contain the remainder, or the locations between the element and the next element in the table. For example let's say we have a 5 element table as shown below

Code:

```

;
;-Some table
;
$1234
FCB    20
FCB    30
FCB    40
FCB    50
FCB    60

```

The table has a span from 0-64. 0 corresponds to value 20 in the table while 64 corresponds to value 60. Let's say that the value for lookup will be 35. So, the commands to perform the lookup might look like the following

Code:

```

LDAA     35          ;Load value for lookup
LDX      $1234       ;Load address of table
JSR      2DLOOKUP    ;Go do lookup

```

Upon entry into the 2D lookup routine, the value in A is multiplied by 16. This gives a result of 560. Keep in mind that in the earlier 8 bit MCUs, the 16 bit results are

stored in registers A and B. This means that A will contain the value  $/256$ , with B containing the remainder. In this case, A will contain the value 2 and B will contain the value 48. This means that the value we will be looking up will be between the values of 40 and 50. Next, we need to interpolate. First, the MCU will add the value in A to the address of the table to get the location of the table element it is to look up. The address in this case will be  $(\$1234 + \$02)$ . This is the address of the element of the table that is equal to 40. If you notice, the command that does the lookup is:

Code:

```
LDD    0,X    ;Load in table value and value following that
```

The LDD stands for Load register D. The 16-bit register D is basically the 8-bit registers A and B combined. In our case, the values 40 and 50 are loaded consecutively. This means that A contains the value 40 and B contains the value 50. To interpolate, we must first determine the span between the two elements. In this case, its  $50-40$ , or 10, which is our span. Now, the remainder is sort of like an imaginary sliding needle that can slide anywhere between the two values of 40 and 50. To determine where that needle is pointing to, we now need to look at the remainder. The remainder can be any value from 1-255. With a remainder of 1, the needle is next to 40. With a value of 255, the needle is next to 50. In our case, the remainder was 48. To interpolate, we first divide 48 by 256. This results in a value of .1875. Next, we multiply by our span, which is 10. The result is 1.875, which is rounded to 2. So, the final output will be  $40 + 2$ , or 42, which is what the routine returns once its complete.

Now, there are also routines to perform 3D lookups. In the 68332 code, these are basically just several TBLU commands performed consecutively, but in the earlier ECMs, its a bit more complicated due to the lack of a built in lookup command. Now, before we dive into the earlier routine, lets take a look at the 3D lookup routine in the 68332. Since we have built in commands, its easier to see what the essence of the routine is.

Code:

```
;
;~~~~~
;
; 3D lookup routine (16-bit)
;
;~~~~~
;
; In:
;   D0 = value for column lookup
;   D1 = value for row lookup
;   D2 = size of row
;   A0 = base addr. of table
;
; Out:
;   D0 = Result
;
LBL_$653E2:
MOVE D1,D3    ;Load value for row lookup into D3
LSR #8,D3     ;/256, now row offset
MULU D2,D3    ;Row size x row offset
ADDA.L D3,A0  ;Add in offset to base address of table for
              ;desired row
```

```

MOVE D0,D3    ;Move value for column lookup into D3 as well
TBLUN.W (A0),D0    ;Look up column value in row and interpolate
TBLUN.W (A0,D2),D3    ;Look up column value in next row and interpolate
TBLU.L D0:D3,D1    ;Interpolate column values between rows
ADDI.L #$00000080,D1    ;Round up if needed
LSR.L #8,D1    ;End result /256
MOVE.L D1,D0    ;Load result into D0
RTS          ;Return
;~~~~~

```

Since we have a built-in routine, it doesn't get much simpler than this. Basically, you come in with an X and a Y rectangular coordinate value that determine where in the table you want to do the lookup. These values are contained in data register D0 and D1. D0 is the column location of the value, and D1 is the row location. Now, lets say for example that we are looking up the spark advance. Lets further assume that the engine is at 2400 RPM and the MAP is 55 kPa. In the table shown below, the column value is 55 kPa, and the row value is 2400 RPM. The value that would be returned by the above routine is shown highlighted in the blue rectangle.

Closed Throttle SA table																	
	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100
6000	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9
5600	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9
5200	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9
4800	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9
4400	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9
4000	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9
3600	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9
3200	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9
2800	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9
2400	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9
2000	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.6	24.6	24.6	24.6	24.6	24.6	24.6
1600	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.6	24.2	24.2	23.5	23.2	23.2	23.2
1200	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.6	23.5	22.5	21.8	21.4	21.4	21.4
1000	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.6	23.5	22.5	21.4	20.4	19.7	19.7	19.7
800	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	23.5	22.5	21.4	20.4	19.0	17.5	17.5	17.5
600	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.2	22.8	21.8	20.7	19.7	18.3	16.1	16.1
400	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	24.9	23.5	22.5	21.4	20.4	19.3	17.9	16.1	16.1



Now, when you have a 3D table, its not really a table, but a surface such as the spark tables shown in the attached pictures above. To get the maximum resolution and approximate a smooth surface, we need to interpolate between the values that define the 3D surface. The value that we want to lookup will typically be bounded by 4 adjacent values if we are in an inbetween location in the table. This gives rise to the need to do two lookups and interpolate three times. These work like an expanded version of what we looked at above with the 2D table. The MCU looks up the first value treating the row as basically a 2D table, interpolating the result. It then looks up the next value in the next row, interpolating that result. The two commands that do this are:

Code:

```
TBLUN.W (A0),D0    ;Look up column value in row and interpolate
TBLUN.W (A0,D2),D3  ;Look up column value in next row and interpolate
```

Now, we have these two interpolated values, but since its a 3D table, we need to interpolate between the interpolated values as well. This is done by the following command:

Code:

```
TBLU.L D0:D3,D1    ;Interpolate column values between rows
```

The interpolation works just like explained above for the 2D table lookup. The D0:D3 defines the span. The span between the adjacent rows is D3-D0. The remainder for interpolation is the value in D1, which spans from 0-65535. After the three interpolations, the result of the lookup is /256 to give a 16-bit value and transferred from D1 to D0.

Now, the routine to perform this WITHOUT the imbedded table lookup commands is considerably more complex. It is shown below:

Code:

```
;
;~~~~~
;
; 3D Lookup Routine
;
; Enter with:
;  A=Row value
;  B=Column. value
;  X=Table address
;
; Exit with:
;  A=value from lookup
;
;~~~~~
;
LF17B    PSHY      ;Reg. Y to stack
PSHB     ;Col. val. to stack
PSHX     ;Table address to stack
SUBA     0,X      ;Subtract row offset
BCC      LF184    ;Bra if no underflow
;
CLRA     ;Clear out col. val.
;
LF184    SUBB      1,X      ;Subtract col offset
BCC      LF189    ;Bra if no underflow
;
```

```

    CLRB      ;Clear underflow
;
LF189 PSHX    ;Table addr. to stack
    PULY      ;Get table addr. into Y
    PSHA      ;Row val. to the stack
    LDAA      #$10 ;Mult. for lookup
    MUL       ;mult. x col. val, A now offset for lookup, B now remainder
    PSHB      ;col. remainder to the stack
    TAB       ;Get MSB into B
    ABX       ;Add the offset into X
    PULA      ;Get col. remainder into A
    PULB      ;Get row value back
    PSHA      ;Put the col. remainder back on the stack
;
    LDAA      #$10 ;mult. for lookup
    MUL       ;row value x mult. for lookup
    PSHB      ;row remainder to the stack
    LDAB      2,Y   ;Load # of cols.
    MUL       ;# cols x row offset, skip to row in table
    ABX       ;Add the offset into X
    PSHX      ;Push address (current row) to stack
    LDAB      2,Y   ;Load # cols
    ABX       ;Add it into X (Now next row)
;
;~~~~~
;-Perform interpolation
;~~~~~
;
;
;      V1= current stored value in table
; V2-V4 are values located
; around the immediate table
; value
;
;
;      -----
;  i  |   V1   |   V2   |
;      -----
; i+1 |   V3   |   V4   |
;      -----
;           j       j+1
;
;

    TSY      ;Get stack pointer + 1 into Y
    LDD      3,X   ;Get values in next row for interpolation
    SBA      ;Subtract table value in next col. from value in current col.,
now delta
    LDAB      3,Y   ;Load col. remainder from stack
    BCC      LF1B4  ;Bra if delta value was negative
;
;~~~~~
; Interpolate values in next row up
;~~~~~
;
;-Interpolate up
;
; V2 + col. remainder x |(V3-V4)|
;

```

```

;
NEGA      ;Make delta positive
MUL       ;remainder from col. value x delta
ADCA      3,X   ;Add it into table value in next row up
BRA       LF1BA   ;Bra
;
;-Interpolate down
;
; V2 - col. remainder x |(v3-V4)|
;
;
LF1B4 MUL      ;remainder from col. value x delta
ADCA      #0   ;round if needed
NEGA      ;Change sign
ADDA      3,X   ;Add it into table value in next row up
;
;~~~~~
; Interpolate table values in current row
;~~~~~
;
LF1BA PULX      ;Get current col/row address back
PSHA      ;Push interpolated table value in next row to stack
LDD       3,X   ;Get table values in current row
SBA       ;Subtract table value in next col. from current value
LDAB      3,Y   ;Load in col. remainder
BCC       LF1CA   ;Bra if next value is less
;
NEGA      ;Make positive
;
;-Interpolate up
;
; V1 + col. remainder x |(V1-V2)|
;
MUL       ;remainder x delta
ADCA      3,X   ;Add it into current value
BRA       LF1D0   ;Bra
;
;-Interpolate down
;
; V1 - col. remainder x (V1-V2)
;
LF1CA MUL      ;remainder x delta
ADCA      #0   ;Round if needed
NEGA      ;change sign
ADDA      3,X   ;Add in current value
;
;-Interpolate between the two rows
;
LF1D0 PULB      ;Get interpolated value from next row
PSHA      ;Push interpolated value from current row to stack
SBA       ;Current interpolated value - interpolated value in next row
LDAB      2,Y   ;Load row remainder
BCC       LF1DF   ;Bra if delta was positive
;
;-Interpolate up between rows
;

```

```

NEGA    ;change sign
MUL      ;row remainder x delta value
ADCA    1,Y    ;add to interpolated value in current row
BRA     LF1E6    ;Bra to
;
;-Interpolate down between rows
;
LF1DF MUL      ;row remainder x delta value
ADCA    #0    ;Round up if needed
NEGA    ;change sign
ADDA    1,Y    ;add it into the current interpolated value
;
;-Clean up the stack and resore X, Y, B
;
LF1E6 INS      ;Inc. stack pointer
PULX      ;
PULX      ;
PULB      ;
PULY      ;
RTS       ;Return
;~~~~~

```

In a nutshell, though, it works in the same manner that the 3D lookup routine for the 68332 does. Since we don't have the built-in commands, the 2D table lookup routine must be duplicated twice to look up and interpolate the values. As shown in the table indicating the 4 surrounding table values, the first row values are looked up, followed by the second. The first row values will be V1 and V2. The next ones are V3 and V4. The routine interpolates between the two of these to return intermediate values of, say, V'1 and V'2. Since the lookup is in 3D, we must also interpolate between the values of V'1 and V'2 to return the result. The process is the same as the other 3D routine, but there are many more commands. You can see the value of having the built-in table lookup routine. The processors with less complex instruction sets will instead have routines looking like the one above.

Now that we've taken a look at how the lookup is performed, we can delve into some basic examples of table lookups. Coming up next, I will show the process used to scale the values for the lookup as well as the lookups themselves using some examples.

---

Ok, now let's take a look at an actual table look up. First we'll use an example of a 2D look up. The first example will be the percent IAC airflow for the proportional term used to control the engine idle speed. From about 93 and on, many of the PCMs that used IACs had full standardized PID idle routines that used a linearizing airflow vs. steps table for the IAC. This allowed for a full linear control algorithm that had all the control parameters as a % of the total airflow of the IAC rather than the IAC steps. As such, there are many % airflow tables for the proportional term, integral gain, DFCO airflow rates, dynamic throttle airflow estimation, etc. The one we will look at is the proportional % airflow term.

This If you notice from the table, the RPM error used for the look up is not linear. Instead, the graduation changes as the RPM error becomes larger. This is because as

the idle speed approaches the desired idle speed, you want to have more resolution to allow you to finely control the idle. At large errors, your more concerned with just trying to get the idle back under control, so resolution doesnt matter as much. Here is the first of seven such tables that contain the proportional airflow for various conditions.

Code:

```
;
;~~~~~
;-Proportional idle % airflow for low idle and in P/N
;~~~~~
;
; % airflow = val/327.68
;
;RPM error
LBL_$0BD6E:
DC.W 0 ; 0.0
DC.W 0 ; 12.5
DC.W 0 ; 25.0
DC.W 115 ; 37.5
DC.W 246 ; 50.0
DC.W 328 ;100.0
DC.W 492 ;150.0
DC.W 655 ;200.0
DC.W 721 ;300.0
DC.W 885 ;400.0
DC.W 983 ;500.0
```

Now that we see what the table looks like, lets see how the code performs the look up. The first step in the process is the RPM error. This is constructed from the desired idle speed, which is handled by a seperate loop of routines. Ultimately, though, the idle RPM error is constructed by subtracting the desired idle speed from the current idle speed. This appears below.

```
;
;~~~~~
; Construct idle RPM error term
;~~~~~
;
LAB_2197:
CLR.L D3 ;Preclear D3
MOVE EXT_1460.W,D3 ;Load D3 with desired idle speed
CLR.L D4 ;Preclear D4
MOVE EXT_1197.W,D4 ;Load D4 with engine RPM
SUB.L D3,D4 ;Subtract desired idle speed from current engine
;speed
CMPI.L #$FFFF8000,D4 ; -6400 RPM, min allowed result
BLT.S LAB_2199 ;Bra if RPM error out of range
;
CMPI.L #$00007FFF,D4 ;6399 RPM, max allowed result
BGT.S LAB_2198 ;Bra if RPM error out of range
;
CMPI #10240,D4 ;Compare result to 2000 RPM, upper limit for RPM
;error
BGT.S LAB_2198 ;Bra if RPM error >
;
CMPI #-10240,D4 ;Compare result to -2000 RPM, lower limit for RPM
;error
```

```

    BLT.S LAB_2199    ;bra if RPM error <
;
    BRA.S LAB_219A    ;Bra to save
;
; -Here to load upper limit for RPM error
;
LAB_2198:
    MOVE #10240,D4    ;Load 2000 RPM, max allowed RPM error
    BRA.S LAB_219A    ;Bra to continue
;
; -Here to load lower limit for RPM error
;
LAB_2199:
    MOVE #-10240,D4   ;Load -2000 RPM, min allowed RPM error
;
LAB_219A:
    MOVE D4,EXT_1464.W ;Save it, idle RPM error term

```

From the loop above, the idle RPM error = desired idle speed - current idle speed, and is limited from -2000 to 2000 RPM. Now that we have our base value used for the table look up, let's see how it's applied. Since the graduation of the RPM error scale used in the table lookup changes, we would suspect that the code would first have to format the RPM error for lookup. This indeed proves to be the case. First, the error is formatted so that the value used is the absolute value of the RPM error. This is due to the fact that signed table lookups weren't originally used in the earlier PCMs from which these routines originated from. Instead, there are separate tables for above and below the desired idle speed. This is shown below.

Code:

```

    MOVE EXT_1464.W,D3 ;Load D3 with idle RPM error
    BGE.S LAB_09BF     ;Bra if idle RPM error >=0
;
    NEG D3             ;Neg result, get abs. value of idle RPM error
;
LAB_09BF:
    MOVE D3,EXT_1101.W ;Save it, abs. idle RPM error

```

Now that we have our value used for the lookup, let's see how it's done. The code then takes the absolute value of the RPM error and applies various formulas to scale it based on the value of the RPM error. This is shown below.

Code:

```

;
; ~~~~~
; Scale idle RPM error for lookup
; ~~~~~
;
    CMPI #1024,D3     ;Compare abs. idle error to 200 RPM
    BLS.S LAB_2804    ;Bra if idle error <=
;
; -Idle RPM error > 200 RPM, RPM = (val - 1280) x .390
;
    LSR #1,D3         ;/2, now RPM / .390
    ADDI #1280,D3     ;Add in offset
    BRA.S LAB_2806    ;Bra to continue

```

```

;
LAB_2804:
    CMPI #256,D3    ;Compare idle RPM error to 50 RPM
    BLS.S LAB_2805  ;Bra if idle RPM error <=
;
;-50 RPM < Idle RPM error <= 200 RPM, RPM = (val - 768) x .195
;
    ADDI #768,D3    ;Add in offset
    BRA.S LAB_2806  ;Bra to continue
;
;-Idle RPM error <= 50 RPM, RPM = val x .0488
;
LAB_2805:
    ASL #2,D3      ;x4, now RPM / .0488
;
LAB_2806:
    CMPI #2560,D3   ;Compare to max allowed value for lookup
    BLS.S LAB_2807  ;Bra if scaled idle RPM error <=
;
    MOVE #2560,D3   ;Load max allowed value for lookup

```

From the formulas utilized in the above code segment, we now know what the graduations are in the table that was originally posted. The scaling goes as follows. For an error of 50 RPM and below, the graduations are in 12.5 RPM increments. For 50 - 200 RPM, its in 50 RPM increments, and from 200 - 500 RPM, its in 100 RPM increments.

Its important to get this right, because if you get the scaling off, you will end up editing the wrong section of the table when you go to tune. This can lead to a lot of insanity as the engine will not respond to your changes, and will end up acting weird in other areas of the table that you inadvertently changed. You can also end up getting the size of the table wrong as well.

Case in point: I've seen tables posted on some tuning sites that are actually two tables partially combined as one from a mistake in the interpretation of the table. The table in question was the force motor current vs. line pressure for the later model electronic transmissions. As an interesting side note, people incorrectly use this table to tune as the creators of the tuning packages dont include the desired line pressure tables that should be used instead. The end result is that the 'table' looks weird in the tuning package. People see this and go 'hmm... thats not right, let me change it to the way it should look'. The next post will usually be something like the following: 'I made these changes, and now my transmission slips and my adaptive modifiers are going crazy, I cant fix it. WTF???' and later maybe followed by 'Now my transmission is TRASHED, and I have to spend [insert dollar ammount here] to have it fixed!!!' What the user didnt know was that the reason they where having all the trouble is that they where actually editing TWO tables at once. The net result was that this threw the force motor calibration straight to hell. The adaptive modifiers are going 'crazy' because the force motor is no longer responding in a linear fashion to the closed loop pressure inputs. From this we can see how critical it is to get the table scaling and start/end addresses correct. My motto is "when in doubt, leave it out". What I mean by this is if your unsure of the scaling or the use of the table, leave it ALONE. You may do more harm than good by trying to edit something that you dont fully understand.


The above table example is by no means comprehensive, but it gives a general idea of how you go about finding the tables. The process usually goes as follows:

- Find out where the table is referenced. This will tell you what the intent of the table is, and what scaling is used for the values stored in the table.
- ID the base values used for the look up.
- Map out the code that formats the values for the lookup.
- Use the min and max values to determine the table size.
- Construct the table in the calibration section.
- Finally, examine the final result. Does it make sense? If not, then double check your work.

Well, that concludes a basic overview of the hacking process. I could write volumes on this, but this should be enough to give you a general overview of reverse engineering a typical late model PCM.

Q & A :.....

Quote:

Originally Posted by **JP86SS** 

*First I'd like to thank you for this writeup. Nice work, I'm sure this was not an overnight effort.*

*I did want to ask if the bin for these newer PCMs are all in one file such as the P4s. My confusion came in from the postings in the other thread where there were several files listed for one application with different bin files for transmission, speedometer and fuel.*

*Does the whole application bin reside in the SOP surface mount IC?*

*Also, The calibration values do not seem to be grouped into the lower areas like in the older stuff.*

*The values seem to be mixed in with the code pretty heavily (or code is mixed in some places)*


*This will make definition files difficult to manage with code changes (when it get to that point)*

*Any thoughts on this or is this just code to arrange the values in memory or something?*

The files you saw are all on one flash chip. Basically, GM organized the bin into sections so that each portion of the algo for the trans, idle, HVAC, system, etc. would have its own section for its calibration. These PCMs also boot from the bottom of the address area, and have a full stand-alone boot code thats used for boot-up and recovery reflashing should someone nuke the main portion of the code. The boot vector tables and boot code are at the bottom of the address map. Following these, there is each of the calibration sections. After the calibration section is the main OS. The main OS also has its own tables. These are large address tables used by the code for executing loops. The tables are used to configure which loops are executed by the routines. The tables come before any of the main code in the OS. IIRC, they start at \$20000, and are followed thereafter by the main OS, which houses all the main OBD/engine/transmission/diagnostics loops. Basically the whole bin looks roughly like this:



[\$7FFFF]  
-Subroutines for main OS  
-Main vector table  
-Main routines for the OS  
-Address jump tables and constants for OS  
-Calibration section  
-Boot kernal  
-Boot vector table  
[\$00000]


Originally Posted by **JP86SS** 

*This will make definition files difficult to manage with code changes (when it get to that point)*

*Any thoughts on this or is this just code to arrange the values in memory or something?*

Code changes may be more difficult to manage. Have to be careful that you set the checksums up right. Much of the empty space on the chip is no-mans land. The checksum is not performed on these sections, and some even overlap the addresses of the internal hardware, and should not be accessed. For a code change, you would need to be sure to update the checksum spans to include the new code and that the section of the chip is addressable. The very end of the chip thats free would be the best place to add stuff.

As far as the difinition files. With those you would need to make one file for each section. For example, the engine, transmission, speedo, etc. will all need their own XDFs. This is due to the fact that each section has its own assigned checksum. There are seven checksums for the calibration and OS as well as one additional checksum for the boot kernal. If you write to a section by accident, and dont update its checksum, the computer will endlessly reboot after it fails the checksum, possibly rendering it a paper-weight unless you manually reflash the chip.

Actually Scot, your correct. I forgot to include the section 

Although, on second thought, some of the hardcore atari sites like atari-forum.com might be better for the basics of disassembly. Most of the coding discussions there aren't really applicable to what we do as theyre working with game consols/dynamic PC type environments as opposed to more static embedded system code like the PCMs have. But... they have a fairly extensive repository of disassemblers and such, and lots of users that have experience working with the 68K assembly language.

For the most part, the only real big issue is finding a good disassmebler. The freeware one I started with didnt support some of the addressing modes like address indirect with indexing. Not an issue until I hit the O2 routines. 4 sensors each all handled in a similar manner, so they extensively use address indirect w/ indexing for saves, stores, and look-ups. Basically the entire O2 logic didn't disassemble at all. Right now I have to use IDA to 'fill in the blanks'

sort of speak. If you can get IDA, its a great program to use. It automatically stops when it hits stuff thats not code, so you can toggle where it actually does the disassembly. It also supports a lot of other features that the basic DOS disassemblers dont.

The other lesser issue is finding whats code and whats tables. The PCM has the boot kernal at the start of the flash chip, with constants organized into sections after that, followed by the constants for the OS, followed by the OS, which itself has some tables thrown in here and there along with the vector table. I'll probably add some more info on how to find some of that up front. The checksum routine can help as that will tell you in a broad fashion where everything is located.

Quote:

Originally Posted by **Zalfrin** 

*Here's a (possibly easy) question: In the case of a line like this: move.w (\$FFFF9116).w,(\$FFFA692).w*

*How does that translate into a physical address, or is it accessing an external device? PCM addresses only go to \$7FFFF... Assuming the first FF is a bug in IDA, it looks like that would be a transfer from some internal register to some other internal register... Does this make sense? I don't know, I'm looking at figure 3-5 in the 68332 user manual, it doesn't make much sense... On the left it says "internal registers" run from \$FF0000 to \$FFFFFF, then it expands into details on the right and defines \$FFF000 to \$FFFFFF... ??? what about \$FF0000 through \$FFF000, and what exactly is the blank space from \$FFF000 to \$FFFA00 supposed to represent? lol, this documentation seems sub-par. 😊 Thoughts?*

The address' are sort of backwards compared to the older ECMs. The first thing in the address map is the flash chip. Since the CPU32 core boots from the bottom of the address map (\$00000), the vector tables on the flash chip must come first. Therefore, the flash chip runs from \$00000-\$7FFFF. All internal registers, hardware, RAM, and TPU RAM are mapped in the \$FFXXXX range at the top of the address map. It varies by PCM, but in mine, the RAM, TPU, regs. and hardware start at \$FF8000 and run onward. The external hardware comes first, followed by the RAM, than TPU/hardware I/O and lastly the internal module registers.

You can differentiate when the flash chip is being accessed as you will see the addresses as *longs* as opposed to *words*. Since the flash chip is an entirely external memory device, a load from address \$A000 on the flash chip would be 'MOVE.L \$0000A000'. All other accesses for things such as teh RAM and internal registers are always done as word addressing. This is because the MPU inherently knows that any word address access is and only is for its internal registers, and not an external adressable memory device. A load from the address \$FFA000 in the RAM would be 'MOVE.W \$A000.W' or 'MOVE.W \$FFA000.W'. Both are interchangeable. The \$FF is truncated as its already assumed that word accesses are for the internal registers.

Not to add too much more confusion, but the internal registers are actually locatable right after the flash chip, or at the very top of the address map. On

the handful of PCMs Ive looked at, they have always been at the top of teh address map at \$FFXXXX and onward.

Quote:

Originally Posted by **ScotSea** 

*So that is what I am really asking. Is there a good 32 bit moto disassembler out there?*

*Thanks, Scot*

IDA is the only real good one Ive come across so far and is the ONLY one that supports the true CPU32 instruction set. The CPU32 instruction set has a few additional commands not present in the standard 680x0 instruction sets. As long as you set things up right in the beginning (start address of code and such), it seems to do a good job.

I used IRA (see 'The cook book approach to getting started with the '411 PCM' thread), or Intelligent Reverse Assembler. For the most part, it did ok, and produced code in the format like we're used to seeing, but it doesnt do some of the addressing modes. Basically, this produced an occasional group of commands that didn't reverse, but in the O2 routines that extensively use certain address modes, they didn't disassemble at all. Had I known this at the start, I probably wouldn't have used it...

There are a few other ones out there that Ive come across. At the time I was looking, there weren't really that many, but a few more have popped up. One additional one is dasm68 (see attached). It seems to work, but when I played with it, I couldnt get it to dump to a .txt file. It kept trying to just echo the disassembly to the dos shell.

No freeware disassembler will ever truely support the CPU32 instruction code since it is used in embedded industrial and automotive systems only, but any good one that works for the 68K should come close to what we need. I saw a few on that atari site that I have yet to try. I'll give them a test drive and see what I think. Even if some of the commands aren't supported, its not the end of the world. Motorola gives all the bitfield encoding in their CPU32 programmers guide. All the instructions are bitfield encoded (binary), so you can look up the instruction manually in the users guide if the compiler doesnt support it.