Dewtronics

M6811DIS v2.00/GDC v2.00

Code-Seeking Disassembler for the Motorola MC68HC11 Microprocessor

Version 1.00 of M6811DIS Written April 15, 1996, © Donna Whisnant
Documentation for Version 1.0 Written September 27, 1998, © Donna Whisnant

Version 1.20 of M6811DIS Written July 29, 1999, © Donna Whisnant
Documentation for Version 1.20 Written January 29, 2000, © Donna Whisnant

M6811DIS Version 2.00 / GDC Version 2.00 Written June 14, 2014, © Donna Whisnant
Documentation for M6811DIS v2.00/GDC v2.00 Written June 21, 2014, © Donna Whisnant
Last Update: June 21, 2014

Copyright

Table Of Contents

Introduction

A disassembler is a program that takes binary memory images and/or object code data files and converts them into the mnemonic equivalents for the processor the code was developed for. It is sort of like decompiling code except that if the original code was written in a higher level language (language other than assembly), you only get the equivalent assembly language code rather than the language the original code was written in.

So what is a disassembler good for? The primary use for a disassembler is to either reverse-engineer or hack a program. In the realm of software, typically reverse-engineering involves taking an entire program apart to figure out exactly how it functions, usually in an effort to understand the overall system and possibly improve upon it or otherwise extend its use. Hacking, on the other hand, typically involves taking a program apart only to the extent of finding one or more particular items of interest usually to modify those parts to achieve some goal, while not necessarily trying to gain a full working knowledge of how the whole system works.

An example of reverse engineering would be a complete disassembly of a vehicle ECM calibration to write out a complete description of the control algorithms used by the vehicle computer. An example of hacking would be disassembling that code only enough to find one or two numbers that contain the speed-limiter on the vehicle, without getting a working knowledge of how this speed-limiter actually works. There are times and places for both reverse-engineering and hacking and even combinations of the two. To any extent, one of the primary tools used by both hackers and reverse-engineers is the disassembler.

What is a code-seeking disassembler and why is it so special? Any binary program image, especially those for processors using Von Neumann architecture, will contain a mix of program bytes and data bytes. Traditional disassemblers typically start at the first address of the binary image and disassemble to the end of the binary image treating everything as code. In the end, you end up with a file that contains most of the correct code mixed with lots of garbage from the data. Sometimes this isn't a problem, especially if the data areas are small and very distinguishable. But, depending on the processor's opcode list, it can cause the disassembly in the good code sections to be skewed – resulting in several incorrect and/or incomplete opcodes that later have to be disassembled by hand. This is where the code-seeking disassembler comes in handy.

Unlike the typical "disassemble everything" method of the traditional disassembler, the code-seeking disassembler actively seeks out and disassembles sections that it sees as code while leaving the rest tagged as data. This is achieved by giving the disassembler one or more initial entry addresses into the code. From these entry points, the disassembler continues to follow through the code as it hits jumps, branches, and returns. In the end, you should have a perfect separation of code and data.

There are some complications to this. For example, what happens on a jump instruction that uses a register to obtain the address of the branch, such as is common with a jump table? The disassembler has no way of knowing exactly what the content of the register is, so it is forced to label the jump instruction as an "undetermined branch". Such tables have to be located by the user and added as additional entry points for the disassembler.

Also, suppose you don't enter all possible entry points – the result will be a file with chunks of code interpreted incorrectly as data. And there are cases where there are unused bits of code that never get executed – those will remain tagged as data. But overall, the code-seeking disassembler is far superior to its traditional counter-part and in many cases, with little user intervention, can produce a perfect separation of code and data, which greatly facilitates the reverse-engineering and/or hacking of the target code.

Is the disassembler output important? For hackers, the answer is "no". This is because a hacker is only interested in the code to the extent of finding the part(s) to achieve his hack. But, for the reverse-engineer, it is a very important aspect. Often after reverse engineering a program, it is desired to reassemble the code either in its original form (to test integrity and validity) or in an altered form after enhancements have been made. Many disassemblers don't address this issue and produce an output that isn't compatible with any existing assembler, resulting in hours of editing and reworking to get the code in the correct form. This disassembler solves the problem by targeting a specific assembler. With the specified assembler, it is guaranteed that the output from the disassembler, when reassembled will result in the original binary. The assembler this disassembler targets is the AS6811 written by Alan Baldwin at Kent State University's Physics Department (**not to be confused with the Motorola AS11 freeware assembler**). Alan's entire assembler set and relocating linker is a superb piece of workmanship, which is why it was chosen as the target output form for this disassembler. See **Reassembling a Disassembly** later in this document for more information on this assembler.

What else you need to know. In order to make sense out of the output from this disassembler and to effectively use this program, it is necessary to first familiarize yourself with the MC6811 microprocessor and have an understanding of assembly language and techniques in general. Such instruction is outside the scope of this document. For it, I refer you to documents such as the "M68HC11 Reference Manual" available from Motorola as document M68HC11RM/AD. And the processor-variant specific pocket reference guides, such as the "MC68HC11F1 Programming Reference Guide" and "MC68HC11E9 Programming Reference Guide" will come in handy as well – the Motorola part numbers for these documents are MC68HC11F1RG/AD and MC68HC11E9RG/AD, respectively. The one(s) you will need will be dependent upon the particular processor used by the device under study. For other variants of the HC11, the document number is typically MC68HC11 followed by the variant code and then "RG/AD". Technical data references for a specific series are also available. These typically have document numbers of MC68HC11 followed by the series code followed by "/D". Examples of these are "MC68HC11N/D" for the "N-Series" and "MC68HC11F1/D" for the F1 series. All of these documents are available from Motorola's website and can either be downloaded as .pdf files or ordered in printed form – or actually, checkout the Freescale website, as Motorola dumped (er, uh, sold off) their SPS chip business to Freescale (https://www.freescale.com/webapp/sps/library/prod_lib.jsp)

You will also need a method of obtaining the original memory image that you wish to disassemble. Again, that is outside of the scope of this document. To obtain the memory image, you will probably need to obtain an EPROM burner and/or reader. This document assumes that you've already obtained the target code and have saved it in a supported file format. M6811DIS uses built-in DFCs (Data File Converters) to load and/or save data files in various formats.

This version of M6811DIS is built with Binary, Intel Hex, and Motorola Hex DFC Libraries. Users can easily add additional file format types by creating a C++ class for it, derived from the base CDFCObject and adding it to the global CDFCArray. See the source code for details on how to do this..

You can also write additional disassemblers for other microprocessors using the base CDisassembler class and use the basic framework of this disassembler to handle the input, output, and control file details. This version of M6811DIS is contains the CM6811Disassembler GDC (Generic Disassembler Class). See the source code for details on how to create additional disassembler classes and add it to the disassembler framework.

In this document, as well as the disassembler output, hexadecimal values are expressed by prepending them with "0x".

Installation

The M6811DIS program has been completely rewritten from its original form and now only uses C++ with STL (Standard Template Library), making it fully cross-platform portable. This version was developed using Linux and has been cross-compiled to Windows and Mac. Native binary distributions of M6811DIS for these platforms are made available on the SourceForge website. But, you can also compile this application for nearly any system or platform.

Most distributions are provided as either a ZIP archive or a self-extracting executable or script. Installation generally consists of unzipping or extracting the contents of the distribution package and running it from that folder. You can make it run system-wide by copying it to a system executable folder, like /usr/local/bin or adding the path you've extracted it to to the system-wide path (such as on Windows).

The following files are distributed with this version (2.00):

• m6811dis	– The main program executable. (Linux and Mac, Statically Linked)
• m6811dis.exe	– The main program executable. (Windows Version)
• libgcc_s_sjlj-1.dll	– GCC runtime library (Windows Version Only)
• libstdc++-6.dll	– STL runtime library (Windows Version Only)
• libwinpthread-1.dll	– Windows pthreads runtime Library (Windows Version Only)
• m6811dis.pdf	– This documentation in PDF (Portable Document File) Format.
• av94bnbh.ctl	– A sample control file to get you started. (in examples folder)
• av94bnbh.log	– stdout/stderr output while disassembling av94bnbh.bin
• base.ctl	– Another sample control file from a real reverse-engineering project
• base.log	– stdout/stderr output while disassembling base.bin
• portsf1.asm	– Assembler file for the F1 HC11. Used when reassembling (AS6811).
• portsf1.h	– Include file for the F1 HC11. Used when reassembling (AS6811).
• portse9.asm	– Assembler file for the E9 HC11. Used when reassembling (AS6811).
• portse9.h	– Include file for the E9 HC11. Used when reassembling (AS6811).
• readme.txt	– Text file containing last minute release notes and installation info.

Note that the "ports" files have nothing to do with the disassembly process itself and are not needed to successfully disassemble a file, but they are rather useful when reassembling the disassembled code. They are written to work with Alan Baldwin's AS6811 disassembler, as is the output of the disassembler. The F1 and E9 are included because they are the most common variants of the HC11.

Usage

Overview

As mentioned in the introduction, this disassembler is a code-seeking disassembler. Therefore, it is necessary to specify all code entry addresses and indirect vectors (such as interrupt vectors) used in the target code. A minimum of only one entry address is required, but often it is necessary to specify multiple entry points and/or indirect vectors, and it is also desirable to be able to specify meaningful names, or labels, for these. It would be cumbersome to have to specify these each time on the command line, not to mention the fact that you'd run out of command-line space on some systems. Therefore, the entering of these entry-points, labels, and indirect-vectors is done with a "Control File".

First, use a text editor of your choice and create a control file for the file you wish to disassemble. The **Control Files** section in this document describes the exact format and available commands to use in creating the Control File. A sample Control File is included there, as well as in the distribution. As a very minimum, your Control File should include an "input" or "load" statement and an "output" statement to specify source and destination files, respectively. And, it should include at least one entry point, in some form, for the code disassembly. If no entry points are specified in the Control File from either "entry" statements or "indirect" statements, the load address of the file is assumed to be an entry point. If the "input", "load", and "output" statements in your Control File do not specify full paths, the current directory will be used.

With your Control File complete, bring up a command prompt, and enter "m6811dis" followed by the name of the Control File that you used, from within the proper directory. It is recommended that you use the extension of ".ctl" for your Control Files, though the disassembler will **not** append the ".ctl" should you fail to specify it with the filename on the command line. Therefore, if you use the ".ctl" extension, you must type it with the filename. The program also allows for multiple Control Files. If using more than one control file, specify the pathnames for each on the command line. The Control Files will be parsed in the order they are specified and will together act as one large Control File, however, this allows you to specify pieces and parts of an overall memory image separately, should your memory image be segmented. Once run, the disassembler will first display its findings from parsing the specified Control File(s), and then it will load the source file(s), resolve any specified indirects, and disassemble the source file(s) to the specified output file.

During the disassembly process, the disassembler will display any labels that are created during the disassembly process, as well as any warning or error messages. Labels are created anytime a direct extended memory reference is encountered, regardless of whether it is an absolute or relative address. For example, suppose a program contains a command to load the 'X-register' from the direct address of 0x103A. This would correspond to the instruction "ldx 0x103A". The label "L103A" will be automatically created and assigned to address 0x103A and the disassembler will output "ldx L103A". If the address wasn't included within the loaded file range, the disassembler will also output an equate of "L103A = 0x103A" so that the assembler will know that L103A is equivalent to the value 0x103A during re-assembly. If the address was within the loaded file's range, then the output line coincident with that address will be prepended with the label followed by a colon – such as "L103A:". However, if the load instruction was an immediate value, rather than an address, such as the instruction "ldx #0x103A", the assembler will not assign a label for 0x103A. The disassembler can only assume, in this latter case, that 0x103A is a constant value and has no address relevancy. This may or may not be the case. If it does have address relevancy, then you must manually rename it in the output file using a search and replace. In most programs, immediate values are usually just constant values, but occasionally you'll run into one that is an address, typically loaded into an index register, for indirect addressing in subsequent instructions.

If you wish to use more meaningful names other than something like "L103A", then you should add "label" commands to the Control File and rerun the disassembler. The disassembler will then use the specified label for the specified address, rather than making up its own "Lxxxx" label. The disassembler supports label names of any length, limited only by the available system memory. However, you should keep in mind any limitations your assembler(s) might have regarding name length, should you plan on re-assembling the file later.

The disassembler is a two-pass disassembler. During the first pass, it iterates through the specified list of entry addresses tagging those locations as code. For each entry address, it continues to tag successive addresses as code until it reaches an instruction that ends the code section – such as an unconditional jump or a return-from-subroutine (RTS) statement. Whenever a jump (or branch) is encountered, the target address, if it is determinable (that is, isn't dependent on a register value or other unknown value), is added to the list of entry points. This process continues until all entry points in the list have been exhausted. During the second pass, it iterates over the entire length of the memory image and writes the output disassembly file. All addresses that were tagged as being code during the first pass will be outputted as code, otherwise, they will be treated as data and outputted as either binary or ASCII data (depending on Control File settings and byte values).

The screen output during the disassembly process, containing new labels and disassembly messages, is send to "stdout". Errors and warning messages are sent to "stderr". This allows the output to be redirected into a log file for later reference using the stdout and/or stderr redirection operators (">", "2>", etc) on the command line – refer to your operation system documentation on how to do input/output redirection and piping.

Step-by-Step Walk-Through

Here is an example dump of the screen output produced while running M6811DIS with the sample Control File shown in the *Control Files* section, captured by using a stdout redirection. It is shown here in its entirety because this it a very typical illustration of what most disassembly runs will be like and it allows us to describe and explain what each part of the screen output is for:

```
M6811 Disassembler V2.00 -- GDC V2.00
Copyright(c)1996-2014 by Donna Whisnant

Reading and Parsing Control File: "av94bnbh.ctl"...
Loading "av94bnbh.bin" at offset 0x4000 using binary library...

        1 Source File: av94bnbh.bin

        Disassembly Output File: av94bnbh.dis

        Functions Output File: av94bnbh.fnc

        Memory Mappings:
            ROM Memory Map:
                0x4000 - 0xFFFF  (Size: 0xC000)
            RAM Memory Map: <Not Defined>
             IO Memory Map: <Not Defined>

        16 Entry Points:
                0x7C0B
                0x7C12
                0x7C1C
                0x7C22
                0x7C35
                0x7C6B
                0x7C7C
                0x7C83
                0x7C9C
                0x7CA0
                0x7CAA
                0x7CAE
                0x7CBE
                0x7CC2
                0x7CCC
                0x7CDD

        0 Exit Functions Defined

        21 Unique Labels Defined:
                0xFFD6=scivect
                0xFFD8=spivect
                0xFFDA=paievect
                0xFFDC=paovect
                0xFFDE=tovfvect
                0xFFE0=ti4o5vect
                0xFFE2=to4vect
                0xFFE4=to3vect
                0xFFE6=to2vect
                0xFFE8=to1vect
                0xFFEA=ti3vect
                0xFFEC=ti2vect
                0xFFEE=ti1vect
                0xFFF0=rtivect
                0xFFF2=irqvect
                0xFFF4=xirqvect
                0xFFF6=swivect
                0xFFF8=ilopvect
                0xFFFA=copvect
                0xFFFC=cmonvect
                0xFFFE=rstvect

Writing byte value comments for ASCII data in disassembly file.

Compiling Indirect Code (branch) Table as specified in Control File...
        21 Indirect Code Vectors:
                [0xFFD6] -> 0xF494
                [0xFFD8] -> 0xF8EE
                [0xFFDA] -> 0xF8E4
                [0xFFDC] -> 0xF8E4
                [0xFFDE] -> 0xF8E4
                [0xFFE0] -> 0x7922
                [0xFFE2] -> 0x7986
                [0xFFE4] -> 0x79EA
                [0xFFE6] -> 0xF8D9
                [0xFFE8] -> 0xCC8A
                [0xFFEA] -> 0xF8D9
                [0xFFEC] -> 0xF8D9
                [0xFFEE] -> 0xF8D9
                [0xFFF0] -> 0xF8D9
                [0xFFF2] -> 0x7597
                [0xFFF4] -> 0xF8B3
                [0xFFF6] -> 0xF8AE
                [0xFFF8] -> 0xF8C4
                [0xFFFA] -> 0xF8C9
                [0xFFFC] -> 0xF8CE
                [0xFFFE] -> 0xF8D3

Compiling Indirect Data Table as specified in Control File...
        0 Indirect Data Vectors


Pass 1 - Finding Code, Data, and Labels...
LC3BA  LCD23  LC3CE  LC4B3  LAE65  LEEAA  LF26C  LD425  L7C31  L7C34
LC6D1  L0002  L7C4A  LE860  LE893  LC6F6  LC71D  L7C59  LD2D7  LD399
L7C7B  LC506  LC923  LCC1E  L7C8C  L7C8F  LCCC9  LCE3D  LA386  LD414
LD1D9  L7CBA  L7CBD  LCA43  LCACF  LD9BB  LDA49  L02BE  L7CD9  L0004
L7CDC  L7CE6  L7CE9  L0044  L759E  LF8BB  L300E  L0062  L3016  L75C3
L0050  L306F  L75DB  L3030  L3031  L082E  L004F  L75E2  L76FD  L0068
L006A  L75F5  L02F0  L75FF  L7873  L0006  LF15E  L7626  L02F2  L78D4
L026F  L7629  L76D7  L76DD  L0019  L7634  L0279  L763E  L0273  L76A4
L7654  L0053  L7678  L4956  LF1ED  L7669  L02F1  L7675  L7907  L768A
L0825  L76A0  L084E  L76BD  L0827  L76D3  L862B  L76E5  L0055  L4D8E
L76FA  L4D8D  L02EC  L7743  L028B  L770B  L7720  L081E  L495A  L028A
L7740  L400C  L7786  L0051  L7767  L77C9  L0277  L7762  L77AF  L0829
L7789  L4920  L77BF  L7812  L77A5  L77BC  L082A  L77D5  L77F9  L7801
L78A4  L081F  L77E8  L7874  L3FFA  L0073  L0067  L450C  L783B  L01FC
L48D0  LF116  L783D  L400B  L7851  L7865  L003F  L7870  L7883  L0821
L789B  L78A3  L78B3  L0823  L78CB  L78D3  L78E4  L78F8  L4D8F  L7906
L5B03  L7921  L7919  L4E65  L7972  L0046  L7954  L025D  L7942  L7945
L7962  L795D  L7983  L79D6  L79B8  L79A6  L79A9  L79C6  L79C1  L79E7
L301A  L3023  L0000  L7A05  LF705  L003A  L7A17  L7A14  L0003  L7A20
L003B  L7A3B  L0391  L0072  L0392  L9286  LF08B  L006F  L7A73  L7A70
L4E5F  LF137  L4E5E  L4139  L0047  L7A7D  L4138  L01E6  LF0F6  L01E9
L7A95  L3032  L3033  L3034  L0148  LEEE2  L7CF6  L7AD4  L3FC4  L080D
L080C  L0146  L3068  L7E04  L7B00  L0043  L7AFF  L00F8  L7AF7  L4953
L7B0F  L7B11  L001A  L7B2D  L7B7C  LAD0C  L303A  L7B51  L924A  LA53C
LD075  LD11D  LF285  L7B49  LB773  LCF65  LAF40  L7E2E  LCC05  L7B6A
L0393  L7B6D  LF081  L7B79  L94B3  LC307  LB3B5  L0082  L7BA9  L7BB7
L7BE6  L7BAC  LC32A  LE50E  LE542  LC78A  LC7E0  LC805  L0016  L7C63
L0017  L7C6A  LF279  L9F7C  LC879  LAD62  LC30B  LE3ED  LE4B9  LE046
L004C  L7D08  L014A  L0149  L7D8B  L014E  L7D17  L0147  L7D22  L7D2D
L7D35  L7D36  L014D  L7D45  L7D51  L7D6B  L7D79  L7D76  L7D88  L008A
L7DC6  L00F6  L6925  L7DC4  L6927  L6926  L00AF  L00A1  L7DBF  L7DDA
L7E03  L6929  L6928  L7E01  L004D  L7E0E  L0096  L7E17  L7E12  L00F5
L7E2D  L5D04  L7E23  L02A1  L7E73  L7E90  L4009  L3039  L7E71  L7E8A
L003E  L0246  L02A0  L0041  L400F  L7EA7  L0085  L7ECA  L0063  L0065
L7EB5  L413B  L01E8  L7ED2  L7ECD  L413A  L01FD  LB270  L0092  L7EE6
L7EEC  L5B23  L48B3  L7EFD  L029E  L48B1  L01F8  L7F10  L48B2  L01F5
L7F23  L91B6  L027A  L7F89  LAF17  L01E3  L01EA  L0201  L01FF  L8695
L7F52  L7F7D  L7F80  L860F  L85D8  L003D  L7F95  L7FFA  L01ED  L48B6
L7FAE  L48B5  L48B7  L7FBB  L7FFD  LF153  L48B8  L7FD4  L7FE6  L0260
L7FF4  L0005  L8001  L0261  L804F  L0200  L8014  L801D  L48B9  L8022
L02A7  L48BB  L803D  L48BA  L0207  L804A  L8053  L0262  L8069  L0208
L8075  L808B  L48CE  L8098  LF17B  L80B8  L80A0  L0232  L4905  L80B5
L0254  L0259  L4D8A  L025B  L0257  L020B  L80ED  L80F0  L0255  L026D
L4D88  L8112  L0271  L01D5  L0275  L026B  L8138  L813B  L81D1  L817F
L8156  L514B  L8153  L084D  L0812  L514C  L514F  L514D  L8170  L514E
L0061  L8179  L8300  L818A  L81E9  L8198  L81A0  L0093  L81A2  L81C5
L495C  L028D  L495B  L81EC  L81DB  L087F  L81FE  L48BC  L8218  L8207
L8213  L48BD  L821A  L8222  L822F  L8247  L025F  L8236  L006E  L48C0
L081D  L8258  L8255  L493A  L8262  L8330  L82CA  L8284  L82E0  L828F
L493B  L82B8  L493C  L01F2  L400E  L82AA  L4933  L82E3  L4934  L0266
L82C6  L4930  L82E8  L0268  L4935  L82DA  L4937  L8325  L82FB  L4931
```

```
L82F6  L4932  L0269  L831E  L4938  L0267  L4939  L8315  L81DF  L83D0
L8341  L8349  L83CA  L082B  L8357  L8362  L492C  L836B  L492D  L837B
L8390  L0265  L839D  L839F  L492F  L83B4  L492E  L83EE  L48D1  L48D2
L48D3  L83F4  L006C  L8425  L8402  L4510  L840E  L8416  L450F  L8422
L842E  L01EF  L8449  L844F  L0292  L8473  L847F  L847B  L848E  L84A0
L859F  L84DE  L84AD  L84D3  L84B7  L4922  L84DC  L496A  L84D9  L8555
L84E9  L854C  L84F2  L8507  L3FCE  LF08A  L3FC0  L8518  L0281  L8535
L027F  L027D  L0280  L027C  L8559  L4928  L8576  L492A  L8599  L858C
L8586  L85AE  L85C9  LF15A  L4967  L85D2  L4963  L4965  L85EB  L85FA
L4969  L8609  L860C  L861B  L8720  LD3CA  L862A  L86F6  L8627  L8A01
L8F97  L864F  L3FF2  L3FFC  L4D8B  L86C4  L866E  L86BA  L86DA  L86F5
L86F2  L4142  L8681  L8689  L8694  L8708  L871F  L8712  L3FCA  L0228
L8745  L872E  L8734  L873F  L0264  L493F  L8756  L8758  L876E  L4940
L0290  L4942  L028F  L8789  L0070  L8790  L4E4E  L028E  L493D  L87A2
L87BB  L87AF  L87B7  L01F7  L87BD  L87CA  L0298  L87D0  L029F  L87D3
L87E9  L87E6  L885C  L8873  L87F1  L0052  L885F  L8859  L880D  L8818
L881C  L8825  L8834  L4901  L4902  L4903  L4904  L0071  L0282  L029D
L89F4  L8889  L0291  L8883  L88A1  L8893  L494B  L889B  L494C  L02A3
L88F6  L4943  L88C9  L88DB  L0285  L88E6  L88F2  L891F  L8908  L8932
L894C  L02A4  L8962  L89B0  L493E  L8970  L897B  L897C  L8998  L4907
L89AD  L8995  L89A3  L4906  L89C5  L4944  L89BD  L89CC  L89E9  L4908
L89FB  L48DD  L8A1B  L8A50  L48C7  L8A35  L01EC  L8A28  L8A32  L02EB
L8A3B  L8A52  L0295  L8A47  L0289  L0287  L8A7B  L8A6B  L48E2  L8A71
L48CF  L8AEE  L8AFC  L48D8  L8AE9  L48D4  L8AC9  L8AB0  L8AC1  L0294
L48D5  L8AD9  L8B05  L8AD3  L48EE  L8AE4  L8B02  L8AFF  L48D6  L8AF9
L8B3F  L8B13  L4952  L48DA  L48DB  L8B39  L082C  L48DC  L8B31  L48E0
L48E1  L4909  L8B49  L4E71  L8B4D  L4955  L8B6B  L4957  L8B64  L8B75
L8D50  L4917  L8B8D  L4918  L0286  L8B9C  L4919  L0203  L0204  L0205
L8BCA  L490D  L8BD2  L8CDF  L8C34  L8BDD  L490A  L8BE8  L490C  L490B
L490E  L0206  L01FE  L8C09  L8C31  L01F0  L8C17  L8C1E  L8CD6  L029A
L4910  L8C3A  L029B  L4911  L8C36  L028C  L4913  L8CA4  L8C5A  L4914
L8C7C  L8C88  L491B  L8C9D  L8CA3  L4912  L8CC0  L4915  L4916  L491A
L8D1B  L0818  L491C  L8CF0  L0299  L490F  L8D0C  L8D15  L8CFB  L8C39
L8D0B  L8D36  L8D49  L0397  L0399  L1800
     *** Warning:  Branch Ref: 0x1800 is outside of Loaded Source File.
L026E  L8D71  L081A  L8D68  L48EC  L8DAE  L8D96  L8DA0  L8DA6  L026A
L8E1D  L8E56  L8E1F  L8DCE  L48EA  L8DCA  L8DD9  L8E04  L48EB  L8E00
L8DFE  L8E1A  L8E18  L8E5C  L48E4  L8E34  L48E5  L48E6  L8E41  L48E7
L48E8  L48E9  L48F2  L8E75  L8E70  L48F0  L8E73  L48EF  L8E7B  LEF1D
L8ED2  L8E99  L8EA2  L8ECC  L026C  L8EC9  L8EBB  L48FF  L48FD  L4900
L48FE  L0263  L8EFD  L48BE  L48BF  L01EE  L8F00  L8F4B  L0297  L8F1E
L8F89  L0296  L8F2C  L48E3  L8F3D  L8F4E  L8F57  L8F82  L8F70  L48F1
L8F6B  L8F7D  L8F7A  L8FA6  L8FAC  L02BC  L0075  L50CE  L02B1  L8FBB
L02B6  L8FC7  L1809
     *** Warning:  Branch Ref: 0x1809 is outside of Loaded Source File.
L8FA8  L50D0  L901B  L8FFA  L8FDA  L8FDD  L02B7  L9016  L50D7  L8FF2
L02A9  L9004  L9007  L50D1  L901E  L9039  L9033  L9042  L90F3  LF136
L905A  L9064  L9067  L02AD  L50D3  L9082  L9085  L02B3  L02B5  L90AF
L50D4  L90A2  L90A5  L90DB  L90C2  L50D8  L90C8  L9176  L90D5  L90D8
L917A  L90E6  L90F0  L9120  L02B0  L9144  L9105  L916E  L912A  L9148
L50DC  L50DB  L913B  L50DD  L9177  L9153  L9161  L9164  L02AA  L9173
L50D9  L919A  L918D  L9197  L91B5  L02AB  L91B2  L50CF  L91BF  L02BD
L91D5  L91D8  L02B8  L02BA  L920E  L91FD  L920A  L9221  L9218  L9205
L922E  L9236  L9244  L0395  L0396  L306C  L9268  L9262  L927C  L9285
L9282  L000A  L9292  L929B  L92A0  L4EAC  L92E3  L0007  L0008  L92AF
L0009  L92B7  L92C2  L92C9  L92D9  L92DD  L92E1  L94BA  L1803
     *** Warning:  Branch Ref: 0x1803 is outside of Loaded Source File.
L0038  L94C4  LA4DD  L94DC  L94D3  L997E  L9387  L94ED  L94EA  L4EA6
L997C  L4EAB  L4EA8  L02CA  L02CC  L93A0  L4EA9  L93A5  L93A7  L4EAA
L02CD  L02CF  L4EA7  L02C0  L02C2  L93C0  L93C5  L93C7  L02C3  L02C5
L50CA  L02D4  LA4D0  L94F4  L961A  L02A6  L0866  L0867  L4EAE  L086B
L4EAF  L086C  L9524  L4E85  L9532  L085C  L9551  L4EB2  L0857  L4EB1
L085D  L4EB0  L085E  L086D  L0879  L087A  L0876  L0877  L0887  L087B
L0864  L086E  L0036  L0037  L0039  L4FB4  L0859  L02D0  L02D2  L02C6
L02C8  L0875  L95DA  L95D5  L95EB  L95E6  L95E3  L95EE  L0862  L0871
L087D  L4E87  L9614  L960E  L9611  L9920  L962A  L9654  L9632  L009C
L9684  L4F19  L4F1B  L4F1A  L4F1F  L9674  L4F20  L4F21  L4F1C  L4F1E
L4F1D  L4F22  L4F23  L9690  L9696  L96C5  L970F  L4F05  L4F06  L96B4
L4F07  L96BC  L4F09  L4F08  L96FF  L4F18  L0878  L96E4  L0252  LF141
L96F8  L4F12  L4F13  L96F0  L4F14  L4F16  L4F15  L971A  L9781  L973C
L9730  L9756  L088F  L0891  L974C  L0895  L0897  L977E  L977B  L9775
L0893  L0899  L4E86  L97A5  L932A  L97A2  L4F24  L4F26  L087C  L4F25
L97AF  L97B3  L9331  L9332  L97BC  L0869  L97C6  L97CA  L0884  L97D3
L97E1  L0858  L97ED  L4F76  L97FA  L4F77  L9807  L4F78  L0882  L0885
L9819  L4F75  L9820  L9829  L4F74  L9AA8  L9D03  L9D89  L9840  L984E
L9855  L9863  L0870  L986A  L9874  L9877  L9891  L988A  L9899  L509F
L98B5  L509E  L98B8  L98C6  L98BE  L98C0  L0880  L98D6  L9913  L98FD
L50A2  L98EE  L98F2  L50B5  L0888  L990B  L50A0  L0889  L9938  L9963
L994F  L50BE  L50BF  L088B  L9AB2  L9ABA  L4EE8  L9AD2  L9ADD  L4EE9
L9AE8  L4EEA  L9B10  L9AFE  L9B09  L9B2A  L9B3B  L9B41  L9CE9  L9B50
L9B82  L9B70  L0868  L504B  L504C  L9B7C  L9CBD  L9B8D  L9BA1  L9B96
L9B9F  L9BB2  L9BBC  L9C31  L50B8  L9BF9  L50B9  L9BDC  L50BA  L50BC
L9BF6  L9C5F  L9C0A  L9C23  L9C1E  L9C2E  L9C49  L9C3E  L9C44  L9C62
L9C56  L9C5C  L9C6D  L9C94  L9C81  L9C8F  L50BB  L9CBC  L9CDC  L9CCC
L9CD3  L9CD9  L9CE7  L0873  L9D13  L9D53  L9D1A  L9D20  L9D26  L4FBF
L9D3D  L9D4B  L4FC0  L9D47  L4FC1  L9D56  L9D5E  L9D72  L9D67  L9D70
L9D95  L9D9F  L9D9D  L9DDF  L9DB7  L9DC2  L9DC8  L9DBD  L5030  L9DC5
L5031  L9DD5  L9DD8  L9F83  LA370  L997F  L9FAD  L4EB6  L4EB7  L9FCD
L50B4  L9FCA  L9FD6  LA01C  L9986  L9AA7  L9992  L99A2  L99AC  L4F7A
L99C6  L4F7B  L4F7C  L99C3  L4F73  L4F79  L99D5  L99ED  L4F7D  L4F7F
L99F2  L085B  L9A20  L4F7E  L9A05  L9A00  L9A15  L4F80  L9A1F  L4F81
L4F82  L4F83  L9A31  L085F  L9A3C  L9A8B  L4F84  L9A50  L4F85  L9A56
L9A7F  L9333  L9A6B  L0860  L9A66  L0861  L9A7B  L4F87  L9A82  L3008
L933F  L9340  L9A73  L9A7E  L4F86  L9AA1  L9AA4  L9FDD  LA05F  L9FE3
L9FEC  L0319  LA031  L50BD  LA000  L50C2  L088D  LA0B1  L0317  L50C3
```

```
L50C5  LA01F  LA0B4  L50C7  LA029  L50C8  LA03A  LA053  L50C1  LA089
L50C0  LA07D  LA06C  LA075  L02D5  L50CD  LA09C  L50C4  LA0A1  L50C6
L92F1  L92FE  L9305  L004B  L9309  LA0BF  LA1AE  L931F  L50CB  L50C9
LA0C7  LA13F  L4EB3  LA0D5  LA0EC  L4EB4  LA0FD  LA114  L4EB5  LA135
LA176  LA1AC  L4EE6  LA15B  LA170  L4EE7  LA1A9  LA1A4  L4EE4  LA18B
LA1A0  L4EE5  LA203  LA1B4  L4FB5  L4FB6  L0018  LA1EB  LA1E3  LA1FA
LA22A  LA226  LA21B  L4F17  LA227  LA238  LA23B  LA35E  LA249  LA250
LA273  L4FBC  LA291  LA269  LA2B1  L4FBB  LA287  LA2A3  LA29E  LA2CF
LA2AC  LA2C3  LA2BE  LA2CC  LA2E0  LA2EB  LA2FF  LA31F  LA313  LA30D
LA30E  LA340  LA319  LA332  LA32D  LA33B  LA352  LA34D  LA35B  LA367
LA37A  LA382  L086A  LA38D  LA4DC  L9DE0  LA3B0  LA39F  L4F27  LA3A5
LA3AD  LA3D0  LA3BC  LA3CD  L9DE9  L9DEB  L9E16  L5064  L9E0F  L9E01
L5065  L9E13  L9E51  L9E20  L9E2A  L5066  L9E3A  L9E33  L507E  L9E46
L507F  L9E4A  L9E70  L02A5  L5090  L9E6D  L5091  L5092  L9EB2  L9E8D
L9ECF  L9EA8  L5096  L9EA5  L9EBE  L9ECC  L0886  L508D  L9ECB  L9EDB
L0883  L9EE9  L9F78  L9EFE  L5094  L9EFA  L5095  L9F13  L9F18  L9F31
L9F2E  L9F48  L5097  L9F45  L9F60  L9F58  L507D  L9F7B  L9F6D  LA3C6
LA400  LA3DC  LA3FD  LA3EC  LA3F6  LA432  LA421  L4FBD  L4FBE  LA46E
L5B02  LA44A  LA457  L50CC  LA465  LA460  LA46B  LA4BC  LA4B6  L50B6
L088E  L50B7  LA4B1  L4EAD  LA53B  LA4F0  LA521  LA516  L0398  LA532
LA548  LA552  LA562  LA56C  L01F1  LA581  LA586  LA5B7  L4E5A  L024C
L4E5C  LA5BC  L4E5B  LA5B2  L4E5D  LA5C2  LA613  L0283  L020D  LA5E0
L0225  LA616  LA5F1  LA657  LA655  L414D  L0215  LA60E  LA671  LA620
LA627  L4134  LA636  L0233  L4143  LA644  LA6BB  L4136  LA66C  L0234
L4144  LA66D  L4958  LA67F  LA683  LA694  L4D8C  LA6B0  L005F  LA6AD
LA6B7  LA6D4  L083D  LA6ED  LA6F4  L50DA  LA721  L02AF  LA72F  LA731
LA742  LA74D  L0837  LA75B  LF0D3  L01E5  LA76B  LA77A  L0209  LA77D
LA78E  LA79F  LA7AF  LA7AA  L083F  LA7B9  L020F  LA7D6  L01E4  LA7EC
LA7FD  L021B  LA810  LA81A  LA830  L4141  L413D  L021C  L400D  LA849
L021E  L0841  L022D  L022E  L413F  LA8A3  LA88D  LA882  LA895  L4854
L0217  L021F  L02EE  L413C  L413E  L0154  L0220  LA8F1  L021D  L083C
LA90C  LA914  L415D  LA91D  LA977  L4159  L415A  LA932  L415B  LA939
L415C  LA954  L415F  LA95D  LA967  LA973  LA97D  LA980  LA9CC  LA995
LA9A2  LA9A8  L450A  LA9C0  L450B  LA9BC  LA9C3  L46E8  LAA40  L46E9
LA9EC  L0838  L083A  LAA18  LAA23  L083B  LAA35  LAA3D  LAA69  L4151
LAA5C  L022B  LAAA3  LAAA6  L414E  L4150  LAA7B  L414F  LAA8D  LAA90
LAAAD  L1812
     *** Warning:  Branch Ref: 0x1812 is outside of Loaded Source File.
LAAE0  LAADD  L4160  LAACD  L4163  LAAD2  L4164  L4161  L4133  L0211
L4145  LAAF9  LAB47  L45D0  LAB24  L0231  LAB17  L0842  L46D5  L46D6
L46D7  L46D8  L0230  LAB60  LAB59  LAB63  LAB91  L45CE  LAB72  L45CF
LABA2  LAB7B  L4E6D  L0843  LAB87  L022F  LAB89  L0844  LABB2  L414B
LABD5  LABD2  LABC9  L4149  L4147  LABF9  LABE9  LABF4  L039A  L0213
LAC0B  LAC1D  LAC1B  LAC24  L180F
     *** Warning:  Branch Ref: 0x180F is outside of Loaded Source File.
L0224  LAC3F  L3FF6  LAC5D  L3FE8  L3FDC  L3FE6  LACEC  LACCB  LACD7
LACE0  LACD1  LACB9  L4E67  LACC8  L024F  L4E68  LACCE  LAD5E  L3FEC
L3FE4  LACFC  LAD5B  LAD59  LAD2C  LAD44  LAD4B  LAD6C  LAE64  LAD76
LAD8B  LAD8E  LAE03  LAE08  L3FE0  L0847  L0849  LADF3  L0845  LADB2
L084B  LAE0D  LADE4  L5119  LAE00  L511A  L0846  LAE21  LAE23  LAE31
LAE38  LAE42  LAE58  LAE6F  LAF16  LAE8B  L5151  LAE99  L5155  L0853
LAF0D  LAEA2  LAF02  L5150  LAF07  L0854  LAEB7  L5152  LAEBA  L5153
LAEC7  LAECC  LAEE3  LAF0A  LAF10  L5154  LAF3C  LAF3F  L3012  L0851
L5181  L5183  LAF5C  LAF62  LB106  LB12A  LB291  LB2DD  LB375  LF42B
LF2EB  LB35E  L00F3  L0076  L3000  LAF6F  L3002  L3060  L3062  L0098
L00D3  LAF9B  LAF96  L0084  LAFB5  LAFAB  LAFA6  LAFB2  LAFC5  LAFCE
LAFD1  L0097  LAFDF  LAFED  L0095  LB001  LB00B  LB01C  L5D23  L019C
LB026  LB02E  LB037  LB054  L009B  LB048  L009D  L0077  LB064  LB06E
LB07D  L00B1  L00CB  LF23F  LB096  LB099  L00B7  L00BE  L5D02  LB0B2
LB0D0  LB0D2  L5D13  L5D12  LE34C  LB0FB  L5D16  LB0F8  L00C6  L0099
L009A  LB121  LB124  LE677  LEA7B  LE4BA  LB13B  LB14E  L001D  LB15D
LB15F  LB15B  L5D0D  LE576  LE5B7  LE38F  LE96F  LB175  LE5F6  LEDD4
LE930  LE3CB  LE4A4  LE3B1  L00A0  LB27D  L5B26  LB28A  LB28D  L0197
L019D  L00BC  L00C0  LB2C9  L00E5  LF0BD  LB2DA  L00E7  LB2FF  LB318
LB2FA  L015D  L016E  LB316  L00CD  LB312  L00CF  L5D18  L0102  LB33A
LB344  L0814  L00D1  L0190  L5B43  LB374  LB370  LB384  L014C  LB38C
LB38F  LB3BE  LB3C7  LB3C8  LB3FB  LB486  L5D1D  LB3F4  L5D1F  L5D21
LB3FA  L00FA  L5D0C  L01C9  L01CA  L01CD  L01CB  L01CE  L5D0E  LB426
LB424  LB40D  LB43D  LB458  L01D0  L0081  LB472  LB47A  L00D5  LB48D
LB4DD  L007F  LB4E8  LB4A5  LB4AC  LB4B6  L0181  L00D4  L00F7  L00D9
L00DA  LB4D3  L008B  L008C  L00AC  L018B  LB552  L00D6  LB504  LB549
LB524  LB51E  LB52D  LB543  LB53B  LB553  LB5FE  LEBB1  LB562  LB574
LB57C  LB586  LB588  LB591  LB597  L0083  LB59F  LB5A1  LB5AB  LB5AD
LB5B3  LB5BF  LE414  LB5CE  LB5DB  LB5E3  L0180  LB5EB  LB5F1  L008D
LB5FD  L0114  LB64D  LB631  LD96D  LB64C  LB665  L00D7  L5FC5  LB67D
LB691  LB69B  LB74E  L601D  LB6AC  LB6CE  LB6BA  LB6C1  LB6C7  L00DB
L0145  L00D8  LB6FA  LB6E9  LB6F8  LB70B  LB701  LB713  LB72A  LB733
L0113  LB77C  LB78B  LB78C  LBCED  LBDF3  LBD4D  LBE8F  L0111  LB794
LBACB  LB7A2  LB7A8  LB7B9  LB7E8  LB7E0  LB7D0  LBACE  LB7F4  LB802
LB882  L6001  LB870  L6004  LB82B  LB822  LBC0C  LB88E  LBCEC  LBB86
LB8BE  LB8A6  LB8D5  LB8D9  LB8E8  LB8E9  LB8F8  LB8FC  LB8FD  LB905
LB917  LB927  LB929  L013E  LB942  L00E2  L00E9  L5FFF  LB94B  L5FFE
LB988  LB954  LB976  L013F  LB987  L0141  LB9AB  L5FE1  LB9A0  LBA76
L5FE3  LB9F7  L5FE2  LB9BA  LB9BF  LB9CA  LB9C3  LB99D  LBA21  LBA7C
LB9E5  L5FDD  L013D  L5FE4  L5FDF  L5FE5  L013C  LBA03  LBA05  LBA18
L0142  LBA2E  LBA3D  LBA5D  LBA4D  L5FDE  LBA54  L5FE0  L0140  LBA63
LBA71  LBA91  LBA8E  L601E  LBA9F  LBAA5  LBAAC  LBABF  LE424  L0112
LBAC8  L00DC  LBADB  LBAE7  LBB44  LBAFD  LBB02  LBB41  LBB1D  LBB23
LBB36  LBB33  L0144  LBBCA  LBBFD  LBBD6  LBBA1  L5FC9  L018C  LBBB8
L5FCA  L018D  L5FCB  L5FCC  LBBD0  LBBC5  L5FCD  L5FCE  L5FCF  L5FD0
L018E  L018F  LBBE9  LBBF2  LBBFB  LBC0B  LBC14  LBC1C  LBCE8  LBC5A
LBC33  L0139  L0138  LBC5C  LBCCB  LBC84  LBC6E  LBC7E  LBC8E  LBC89
LBCAE  LBC9A  L013A  LBCC9  LBCD8  LBCF5  LBD44  LBCFE  LBD00  L0143
LBD11  LBD29  LBD1B  LBD21  LBD2F  LBD3C  LBD3E  L0184  LBD59  L00AD
LBD68  LBD71  LBD77  L0183  LBD7A  LBD86  LBD8D  LBD99  LBDA1  LBDB7
```

```
LBDB4  LBDF2  L0087  LBDCC  L0119  L674A  LBDED  L011F  L674E  LBDEA
LBDFC  L011B  LBE03  L674C  LBE1D  L6750  LBE1A  L6748  LBE2C  LBE46
L0121  L6752  LBE43  LBE49  L0129  LBE52  L012A  LBE5B  LBE6C  L0123
L0182  LBE74  LBE7A  LBE7D  L0122  LBE85  LBE8B  LBE8E  L0086  LBE9E
LBEA4  LBEA1  LC065  LC0B7  LBF79  LBEC9  L011D  LBEC6  LBEC0  LC06A
LBEE5  LBEFF  L691F  LBEF6  LBF0B  LBF1D  L6920  LBF14  L691B  L0124
L6923  LBF76  LD90A  LBF35  LC0B8  L0110  LBF53  LBF4E  L0185  L0125
LBF62  LBF6A  L00EB  LBF71  L00ED  LC046  LC054  LBF84  LBF95  LBFAE
L6924  LBFE6  LBFD2  L6921  LBFC6  LBFDA  L6922  LBFE0  LBFF3  LC01C
LC00A  L6747  LC025  LC017  LC02B  LC034  LC03F  LC05C  L0127  LC05F
LC083  L012B  LC08E  LC0A8  L691D  LC0A3  L691E  LC0AB  L010A  LC0B4
LC10A  L0188  LC0CC  L0186  LC0F8  LC107  LC118  LB185  LB1B5  LB199
L015B  LB1A6  LB1B7  L015C  LB1B2  L019B  LB219  LB223  LB213  LB1CD
L00F1  LB1D8  L0094  LB201  L5B2A  LB205  L5B29  L5B24  LB1FB  L0199
L019A  L0198  L5B22  LB221  L00AA  L5B28  L0101  LB230  L5D1C  LB23E
L5D1B  LB24C  LE6B4  LE74D  LEC7F  LED0A  LED63  LE9DC  LEAAE  LEDD5
LEDD6  LEE44  LC348  LC35C  LC361  L001C  LC364  LC37B  LC383  LC3A5
LC39F  L306A  LC3B9  L012E  L012F  LEF12  LEEFC  LC3F1  L0316  LC454
LC44A  LC464  LC40D  L4153  LC45F  L4154  L4158  L4155  LC42E  L4156
LC440  LC451  L4157  LC46B  LC46E  LC47A  LC4C4  L001B  LC4D1  LC4D5
LC503  LC4DF  LC4EC  LC4F0  L5D05  L5D06  L00A6  L0088  LC51D  LC530
LC529  LC543  LC53C  L00A3  LC54A  LC55D  LC55A  LC569  LC575  L0089
LC595  LC58E  LC5A3  LC5B1  LC5E2  LE434  LC5D5  LC5C4  LC5D2  LC5EC
LC622  LC606  LC603  LC6CA  LC614  LC611  LC6C2  LC6D0  LC62E  LC6BD
LC635  LC64A  LC65E  LC66D  L6753  LC668  L6754  LC677  LC68F  L0126
LC6A1  LC6AB  LD934  LC6B7  LEC80  LECCD  LEAEE  LEB2F  LEB70  LE7EF
LE82B  LEA28  LE8C6  LE8C7  LE8C8  LE8FC  L009F  LC706  L004E  LC717
LC71A  L00AE  LC72A  LC732  LC734  L672E  LC74B  L672D  LC754  L5D28
LC75F  L5D29  LC767  L5D2A  LC770  LC787  L5D2B  L00A5  L082D  LC79B
L00A2  LC7AE  LC7B1  LC7C6  L5B1A  L5B18  LC7C2  L5B1B  LC7CA  LC7D6
LC7DC  LC7ED  LC7F5  LC7F7  LC829  LC83E  L48D9  L48DE  L01DE  LC862
LC868  LC922  LC8BF  LC88C  LC893  LC8A4  LC8AD  L4140  LC8BB  LC8E0
LC90B  LC91F  LC916  L082F  LC93B  LC93E  LC949  LCA42  L4022  LC955
LC963  L4012  L0830  L4013  L0831  LC994  LC96D  L4011  LC977  L4010
LC988  L0834  LC9C8  LC9C3  L4020  LC982  LC9C0  L4019  LC9AD  L0833
L4016  LC9BA  L401B  LC9D2  L401A  LC95B  L401C  LC9E0  L401D  LCA0A
L401E  LC9F9  L401F  L4014  L0836  L4017  LCA14  L4018  LCA1D  L0832
LCA2B  L4015  LCA2F  LCA32  LCA3F  LCA56  L48B4  L01F3  LCA59  L03B2
LCA61  LCA7C  LCA80  L3061  L3063  L3065  L4E7F  LCAE8  LCB3A  LCAFE
LCB2F  LCB34  L4E82  LCB18  L029C  L4E83  L4E80  L4E81  LCB37  LCB3C
LCB4B  LCB66  LCB4E  LCB75  L01F9  LCB7F  L3FDA  LCB92  LCB95  L3FD6
LCBB2  LCBD9  LCBC7  L50D2  LCBD6  L3FD4  LCBF0  LCBED  LCC04  LCC01
LCC18  LCC15  L01D2  L3FD8  LCC89  L022C  LCC72  L45AC  LCC86  L45AD
LCC44  L45AE  L45B0  LCC66  L45AF  LCC5D  LCC78  L45B1  L45B2  L45B3
L3FC8  L0226  LCC9C  LCCC8  LCCBD  LCCC5  LCCE9  LCCE0  LCCE6  L4E6B
LCD0B  L4E6C  LCD1D  L022A  L4E6A  LCD18  L0130  LCD2C  L0118  LCD41
LCD78  LCD4A  LCD58  LCD60  L5E8E  LCD7B  L5E8B  LCD75  L5E8A  LCE3C
LCD81  LCD98  LCD8C  LCD95  LCDA5  L5E8C  LCE29  LCDB2  L5E8D  LCDBC
LCDC5  LCDD1  LCDD6  LCE07  LCDFE  LCE04  L5E8F  LCE20  LCE26  LCE39
LCE55  L0253  LCE52  LCEB0  LCE64  LCE6A  LCE70  LCE78  LCE82  LCEA0
LCE93  LCE9D  LCEAA  LCEB4  LCF37  LCEC3  L46F4  LCECB  LCED5  LCEF5
LCF0B  LCF15  LCF22  L46FF  LCF3E  LCF48  LCF49  L01D1  LCF53  L1815
     *** Warning:  Branch Ref: 0x1815 is outside of Loaded Source File.
LCF64  LCF70  LCFAF  LCF80  LCF86  LCF83  LCFAB  LCF94  LCF9F  LCFA7
LD048  LD02A  LCFF3  LCFDE  LCFCD  L01DC  L4850  L4851  LCFE6  L4853
LCFF6  LD00E  L01D3  LD020  LD02B  L01D4  L47DA  LD038  LD047  LD059
LD063  LD083  LD11C  LD0AA  LD09D  L01D7  L01D8  LD0CA  L489D  L489C
LD0C7  LD0D5  L489A  LD0E2  L01DA  LD100  L489B  LD119  LD15A  LD12E
LD1D5  LD139  LD141  L006D  LD150  L01DD  LD160  LD1CD  LD1CE  L01DF
LD172  L084F  LD193  LD191  LD19A  LD1B0  L01E1  LD1C4  LD1C2  LD1E0
LD2D6  LD21A  L402C  LD1FE  L402D  LD2CB  L402B  LD213  LD22E  L01FB
L402F  LD22B  L01FA  LD28D  L4024  LD288  L4025  LD266  LD2A8  L4026
LD27E  L4027  L402A  LD285  L4029  L4028  LD2A3  LD2E1  LD398  LD2EB
LD2EE  L4074  LD2FB  L007E  LD313  L4075  LD309  L4076  L007C  L007B
LD333  LD338  L007D  LD34D  LD352  L007A  L0078  LD380  L406D  LD37B
L084C  LD371  L4073  LD395  LD386  L406E  LD390  L406F  L4070  LD3A2
LD3C6  LD3B0  LD3C9  LD3DD  L0811  LD3D8  LD410  LD405  L080F  L4132
LD402  LE444  LE46C  LE45C  L0080  LD491  LD8FE  LD444  LD478  LD8DC
LD451  LD46C  LD854  LD8B7  LD47B  LD7FB  LD48C  LD4DA  LD4D9  LD4A4
LD4CF  LD4B1  LD4C3  LD4E3  LD4EB  LD4F1  LD4F7  LD50F  LD505  LD50D
LD517  LD519  LD549  LD527  LD547  L5D2D  LD556  L5B2C  LD567  LD564
LD57D  LD589  L0131  LD5C2  LD59A  LD5B8  L5D26  L5D27  L5D25  LD5C0
L5D24  LD5D6  LE322  L0103  LD5F4  LD5EC  LD602  LD604  LD5FF  LD81B
LD82A  LD852  LD984  LD869  LD879  LD886  LD895  LD8B5  LD8C3  LD8CE
LD8FA  LD8FC  LD906  LD908  LD91B  LD92C  LD927  LD932  LD94C  LD94F
LD94A  LD959  LD967  LD97B  LD982  LD98D  LD99F  LD9AF  LD9B8  L4E3B
LDA32  LD9FE  L4E37  L0240  L4E38  LDA03  L4E3D  LDA06  LDA2A  L4E3F
L0241  L4E40  LDA2F  L0855  LDA45  L4E3E  LDA53  LDA60  LDAB5  L024E
LDA70  LDAC6  LDA82  LDAAA  LDA9E  LDA93  LDA74  LDAA2  LDAA7  LDA8D
L003C  L0239  LDAD4  LDAC2  LDFA8  L024B  L023C  L023D  LDEDA  LDB00
L00F9  L0014  L0015  L5B12  LDAF2  LEFC3  LDB72  LDB18  L4E28  L0158
LDB29  L4E29  LDB2D  L4E2A  L4E2C  L4E2B  L023F  L4E2E  LDB77  LDB7A
L4E2F  L4E2D  LDB6D  LDBAE  L4E30  L0817  L4E35  LDBC4  L4E33  L4E34
L4E31  L4E32  LDBB3  LDBC7  L4E36  LDBFA  L5B01  LDBE2  L081C  L4E39
LDBEF  L4E3A  LDC16  LDC10  L4E41  LDC24  LDC27  LDD14  LDC35  LDD0D
LDC3C  LDCFD  LDC39  L0242  LDC52  LDC4B  LDC4F  LDD03  L4E47  LDC8E
L4E43  L4E44  L4E45  L4E46  LDC7A  L4E4A  L4E4D  LDC90  LDCBB  LDC9C
L48ED  L0243  L4E48  LDCAC  LDCB6  L0245  LDCC5  L4E4C  LDCE5  L0244
L4E4B  LDCD3  L4E49  LDCE0  LDCEB  L4E42  LDD8A  LDD7E  L4E4F  LDD41
L024A  L4E55  LDD3B  LDD7B  LDD44  LDD59  L0248  L4E53  LDD56  L4E52
LDD86  L4E50  L0247  L4E51  LDD74  LDD87  LDD83  LDDC1  L4E57  LDDA0
L4E58  LDDC6  L4E56  L4E59  LDDCB  LDE01  L4E66  LDDFB  L4E69  LDDF8
L4E6E  LDE39  LDE1E  L4E70  LDE41  LDE44  LDE30  L4E6F  LDE28  LDE3C
LDE4E  L4E73  LDE96  LDE6F  L4E72  LDE6C  L4E74  LDE9E  LDE74  L4E76
L4E78  LDE89  L4E79  LDEA1  LDEB6  LDEB3  LEF2B  L0856  L4E7A  LDED7
```

```
LDEE1  LDFAB  LDEEB  LDFAC  LDF9F  LDEF2  LDEFA  LDF20  LDF05  LDF1B
LDF87  LDF24  LDF40  LDF44  L4E26  LDF5B  LDF84  LDF76  LDF98  LDF9B
LE018  L023B  LDFCF  LE033  LDFC3  LDFD1  LE039  LDFDC  LDFE8  LE01B
LDFF7  LE036  LE010  L023A  L0237  LE042  LE045  LE088  LE085  L4E7E
LE06C  L4E7D  L024D  L4E7B  LE082  L4E7C  LE0AD  LE0A2  LE0AB  LE0CB
LE0EE  LE14F  LE136  L02EF  L491F  L02ED  L0222  LE237  L0066  LE15D
L494A  LE164  LE172  LE178  LE1F4  LE188  LE1FC  LE1A6  L494F  LE19D
L494D  L0293  LE1F9  LE1C2  LE1CF  L491E  LE1EF  L494E  LE20C  L0223
LE21B  LE224  LE250  LE241  LE24D  LE253  LE2D6  LE264  LE2BC  L48C8
LE272  LE275  LE2AA  L48C2  LE293  L48C5  LE2CB  LE2A4  L48C1  LE29F
L48C4  L48C3  L48C6  LE2D9  L0801  LE2DE  LE321  L48DF  LE31E  L48CC
L48C9  LE308  L48CB  L48CA  L48CD  LE32A  LE33B  LE364  LE35F  LE38E
LE37B  L5B34  L5B2B  LE3A9  L5B35  LE3AD  L5B2F  LE3C3  LE3C7  L0100
LE3E5  L5B3A  L5B31  LE3E9  LE410  LE3F3  LE47C  LE420  LE430  LE440
LE458  LE454  LE468  LE478  LE49F  LE486  LE4B5  LE4CB  LE541  LE53D
L0159  LE531  LE53E  LE575  LE571  L015A  LE565  LE572  LE5B6  LE5B2
LE5A5  L008E  L001F  L5B1E  L5B1D  LE5B3  LE5F5  LE5F1  LE5E5  L0020
LE5F2  LE676  LE672  LE61E  L0021  L5B45  L5B46  L5B47  L5B48  LE66A
LE673  LE6B3  LE6AF  L015E  LE6A3  L0022  LE6B0  LE6E5  LE6DF  LE6E8
L015F  LE6E2  L0023  LE748  LE749  LE74C  LE6FE  L017B  LE71A  LE72E
LE715  L017C  LE727  LE733  LE781  LE77B  LE784  L0160  LE77E  L0024
LE7EA  LE7EB  LE7EE  LE7AC  LE796  L017E  LE7A7  L017D  LE7C3  LE7D1
LE7D9  LE82A  LE822  L0162  LE814  L0091  LE826  LE85F  LE85B  L0164
LE84F  L008F  LE85C  LE892  LE88E  L0165  LE882  LE88F  LE8C5  LE8C1
L0166  LE8B5  LE8C2  LE8FB  LE8F7  L0167  LE8EB  LE8F8  LE92F  LE92B
L0168  LE91F  LE92C  LE96E  LE96A  L016D  LE954  LE96B  LE9DB  LE9D7
LE984  LE98E  L5B87  LE992  L5B85  LE9AF  L0029  L5B8B  L5B89  L5B8C
LE9D8  LEA27  LEA23  L016F  LEA12  LE9F4  L002A  L0171  LEA24  LEA7A
LEA76  L5B95  LEA4B  LEA54  LEA6A  LEA77  LEAAD  LEAA9  L0172  LEA9D
LEAAA  LEAED  LEAE5  L0173  LEAD3  LEAE9  LEB2E  LEB2A  L0175  LEB04
LEB1E  L0090  L002B  LEB2B  LEB6F  LEB6B  L0176  LEB45  LEB5F  L002C
LEB6C  LEBB0  LEBAC  L0177  LEB86  LEBA0  L002D  LEBAD  LEC13  LEBD6
L001E  L0031  LEBE1  LEBE5  LEC15  L0132  LEC4E  LEC4F  L0032  LEC4B
LEC30  L0133  L0033  L0134  L0034  LEC7E  LECCC  LECC8  L0169  LECAF
L0026  LECC9  LED09  LED05  L016A  LECF9  L0027  LED06  LED62  LED5E
L016B  LED32  L0028  LED5F  LEDD3  LEDCF  L016C  LED81  LEDAA  LEDD0
LEDFE  LEDFC  L0179  LEE00  LEE3F  L002F  LEE43  LEE40  LEE6C  LEE6A
L017A  LEE6E  L0030  LEEA5  LEEA9  LEEA6  L0192  LEEFB  LEEF6  LEF11
LEF04  LEF22  L48F3  L48F4  LEF3A  LEF40  LEF34  LEFC9  L3066  LF09A
L3064  L3067  LF0CA  LF0E4  LF0E0  LF100  LF103  LF10D  LF121  LF125
LF13D  LF148  LF14E  LF155  LF162  LF172  LF178  LF184  LF189  LF1B4
LF1BA  LF1CA  LF1D0  LF1DF  LF1E6  LF273  L1806
     *** Warning:  Branch Ref: 0x1806 is outside of Loaded Source File.
L0F00  LF284  L180C
     *** Warning:  Branch Ref: 0x180C is outside of Loaded Source File.
L0042  LF2EA  LF2E7  L0364  L302E  L302F  L0363  L0367  LF2CF  LF2BF
L0362  L302D  L012C  L012D  LF36A  L039B  LF39F  L036C  LF35E  LF339
LF343  LF35A  LF36C  L03A1  LF376  LF399  L031B  L031D  L031F  L0321
L0323  L0325  L03A3  LF3BF  LF3AC  LF3E3  L0390  LF3E0  L0045  LF3F2
LF3FE  LF401  LEFD0  LEFD7  LF411  LF414  LF01D  LF024  LF42A  LF421
LF464  L0369  LF43D  LF47D  L036A  L51B1  LF493  L51B3  LF4A4  LF4CB
LF5B5  LF4B1  LF4CC  LF4FC  LF4F4  L0366  LF4E9  LF590  LF513  LF510
LF58A  LF57E  LF52B  LF536  LF530  LF562  LF54C  LF55E  LF559  LF586
LF573  L0368  LF59E  LF5B4  LF5E3  LF5FF  LF5DB  LF5E5  LF660  LF5EE
LF5F5  LF65A  LF60F  LF663  LF62E  LF655  L036E  LF646  LF704  LF6F7
LF6BA  LF680  LF69B  LF695  LF6CB  LF6B0  LF6AB  LF6C0  LF72C  LF72D
LF749  LF753  L0001  LF771  LF774  LF763  LF87E  LF7E3  L3FCC  LF880
L3FEA  LF7B7  LF7CE  LF7D0  LF7F4  LF842  L306E  LF6D2  LF8B1  LF8C7
LF8D6  LF8D1  L7200  LCA8A  L723A  L725B  LEF9A  L7282  L72C3  L7548
L303F  L72D2  L72EC  LEF47  L4008  L4006  L72E5  L730C  L000F  L7308
L733C  L7346  L735A  L7351  L0219  L736D  L7379  L5B21  L7385  L5B42
L7391  L5B52  L739D  L5B58  L73A9  L5B5E  L0025  L73B5  L5B78  L73C1
L5B72  L73CD  L5B74  L73D9  L5B79  L73E5  L73F1  L5B8F  L73FD  L5B9D
L7409  L5B9F  L7415  L5BA1  L002E  L7421  L5BA2  L742D  L5BA8  L7439
L5BAE  L7445  L5BB8  L7457  L7467  L7472  L747A  L7482  L7499  L4950
L74CA  L4951  L9341  LB24D  L0193  L755E  L4071  L750C  L4072  L0049
L3022  L756A  L756C  L7583  L9355  L935E  L93D4  L9361  L94A8  L937F
L9377  L937C  L9323  L93F2  L508E  L93E4  L93EF  L9410  L508F  L9402
L940D  L9423  L9494  L943D  L9442  L9447  L02C4  L9457  L945C  L945E
L946E  L9473  L9478  L02CE  L9488  L948D  L948F  L94A7  L94B2  LB26F
LEF5F  LEF56  LEF62  LEFA8  LEFC2  LEFB2  L3024  L3025

Pass 2 - Disassembling to Output File...

     *** Warning:  Branch Ref: 0x1800 is outside of Loaded Source File.
     *** Warning:  Branch Ref: 0x1809 is outside of Loaded Source File.
     *** Warning:  Branch Ref: 0x1803 is outside of Loaded Source File.
     *** Warning:  Branch Ref: 0x1812 is outside of Loaded Source File.
     *** Warning:  Branch Ref: 0x180F is outside of Loaded Source File.
     *** Warning:  Branch Ref: 0x1815 is outside of Loaded Source File.
     *** Warning:  Branch Ref: 0x1806 is outside of Loaded Source File.
     *** Warning:  Branch Ref: 0x180C is outside of Loaded Source File.

Pass 3 - Creating Functions Output File...
     *** Warning:  Branch Ref: 0x1800 is outside of Loaded Source File.
     *** Warning:  Branch Ref: 0x1809 is outside of Loaded Source File.
     *** Warning:  Branch Ref: 0x1803 is outside of Loaded Source File.
     *** Warning:  Branch Ref: 0x1812 is outside of Loaded Source File.
     *** Warning:  Branch Ref: 0x180F is outside of Loaded Source File.
     *** Warning:  Branch Ref: 0x1815 is outside of Loaded Source File.
     *** Warning:  Branch Ref: 0x1806 is outside of Loaded Source File.
     *** Warning:  Branch Ref: 0x180C is outside of Loaded Source File.

Disassembly Complete
```

Due to copyright issues, the source binary file and resulting disassembly and function output files cannot be included in the distribution of M6811DIS.

By examining this output, we can better illustrate the order of operation within the disassembler. First, we see that after initializing itself, the disassembler reads the Control File. From the Control File, the disassembler sets the load address to 0x4000, sets the input filename to "av94bnbh.bin", sets the disassembly output filename to "av94bnbh.dis", sets the functions output filename to "av94bnbh.fnc", sets 21 user-specified entry points and 16 user-specified labels. Notice that the indirect vectors specified in the Control File isn't processed until later – this is because the source file has to be loaded before the indirects can be resolved.

More than one input file can be specified and since each may be in a different file format and have different loading addresses, each file is specified with its load address and file type with separate "load" commands in the Control File. This is the preferred method to load input files.

You can alternatively use the deprecated "input" command to specify a single input file to load and the "dfc" command to specify its file type (which defaults to binary if not specified), and the "load" command without a filename to specify its load address (which defaults to 0x0000 if not specified). This mechanism is deprecated and provided for backward compatibility with old Control Files used in previous versions of M6811DIS. It is recommended that you use the "load" command to specify and load the input file, allowing you to specify different loading addresses and different DFC libraries for each file. In all cases, overlaps in the data from the input files are not permitted.

Notice that this particular file is 0xC000 bytes or 48K. Since it started at 0x4000, the loaded code consumes the upper 48K of the HC11's 64K program-space. Typically you should **never** have a file that occupies the entire 64K address-space of the HC11. This is because part of that space is really RAM space and HC11 Register Space. RAM and Register space (and anything else that isn't part of the program and program data) should **not** be included, simply because the information in those areas is not valid.

After loading the source file(s), the disassembler can now resolve the indirects specified in the Control File. It reads the address stored at each specified indirect location and adds the found address to the internal entry table. You can optionally specify in the Control File, whether an indirect address is a vector to (or pointer to) code or data. If not specified, it will be assumed that the vectors point to code. This example Control File illustrates the best known and type of indirect code vector table – the interrupt table.

Now that everything has been loaded, the disassembler will begin pass 1 through the source that it has loaded into memory. During this pass the disassembler will tag all loaded memory locations as being either data or code. And, each time it encounters a new label, the new label is added to the label table and outputted to stdout. This is why the labels appear seemingly in random order.

Notice the warning messages like "*** Warning: Branch Ref: 0x1803 is outside of Loaded Source File.". These indicate that a branch or jump instruction specified an address that was outside of the memory area loaded from the source file. This is typical in applications that have more than one source for the program code or from some dynamically loaded source, like a bootloader. It can result from any of the following "system" reasons:

- The code is split into multiple ROM chips
- The code is simply broken into multiple parts
- A second processor with common dual-port memory space
- Dynamic code that is loaded into RAM either by this program or some other bootloader
- One of many other possible "system" reasons

However, this warning message can result from any of the following "user" reasons:

- An incorrect load address for the binary image was specified
- An incorrect entry point was specified
- The memory image is corrupt or incorrect
- One of many other possible “user” reasons

If you see warnings of this nature, check your source file. Make sure it is the correct length and that you have specified the load addresses correctly. Make sure that you have all the pieces of the memory image and that they are either concatenated or loaded correctly as individual parts.

In this example, we can ascertain that in this particular case, it is the result of a “hardware” reason. We notice that the 8 different “outside” addresses are spaced every 3 bytes apart. Peculiarly enough, the extended memory jump instructions in the HC11 happen to also be 3 bytes long. After double checking our original ROM and after further study of the device under test, we find that these “outside” addresses create a dynamic jump-table and that these addresses appear in a secondary memory device – possibly EEPROM or a dual-port memory interfacing this HC11 with a coprocessor. By reading the HC11 memory space in-circuit, we can actually see this dynamic jump table and how it is created. But, the knowledge, instruction, and methodology on how to do in-circuit testing and other more in-depth reverse engineering techniques are outside the scope of this document.

After the disassembler has completely exhausted all entry point table values, the disassembler then starts pass two. Pass two is simply an iteration through the entire loaded memory space, while outputting the disassembly to the output file. After the iteration through the source is complete, the disassembly process is finished. The resulting assembly file, when reassembled according to the section **Reassembling a Disassembly**, will result in a memory image file that is guaranteed to be byte-for-byte identical to the original binary source file. This eliminates many of the headaches and hard work required by most disassemblers that aren’t targeted for a specific assembler – on those disassemblers, it is often necessary to completely rework the output file before it will even assemble, much less assemble back into the original image. That is what makes this type of disassembler so appealing to the reverse-engineer, who often needs to disassemble a program, add or change some functionality, and then reassemble and be able to do so without reworking the entire source by hand.

Control Files

Control files are the means for telling the disassembler how to load, interpret, and disassemble the desired memory image source file(s). The reason there is a Control File is that the disassembler needs to know more than just the name of the original file – so much more that it would be cumbersome to have to repeatedly specify these options on the command line when running the disassembler, not to mention you'd run out of command line space!

A Control File is nothing more than a simple text file that you create using your favorite text editor. This text file is a listing of commands with associated arguments for the disassembler. Each command must be listed on a separate line. By default, all numbers, such as addresses, specified in the Control File are treated as a "base" of hexadecimal. However, the "base" command can be used to change the numeric base at any point in the Control File from that point forward. When a base has been specified (or the default hexadecimal mode is in use), numbers should be entered as-is in the base specified with no prefix or suffix – that is, you do **not** put a '0x' or '$' or any other symbol before or after to indicate a hexadecimal value. You can, however, select either a different base or turn off the automatic-base mode entirely – where a base-denoting prefix, such as '0x', **can** be used to specify the base. Refer to the "base" command in this document.

Comments can be placed anywhere in the command file by using a semicolon (';'). Any text on a line following the ';' will be ignored by the disassembler. Blank lines are also ignored.

Note that when using the disassembler to read more than one Control File, **all command default values, such as "dfc" and "base", are reset to their initial internal default between files**. That is, parser settings don't carry over from one file to the next. This keeps each control file independent and self-contained, allowing you to specify the control files on the command line without regard to their order.

The commands are **not** case sensitive, as everything is internally converted to uppercase.

A bare minimum Control File can consist of only 2 lines – an "input" or "load" statement and an "output" statement, as you'll see from analyzing the example and command descriptions below. The default load address for the source memory image file is 0x0000 if it isn't explicitly specified. And, if no "entry" or "indirect" statements are specified to give entry points, then the file load address will be used as an entry point. **However, you should note that the load address is NOT assumed to be an entry point if any other entry point (either direct or indirect) is used. If the load address is a correct entry point and you have other entry points and/or indirects specified, you must include an additional "entry" command to add the load address to the list of entry points.**

All of the Control File commands fall into one of four categories: 1) Switch Commands, 2) Value Commands, 3) List Entry Commands, and 4) Parser Setting Commands. Each Switch Command switches a yes/no option in the disassembler. The following Switch Commands are supported: ADDRESSES, ASCII, ASCIIBYTES, DATAOPBYTES, OPCODES/OPBYTES, SPIT, and TABS. The Value Commands allow you to specify a value for a disassembler option. The following Value Commands are supported: INPUT, LOAD, MAXNONPRINT, MAXPRINT, MEMMAP, OUTPUT, and TABWIDTH. The List Entry Commands allow you to add an entry to one of the disassembler's internal lists, such as the entry point table, the label table, or the indirects table. The following List Entry Commands are supported: ENTRY, EXITFUNCTION, INDIRECT, and LABEL. The Parser Setting Commands allow you to specify default value settings for how the parser will interpret the rest of the Command File. The following Parser Setting Commands are supported: BASE and DFC. Each of the supported commands is described in detail in the following sections of this document.

Example Control File

Below is the Control File that was used in an actual disassembly/reverse-engineering effort. This Control File is the one that was used to create the previously discussed screen output in the *Step-by-Step Walk-Through* section:

```
;
; M6811DIS Control File for:
;
;  '94 Astro Van computer code: BNBH
;

load 4000 av94bnbh.bin binary
output disassembly av94bnbh.dis
output functions av94bnbh.fnc

MemMap ROM 4000 C000

tabs off
;addresses
;ascii

label ffd6 scivect
label ffd8 spivect
label ffda paievect
label ffdc paovect
label ffde tovfvect
label ffe0 ti4o5vect
label ffe2 to4vect
label ffe4 to3vect
label ffe6 to2vect
label ffe8 to1vect
label ffea ti3vect
label ffec ti2vect
label ffee ti1vect
label fff0 rtivect
label fff2 irqvect
label fff4 xirqvect
label fff6 swivect
label fff8 ilopvect
label fffa copvect
label fffc cmonvect
label fffe rstvect

indirect ffd6 scirtn
indirect ffd8 spirtn
indirect ffda paiertn
indirect ffdc paortn
indirect ffde tovfrtn
indirect ffe0 ti4o5rtn
indirect ffe2 to4rtn
indirect ffe4 to3rtn
indirect ffe6 to2rtn
indirect ffe8 to1rtn
indirect ffea ti3rtn
indirect ffec ti2rtn
indirect ffee ti1rtn
indirect fff0 rtirtn
indirect fff2 irqrtn
indirect fff4 xirqrtn
indirect fff6 swirtn
indirect fff8 iloprtn
indirect fffa coprtn
indirect fffc cmonrtn
indirect fffe reset

entry 7C0B
entry 7C12
entry 7C1C
entry 7C22
entry 7C35
entry 7C6B
entry 7C7C
entry 7C83
entry 7C9C
entry 7CA0
entry 7CAA
entry 7CAE
entry 7CBE
entry 7CC2
entry 7CCC
entry 7CDD
```

This is the same example Control File that is provided in soft form with the distribution of this disassembler and was used to disassemble the code from a 1994 4.3L CPI Vortec Astro Van vehicle computer, which happens to use a Motorola 68HC11 variant. Let's analyze this Control File piece by piece.

First off, we define the input and output files we will be using. I typically use the convention of .bin for all source binary format files and .dis for all disassembler output files. After I have a chance to go through the disassembled output, clean it up, and comment it, I'll rename the "clean" version with an .asm extension. You may, however, adopt a different extension standard, as the disassembler doesn't care what you name the input and output files as long as they are properly specified in the Control File. It will complain if you don't specify these and it makes no assumptions as to what the extensions are.

We then tell the disassembler that the load address for this file is 0x4000. Even though the ROM is a 27512 or 64K byte ROM, only the upper 48K is used – with the lower 16K being all 0xFF bytes. This makes sense considering the RAM and CPU Registers exists in the lower memory. So after reading the entire 64K ROM, the binary image was trimmed down to 48K by using a hex editor (a decent hex editor for Windows is written by BreakPoint Software and can be found at http://www.bpsoft.com/. For Linux, most distros have Okteta http://utils.kde.org/projects/okteta/). As mentioned earlier, it is **not** good to include parts in the binary source image that aren't actually part of the code or data being disassembled – in this case the 0xFF padding. This Control File shows the preferred method of specifying specifying the input file, which is "load 4000 av94bnbh.bin binary". The "binary" DFC name could have been omitted, as it is the default. Alternately, the input file could have been specified with the now deprecated "input" and short-form "load" commands by using: "input av94bnbh.bin" and "load 4000".

In our output, if you'd like for the disassembler to include the actual memory address for each instruction in the disassembly output, the "addresses" switch can uncommented to enable that. Similarly, if you'd also like for it to attempt to decode any areas that doesn't appear to be code as possibly being ASCII data and to output them as strings if they exist, you can uncomment the "ascii" switch to enable that.

Since we know that this is an HC11 processor and that the ROM exists in the upper memory, it only makes sense that the image includes the interrupt vector table. So we've included a list of the basic HC11 interrupt vectors. (Because of their complexity and variance from one HC11 family to another, the details and specifics of these interrupts are not included in this document, but are available in the aforementioned HC11 reference manuals.) We first define a name for each of the vector locations themselves. This is optional, but allows us, when looking at a disassembly output, to easily tell which vector is which. We then list each vector as being an indirect – which will cause the disassembler to add the address located at the vector address to the list of entry points. It also allows us to specify a label name for the routine that is being indirectly referenced. So, we'll give them names like "reset" and "swirtn", things that will be meaningful when we later examine the disassembly.

If this were the first time we've encountered this memory image, that is about all that we can enter into the Control File, as we won't yet know of any additional entry points. So we run the disassembler with a Control File that doesn't have the "entry" commands listed. In a quick look over the resulting output, we'll find several jump tables and code that gets called indirectly during execution. Typically, these are done by loading one of the index registers with a lookup table address and doing a "jsr" relative to the index register value. So, look for "Undetermined Branch Address" comments in the disassembler output. Anytime the disassembler encounters a jump it cannot trace, it will comment the instruction as such. This will typically be a clue to you that you need to look for a branch table or similar, add "entry" values or additional "indirect" values to the Control File, and run the disassembler again. That is what was done here and is where the addresses came from that you see in the "entry" commands. Most of these were actually indirects, instead of plain entries, but I chose to use the "entry" command so you can see additional Control File commands in-use.

Control File Commands

Switch Commands

ADDRESSES

Format: addresses [OFF | ON | TRUE | FALSE | YES | NO]

The “addresses” switch instructs the disassembler to output the address of the start of each instruction in front of the actual disassembled instruction in the output line. Here is an output example with “addresses” turned on:

```
EBAC      LEBAC:    clra
EBAD      LEBAD:    staa      L0177
EBB0      LEBB0:    rts
EBB1      LEBB1:    brset     *L003B,#0x04,LEC13
EBB5                ldx       #0x5B00
EBB8                brclr     0x08,x,#0x04,LEC13
EBBC                brset     *L0090,#0x40,LEBD6
EBC0                bset      *L0090,#0x40
EBC3                brclr     *L001E,#0x04,LEC13
EBC7                ldab      *L0031
EBC9                cmpb      0xB8,x
EBCB                bcc       LEC13
```

Having a copy of the output with the addresses on each line is very useful when hunting down references, finding data labels, etc. However, having the addresses present prevents the code from directly assembling. Since not all editors allow you to do block or column deletes and easily delete the addresses, this switch is provided so that you can enable/disable address generation. This way, you can turn them off and create a file that is compatible with direct re-assembly, or turn them on and create a file that is easier to sort through when deciphering and commenting the resulting disassembled code.

The default mode if “addresses” is not specified is ADDRESSES OFF.

If “addresses” was not specified in the control file, the above code example would have appeared as follows:

```
LEBAC:    clra
LEBAD:    staa      L0177
LEBB0:    rts
LEBB1:    brset     *L003B,#0x04,LEC13
          ldx       #0x5B00
          brclr     0x08,x,#0x04,LEC13
          brset     *L0090,#0x40,LEBD6
          bset      *L0090,#0x40
          brclr     *L001E,#0x04,LEC13
          ldab      *L0031
          cmpb      0xB8,x
          bcc       LEC13
```

ASCII

Format: ascii [OFF | ON | TRUE | FALSE | YES | NO]

ASCIIBYTES

Format: asciibytes [OFF | ON | TRUE | FALSE | YES | NO]

The “ascii” switch causes the disassembler to look at the data areas when creating the output file and to try and group adjacent bytes if they are ASCII printable characters. The “asciibytes” switch causes the disassembler to output, in addition to the ASCII characters, the actual byte values themselves. Here is an output example with both “ascii” turned on and “asciibytes” turned on:

```
; 432A: 59,55,52,4E,4E,4D,4C,4B
; 4332: 47,44,40,3B,52,63,66,61
; 433A: 5F,5B,56,52,4F,4E,4E,4D
; 4342: 4B,47,44,42,3E,52,63,66
; 434A: 5F,55,52,4F,4E,4D,4C,4B
        .ascii    'YURNNMLKGD@;Rcfa_[VRONNMKGDB>Rcf_URONMLK'
```

Here is an output example with “ascii” turned on and “asciibytes” turned off:

```
        .ascii    'YURNNMLKGD@;Rcfa_[VRONNMKGDB>Rcf_URONMLK'
```

Outputting the “asciibytes” in addition to the “ascii” is useful in case the data really isn’t text, as in the case above. And, sometimes there will be real text preceded or followed by data that just happens to be in the printable ASCII range. Depending on whether your file has more printable text or not will determine if you will want to run the disassembler with “ascii” and/or “asciibytes” on or off. If it has a lot of printable text, running with “ascii” on will save a lot of typing in your “cleaned-up” version of the disassembly. But if there isn’t very much printable text, running with it off will keep you from having to convert those misinterpreted areas back to bytes. I suggest first running with “ascii” on and see what ASCII strings it produces and then decide from there.

The default mode if “ascii” is not specified is ASCII OFF. The default mode if “asciibytes” is not specified is ASCIIBYTES ON.

If “ascii” had not been specified on the above, the output would have appeared as follows, regardless of whether or not “asciibytes” was on or off:

```
.byte     0x59,0x55,0x52,0x4E,0x4E,0x4D,0x4C,0x4B
.byte     0x47,0x44,0x40,0x3B,0x52,0x63,0x66,0x61
.byte     0x5F,0x5B,0x56,0x52,0x4F,0x4E,0x4E,0x4D
.byte     0x4B,0x47,0x44,0x42,0x3E,0x52,0x63,0x66
.byte     0x5F,0x55,0x52,0x4F,0x4E,0x4D,0x4C,0x4B
```

Note that regardless of whether “ascii” and/or “asciibytes” is on or off, the output file will still reassemble back into the original binary. This is because the “.ascii” assembler directive and the “.byte” assembler directive will produce the same value bytes in the assembly process, and the extra “real byte values” are outputted as comments for the assembler (as can be seen above).

OPBYTES/OPCODES

Format: opbytes [OFF | ON | TRUE | FALSE | YES | NO]
Format: opcodes [OFF | ON | TRUE | FALSE | YES | NO]

DATAOPBYTES

Format: dataopbytes [OFF | ON | TRUE | FALSE | YES | NO]

The “opbytes”, which is also synonymous with the “opcodes” switch, causes the disassembler to create an extra field, which precedes the label field, in the output. This extra field will contain the actual byte values for the opcode disassembled on that particular line. However, this field is left blank in data areas unless the “dataopbytes” switch is also turned on.

Here is an example of code that has been disassembled with “opbytes” turned on, but with “dataopbytes” turned off. Note that this is the same code as was used for the “addresses” example above and the same ASCII data as used for the “ascii” example above, which illustrates this code with these switches both turned off:

```
                    .ascii    'YURNNMLKGD@;Rcfa_[VRONNMKGDB>Rcf_URONMLK'
4F          LEBAC:  clra
B7 01 77    LEBAD:  staa      L0177
39          LEBB0:  rts
12 3B 04 5E LEBB1:  brset     *L003B,#0x04,LEC13
CE 5B 00            ldx       #0x5B00
1F 08 04 57         brclr     0x08,x,#0x04,LEC13
12 90 40 16         brset     *L0090,#0x40,LEBD6
14 90 40            bset      *L0090,#0x40
13 1E 04 4C         brclr     *L001E,#0x04,LEC13
D6 31               ldab      *L0031
E1 B8               cmpb      0xB8,x
24 46               bcc       LEC13
```

If you were to also turn on “dataopbytes”, it would look like this:

```
59 55 52 4E         .ascii    'YURNNMLKGD@;Rcfa_[VRONNMKGDB>Rcf_URONMLK'
4E 4D 4C 4B
47 44 40 3B
52 63 66 61
5F 5B 56 52
4F 4E 4E 4D
4B 47 44 42
3E 52 63 66
5F 55 52 4F
4E 4D 4C 4B
4F          LEBAC:  clra
B7 01 77    LEBAD:  staa      L0177
39          LEBB0:  rts
12 3B 04 5E LEBB1:  brset     *L003B,#0x04,LEC13
CE 5B 00            ldx       #0x5B00
1F 08 04 57         brclr     0x08,x,#0x04,LEC13
12 90 40 16         brset     *L0090,#0x40,LEBD6
14 90 40            bset      *L0090,#0x40
13 1E 04 4C         brclr     *L001E,#0x04,LEC13
D6 31               ldab      *L0031
E1 B8               cmpb      0xB8,x
24 46               bcc       LEC13
```

Turning “dataopbytes” on without also turning “opbytes” on has no effect.

These switches are useful if you just want to see what the bytes are to help with your understanding of the actual HC11 machine code, or if you are tracking a section that you think might be misinterpreted as code that really should be data. This way you can see it in both forms.

The opbytes field resembles the list file output of many assemblers. However, when the “opbytes” switch is set, the output most likely cannot be directly assembled unless you first delete the opbytes field, since the assembler won't know how to interpret them.

The default mode if “opbytes” is not specified is OPBYTES OFF. The default mode if “dataopbytes” is not specified is DATAOPBYTES OFF.

In most cases, since the disassembler does a good job with separating code and data, you will probably want to leave this option turned off (which is why I didn't include it in the sample Control File we previously examined). It will only make the output file bigger and prevent it from being directly reassembled. This option was originally implemented for debugging purposes when the disassembler was written, but was left as an option to be used as a learning tool for newcomers to the HC11 processor and to be used in the rare case of data being misinterpreted as code.

To see what the output would look like with the "opbytes" and "dataopbytes" switches left off, please refer to the "addresses" and "ascii" commands.

SPIT

Format: spit [OFF | ON | TRUE | FALSE | YES | NO]

The “spit” switch was added because of popular demand. This switch turns off the code-seeking logic and causes this disassembler to perform like the traditional unintelligent “disassemble everything” disassemblers. In this mode, the disassembler will start at address 0x0000 and disassemble through the entire size of the loaded memory image attempting to interpret everything as code, which can produce an enormous output of data incorrectly interpreted as code.

Why is this mode useful if it can produce erroneous output? It is useful if you happen not to know any entry points into the code (which is unlikely) or if there is an extremely large number of jump tables and you’d rather just dump the output and cut-and-paste the results rather than finding and entering all of the jump table entries. Many hackers prefer this mode, as it often gets them to the code quicker without having to locate and understand the jump tables.

The default mode if “spit” is not specified is SPIT OFF.

TABS

Format: tabs [OFF | ON | TRUE | FALSE | YES | NO]

This option allows you to control whether or not the fields in the output file will be aligned with tabs or spaces. In previous versions, tabs were used exclusively, but because of the non-standard interpretation of tabs and variances between various editing and viewing software as to just how many spaces is represented by a tab, this switch was added so that the fields can be properly aligned. Turning this switch on causes the fields to be aligned using tab characters. If turned off, fields will be padded with spaces. In either case, the field width is set by the “tabwidth” value command – see “tabwidth” for additional information.

The default mode if “tabs” is not specified is TABS ON.

Value Commands

INPUT

Format: input <filename>

The “input” command is deprecated. For new projects, you should consider using the “load” command instead. It is maintained for backward compatibility with old projects.

The “input” command allows you to specify the name of an input memory image file for the disassembler to read. If a path is not specified, the file must reside in the current directory. An extension need not be given, and if none is given, none will be appended. The file must be in the format specified by the last “dfc” command or binary (the DFC default) if no “dfc” command is specified.

The file will be loaded at the offset address optionally specified by the last “load <address>” command in the file. If no “load” command is specified, then the address 0x0000 will be used.

The source file(s) must fit within the memory bounds of the HC11 processor. That is, it cannot be bigger than 64K if loaded at 0x0000. If loaded higher than 0x0000, the size must also reflect this. For example, if the load address is specified as 0x4000, then the file can be no bigger than 48K. Note that the file(s) need not fill the entire memory. If you are disassembling a 1K chunk of code that is originated at 0x0800 (for example), then the image needs to only contain the 1K chunk and a load address of 0x0800 should be specified.

In reality, you should not include bytes that aren’t either code or data for the source you are disassembling. For example, RAM areas in the processor address space should not be included. Typically, if these locations have corresponding ROM addresses, they will be filled with 0xFF or 0x00 (depending on the source), which you should omit. You should also not include processor control registers either. You may, however, wish to define labels for the control registers or even labels for RAM variables. The disassembler will properly tag these in the disassembly and setup equates for you. But, since these are in “volatile” memory, the actual bytes should not be included directly in the memory image – unless of course you are using the disassembler to disassemble a chunk of code that is transferred to the HC11 and run from RAM.

It should also be noted that the load address specified by the “load <address>” command is a base relocator for the data file’s addressing scheme. In other words, if the data file type supports address information, then the actual address used is the data file’s specified address plus the specified load address – effectively making the Control File specified address an offset address for the addresses contained in the data file, rather than an absolute address. If the data file type doesn’t support address information, such as a binary image, then the address used is simply the one specified in the Control File, making the offset the same as the absolute address.

To illustrate, an Intel Hex file supports address information. Suppose you had a 256-byte image in an Intel Hex format file and the Intel Hex file specified a starting address of 0x0040 for the 256-byte image. If you specify a load address of 0x1000, the file will actually be loaded into 0x1040. On the other hand, if you had the same 256-byte image in a binary format file and specified a load address of 0x1000, it will be loaded at 0x1000 since the binary format contains no address information.

Only one source file can be specified with the “input” command in a single Control File. Any additional “input” commands override previous “input” commands and only the last specified file will be loaded. To load more than one file, you must use the “load” command. You should also consider changing your projects to just use the “load” command exclusively, since the “input” command is deprecated.

Between the “input” and “load” commands, failure to specify at least one source input file will cause the disassembler to halt with an error. And in all cases, bytes in the various files cannot overlap.

LOAD

Format1: load <address> **Format2:** load <address> <filename> [<library>]

The “load” command has two forms. The “load <address>” form, which is deprecated and provided to maintain backward compatibility with existing projects, allows you to specify the relative load address for the data file specified by the “input” command, which is also deprecated.

The second form, “load <address> <filename> [<library>]” lets you actually load a specific file at a specific address using a specific DFC Library. This enables you to load more than one file simultaneously, with each possibly having different relative load addresses and different file formats, by using multiple “load” commands.

The source file(s), when combined, must fit within the memory bounds of the HC11 processor. That is, it cannot be bigger than 64K if loaded at 0x0000. If loaded higher than 0x0000, the size must also reflect this. For example, if the load address is specified as 0x4000, then the file can be no bigger than 48K. Note that the file(s) need not fill the entire memory. If you are disassembling a 1K chunk of code that is originated at 0x0800 (for example), then the image needs to only contain the 1K chunk and a load address of 0x0800 should be specified.

In reality, you should not include bytes that aren’t either code or data for the source you are disassembling. For example, RAM areas in the processor address space should not be included. Typically, if these locations have corresponding ROM addresses, they will be filled with 0xFF or 0x00 (depending on the source), which you should omit. You should also not include processor control registers either. You may, however, wish to define labels for the control registers or even labels for RAM variables. The disassembler will properly tag these in the disassembly and setup equates for you. But, since these are in “volatile” memory, the actual bytes should not be included directly in the memory image – unless of course you are using the disassembler to disassemble a chunk of code that is transferred to the HC11 and run from RAM.

The base of the <address> argument is dependent upon the current base setting – see the “base” command. An example, to specify a load address of 0x4000 for a file that will be loaded by the “input” command, would be:

```
load 4000
```

An example, to load an Intel Hex format file named “foo.hex” to a relative address offset of 0x0800 using the “intelhex” DFC Library, would be:

```
load 0800 foo.hex intelhex
```

If no DFC Library is specified for the optional <library> argument, the default library specified by the last “dfc” command will be used. If no “dfc” command preceded the “load” command in the Control File, the “binary” DFC Library will be used. See the “dfc” command for additional information.

It should also be noted that the load address specified by the <address> argument is a base relocator for the data file’s addressing scheme. In other words, if the data file type supports address information, then the actual address used is the data file’s specified address plus the specified load address – effectively making the Control File specified address an offset address for the addresses contained in the data file. If the data file type doesn’t support address information, such as a binary file, then the address used is simply the one specified in the Control File.

To illustrate, an Intel Hex file supports address information. Suppose you had a 256-byte image in an Intel Hex format file and the Intel Hex file specified a starting address of 0x0040 for the 256-byte image. If you specify a load address of 0x1000, the file will actually be loaded into 0x1040. On the other hand, if you had the same 256-byte image in a binary format file and specified a load address of 0x1000, it will be loaded at 0x1000 since the binary format contains no address information.

If no “load <address>” command is specified, 0x0000 is used as the load address for any file specified by the “input” command. Between the “input” and “load” commands, failure to specify at least one source input file will cause the disassembler to halt with an error. And in any case, bytes in the various files cannot overlap.

MAXNONPRINT

Format: maxnonprint <count>

The “maxnonprint” command sets the maximum number of non-printable data bytes to be outputted to a single line in a data section of the output file. The default value for “maxnonprint” is 8, if “maxnonprint” is not specified.

The base of the <count> argument is dependent upon the current base setting – see the “base” command. An example of setting the “maxnonprint” to 10 characters, with the default base of hexadecimal, would be:

```
maxnonprint A
```

This would produce an output similar to the following:

```
.byte     0x59,0x55,0x52,0x4E,0x4E,0x4D,0x4C,0x4B,0x47,0x44
.byte     0x40,0x3B,0x52,0x63,0x66,0x61,0x5F,0x5B,0x56,0x52
```

Depending on the <count> number used, keeping the default base makes this command less intuitive. For example, “maxnonprint 12” with the default hexadecimal base would really be 18 bytes instead of 12. Therefore, you may wish to use the “base” command in the Control File to set the default base to either decimal or to nothing (see the “base” command). If you set it to nothing, you can use the “0x” prefix to specify other numbers in the Control File in hexadecimal format:

```
base none
maxnonprint 12                ; This is 12 bytes, not 18
```

If more than one “maxnonprint” command is encountered, the final one encountered is the one that is actually used, and this applies to <u>all</u> Control Files when multiple Control Files are used.

MAXPRINT

Format: maxprint <count>

The “maxprint” command sets the maximum number of printable characters to be outputted to a single line in a data section of the output file. The default of 40 is used, if “maxprint” is not specified.

The base of the <count> argument is dependent upon the current base setting – see the “base” command. An example of setting the “maxprint” to 50 characters, with the default base of hexadecimal, would be:

```
maxprint 32                ; With the default base, this is 50 not 32
```

This would produce an output similar to the following:

```
.ascii    'YURNNMLKGD@;Rcfa_[VRONNMKGDB>Rcf_URONMLKYURNNMLKGD'
```

Depending on the <count> number used, keeping the default base makes this command less intuitive. Just as in the example above, with the default hexadecimal base it is really 50 bytes instead of 32. Therefore, you may wish to use the “base” command in the Control File to set the default base to either decimal or to nothing (see the “base” command). If you set it to nothing, you can use the “0x” prefix to specify other numbers in the Control File in hexadecimal format:

```
base none
maxprint 32                ; This is 32 bytes, not 50
```

If more than one “maxprint” command is encountered, the final one encountered is the one that is actually used, and this applies to <u>all</u> Control Files when multiple Control Files are used.

MEMMAP

Format: memmap <mem-type> <address> <size>
<mem-type>: [RAM | ROM | IO]

The “memmap” command lets you define the memory structure of the device or system you are working on. Currently, memory types of RAM, ROM, and IO are supported. Multiple entries for each memory type can be specified if multiple discontiguous sections of memory exist. The <address> is the starting address for the specified memory range and <size> is the size, in bytes, of the specified memory range. These values are specified in the base currently set by the “base” command (see the “base” command for more details).

The purpose of the “memmap” command is for the generation of the Functions Output File, used as input to the Fuzzy Function Analyzer, which takes the type of memory being accessed into account. For basic disassembly functionality, it doesn't serve much purpose, but may be useful for you to document your system's memory layout.

OUTPUT

Format: output [DISASSEMBLY | FUNCTIONS] <filename>

The “output” command allows you to specify the name of the output text file for the disassembler to write for either the disassembly or the functions output files. If “disassembly” or “functions” is not specified, the default is “disassembly”. If a path is not specified, the file will be placed in the current directory. An extension need not be given, and if none is given, none will be appended.

Failure to specify a Disassembly Output File will cause the disassembler to halt with an error. Multiple “output” commands can be used for the different types of output files that can be written. However, specifying a different output file for a given type replaces the output filename used for that file type, and this applies across all input Control Files, as only one output file, for each output file type, is written for a single disassembly run.

The entire portion of HC11 memory that is “loaded” (that is has a corresponding byte in the input source file) will be disassembled and written to the Disassembly Output File. The Disassembly Output File can then be viewed, edited, and/or printed by any favorite text editor. This Disassembly Output File can be reassembled if need be – see *Reassembling a Disassembly* in this document.

Specifying a Functions Output File, however, is optional. If specified, the disassembler will write a Functions Definition File in addition to the Disassembly Output File. The Functions Definition File is used as input to the Fuzzy Function Analyzer when cross-comparing the output of multiple disassemblies of different, but similar binary files.

Warning: If the specified Output File exists, the disassembler will overwrite it with the new disassembly and/or functions file without prompting you for confirmation. Any edits or changes you made by hand to the file will be lost. Therefore, I suggest that after you’ve finished running the disassembler, and before you start doing any manual editing or changes to the file, you rename the file. That way, if for some reason you need to rerun the disassembler, or even accidentally rerun it, you won’t inadvertently overwrite the previously edited file. Similarly, be sure to not accidentally specify the name of an existing file that you want to keep. You have been warned.

I typically use an extension of “.dis” for the disassembler output file. I then rename it to have a “.asm” extension before editing it. The “.asm” file then will become my cleaned up, commented version of the disassembled code. That way, if I need to rerun the disassembler to, perhaps, disassemble some missed portion of code from the previous attempt (such as an indirect branch table), I can do so and then just cut and paste as needed from the new “.dis” file into the “.asm” file without losing my edits. You may, however, have a different system that you prefer.

For the Functions Definition File, I usually use the extension of “.fnc”. The definition of the Functions Definition File is unique and specific to the Fuzzy Function Analyzer and isn't covered in this document.

TABWIDTH

Format: tabwidth <width>

The "tabwidth" command sets the number of characters that a tab should be defined as in the output file. If tabs are used (see "tabs" command), then enough tabs are inserted, based on this width, to make the fields properly align in the output file. If tabs are turned off, this value is used to determine how many spaces are needed to pad each field to make the fields align in the output file. If "tabwidth" is not specified, the default is 4.

The base of the <width> argument is dependent upon the current base setting – see the "base" command. An example of setting the "tabwidth" to 8 characters, with the default base of hexadecimal, would be:

```
tabwidth 8
```

Depending on the <width> number used, keeping the default base makes this command less intuitive. For example, "tabwidth 12" with the default hexadecimal base would really be 18 instead of 12. Therefore, you may wish to use the "base" command in the Control File to set the default base to either decimal or to nothing (see the "base" command). If you set it to nothing, you can use the "0x" prefix to specify other numbers in the Control File in hexadecimal format:

```
base none
tabwidth 12                 ; This is 12 bytes, not 18
```

If more than one "tabwidth" command is encountered, the final one encountered is the one that is actually used, and this applies to <u>all</u> Control Files when multiple Control Files are used.

List Entry Commands

ENTRY

Format: entry <address>

The code-seeking portion of the disassembler works by creating a list of entry points and then scanning the code starting with each entry point. If a branch or jump is encountered, the address for it, if it is a resolvable address that isn't already in the entry point list, is added to the entry point list. When a terminating instruction has been reached – such as an unconditional branch or return – then scanning with that code portion ends and the next entry in the entry point list is used. This continues until all entry points are exhausted.

The "entry" command allows you to specify hard entry points within the source. Typically, most of the entry points in the source can be specified with indirects (see the "indirect" command). But occasionally you'll run into a portion of code that for some reason has no indirect vector and requires a hard entry point. This is where the "entry" command comes into play.

Most of the "entry" commands in the sample Control File actually should have been entered as "indirect" since they were from jump tables. However, in the example I was trying to illustrate the functionality differences between "entry" and "indirect".

The base of the <address> argument is dependent upon the current base setting – see the "base" command. An example, to specify a code entry point at 0x7C12 with the default hexadecimal base, would be:

```
entry 7c12
```

Between the hard "entry" commands and the "indirect" commands in the control file, at least one entry point into the source must be ascertained in order for the disassembler to produce any code output. If no "entry" commands exist and no "indirect" commands exist, then the file's load address will be assumed to be a code entry point and automatically added to the list. However, if any entry point is specified (either from an "entry" or from an "indirect") then the source file load address is NOT added to the list of entry points.

EXITFUNCTION

Format: exitfunction <address> [<label>]

The “exitfunction” command is used in conjunction with the automatic function locating logic of this code-seeking disassembler. The disassembler looks for hard branches and return instructions to ascertain the beginning and ending of functions, and to determine which parts of the memory image is code and which parts are data. However, sometimes it incorrectly misses a “function exit”, such as when a “jsr” instruction is used as a branch into a jump table or v-table for an object where the “jsr” itself is followed by data containing details about the jump, but isn't code itself.

This can cause the disassembler to accidentally treat data following the “jsr” as being code rather than being data for the function call. Therefore, this command was added to allow jumps and branches to a specific address to be treated as an exit-of-current-function and to treat the bytes following the call as data instead of code.

The <address> specified is interpreted as being specified in the current base (see the “base” command). And the <label> if specified, is a shortcut that allows you to easily give the <address> a label without having to separately use the “label” command for the same address. The label, if specified, must follow typical variable naming conventions – that is, it should only contain alphanumeric characters and underscore (‘_’) and must start with a non-numeric first character.

Here's an example of using “exitfunction” to mark calls to a functions that are jump table functions:

```
exitfunction F948 VI_SWITCH_L06
```

This will mark the function at 0xF948 to be a jump table function and will give it the name “VI_SWITCH_L06”. A typical example would be something like:

```
VI_SWITCH_L06: pulx
; -----------------------------------------------------------------
LF949:         tst     0x04,x
               bne     LF951
               tst     0x05,x
               beq     LF95B
; -----------------------------------------------------------------
LF951:         subd    0x02,x
               beq     LF95B
               inx
               inx
               inx
               inx
               bra     LF949
; -----------------------------------------------------------------
LF95B:         ldx     0x00,x
               jmp     0x00,x                                ; Undetermined Branch Address
```

This will cause the disassembler to treat calls to 0xF948 as being the end of the current function, because the “jsr” to that function is followed by data for a jump table. For example:

```
; -----------------------------------------------------------------
LDABC:         ldab    LBD5B
               clra
               jsr     VI_SWITCH_L06
; =================================================================

               .word   LDB28
               .byte   0x00,0x7F
               .word   LDB20
               .byte   0x00,0x40
               .word   LDB18
               .byte   0x00,0x20
               .word   LDB10
               .byte   0x00,0x10
               .word   LDB08
               .byte   0x00,0x08
               .word   LDAFF
               .byte   0x00,0x04
               .word   LDAF6
               .byte   0x00,0x02
               .word   LDAED
               .byte   0x00,0x01
               .word   LDB30
```

```
              .byte   0x00,0x01
              .word   LDB89
              .byte   0xFF,0x01,0x00,0x00

; ====================================================================
LDAED:        ldd     #0x0051
              std     LBD57
              jmp     LDB89
; ====================================================================
```

In the above example, the function at 0xF948, which we are calling "VI_SWITCH_L06" is a function that treats the data after the "jsr" that called it (i.e. the return address) as being data for the jump table. We don't want the disassembler to incorrectly try to disassemble the data bytes following the "jsr" as code. So, by specifying 0xF948 as an "exitfunction", the disassembler knows that calls to that will be followed by data and it should exit the current function being disassembled and switch back to data mode.

This also means that you must manually add an "entry" command to your Control File for EVERY "jsr" to 0xF948 (or "VI_SWITCH_L06") so that the disassembler will know where to resume. This has to be done manually, because the disassembler has no idea exactly how big the jump table might be. In the above example, after we looked at the disassembler output and determined how big the jump table actually was, the following command was added to the Control File so that the disassembler would know to continue disassembling after the jump table:

```
entry DAED
```

In fact, I also added "indrect" commands to the Control File for all of the entries in the jump table so that the disassembler will know to treat each function reachable via the jump table as code:

```
indirect code DAC3                ; fdb loc_0_DB28
indirect code DAC7                ; fdb loc_0_DB20
indirect code DACB                ; fdb loc_0_DB18
indirect code DACF                ; fdb loc_0_DB10
indirect code DAD3                ; fdb loc_0_DB08
indirect code DAD7                ; fdb loc_0_DAFF
indirect code DADB                ; fdb loc_0_DAF6
indirect code DADF                ; fdb loc_0_DAED
indirect code DAE3                ; fdb loc_0_DB30
indirect code DAE7                ; fdb loc_0_DB89
entry DAED
```

In addition to dealing with jump tables, the "exitfunction" command is also useful for cases where the disassembler happens to incorrectly combine multiple functions together, which will adversely affect your cross-comparison of functions from several similar, but different binary files when using the Fuzzy Function Analyzer. If some particular jump or branch should have been picked up as a function exit criteria, then you can mark it as such so that any function or code following that jump will be treated as a separate function.

INDIRECT

Format: indirect [CODE | DATA] <address> [<label>]

Most all jump tables and vector tables on any processor are done by using indirect addressing. An indirect address is a memory location that contains a memory address to another location. These "indirect addresses" can be data or code, but in most cases are code pointers. A prime example of an indirect address on the HC11 is 0xFFFE, which is the reset vector for the processor. Immediately after power-on, the HC11 reads the 2 bytes starting at 0xFFFE and uses those 2-bytes as the address for starting the execution of the HC11 startup code.

The argument following the "indirect" command can specify whether the indirect vector points to code or to data. If this argument isn't included, then "code" is assumed. The base of the <address> argument is dependent upon the current base setting – see the "base" command. The <label> argument specifies the text that you want to use for the label of the vectored location (not the vector itself). The label must follow typical variable naming conventions – that is, it should only contain alphanumeric characters and underscore ('_') and must start with a non-numeric first character. The <label> argument is optional. If not specified, a label will be generated from the vectored location address in the 'Lxxxx' form. An example, to specify the reset vector located at 0xFFFE with the default hexadecimal base, would be:

```
indirect fffe reset
```

This is equivalent to:

```
Indirect code fffe reset
```

This example is illustrated as follows:

```
RESET:    .....
          ....
          ... reset program code ...
          ...

RSTVECT:  .word     RESET               <-- this is at FFFE and specs the vector data
```

This allows us to specify the address for the indirect without having to specifically look it up and resolve it by hand. The disassembler will look at address 0xFFFE and add the 2-byte value it finds there to the entry point table. And, it will also add the label specified, which is "RESET" in the above example, to the label table with the indirected address. Note that the label is assigned to the resolved address and not to the vector itself! If you want to assign a label to the indirect vector itself, you should also use the "label" command to add the label for the vector – this is illustrated in the example Control File. That is where the "RSTVECT" comes from in the above illustration:

```
label fffe rstvect
```

There is no character length limit for the label names, nor is there a limit to the number of indirect vectors that can be specified. However, you should take care not to exceed the name length limits of your target assembler if you are planning on reassembling the code, as it might have a limit.

Code-Indirect entries are another way of specifying entry points into the source. Between the hard "entry" commands and the "indirect" commands in the control file, at least one entry point into the source must be ascertained in order for the disassembler to produce any code output. If no "entry" commands exist and no "indirect" commands exist, then the file's load address will be assumed to be a code entry point and automatically added to the list. However, if any entry point is specified (either from an "entry" or from an "indirect") then the source file load address is NOT added to the list of entry points.

LABEL

Format: label <addr> <label>

The “label” command lets you assign a meaningful name to an address. If a label is needed for a particular memory address during the disassembly and you have not assigned a name to that address, the disassembler will create one in the form of “Lxxxx” where “xxxx” is the hexadecimal address of the memory location.

The base of the <address> argument is dependent upon the current base setting – see the “base” command. The <label> argument specifies the text that you want to use for the label. The label must follow typical variable naming conventions – that is, it should only contain alphanumeric characters and underscore (‘_’) and must start with a non-numeric first character. An example, to specify a label for the reset vector located at 0xFFFE with the default hexadecimal base, would be:

```
label fffe rstvect
```

This allows us, when looking at a disassembly output, to easily know what is what from the meaningful names. Instead of seeing either nothing or an Lxxxx name, we have a name that when we see it in the code, we will recognize it as to what it is referring to.

There is no character length limit for the label names, nor is there a limit to the number of labels that can be specified. However, you should take care not to exceed the name length limits of your target assembler if you are planning on reassembling the code, as it might have a limit.

If you attempt to add a label that is already in the label table, the addition will be ignored – keeping its original definition.

Parser Setting Commands

BASE

Format: base [<basetype>]

The “base” command allows you to set the numeric base to use when converting numeric arguments in the Control File. The default is hexadecimal (chosen to maintain backward compatibility with the Control Files on older versions of M6811DIS), meaning that all numbers in the Control File must be in hexadecimal <u>without</u> any leading or trailing base designator symbols such as “0x”, “$”, or “h”. This command allows you to override that default to use a different base or to set the base to “none”.

The <basetype> can be BINARY, OCTAL, DECIMAL, or HEXADECIMAL. These can be abbreviated as BIN, OCT, DEC, and HEX respectively. You can also use either NONE or OFF to turn off automatic typing. If you turn off automatic typing, you <u>can</u> then use standard C-Style typing as specified by the C “strtoul” function. That is, a number starting with “0x” is hexadecimal. Any number starting with a digit other than zero (0) is assumed to be in decimal. Any number starting with a zero (0), but not followed by an “x” is assumed to be in octal.

The “base” command, as well as all parser setting commands, do **<u>not</u>** carry over from one Control File to the next when using M6811DIS on multiple Control Files. This allows you to specify the Control File names on the command line without regard to order.

DFC

Format: dfc <library>

The “dfc” command allows you to specify a default Data File Converter Library to use in reading input memory image files. The <library> argument is the library name of a DFC to use. Data File Converters are included for “binary” (raw binary files), “intel” (Intel Hex format files), and “motorola” (Motorola S-Record Hex format files). Additional Data File Converters can be added and compiled into the code to support additional formats. To call those functions, use the name you defined in that class with this command.

The “dfc” command specifies the default DFC library used for loading source files as specified by the “load” command when a library is not specified along with the “load” command. It is also used to specify the library to use with files loaded with the (deprecated) “input” command.

The “dfc” command, as well as all parser setting commands, does **not** carry over from one Control File to the next when using M6811DIS on multiple Control Files. This allows you to specify the Control File names on the command line without regard to order.

Error and Warning Messages

The following Error and Warning messages can be reported during the execution of this version of the disassembler:

Error Messages

*** Error: Opening Control File "<filename>" for reading…

This indicates that the disassembler had trouble either locating or opening the specified Control File. Check to make sure that the file exists and is accessible to the disassembler and that you typed the name and/or path correctly on the command line.

*** Error: Unknown error

This error message should never appear. But if it does, it means that the Control File parser or one of its overrides in a child GDC class reported an error, but didn't set the error message to report.

*** Error: At least one input file must be specified in the control file(s) and successfully loaded

Check to make sure that you have at least one input file specified in the Control File with either the "input" command or the "load" command. Also, check to make sure that the files exist and that the DFC libraries you are specifying are compiled into your disassembler.

*** Error: Output file must be specified in the control file(s)

Make sure you have an "output" command properly specified in one of the Control Files for a Disassembly Output File.

*** Error: Can't open DFC library <library> to read "<filename>"

Check to see that the specified DFC library has been compiled into your disassembler. This version of M6811DIS comes with DFC libraries for "binary", "intel", and "motorola".

*** Error: Can't open file "<filename>" for reading

Make sure that the specified file exists, is accessible, that it isn't being used by another process, and that you have access rights to it.

*** Error: Reading file "<filename>"

Check the drive and make sure the media is still in place. Check to the file to make sure it isn't correct. If working on a network, check the network status.

*** Error: Unexpected end-of-file reading "<filename>"

The DFC indicated that it reached the end of the input file before it was expecting to. Such as reaching the end of an Intel Hex file without having read the EOF coding in the file. Check your file to make sure it isn't corrupt. Make sure you specified the correct DFC.

*** Error: Checksum error reading file "<filename>"

The DFC indicated that it encountered a checksum error in the data of the file it was reading. Check your file to make sure it isn't corrupt. Make sure you specified the correct DFC.

*** Error: Reading file "<filename>" extends past the defined memory limits of this processor

This indicates that during the loading of the Source File, the file went past the 64K limit of the processor space. The combined length (or size) of the Source File(s) and the specified starting or "load" offset must not exceed 64K or 0xFFFF + 1. For example from the sample Control File, we have a Source File that is 48K (or 0xC000 bytes) and a load offset of 0x4000 (or 16K). Together that is 64K or 0x10000, which is 0xFFFF + 1. If we were to specify a starting address higher than 0x4000, the file would attempt to extend past the end of the 64K boundary of the processor. If your file is too large to fit inside of 64K, such as that from a 128K Flash ROM or EPROM, then look for how the OEM of the equipment is bank selecting the data. I guarantee you that no more than 64K is actually visible to the HC11 at any given time. This means you'll have to break the source into multiple smaller parts and figure out how they relate. And yes, I've seen 128K and even larger files for the HC11 that were bank selected. This version provides no support for bank selecting, as the HC11 has no internal means (no machine instructions, etc) for doing bank selecting of external memory – meaning that every implementation of it will be unique. Since the Source Code for M6811DIS is freely available, you can modify it to add a different GDC definition that provides a bank selection mechanism and can override this restriction. But in any case you'll still have to reverse engineer the system enough to figure out what the bank switching methods are on the target device and write a function or interface for the disassembler to emulate it. And, it may not be possible to emulate all methods.

*** Error: Unknown DFC Error encountered while reading file "<filename>"

The DFC returned an error message that isn't supported by this GDC version. This typically indicates that you have a newer version of DFC than GDC. Apparently there's an error in reading the file, but the disassembler can't interpret the error message to tell you what it is. Since the DFC and GDC code is all compiled together, this means you compiled in source code from different disassembler versions that are incompatible with each other.

*** Error: Opening file "<filename>" for writing…

This indicates that the disassembler had trouble opening the Output File specified in the Control File. Check to make sure that you typed the name and/or path correctly in the Control File and that the target directory exists and has sufficient free space and that you have write-access rights to that directory.

*** Error: No entry addresses or indirect code vectors have been specified in the control file(s)

You must specify at least one valid entry point into the code of the memory image you are disassembling. Check the Control Files.

*** Error: Unknown command '<command>'

This means that one or more commands in the Control File(s) were not recognized. Check to make sure that all lines are either blank, start with a semicolon (';') for a comment line, or begin with one of the valid commands described in this document – perhaps it is just a typo. The commands can be uppercase or lowercase or mixed, as everything is converted internally to uppercase. Check to make sure that numbers are in the correct base as specified by the "base" command or to the default if no "base" command is used.

*** Error: Not enough arguments for '<command>' command

You didn't specify enough arguments for the specified command, or you mistyped one of the arguments and the parser misinterpreted them.

*** Error: Too many arguments for '<command>' command

You entered too many arguments for the specified command, or you mistyped one of the arguments and the parser misinterpreted them.

*** Error: Illegal argument for '<command>' command

The argument you specified for the command isn't valid for that command. Make sure that you typed the argument and the command correctly.

*** Error: Writing Output File "<filename>"…

This indicates that the disassembler had trouble while writing a line to the Output File. Check to make sure you haven't run out of disk space and that the disk is still properly mounted and accessible or if it is via network that the network isn't down.

Warning Messages

*** Warning: Branch Ref: <address> is outside of Loaded Source File

This indicates that the branch that was added to the branch table, during the code-seeking portion of the disassembler, referenced an address (indicated by <address> above) that was outside of the area loaded from the source file(s). This can occur normally when there are other memory sources, such as dynamic RAM routines, etc, that may not have been in the ROM image that was read and feed into the disassembler. Therefore, this warning could be of little consequence. However, it can also indicate that either the source file was not of the right size or that a load offset specified in the Control File was incorrect and caused a Source File to be loaded at incorrect memory locations. So check the source and make sure the warning makes sense.

*** Warning: Entry Point <address> is outside of loaded source file(s)…

This warning is basically synonymous with Branch Ref outside of Loaded Source File. The difference is that this applies to the entry points specified in the Control File, where as branch addresses are from branches found by the disassembler. However, the same guidelines apply to this warning as does the Branch Ref warning – so see "*** Warning: Branch Ref <address> is outside of Loaded Source File(s)…" for more information. Also, check to make sure that you have typed the entry point correctly in the Control File.

*** Warning: Vector Address <address> is outside of loaded source files(s)…

This warning is similar to "Entry Point … outside of loaded source file(s)", except that it applies to indirect vectors specified in the Control File(s). Follow the guidelines for "*** Warning: Entry Point <address> is outside of loaded source files(s)…" for additional information.

*** Warning: Indirect Address [<address>] -> <address> is outside of loaded source file(s)…

This warning indicates that the address specified by an indirect vector is outside of the loaded source file(s). It is similar to the "Vector Address … is outside of loaded source file(s)" except it applies to the address resolved from an indirect vector. Follow the guidelines for "*** Warning: Vector Address <address> is outside of loaded source file(s)…" for additional information.

*** Warning: Duplicate Label

This warning is displayed anytime a label that has already been defined in the Control File(s) is redefined again within the Control File(s) for a different address. Check to see if you've used the same label on different locations.

*** Warning: Duplicate indirect

This warning is displayed anytime an indirect vector that has already been defined in the Control File(s) is redefined again within the Control File(s).

*** Warning: Duplicate entry address

This warning is issued whenever an entry point address is specified in a Control File that has already been specified.

*** Warning: Input filename already defined

This warning is issued whenever an Input Filename is specified in a Control File after an Input Filename has already been specified. Note that the "input" command can only load one file within one Control File. To load multiple files, either multiple Control Files must be used or the "load" command must be used. It is recommended that you use the "load" command, as the "input" command is deprecated.

*** Warning: Disassembly Output filename already defined

This warning is issued whenever a Disassembly Output Filename is specified in a Control File after a Disassembly Output Filename has already been specified. Only one Disassembly Output Filename can be used.

*** Warning: Functions Output filename already defined

This warning is issued whenever a Functions Output Filename is specified in a Control File after a Functions Output Filename has already been specified. Only one Functions Output Filename can be used.

*** Warning: Reading file "<filename>" overlaps previously loaded files

This indicates that one or more of the Source File(s) attempted to overlap data previously loaded. Since a single memory address can only store one byte, so this warning is to notify you that at least one byte has been overwritten. You may have done this intentionally to combine one or more segments of a file without taking time to properly trim each file beforehand. However, that is not a good practice because while this version does load the files sequentially in the order specified in the Control File(s), there is no guarantee that future versions will.

Disassembly Pitfalls

There are many pitfalls often encountered when reverse engineering and/or hacking a particular system. Many aren't specific to any system, and since this document is **not** an explanation of how to do reverse engineering, we will only talk about things specific with the HC11 and more specifically about this disassembler.

Code Inline Data

The first big quirk or pitfall that comes to mind is data bytes passed on "jsr" or "bsr" instructions inline with the code. Some HC11 compilers, such as Cosmic C, make standard practice of this. For example, suppose you have the following code:

```
A_FUNC:   .set      OFST=12
          jsr       c_ents
          .byte     12
          ldd       #3
          jsr       getvalue
          clr       2,x
          clr       3,x
          std       OFST-2,x
          ldd       2,x
          std       OFST-4,x
          clra
          clrb
          std       2,x
```

This is a snippet of code from a real compiled program. Notice the ".byte 12" after the "jsr". That is an inline data argument passed to the function "c_ents". It is cleaner and has less overhead than pushing and popping the argument on the stack, but it causes problems with any disassembler. The problem is that the disassembler has no way of knowing that the "12" (or 0x0C) after the "jsr" is in fact a data byte. It will be assuming that the bytes immediately following the "jsr" will be the next instruction. In this particular case, since 0x0C is equivalent to the "clc" instruction – which happens to be a one byte immediate instruction, this will be interpreted by the disassembler as the follow (note that OFST=12, so OFST-2=10):

```
A_FUNC:   jsr       C_ENTS
          clc
          ldd       #3
          jsr       GETVAL
          clr       2,x
          clr       3,x
          std       10,x
          ldd       2,x
          std       8,x
          clra
          clrb
          std       2,x
```

In this case, it is only a bit confusing as you may think the "clc" (or clear-carry instruction) is a legitimate command and that may cause you to incorrectly interpret the code following the "jsr". In other cases, it can be more extreme. Suppose that instead of a simple one-byte immediate instruction, the byte happened to be the first byte of a two, three, or more, byte instruction? Then it could be that the "ldd #3" that follows and possibly even more instructions would get mangled as well, into erroneous instructions. Eventually, either the number of bytes will happen to fall back on track or you'll encounter an illegal byte that creates an unknown instruction for the processor – either will tend to get the disassembly back on track. But, this can cause problems with the code seeker, because suppose that one of the erroneous instructions happened to be a branch or jump of some sort, or worse yet is a return instruction. Or what if one of the mangled instructions was supposed to be a jump or branch. In the first case, you'd be adding extra incorrect branches (and may possibly pre-maturely end the current code section) and in the last case you'll fail to add a branch that should be added, which unless it is called elsewhere will result in code sections that will be outputted as data.

The fix for this problem isn't as easy as it appears. If each function had inline data of a fixed length, it would be fairly easy – you simply implement another list in the disassembler and specify that function "xyz" always has, for example, 2 bytes of data following any jump or branch to that function. The disassembler, when it encounters a call to function "xyz", would simply treat the 2 bytes following the call as data. But, the problem is that first you have to realize that that particular function uses data bytes in that fashion and tell the disassembler and that it is always 2 data bytes. What do we do if the number of bytes is variable? How can the disassembler know? An example of variable length would be a null-terminated string passed as inline data after a call. The length is determined by where the null is placed. Or what if, instead, it is a length/string argument where the first byte after the call is the length of the string or data that follows?

As you can see, there isn't an immediate, simple, fix-all solution. This version of the disassembler lets you deal with this problem, but you'll still have to figure out what functions use data passed inline and tag them using the "exitfunction" command in your Control File. And, you'll have to manually check every call to that function and add an "entry" command to your Control File so the disassembly can pick back up after the "jsr" or "bsr" call. For details and an example on how to do this, see the "exitfunction" command in this document, which shows an inline jump table example.

I've been contemplating adding additional commands to better automate the "fixed length" data variation and if there's enough interest, I may add code to do that in the future. And of course, since you have the source code to this disassembler, you can add your own GDC class override to deal with special cases of such calls, if it's posing a major difficulty in your disassembly efforts.

Undetermined Branch Address

Another very common pitfall is when the disassembler encounters a branch that it simply cannot figure out – such as a branch that is based off of register value. An example:

```
                        jsr       0,x
```

Since the disassembler has no way of knowing what value is contained in "x" it will not know what address the "jsr" branches to. When this occurs, the disassembler will comment the output file with "Undetermined Branch Address". Fortunately, many of these are simple jump tables. Look at the code preceding the "jsr" for any loading of the "x" register. Often you'll see the address of a branch table loaded and then an offset in the table added to it. That will be followed by something like "ldx 0,x" to load the actual vector from the table and then you'll have the "jsr 0,x".

If you encounter these, all you have to do is add "indirect" commands to the Control File for each entry in the branch table and then re-run the disassembler. This will allow the disassembler to track and disassemble all of the code that is there. I usually use the convention of naming the first indirect in the first jump table as "JT1R1" (for jump table 1 routine 1), the next routine as "JT1R2", and so on. When I come to the next table, I use "JT2R1", etc. Later on, once I actually figure out what "JT1R1", etc, really do, I'll give them more meaningful names. You may have a better method – so use whatever works well for you.

Unfortunately, there are still occasional calls, jumps, or branches that are not determinable by the disassembler and that even when you look at them, you can't figure out what they are as they may have no obvious jump table. The only solution for this is to figure out what the rest of the code does and work to figure out exactly what is called by the illusive function. I've always found that by working on other parts, parts that were more obvious, then eventually, before all was said and done, that I knew exactly what this call was for and why it was so illusive. But in any case, it does make life more difficult.

Addresses as Immediate Values

When a disassembler encounters an immediate value for an instruction, it has no way of knowing whether it should be treated strictly as a value or if it is really an address, or worse yet, an offset to some address. For example, suppose you encounter a system that, for the sake of argument, still has the HC11 registers located at 0x1000 in the HC11 memory space. And, you come across a routine that reads/writes from the SCI data register at 0x102F. You may encounter simple reads and writes directly to this address, such as:

```
                lda       $102F
```

(which is an extended addressing mode) which would be interpreted by the disassembler as:

```
                lda       L102F
```

You can later equate L102F with SCDR by adding a label for 102F to your Control File, rerun the disassembler, and your done:

```
                lda       SCDR
```

But, you may encounter something like:

```
                ldy #$102F
                lda 0,y
```

In this case, since the 0x102F is an immediate value, the disassembler has no way of knowing that it really corresponds to an address. For these, you'll have no choice but to manually change the $102F to SCDR (don't forget to add the label for 0x102F as SCDR in the Control File so your code will reassemble):

```
                ldy #SCDR
                lda 0,y
```

Worse yet, you can also encounter the following:

```
                ldy #$1000
                lda $2F,y
```

Here, the immediate value 0x1000 is the base address of the registers and 0x2F is the offset. This is why on the Ports files that I included, you'll see both direct addresses and register base relative addresses. That way, you can manually convert this to:

```
                ldy #REGBASE
                lda PSCDR,y
```

Where "regbase" is defined by you as the base for the HC11 registers and should also be the address that you originate the corresponding "ports" file when later re-assembling. In the example "ports" files, you'll notice that I've defined, in this particular case, SCDR as being the direct full address of the SCDR register or 0x102F (or wherever you originate the ports file) and PSCDR as being a pointer to the register relative address of SCDR or 0x2F. This allows us to use both names in the code to cover whatever form they are in.

As I've said, there is no way for the disassembler to know when it encounters an immediate value if that immediate value is really an address or if it is only data, and register-offset addresses (such as the lda $2F,y) are even worse yet. Thus the only recourse is to manually edit it in the Disassembly Output File, if it's important to you for your project – usually, search-and-replace works well for doing this.

Code Paging

The HC11 can only directly access 64K bytes of memory. This includes all RAM, ROM, Registers, Memory-Mapped Devices, etc. Unfortunately, many programs, especially those written by today's inefficient compilers, exceed this limit. This causes the designer to have to implement workarounds. The most common is to implement a method of paging in multiple banks of memory. Unfortunately, not only does the HC11 not support more than 64K of direct access, but also it has no built-in means for performing paging either. Thus, the designer is left to implement his or her own unique solution to the paging problem.

The biggest problem for the designer isn't so much that there must be some external paging means, but that there can be no direct access of data from one page to another across page boundaries – since there are no code-segment and data-segment registers as can be found in processors like the 8088. For the reverse-engineer and/or hacker, this further complicates things because there is no "standard method" for implementing the paging technique, and so no way that a disassembler can automatically sort this out for you.

One common way to implement a simple 2-bank method is to use a 128K byte ROM and connect the upper address line to an output port pin of the HC11. Part of the ROM's code is duplicated from one half of the ROM to the other so that it is accessible by the HC11 in both pages – i.e. the common page. The common page is the page that must do the actual page swaps (since it doesn't change during the swap). The other half of the ROM is unique to that page and allows an extra extension of the ROM in the range of 32K to 64K depending on the sections that must be common and/or duplicated across pages.

Another common technique is to use PAL or PLD logic to serve as a page register. This allows for more intricate design and layout of the pages and would allow for more flexibility as to where the pages get "banked-in" and when. But, this really makes life difficult for the hacker or reverse-engineer.

This list could go on and on – unfortunately – as there is no single technique, not even a unique dozen, that can describe all the different methods people have used for paging. Therefore, when reverse-engineering large projects, it will be necessary to determine the paging technique (if any) by hand. It also means, since the disassembler can only deal with the HC11's direct 64K space, that you must divide the source up into multiple files – each corresponding to unique pages – and run them separately through the disassembler. And, not only does the disassembler have problems with multiple pages, but HC11 assemblers have problems as well. Most assemblers will require that you uniquely assemble each section and then link them manually into the correct positions in the final output file.

Laziness

"But I don't want to have to go through the program and tag all of the indirect vectors and entry point locations; I just want it to dump out the code." Well, there isn't much that can be done about being lazy, but there can be a "spit" mode that disables the code-seeking portion – or more correctly, labels everything as code – and dumps or "spits" the disassembly out. This can be useful in systems that have a large number of indirects and you want to do a quick hack on the file and don't really care about truly reverse engineering the code.

Originally, when this program was developed, the initial goal was reverse engineering, not hacking. Therefore, the early versions had no "spit" mode. However, because of many requests from hackers that want quick results, this version now supports a "spit" command in the Control File that will disable the code-seeker and simply output a disassembly of everything in much the same form of an ordinary "dumb" disassembler.

Others

Well, this is about all I can think of including at this present time. I'm sure there are many more that warrant being added to this document. If you know of any, let me know and it will possibly be included in future editions. See the *Support* section in this document for contact information.

MC68HC11 Overview

As previously stated, the purpose of this document isn't to teach you about the functionality of the HC11 – that's what Motorola's documentation is for. However, for completeness, I thought it wise to include a list of opcodes and corresponding mnemonics that the disassembler processes as well as how the disassembler's code-seeker behaves with each – and that is what the following table is all about. For everything else, check out Motorola's website – or actually, checkout the Freescale website, as Motorola dumped (er, uh, sold off) their SPS chip business to Freescale (https://www.freescale.com/webapp/sps/library/prod_lib.jsp).

Mnemonic	Machine Code	Form	Disassembler Action	Discontinue Disassembly
test	00	test		
nop	01	nop		
idiv	02	idiv		
fdiv	03	fdiv		
lsrd	04	lsrd		
lsld	05	lsld		
tap	06	tap		
tpa	07	tpa		
inx	08	inx		
dex	09	dex		
clv	0A	clv		
sev	0B	sev		
clc	0C	clc		
sec	0D	sec		
cli	0E	cli		
sei	0F	sei		
sba	10	sba		
cba	11	cba		
brset	12 dd mm rr	brset *dd,#mm,.+rr	Add Data Label, Add Branch Addr & Label	
brclr	13 dd mm rr	brclr *dd,#mm,.+rr	Add Data Label, Add Branch Addr & Label	
bset	14 dd mm	bset *dd,#mm	Add Data Label	
bclr	15 dd mm	bclr *dd,#mm	Add Data Label	
tab	16	tab		
tba	17	tba		
iny	18 08	iny		
dey	18 09	dey		
bset	18 1C ff mm	bset ff,y,#mm		
bclr	18 1D ff mm	bclr ff,y,#mm		
brset	18 1E ff mm rr	brset ff,y,#mm,.+rr	Add Branch Addr & Label	
brclr	18 1F ff mm rr	brclr ff,y,#mm,.+rr	Add Branch Addr & Label	
tsy	18 30	tsy		
tys	18 35	tys		
puly	18 38	puly		
aby	18 3A	aby		
pshy	18 3C	pshy		
neg	18 60 ff	neg ff,y		
com	18 63 ff	com ff,y		
lsr	18 64 ff	lsr ff,y		
ror	18 66 ff	ror ff,y		
asr	18 67 ff	asr ff,y		
lsl	18 68 ff	lsl ff,y		
rol	18 69 ff	rol ff,y		
dec	18 6A ff	dec ff,y		
inc	18 6C ff	inc ff,y		
tst	18 6D ff	tst ff,y		
jmp	18 6E ff	jmp ff,y	Undeterminable Branch	discontinue
clr	18 6F ff	clr ff,y		
cpy	18 8C jj kk	cpy #jjkk		
xgdy	18 8F	xgdy		
cpy	18 9C dd	cpy *dd	Add Data Label	
suba	18 A0 ff	suba ff,y		
cmpa	18 A1 ff	cmpa ff,y		
sbca	18 A2 ff	sbca ff,y		
subd	18 A3 ff	subd ff,y		
anda	18 A4 ff	anda ff,y		
bita	18 A5 ff	bita ff,y		
ldaa	18 A6 ff	ldaa ff,y		
staa	18 A7 ff	staa ff,y		
eora	18 A8 ff	eora ff,y		
adca	18 A9 ff	adca ff,y		
oraa	18 AA ff	oraa ff,y		
adda	18 AB ff	adda ff,y		
cpy	18 AC ff	cpy ff,y		
jsr	18 AD ff	jsr ff,y	Undeterminable Branch	
lds	18 AE ff	lds ff,y		
sts	18 AF ff	sts ff,y		

cpy	18 BC hh ll	cpy hhll	Add Data Label	
ldy	18 CE jj kk	ldy #jjkk		
ldy	18 DE dd	ldy *dd	Add Data Label	
sty	18 DF dd	sty *dd	Add Data Label	
subb	18 E0 ff	subb ff,y		
cmpb	18 E1 ff	cmpb ff,y		
sbcb	18 E2 ff	sbcb ff,y		
addd	18 E3 ff	addd ff,y		
andb	18 E4 ff	andb ff,y		
bitb	18 E5 ff	bitb ff,y		
ldab	18 E6 ff	ldab ff,y		
stab	18 E7 ff	stab ff,y		
eorb	18 E8 ff	eorb ff,y		
adcb	18 E9 ff	adcb ff,y		
orab	18 EA ff	orab ff,y		
addb	18 EB ff	addb ff,y		
ldd	18 EC ff	ldd ff,y		
std	18 ED ff	std ff,y		
ldy	18 EE ff	ldy ff,y		
sty	18 EF ff	sty ff,y		
ldy	18 FE hh ll	ldy hhll	Add Data Label	
sty	18 FF hh ll	sty hhll	Add Data Label	
daa	19	daa		
cpd	1A 83 jj kk	cpd #jjkk		
cpd	1A 93 dd	cpd *dd	Add Data Label	
cpd	1A A3 ff	cpd ff,x		
cpy	1A AC ff	cpy ff,x		
cpd	1A B3 hh ll	cpd hhll	Add Data Label	
ldy	1A EE ff	ldy ff,x		
sty	1A EF ff	sty ff,x		
aba	1B	aba		
bset	1C ff mm	bset ff,x,#mm		
bclr	1D ff mm	bclr ff,x,#mm		
brset	1E ff mm rr	brset ff,x,#mm,.+rr	Add Branch Addr & Label	
brclr	1F ff mm rr	brclr ff,x,#mm,.+rr	Add Branch Addr & Label	
bra	20 rr	bra .+rr	Add Branch Addr & Label	discontinue
brn	21 rr	brn .+rr	Add Branch Addr & Label	
bhi	22 rr	bhi .+rr	Add Branch Addr & Label	
bls	23 rr	bls .+rr	Add Branch Addr & Label	
bcc	24 rr	bcc .+rr	Add Branch Addr & Label	
bcs	25 rr	bcs .+rr	Add Branch Addr & Label	
bne	26 rr	bne .+rr	Add Branch Addr & Label	
beq	27 rr	beq .+rr	Add Branch Addr & Label	
bvc	28 rr	bvc .+rr	Add Branch Addr & Label	
bvs	29 rr	bvs .+rr	Add Branch Addr & Label	
bpl	2A rr	bpl .+rr	Add Branch Addr & Label	
bmi	2B rr	bmi .+rr	Add Branch Addr & Label	
bge	2C rr	bge .+rr	Add Branch Addr & Label	
blt	2D rr	blt .+rr	Add Branch Addr & Label	
bgt	2E rr	bgt .+rr	Add Branch Addr & Label	
ble	2F rr	ble .+rr	Add Branch Addr & Label	
tsx	30	tsx		
ins	31	ins		
pula	32	pula		
pulb	33	pulb		
des	34	des		
txs	35	txs		
psha	36	psha		
pshb	37	pshb		
pulx	38	pulx		
rts	39	rts		discontinue
abx	3A	abx		
rti	3B	rti		discontinue
pshx	3C	pshx		
mul	3D	mul		
wai	3E	wai		
swi	3F	swi		
nega	40	nega		
coma	43	coma		
lsra	44	lsra		
rora	46	rora		
asra	47	asra		
lsla	48	lsla		
rola	49	rola		
deca	4A	deca		
inca	4C	inca		
tsta	4D	tsta		
clra	4F	clra		
negb	50	negb		
comb	53	comb		
lsrb	54	lsrb		
rorb	56	rorb		
asrb	57	asrb		
lslb	58	lslb		
rolb	59	rolb		

decb	5A	decb		
incb	5C	incb		
tstb	5D	tstb		
clrb	5F	clrb		
neg	60 ff	neg ff,x		
com	63 ff	com ff,x		
lsr	64 ff	lsr ff,x		
ror	66 ff	ror ff,x		
asr	67 ff	asr ff,x		
lsl	68 ff	lsl ff,x		
rol	69 ff	rol ff,x		
dec	6A ff	dec ff,x		
inc	6C ff	inc ff,x		
tst	6D ff	tst ff,x		
jmp	6E ff	jmp ff,x	Undeterminable Branch	discontinue
clr	6F ff	clr ff,x		
neg	70 hh ll	neg hhll	Add Data Label	
com	73 hh ll	com hhll	Add Data Label	
lsr	74 hh ll	lsr hhll	Add Data Label	
ror	76 hh ll	ror hhll	Add Data Label	
asr	77 hh ll	asr hhll	Add Data Label	
lsl	78 hh ll	lsl hhll	Add Data Label	
rol	79 hh ll	rol hhll	Add Data Label	
dec	7A hh ll	dec hhll	Add Data Label	
inc	7C hh ll	inc hhll	Add Data Label	
tst	7D hh ll	tst hhll	Add Data Label	
jmp	7E hh ll	jmp hhll	Add Branch Addr & Label	discontinue
clr	7F hh ll	clr hhll	Add Data Label	
suba	80 ii	suba #ii		
cmpa	81 ii	cmpa #ii		
sbca	82 ii	sbca #ii		
subd	83 jj kk	subd #jjkk		
anda	84 ii	anda #ii		
bita	85 ii	bita #ii		
ldaa	86 ii	ldaa #ii		
eora	88 ii	eora #ii		
adca	89 ii	adca #ii		
oraa	8A ii	oraa #ii		
adda	8B ii	adda #ii		
cpx	8C jj kk	cpx #jjkk		
bsr	8D rr	bsr .+rr	Add Branch Addr & Label	
lds	8E jj kk	lds #jjkk		
xgdx	8F	xgdx		
suba	90 dd	suba *dd	Add Data Label	
cmpa	91 dd	cmpa *dd	Add Data Label	
sbca	92 dd	sbca *dd	Add Data Label	
subd	93 dd	subd *dd	Add Data Label	
anda	94 dd	anda *dd	Add Data Label	
bita	95 dd	bita *dd	Add Data Label	
ldaa	96 dd	ldaa *dd	Add Data Label	
staa	97 dd	staa *dd	Add Data Label	
eora	98 dd	eora *dd	Add Data Label	
adca	99 dd	adca *dd	Add Data Label	
oraa	9A dd	oraa *dd	Add Data Label	
adda	9B dd	adda *dd	Add Data Label	
cpx	9C dd	cpx *dd	Add Data Label	
jsr	9D dd	jsr *dd	Add Branch Addr & Label	
lds	9E dd	lds *dd	Add Data Label	
sts	9F dd	sts *dd	Add Data Label	
suba	A0 ff	suba ff,x		
cmpa	A1 ff	cmpa ff,x		
sbca	A2 ff	sbca ff,x		
subd	A3 ff	subd ff,x		
anda	A4 ff	anda ff,x		
bita	A5 ff	bita ff,x		
ldaa	A6 ff	ldaa ff,x		
staa	A7 ff	staa ff,x		
eora	A8 ff	eora ff,x		
adca	A9 ff	adca ff,x		
oraa	AA ff	oraa ff,x		
adda	AB ff	adda ff,x		
cpx	AC ff	cpx ff,x		
jsr	AD ff	jsr ff,x	Undeterminable Branch	
lds	AE ff	lds ff,x		
sts	AF ff	sts ff,x		
suba	B0 hh ll	suba hhll	Add Data Label	
cmpa	B1 hh ll	cmpa hhll	Add Data Label	
sbca	B2 hh ll	sbca hhll	Add Data Label	
subd	B3 hh ll	subd hhll	Add Data Label	
anda	B4 hh ll	anda hhll	Add Data Label	
bita	B5 hh ll	bita hhll	Add Data Label	
ldaa	B6 hh ll	ldaa hhll	Add Data Label	
staa	B7 hh ll	staa hhll	Add Data Label	
eora	B8 hh ll	eora hhll	Add Data Label	
adca	B9 hh ll	adca hhll	Add Data Label	

oraa	BA hh ll	oraa hhll	Add Data Label
adda	BB hh ll	adda hhll	Add Data Label
cpx	BC hh ll	cpx hhll	Add Data Label
jsr	BD hh ll	jsr hhll	Add Branch Addr & Label
lds	BE hh ll	lds hhll	Add Data Label
sts	BF hh ll	sts hhll	Add Data Label
subb	C0 ii	subb #ii	
cmpb	C1 ii	cmpb #ii	
sbcb	C2 ii	sbcb #ii	
addd	C3 jj kk	addd #jjkk	
andb	C4 ii	andb #ii	
bitb	C5 ii	bitb #ii	
ldab	C6 ii	ldab #ii	
eorb	C8 ii	eorb #ii	
adcb	C9 ii	adcb #ii	
orab	CA ii	orab #ii	
addb	CB ii	addb #ii	
ldd	CC jj kk	ldd #jjkk	
cpd	CD A3 ff	cpd ff,y	
cpx	CD AC ff	cpx ff,y	
ldx	CD EE ff	ldx ff,y	
stx	CD EF ff	stx ff,y	
ldx	CE jj kk	ldx #jjkk	
stop	CF	stop	
subb	D0 dd	subb *dd	Add Data Label
cmpb	D1 dd	cmpb *dd	Add Data Label
sbcb	D2 dd	sbcb *dd	Add Data Label
addd	D3 dd	addd *dd	Add Data Label
andb	D4 dd	andb *dd	Add Data Label
bitb	D5 dd	bitb *dd	Add Data Label
ldab	D6 dd	ldab *dd	Add Data Label
stab	D7 dd	stab *dd	Add Data Label
eorb	D8 dd	eorb *dd	Add Data Label
adcb	D9 dd	adcb *dd	Add Data Label
orab	DA dd	orab *dd	Add Data Label
addb	DB dd	addb *dd	Add Data Label
ldd	DC dd	ldd *dd	Add Data Label
std	DD dd	std *dd	Add Data Label
ldx	DE dd	ldx *dd	Add Data Label
stx	DF dd	stx *dd	Add Data Label
subb	E0 ff	subb ff,x	
cmpb	E1 ff	cmpb ff,x	
sbcb	E2 ff	sbcb ff,x	
addd	E3 ff	addd ff,x	
andb	E4 ff	andb ff,x	
bitb	E5 ff	bitb ff,x	
ldab	E6 ff	ldab ff,x	
stab	E7 ff	stab ff,x	
eorb	E8 ff	eorb ff,x	
adcb	E9 ff	adcb ff,x	
orab	EA ff	orab ff,x	
addb	EB ff	addb ff,x	
ldd	EC ff	ldd ff,x	
std	ED ff	std ff,x	
ldx	EE ff	ldx ff,x	
stx	EF ff	stx ff,x	
subb	F0 hh ll	subb hhll	Add Data Label
cmpb	F1 hh ll	cmpb hhll	Add Data Label
sbcb	F2 hh ll	sbcb hhll	Add Data Label
addd	F3 hh ll	addd hhll	Add Data Label
andb	F4 hh ll	andb hhll	Add Data Label
bitb	F5 hh ll	bitb hhll	Add Data Label
ldab	F6 hh ll	ldab hhll	Add Data Label
stab	F7 hh ll	stab hhll	Add Data Label
eorb	F8 hh ll	eorb hhll	Add Data Label
adcb	F9 hh ll	adcb hhll	Add Data Label
orab	FA hh ll	orab hhll	Add Data Label
addb	FB hh ll	addb hhll	Add Data Label
ldd	FC hh ll	ldd hhll	Add Data Label
std	FD hh ll	std hhll	Add Data Label
ldx	FE hh ll	ldx hhll	Add Data Label
stx	FF hh ll	stx hhll	Add Data Label

Where:

dd	=	8-Bit Direct Address (0x0000 – 0x00FF). High byte assumed to be 0x00.
ff	=	8-Bit Positive Offset 0x00 (0) to 0xFF (255) added to index register value.
hh	=	High Order Byte of a 16-bit Extended Address.
ii	=	Single Byte of Immediate Data.
jj	=	High Order Byte of 16-Bit Immediate Data.
kk	=	Low Order Byte of 16-Bit Immediate Data.
ll	=	Low Order Byte of a 16-bit Extended Address.

mm	=	8-Bit Mask (Bits that are set are the bits that will be affected).
rr	=	Signed Relative Offset 0x80 (-128) to 0x7F (127). Offset is relative to the address **<u>following</u>** the machine code offset byte.

Instructions listed as “discontinue disassembly” are instructions that end the current stream of code. The code-seeker starts with the first entry point and disassembles until one of the following conditions is satisfied:

- An instruction flagged as “discontinue” is encountered (these are hard jumps or returns)
- It encounters code that has already been tested.
- It encounters an illegal instruction (an opcode byte that isn’t in the above table)
- It encounters a jump or branch to a function tagged with the “exitfunction” command

It then reads the next entry point, and continues iterating until all entry points are exhausted.

Reassembling a Disassembly

As stated earlier in this document, there are typically two approaches to disassembly work – hacking and reverse-engineering. Typically, the hacker is only interested in finding out what is in a program or binary enough to complete a hack and isn't interested in the overall scheme of figuring out the how and why of the workings of the entire system. So for the hacker, being able to reassemble a disassembly is probably of little or no importance. However, to the serious person working on fully reverse engineering a system, being able to easily reassemble a disassembly is a life-send. The M6811 Code-Seeking Disassembler was designed for the reverse-engineer and so the output was targeted for a specific assembler.

This disassembler targets the AS6811 assembler written by Alan Baldwin at Kent State University's Physics Department (**not to be confused with the Motorola AS11 freeware assembler**). Alan's entire assembler set and relocating linker is a superb piece of workmanship, which is why it was chosen as the target output form for this disassembler.

The AS6811 assembler is available in freeware/shareware form with complete source code and can be found on many freeware CD-ROM sets, including several by Walnut Creek. So, you should be able to locate the assembler without problem. If not, you can download it from my website, or other site, as described in the *Support* Section of this document.

For most programs, you can use the assembler unmodified to reassemble this disassembler's output back into the original binary. However, there are a few considerations you should keep in mind. The first one is the memory model that the pre-compiled version of the assembler is designed for. Sometimes it is necessary to rebuild the assembler from the source using a larger memory model in order to assemble very large HC11 programs.

Another problem, and probably the main problem, is the extended address optimizer in the assembler. Alan's assembler, when given an address in the 0x0000 to 0x00FF range, will optimize the assembled code to be that of a Direct Address Mode instruction (see the tables in the *MC68HC11 Overview* section of this document) regardless of whether you specify it as a Direct Mode instruction ("*" operator) or not. Ordinarily, this is a good feature of the assembler and allows you to produce binary files that are properly space optimized. However, the HC11 also allows for Extended Address Mode instructions to also access the range of 0x0000 to 0x00FF. This can cause output from the disassembled code, when reassembled, to not match that of the original binary if the original binary has one or more instructions that are not optimized into the Direct Mode form – not a good thing when reverse-engineering.

To solve this problem, this disassembler will use the "*" operator for the target address on all Direct Address Mode instructions. This should signify to the assembler that it is indeed a Direct Address in the 0x0000 to 0x00FF range. Any extended addresses will be outputted by the disassembler with **no** prefix operator. The assembler should interpret these addresses as being Extended Mode addresses, regardless of the fact that they might be in the 0x0000 to 0x00FF range. Unfortunately, this means that Alan's assembler source code must be altered to disable this optimizing feature in order to produce truly compatible binaries.

So, I recommend you download the source code version of the assembler, disable the optimizer, and recompile in a large memory model. Again, refer to the *Support* section in this document, as I do offer an already modified version of the assembler both in source and in binary form – but, to maintain proper redistribution policies of his license, I do have it in the true, unmodified form. So while you can use the unmodified version for reassembling code that is anatomically correct, you should be aware that if you reassemble and the sizes and/or addresses don't seem to match, start looking for optimized verses non-optimized instructions. Typically, the first point of divergence will be the first non-optimized instruction.

Other details of assembly are out of the scope of this document – for those, I refer you to Alan's manual that comes with the assembler, as well as Motorola documentation.

Additional Examples

In addition to the “av94bnbh.ctl” Control File in the examples folder, I also included “base.ctl”, which was the Control File I used for a real reverse-engineering code-recovery project. Non-Disclosure-Agreements and copyright limitations prevent me from being able to include the binaries and resulting disassembled code from that project, but the Control File may be helpful in learning how to setup some common things for the disassembler.

The “base.ctl” Control File illustrates dealing with a microprocessor where the I/O ports of the HC11 have been relocated. And it shows how to use the “exitfunction” command, in conjunction with additional “indirect” and “entry” commands, to deal with jump tables.

Fuzzy Function Analyzer

The project where I was using the "base.ctl" Control File was one where I was hired by a company who had managed to lose the C Source Code for the binary they were currently shipping in their product. They still had the C Source Code for an old version and they wanted to recover the C Source Code for the current binary.

That's when I first invented the Fuzzy Function Analyzer. The Fuzzy Function Analyzer uses "A Greedy Algorithm for Aligning DNA Sequences" (see: http://pipmaker.bx.psu.edu/dist/greedy.pdf, also checked into my Source Code Repository under "articles") to do a cross-comparison of every function from one disassembly (one Functions Output File) with every function from another disassembly (a second Functions Output File).

This allows you to disassemble the binary for the copy of the code you have from the known C Source Code and disassemble the binary for the newer code that you wish to recover, and find all of the places where they match, by percentage of match.

What makes the "Fuzzy Function Analyzer" "fuzzy" is the fact that it's able to fuzzify addressing, both absolute and relative, in the comparison process and assign them a weight in the comparison process rather than requiring that the explicitly match. This allows code that is functionally equivalent, but only relocated to a different address or perhaps has had only a couple lines of code added to it, show up as being a "best match", allowing you to figure out exactly what is the same and what is different between two binaries.

This same technique can also be applied solely to reverse-engineering where you've already completely figured out one binary, and now have a similar, but different, binary file and want to apply everything you figured out from the previous binary to the new one. An example might be two different ECM or ECU modules for automotive, where you've reverse engineered one controller and now you have a new controller with a similar code base, but where things have been added or changed, and you want to see and analyze those changes without starting all over with your reverse-engineering efforts.

The specifics of the Fuzzy Function Analyzer are not covered in this document, other than to point out that that tool uses the Functions Output File from this disassembler, which is why the Functions Output File was developed.

The Functions Output File format will be documented in the documentation for the Fuzzy Function Analyzer, and not this document, but in short, it's a processor independent way of representing the assembly instructions in a form that can be easily compared with an algorithm, such as a DNA Sequence Alignment Algorithm. It tags addresses that are relative and includes the corresponding relative offsets. It also marks data differently from code so that can be taken into consideration during the comparison process.

The code for the Fuzzy Function Analyzer is also checked into the same Source Code Repository as this disassembler is. If you wish to pursue it in more detail, then checkout that code and see if you can apply it to your project.

Bugs

With any software application, it is likely that at least one bug will exist somewhere. The previous versions had a few. But at the current time of writing this document, I don't know of any in this version. If you do find a bug, or think you have, please contact me and let me know – See the *Support* section for contact information.

Support

The Disassembler

This version, and future M6811 disassembler versions will be provided free of charge and can be freely distributed provided you supply the disassembler in its entirety, including support files, without changes or modifications. It's available in both Binary and Source Code forms and accessible on SourceForge at: https://sourceforge.net/projects/m6811dis/

There you'll find this M6811 Code-Seeking Disassembler (in any of the versions I create), as well as the AS6811 assembler both in virgin and in modified forms as described earlier in this document. I will also have copies of some of the Motorola documentation in .pdf format, to keep you from having to dig and search on Motorola's website.

If you find any bugs, have suggestions or ideas for program enhancement, or have any questions in general, you can email me at: dwhisnant@dewtronics.com. You can visit my personal website at: http://www.dewtronics.com/.

Motorola

Documentation on the MC68HC11 processor family, as well as other assemblers, disassemblers, and support utilities, can be found on Motorola's website – if you look hard enough that is. Unless they've improved their site recently, it will take a bit of hunting and searching around – though keep looking, because it is there somewhere – or actually, checkout the Freescale website, as Motorola dumped (er, uh, sold off) their SPS chip business to Freescale (https://www.freescale.com/webapp/sps/library/prod_lib.jsp)

Third Party (Assemblers, etc)

Alan Baldwin's M6811 assembler can be found on my website as well as from many freeware/shareware CD distribution houses, such as Walnut Creek. The version I originally tested and developed against, Version 1.50, with source, came directly from Alan himself back in April of 1995. Since then, he has generated newer versions and made additional enhancements. At the time of this writing, the latest version is 5.00. But, the last version I've actually tested and used was version 2.21, which worked without problems, but had to be altered in the same way as described in *Reassembling a Disassembly*. As I come across other versions and resources, I will post them online to be downloaded, but I will always try to keep a version online that I have tested and verified to be working with the disassembler.

According to the AS6811 documentation, Alan can be reached at the following address:

Alan R. Baldwin
Kent State University
Physics Department
Kent, Ohio 44242
Phone: 330-672-2531
Fax: 330-672-2959

His documentation also states that the assembler is available via anonymous FTP to: shop-pdp.kent.edu. And that it is also available from the C Users' Group:

The C Users' Group
1601 W. 23rd Street, Suite 200
Lawrence, KS 66046-2700 USA
Phone: 913-841-1631
Fax: 913-841-2624

From a web search, it appears that the documentation for his ASxxxx assemblers can be found at:
http://shop-pdp.net/ashtml/asxdoc.htm

And the assemblers can be downloaded from:
http://shop-pdp.net/ashtml/asxget.php

The ASxxxx collection contains cross assemblers for the 6800(6802/6808), 6801(hd6303), 6804, 6805, 68HC08, 6809, 68HC11, 68HC12, 68HC16, 8051, 8085(8080), z80(hd64180), H8/3xx, and 6500 series microprocessors.

You will probably want to have a good hex editor as well. A decent hex editor for Windows can be found at BreakPoint Software, called Hex Workshop, at www.bpsoft.com. And most Linux distributions have Okteta http://utils.kde.org/projects/okteta/.

And, a good text editor won't hurt any either. One of the better text editors I've found for Windows is made by Helios Software, and is called TextPad – available at www.textpad.com.

Version History

Below is a list of what is planned with each version. Some of these have been put into place, others still have to be worked in:

Version 1.0 – DOS 16-Bit – Written April 15, 1996
First release version. Somewhat limited, but it runs in DOS on nearly any machine and is a great entry level version. This version was written in Borland Pascal.

Version 1.1 – DOS 16-Bit
This was an intermediate stepping stone version that was never released to the public. It added multiple source files, has "spit" output mode capabilities, and a few other slight features – but was never refined.

Version 1.2 – DOS 32-Bit – Written July 29, 1999
Upgraded to a 32-bit application so there are no memory limits other than the machine's physical memory. DFC (Data File Converter) DLL's were added to support any source data file type. Multiple Source Files is supported, as is multiple Control Files. It can support the "spit" mode or code-seeking. Mixed number bases is supported in the Control File. The disassembler has been converted into a GDC (Generic Disassembly Class) that will facilitate the development of disassemblers for other processors. Since there are "no memory limits", label names can be of any size and there can be as many labels, branch references, indirects, and entry points as needed, and Indirect Data Vectors are supported in addition to Indirect Code Vectors. This version was first written in Borland VC++ 5.0 and then reworked to MSVC++ 4.0 and 5.0 and used MFC classes, like CMap, CString, etc.

Version 2.0 – Generic Multi-Platform – Written June 14, 2014
For this version, I dropped all dependencies on Microsoft and Windows, as I now exclusively use Linux for my personal operation system. I replaced all MFC class usage with STL classes, making it completely generic and portable. This version will compile and run on Linux, Windows, and Mac and nearly any system that has a GCC compiler or cross-compiler with C++ Standard Template Library support. This version was created because of the continued ongoing demand for this disassembler and done to make it easier for me to support and make available, since it's no longer dependent on Microsoft products. Having the DFC and GDC libraries as separate DLL's were done away with and compiled into the program itself, eliminating the "DLL Hell" of Windows and keeping it truly platform independent. The code separation still exists, but as C++ classes. Additional "libraries" can be added as new classes inherited from the base classes and compiled into the program to add additional functionality. Also, as of this version, the M6811 Code-Seeking Disassembler is now provided as an Open Source application and is available on SourceForge.

Enjoy the disassembler! I hope it proves to be most helpful. Please visit the SourceForge website (see the *Support* section) and leave reveiws and report and issues. I enjoy tracking the progress of my software and like to see how many different countries it ends up in. Presently, its being used in over 137 different countries.