

Dewtronics

M6811DIS v1.0 (DOS)

Code-Seeking Disassembler for the Motorola MC68HC11
Microprocessor

Version 1.0 of Software written April 15, 1996, © Donald Whisnant
Documentation for version 1.0 written September 27, 1998, © Donald Whisnant
Last Update: June 28, 1999

Table Of Contents

Table Of Contents.....	2
Introduction	3
Installation	5
Usage	6
Overview	6
Step-by-Step Walk-Through.....	7
Control Files	15
Example Control File.....	16
Control File Commands.....	18
Switch Commands	18
ADDRESSES	18
ASCII.....	19
OPCODES.....	20
Value Commands.....	21
INPUT	21
LOAD	22
OUTPUT	23
List Entry Commands	24
ENTRY.....	24
INDIRECT.....	25
LABEL	26
Error and Warning Messages.....	27
Error Messages	27
Warning Messages.....	28
Disassembly Pitfalls.....	30
Code Inline Data.....	30
Undetermined Branch Address	31
Addresses as Immediate Values.....	31
Code Paging.....	32
Laziness	33
Others	33
MC68HC11 Overview.....	34
Reassembling a Disassembly	38
Limitations in This Version	39
Bugs.....	40
Support	41
The Disassembler.....	41
Motorola	41
Third Party (Assemblers, etc)	41
Future Versions.....	43

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 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 hunks 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 M68HC11 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 (www.mot-sps.com) and can either be downloaded as .pdf files or ordered in printed form.

You will also need a method of obtaining the original binary that you wish to disassemble. Again, that is outside of the scope of this document. To obtain the binary, 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 as a binary format file. This version of M6811DIS only supports binary source files. Future versions will include support for other formats.

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

Installation

Installation of the M6811DIS program is simple and straightforward. Simply create a directory for the program and place all of the distributed files in that directory. The current method of file distribution is PKWare's PK-ZIP version 2.04G. If you are installing from that zip file, then you'll obviously need to have a copy of the unzip tool. If you don't already have a copy, it can be downloaded from www.pkware.com. Chances are if you are reading this document, you already have that tool, since this document is also zipped with the other files and placed into the M6811DIS.ZIP archive.

Once all of the files are placed into a single directory, you can simply change to that directory and run the program. Note that you can run the program from a path in your path-statement, but the M6811DIS.OP file that is used with this version must reside in the current directory at the time the program is run. This means if you use the system path to execute the program from a different directory, you must first copy the .OP file to the current directory prior to running the program. Future versions of this program will do away with the .OP file and eliminate the need for copying it all over the place.

The following files are distributed with this version (1.0):

- M6811DIS.EXE – The main program executable.
- M6811DIS.OP – The opcodes file used by the executable.
- M6811DIS.DOC – This document in MS-Word 97 Format.
- M6811DIS.TXT – This document in Plain-Text Format.
- M6811DIS.PS – This document in PostScript Format.
- M6811DIS.PDF – This document in PDF Format.
- AV94BNBH.CTL – A sample control file to get your started.
- 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).

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. 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 section entitled "Control Files" 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" 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" 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 DOS window (or run the machine in DOS mode), and enter "m681 ldis" 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 this version 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. Future versions will make better use of the ".ctl" extension. Once run, the disassembler will first display its findings from parsing the specified Control File, and then it will load the source file, resolve any specified indirects, and disassemble the source file 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. However, on this version of the disassembler, labels are limited to no more than six (6) characters. Future versions will support longer label sizes.

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 that 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 warning messages, is sent to “stdout”. This allows the output to be redirected into a log file for later reference using the stdout redirect operator (“>”) on the command line – refer to DOS 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 is 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 V1.0
Copyright(c)1996 by Donald Whisnant

Initializing...
Reading Control File...
Load Address: 0x4000
16 Entry Points:
    0x7C0B
    0x7C12
    0x7C1C
    0x7C22
    0x7C35
    0x7C6B
    0x7C7C
    0x7C83
    0x7C9C
    0x7CA0
    0x7CAA
    0x7CAE
    0x7CBE
    0x7CC2
    0x7CCC
    0x7CDD
Source File: AV94BNBH.BIN
Destination File: AV94BNBH.DIS
21 Labels Defined:
    0xFFD6=SCIVEC:
    0xFFD8=SPIVEC:
    0xFFDA=PAIEVE:
    0xFFDC=PAOVEC:
    0xFFDE=TOVFVE:
    0xFFE0=TI405V:
    0xFFE2=TO4VEC:
    0xFFE4=TO3VEC:
    0xFFE6=TO2VEC:
    0xFFE8=TO1VEC:
    0xFFEA=TI3VEC:
    0xFFEC=TI2VEC:
    0xFFEE=TI1VEC:
    0xFFE0=RTIVEC:
    0xFFE2=IRQVEC:
    0xFFE4=XIRQVE:
    0xFFE6=SWIVEC:
    0xFFE8=ILOPVE:
    0xFFEA=COPVEC:
    0xFFEC=CMONVE:
    0xFFEE=RSTVEC:
Writing program counter addresses to disassembly file.
Reading Opcodes File...308 opcodes read.
Reading Source File...File Size: 0xC000
Compiling Indirect Branch Table as specified in Control File...
[0xFFD6] -> 0xF494 = SCIRTN
[0xFFD8] -> 0xF8EE = SPIRTN
[0xFFDA] -> 0xF8E4 = PAIERT
[0xFFE0] -> 0x7922 = TI405R
[0xFFE2] -> 0x7986 = TO4RTN
[0xFFE4] -> 0x79EA = TO3RTN
[0xFFE6] -> 0xF8D9 = TO2RTN

```

```

[0xFFE8] -> 0xCC8A = TOIRTN
[0xFFF2] -> 0x7597 = IRQRTN
[0xFFF4] -> 0xF8B3 = XIRQRT
[0xFFF6] -> 0xF8AE = SWIRTN
[0xFFF8] -> 0xF8C4 = ILOPRT
[0xFFFA] -> 0xF8C9 = COPRTN
[0xFFFC] -> 0xF8CE = CMONRT
[0xFFFE] -> 0xF8D3 = RESET

```

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 LF4A4 LF4CB
LF5B5 L3024 L3025 L7972 L0046 L7954 L0273 L025D L7942
L7945 L0825 L7962 L79D6 L79B8 L79A6 L79A9 L0827 L79C6
L301A L3023 L0000 L0044 L7A05 LF705 L3022 L3FC8 L0226
LCC9C L0062 LCCC8 LCCBD LCCC5 L759E LF8BB LF8B1 LF8C7
LF8D6 LF8D1 L7200 LEF12 LEEFC L0055 L0130 LCD2C L0118
LCD41 L012E L007F LCD78 LC3F1 L0316 LC454 LC4C4 L001B
LCD4D L400F LAE6F LAF16 L082D L0192 L082F L009F L00A1
LEEFB L0082 LF273 L1806

```

*** Warning: Branch Ref: 0x1806 is outside of Loaded Source File.

```

L0F00 LE444 LE46C LE45C L0080 L00D3 LD491 LD8FE LD444
LD478 LC32A LEC80 LECCD LEAEE LEB2F LEB70 LE7EF LE82B
LEA28 LE8C6 LE8C7 LE8C8 LE8FC LE50E LE542 LC78A LC7E0
LC805 L006F L0016 L7C63 L0017 L7C6A L003B LE892 LE88E
L0165 LE882 LE88F LE8C5 LE8C1 L0166 LE8B5 LE8C2 L0043
LC706 L004E LC717 LC71A L00AE LC72A LC732 LC734 L00AF
LF0D3 L672E L0083 LC74B L672D LC754 L00A3 L5D28 LC75F
L5D29 LC767 L5D2A LC770 LC787 LD2E1 LD398 LD3A2 LD3C6
L0088 LC51D L009C LC530 L0087 LC529 LC543 LC53C L0096
LC54A LC55D L0183 L0085 LC55A LC569 L0182 LC575 L0122
L0089 LC595 LC58E L00D1 LC5A3 LC5B1 LC5E2 LE434 L0092
LC5D5 LC5C4 LC5D2 L0050 LC93B LC93E LCC89 L022C L0019
LCC72 L0006 L45AC LCC86 L01E3 L45AD L01E5 LCC44 L45AE
L022F L45B0 LCC66 L45AF L0052 LCC5D LCC78 LF279 LF97C
LC879 LAD62 LCC9E LF15E LCC60 LCC66 L5B03 L4E6B LCD0B
L4E6C LCD1D L022A L4E6A LCD18 L400B LCE55 L0253 L006E
LCE52 LCEB0 L02A5 LCE64 LCE6A L004F LA38D LA4CD L0812
L02A7 LD1E0 LD2D6 LCA56 L48B4 L01F3 L01F8 LCA59 L4E7F
LF0F6 LCAE8 LCB3A L00F8 L4E3B LDA32 LD9FE L4E37 L0240
L4E38 LDA03 L4E3D LDA06 LDA53 LDA60 L0071 LDAB5 LC30B
LE3ED LE4B9 LE046 LF4B1 LF4CC L302E L302F L0364 L0369
LF5E3 L0363 L0362 LF5FF LF5DB LF5E5 LF660 L0821 L7983
L795D L0823 L79E7 L79C1 L003A L7A17 L7A14 L0003 L7A20
L7A3B L0391 L0072 L0392 L9286 LF08B L082E L7A73 L7A70
L4E5F L01FC LF137 L4E5E L4139 L0047 L7A7D L4138 L01E6
L01E9 L3030 L7A95 L3031 L3032 L3033 L3034 L0148 LEEE2
L7CF6 L3FFA L0073 L7AD4 L3FC4 L080D L080C L0146 L3068
L7E04 L7B00 L7AFF L7AF7 L4953 L0051 L7B0F L7B11 L001A
L7B2D L7B7C LAD0C L303A LF72C LF72D L0366 LF749 LF081
LF753 L0001 L0390 LF771 L0230 L300E L3016 L75C3 L306F
L75DB L75E2 L76FD LCA8A L306E L723A L3FFC LF08A L725B
LEF9A L7282 L72C3 L00A0 L7548 LEF11 LEF04 L0093 LCD4A
LCE3C LC44A L0317 LC464 LC40D L0061 L4153 LC45F L4154
L4158 L4155 L0201 LC42E L4156 LC440 LF17B LC451 L4157
LC46B LC4D5 L0018 LC503 LAE8B L5151 LAE99 L5155 L0853
LAF0D L001D LE458 LE454 LE47C LE478 LE468 LD4D9 LD4A4
LD4CF LD90A LD906 LD908 LD8DC LD451 LD48C L019D LD4DA
L004C LC348 LC35C LC361 LECCC LCC88 L0169 L0084 LECAF
L0090 L001C L0026 LED09 LED05 L016A LECF9 L008F L0027
LEB2E LEB2A L0175 L0098 LEB04 LEB1E L002B LEB6F LEB6B
L0176 LEB45 LEB5F L002C LEBB0 LEBAC L0177 LEB86 LEBA0
L002D LE82A LE822 L0162 LE814 L0091 LE85F LE85B L0164
LE84F LEA7A LEA76 L5B95 LEA4B LEA54 LE8FB LE8F7 L0167
LE8EB LE8F8 LE92F LE92B L0168 LE91F LE92C LE541 LE53D
L0159 LE531 LE53E LE575 LE571 L015A LE565 LE572 LC79B
L00A2 LC7AE LC7B1 LC7ED LC7F5 LC7F7 L02A6 L0208 L0266
LC829 L003F LC83E L48D9 L48DE L0295 L01EE L01DE LC862
LC868 LF136 L0842 LC922 LF0E0 LF0E4 L5D2B L00A5 LD2EB
LD2EE LD3B0 LD3C9 L00DC LC5EC L0127 L010A LC622 LC606
L0086 LC603 LC6CA LE440 LC949 LCA42 L45B2 L45B3 L022D
L45B1 L1800 LF284 L180C

```

*** Warning: Branch Ref: 0x180C is outside of Loaded Source File.

```

L9F83 LA370 LC8BF L0252 LC88C LC893 LAD6C LAE64 LF172
LF178 LCF37 LCE70 L9DE0 L087B LA3B0 L087C LA39F L4F27
LA3A5 LA3AD L4F26 L086B LA3D0 LA3BC LA3CD L400C LD21A
L402C LD1FE L402D L01F9 LD2CB L03B2 LCA61 LCA7C LCA80
LF100 LF103 LCAFE LCB2F LCB34 LCB3C L0855 LDA45 L4E3E
LDA2A L4E3F L0241 L4E40 LDA2F L024E LDA70 LDAC6 L003C
L0239 LDAC2 LDF48 LE6B4 LE74D LEC7F LED0A LED63 LE9DC
LEAAE LEDD5 LEDD6 LEE44 LE410 LE3F3 LE088 LE085 L4E7E
LE06C L4E7D LF4FC L0042 LF4F4 LF4E9 L039B L0367 LF590
LF60F LF65A LF5EE LF5F5 LF6F7 L000A L9292 L929B LF09A
L3060 L004B L3062 L3064 L004D L3067 LF13D LF15A LEEF6
L7D08 L014A L0149 L7D8B L7E0E L7E17 L7E12 L7B51 L924A
LA53C LD075 LD11D LF285 L7B49 LB773 L7BA9 L7BB7 L7BE6
L7BAC LAD59 L3FC0 L005F L020B L4142 LAD2C LF1ED LAD44
LAD4B L3066 LF774 LF763 LF87E L0068 L006A L75F5 L02F0
L75FF L02EC L7743 L028B L770B L02F1 L7720 L081E L495A
L028A L7740 L7786 L7767 L0053 L77C9 L0277 L7762 L77AF
L3061 L3063 L3065 LEFA8 LEFC2 L0005 L303F L72D2 L72EC
LCD58 LCD60 L5E8E LCD7B L5E8B LCD75 L5E8A LC47A L01EF
L0207 L0206 L01F0 L01F1 LC46E LF184 LF189 LF1BA LF1BA
LC4DF LC4EC L00A6 LAAE2 LAF02 L0854 LE49F LE486 LD4E3
L0180 LD4EB LD4F1 LD4F7 LD50F LD505 LD50D LD4B1 LD91B
LD92C L0099 LD927 LD932 LD8FA LD8FC LD46C LD854 LD8B7
LD47B L008A LC364 LC37B LC383 LCC99 LED06 LEB2B LEB6C
LEBAD LE826 LE85C L0171 LEA6A LEA77 LC7C6 L5B1A L5B18

```

LC7C2	L5B1B	LC7CA	L4074	LD2FB	L007E	LD313	LC62E	L0124
LC6BD	LC614	LC611	LC6C2	L4022	LC955	LC963	L4012	L0830
L4013	L0831	LC994	L997F	L0009	L9FAD	L4EB6	L4EB7	L02C6
L02D0	L0887	L9FCD	L0888	L50B5	L50B4	L9FCA	L0038	L4E85
L9FD6	LA01C	L0036	LA37A	LA382	L086A	L021D	L45D0	LC8E0
L003D	L029B	L021F	L0841	L0843	LC90B	LC91F	LC8A4	LC8AD
LAD76	LAD8B	L46FF	LCF3E	LCF48	LCE78	LCE82	L9DE9	L9DEB
L0882	L9E16	L5064	L9E0F	L9E01	L0884	L5065	L9E13	L9E51
L085C	LA400	LA3DC	LA3FD	L086C	LA3C6	L4EAF	L01FB	L402F
LD22B	L01FA	L402B	L0041	LD213	LD22E	LF10D	L4E82	LCB18
L029C	L4E83	LCB37	LCB4B	LCB66	LCB4E	L003E	LDA82	LDAAA
LDA9E	L024B	L023C	L023D	LDEDA	LDAD4	LE6E5	LE6DF	LE6E8
L008E	L015F	LE6E2	L0023	LE748	LE781	LE77B	LE784	L0160
LE77E	L0024	LE7EA	LED62	LED5E	L016B	LED32	L0028	LEDD3
LEDCF	L016C	LED81	LEA27	LEA23	L016F	LEA12	LE9F4	L002A
LEAED	LEAE5	L0173	LEAD3	LEAE9	LEDFE	LEDFC	L0179	LEE0A
L001E	LEE3F	L002F	LEE6C	LEE6A	L017A	LEE6E	L0030	LEE5A
LE0AD	L0255	LE0A2	LE0AB	L0259	L0279	LE0CB	L024D	L4E7B
LE082	L4E7C	LF513	LF510	LF5B4	LF663	LF62E	LF655	LF704
L302D	L92A0	L4EAC	L92E3	L014E	L7D17	L0147	L7D22	L7D2D
L7D35	L7D36	L014D	L7D45	L7D51	L7D6B	L7D79	L7DC6	L00FE
L6925	L7DC4	L6927	L6926	L7DBF	L00F5	L7E2D	L5D04	L7E23
L7E2E	LCC05	L7B6A	L0393	L7B6D	L9268	L9262	LA548	LA552
L01EC	LA562	LA56C	L400D	LD083	L01DC	LD11C	L0070	LD15A
LD12E	L01D1	LD1D5	LF2EA	LF2E7	LF2CF	LF2BF	LCF65	LAF40
LB77C	LB78B	LAD5B	LF7E3	L3FCC	LF880	L3FEA	L3FCE	L3PDC
L3FF6	LF7B7	LF7CE	LF7D0	L7873	L7626	L02F2	L78D4	L026F
L7629	L76D7	L7812	L0829	L7789	L4920	L77BF	L77D5	L77F9
L084E	L7801	L78A4	L081F	L77BC	L082A	LEFB2	LEF47	L4008
L4006	L72E5	L730C	L000F	LEFC3	L0015	L7308	L733C	L5B24
L0319	LEF1D	L7346	L735A	L7351	L4133	L0219	L001F	L736D
L5B1E	L0020	L7379	L5B21	L0021	L7385	L5B42	L0022	L7391
L5B52	L739D	L5B58	L73A9	L5B5E	L0025	L73B5	L5B78	L73C1
L5B72	L73CD	L5B74	L73D9	L5B79	L0029	L73E5	L5B8B	L73F1
L5B8F	L73FD	L5B9D	L7409	L5B9F	L7415	L5BA1	L002E	L7421
L5BA2	L742D	L5BA8	L7439	L5BAE	L0031	L7445	L5BB8	L7457
L7467	L0032	L7472	L0033	L747A	L0034	L7482	L7499	L01EA
L4950	L74CA	L4951	L9341	L01F5	LB24D	LB185	L01FF	L01FD
L01FE	L489B	L01D8	L0193	L755E	L4071	L750C	L4072	L0078
L3FCA	L0228	L3012	L0851	L48DD	L026D	L026B	L0049	LEF2B
L4D8C	LCD81	LCD98	LF1CA	LF1D0	LC4F0	L5150	LAF07	LAEB7
L5152	LAEBA	LAF10	LD517	LD519	LD549	LD527	LD547	LD4C3
LD7FB	LD869	LD879	LD886	LD895	LD8B5	LD8C3	LD8CE	LD984
LD934	LC3A5	L691B	LC39F	L306A	LC3B9	L012F	L306C	LC7D6
LC7DC	L4075	LD309	L4076	L007C	L007B	LD333	LD338	L007D
LD34D	LD352	LC635	LC64A	LC6D0	LC96D	LC9C0	L4019	LC9AD
L0833	L9986	L9AA7	L0037	L9FDD	LA05F	LA0B4	LC916	L0844
L4140	LC8BB	LAD8E	LAE03	LAE08	LCF49	LCF53	L1815	

*** Warning: Branch Ref: 0x1815 is outside of Loaded Source File.

LCF64	L0395	L0396	LCEA0	LCE93	LCE9D	L9E20	L9E70	L5090
L9E6D	L5091	L5092	LA432	LA421	L02CC	L4FBD	L02CA	L085E
LA3EC	L027A	L026A	LD28D	L4024	LD288	L4025	LD266	LD2A8
L4E80	L4E81	LCB75	LCB7F	L3FDA	LCB92	LCB95	LDA93	LDA74
LDA2A	LDA77	LDEE1	LDFAB	LDB00	L00F9	L0014	L5B12	LDAF2
LDB72	LDB18	L4E28	L0158	LDB29	L4E29	LE74C	LE6FE	L017B
LE749	LE7EE	LE7AC	LE796	L017E	LE7A7	L017D	LE7EB	L00E5
LED5F	L00AC	LEDAA	LEA24	LEE43	L0190	LEE40	LEEA9	LEEA6
LF153	L0265	LE0EE	L025F	L0260	LE14F	LE136	L02EB	L02EE
L02EF	L491F	L02ED	L0222	LE237	LF58A	LF57E	LF52B	LF536
LF6BA	LF680	L036E	LF69B	LF695	LF6CB	LF646	L0007	L0008
L92AF	L7D76	L7D88	L7DDA	L7E03	L6929	L6928	L7E01	L02A1
L7E73	L7E90	L4009	L3039	L7E71	LCC18	LCC15	L7B79	L94B3
LC307	L927C	L9285	L9282	LA581	LA586	L01F2	LA5B7	L4E5A
L024C	L4E5C	LA5BC	L4E5B	LA5B2	L4E5D	LA5C2	LD00A	LD09D
L01D7	LD0CA	L489D	L489C	LD0C7	LD0D5	L489A	LD0E2	L01DA
LD100	L01DD	LD1CE	LD139	LD141	L006D	LD150	LD160	LD1CD
L01D2	LCF70	LCFAF	LCF80	LCF86	LCF83	LCFAB	LAF5C	LAF62
LB106	LB12A	LB291	LB2DD	LB375	LF42B	LF2EB	LB78C	LBCEB
LBDF3	LB4D4	LB8E8	LF842	L3FD4	L3FD6	L3FD8	LF7F4	L76DD
L78E4	L78F8	L7634	L862B	L76E5	L4D8E	L76FA	L4D8D	L0067
L450C	L783B	L48D0	LF116	L783D	L77A5	L7907	L77E8	L7874
L78B3	L78CB	L78D3	LEF5F	LEF56	LEF62	LEFC9	LEF22	L4EE8
L0885	L9355	L4EA6	L9387	L935E	L93D4	L9361	LB26F	LB1B5
LB199	L015B	LB1A6	LB1B7	L015C	LB1B2	L019B	LB219	L756A
L756C	L48F3	L48F4	LEF3A	LEF40	LEF34	LCD8C	LCD95	LCDA5
L5E8C	LCE29	LF1DF	LF1E6	L5D05	L5D06	L5154	L5153	LAEC7
LAEEC	LAEE3	L0063	LAFOA	LD556	L5B2C	LD567	LD564	L5D2D
LD81B	LD82A	LD852	LD98D	LD99F	LD9AF	LD9B8	LD94C	LD94F
LD94A	L007A	LD380	L406D	LD37B	L084C	LD371	L4073	LD395
LD386	L0184	LC65E	LC66D	L6753	LC668	L6754	LC677	LC68F
L0126	LC6A1	L4011	LC977	L4010	LC988	L0834	LC9C8	LC9C3
L4016	LC9BA	L9992	L99A2	L99AC	L9FE3	L9FEC	L088D	LA07D
LA06C	LA075	L02C8	L02D2	LA0BF	LA1AE	L3FE0	L0847	L0849
LADF3	L0845	LADB2	L084B	LAE0D	LAD4E	L5119	LAE00	L511A
L0846	LCEAA	LCEB4	L9E2A	L5066	L9E3A	L0039	L9EB2	L9E8D
L9ECF	LA46E	L5B02	LA44A	L02CF	L4FBE	L02CD	L085D	LA3F6
L4EB1	L4028	LD2A3	L0232	L4026	LD27E	L4027	L402A	LD285
LCBB2	LCBD9	LDA8D	LDEEB	LDFAC	LDF9F	L023F	LDB7A	LDB2D
L4E2A	L4E2C	L4E2B	L4E2E	LDB77	L4E2F	L4E2D	LDB6D	LE71A
LE72E	LE715	L017C	LE7C3	LEDD0	LF162	L0066	LE15D	L494A
LE164	L0858	L4FB4	LE172	LE178	LE1P4	LE188	LE1FC	LE250
LE241	LE24D	LF59E	L0368	LF530	LF562	LF54C	LF55E	LF559
LF6C0	LF6B0	LF6AB	L92B7	L92C2	L92C9	L7E8A	L0246	L02A0
L7EA7	L7ECA	L0065	L7EB5	L413B	L01E8	L7ED2	L7ECD	LB3B5
L94BA	L1803							

*** Warning: Branch Ref: 0x1803 is outside of Loaded Source File.

L94C4	LA4DD	L94DC	L94D3	L997E	LA613	L0283	L020D	LA5E0
L0225	LA616	LA5F1	LA657	LA655	L414D	L0215	LA60E	LD119
L01DF	L01E1	LD172	L084F	LD193	LD191	LD19A	LD02A	LCFF3
LCFDE	LCFCD	L4850	L4851	LCFE6	LCF94	LCF9F	LD048	LB35E
L00F3	L0076	L3000	LAF6F	L3002	LAF9B	LAF96	LAFB5	L009A

LB121	LB124	LE677	LEA7B	LE4BA	LB13B	LB14E	LB15D	L00B1
L00B7	L00BC	L00BE	L00C0	LB2C9	LF0BD	LB2DA	L00E7	LB2FF
LB318	L5D16	LF23F	LB2FA	LB384	L014C	LB38C	LB38F	LF464
LF43D	L0045	LF47D	L036A	L51B1	LF493	L036C	L012C	L00F7
L00B8	L008C	L012D	LF36A	LF39F	LF35E	LF339	LF343	LF35A
L0103	LF36C	L03A1	LF376	L0111	LB794	LBACB	LBCE5	LBDA4
L0119	LBDFC	L011B	LBEO3	L674C	LBED1	L011F	L6750	LBEL1A
L6748	LBEO2C	LBEO4E	L0121	L6752	LBEO43	LBEO49	LBEO59	L00AD
LBEO68	LBEO71	LBEO77	LBEO9E	LBEOA4	LBEOA1	LC065	LF6D2	L4D8F
L7906	L763E	L76A4	L7654	L7678	L864F	L4967	L3FF2	L7851
L7865	L7870	LF121	LF125	L7921	L7919	L4E65	L7883	L789B
L78A3	L4EAB	L4EA8	L93A0	L4EA9	L93A5	L93A7	L94A8	L937F
L50C4	L9377	L50BD	L937C	L93F2	L508E	L93E4	L50B9	L93EF
L9410	L508F	L9402	L50BA	L940D	L4E86	L9423	L02C2	L02C0
L9494	L943D	L9442	L9447	LB213	LB1CD	L00F1	LB1D8	L0094
LB201	LB223	L00AA	L5B28	L0101	LB230	L5D1C	LB23E	L5D1B
LB24C	L7583	LCDB2	L5E8D	LCE39	LD57D	LD589	LD959	LD967
LD390	L406E	L406F	L4070	LC6AB	LC6B7	L4020	LC982	L401B
LC9D2	L401A	LC95B	L401C	LC9E0	L401D	LCA0A	L401E	LC9F9
L401F	L4014	L0836	L4F7A	L99C6	L4F7B	L4F7C	L99C3	L0859
L4F73	L4F79	L99D5	L99ED	L4F7D	L4F7F	L99F2	L085B	L9A20
L02D4	LA031	LA000	L50C2	LA0B1	LA09C	L02D5	L50C0	LA089
L0875	LA0C7	LA13F	LA1B4	LA203	LAE21	LAE23	LAE31	LAE38
LAE42	LAE58	LCCEC3	L46F4	LCCECB	LCED5	L9E33	L507E	L9E46
L507F	L9E4A	L9ECC	L9EA8	L5096	L9EA5	L9EBE	L9EDB	L0883
L9EB9	L9F78	LA4BC	LA4B6	L50B6	L088E	L50B7	LA4B1	LA457
L50CC	LA465	LA460	LA46B	L4029	LCBC7	L50D2	LCBD6	LCBF0
LCBED	LCC04	LCC01	LDEF2	LE018	L023B	LDFCF	LE033	LDFC3
LDFD1	LE039	LDBAE	L4E30	L0817	L4E35	LDBC4	L4E33	L4E34
L4E31	L4E32	LDBB3	LDBC7	LE727	LE733	LE7D1	LE7D9	LE1A6
L494F	LE19D	L494D	L0293	LE1F9	LE1C2	LE1CF	L491E	LE1EF
L494E	LE20C	LE2D6	LE253	L0264	LF586	LF573	L92D9	L92DD
L92E1	L413A	LB270	L7EE6	L7EEC	LB3BE	LB3C7	LA53B	LA4F0
LA4D0	L94ED	L94EA	L997C	LA671	LA620	L4136	LA66C	L0234
L4144	LA66D	LA6BB	LD1B0	LD1C4	LD1C2	L01D4	L47DA	LD038
LD047	LD059	LD063	L01D5	LD00E	L01D3	LD020	LD02B	L4853
LCFF6	LCFA7	L5B43	LB374	LB370	LAFAB	LAFAG	LAFCS	LAFCE
LAFD1	LE6B3	LE6AF	L015E	LE6A3	LEAAD	LEAA9	L0172	LEA9D
L5B35	LE4CB	LB15F	LB15B	L5D0D	LE576	LE5B7	LE38F	LE96F
LB175	LE5F6	LEDD4	LE930	LE3CB	LE4A4	LE3B1	LF0CA	L015D
L016E	LB316	L00C6	L00CD	LB312	L00CF	L00CB	L5D18	L0102
LB33A	LB344	L0814	L51B3	LF3BF	LF3AC	LF3E3	LF3E0	LF399
L031B	L031D	L031F	L0321	L0323	L0325	L03A3	LB7A2	LB7A8
LB7B9	LB7E8	LB7E0	LB7D0	LBACE	LB7F4	LB802	LB882	LBCEA
LBD00	L0143	LBD11	LBD29	LBD1B	LBD21	L0129	LBEE2	L012A
LBEB5	LBEB6C	L0123	LBEB7A	LBEB7A	LBD7A	LBD86	LBD8D	LC0B7
LBEC9	L011D	LBEC6	LBEC0	LC06A	LBFB9	LC083	L6921	L012B
LC08E	LC0A8	L76BD	L76D3	L4956	L7669	L7675	L768A	L76A0
L4EAA	L4EA7	L93C0	L93C5	L93C7	L94B2	L9323	L92F1	L94A7
L02C5	L02C3	L02C4	L9457	L945C	L945E	L5B22	LB221	L5B2A
LB205	L5B29	LB1FB	L0199	L019A	L0197	L0198	LCDCB	LCDC5
LCDD1	LCDD6	LCE07	LCDFE	LCE04	L0131	LD5C2	LD59A	LD5B8
L5D26	L5D27	L5D25	LD5C0	L5D24	L0081	LD5D6	L018D	LE322
LD5F4	LD5EC	LD602	L4017	LCA14	L4018	LCA1D	L0832	LCA2B
L4F7E	L9A05	L9A00	L9A31	L4EB2	L085F	L9A3C	L9A8B	L4F84
L9A50	L4F85	L9A56	L9A7F	LA03A	LA053	L50C1	L50C0	L50C3
L50C5	LA01F	L9309	LA0A1	L50C6	L4EB3	LA0D5	LA0EC	L4EB4
LA0FD	LA114	L4EB5	LA135	LA176	LA1AC	L4EB6	LA15B	LA170
L4EE7	LA1A9	L0899	LA1A4	L4FB5	L4FB6	LA1EB	LA1E3	L0876
LA22A	L0877	LA226	L0878	LA21B	L4F18	L4F17	LA227	LA238
LA23B	LA35E	LCEF5	LCF0B	LCF15	LCF22	L0886	L508D	L9E3B
L9EFE	L5094	L9EFA	L5095	L9F13	L9F18	L9F31	L0880	L4EAD
LDEFA	LDF20	LDF05	LDF1B	LDF87	L0237	LE042	LE045	LDFDC
LDBFA	L5B01	LDBE2	L081C	L4E36	L0223	LE21B	LE321	LE264
LE2BC	LB27D	L5B26	LB28A	LB28D	L5B23	L48B3	L7EFD	L029E
L48B1	L7F10	L48B2	L7F23	L91B6	L7F89	LAF17	L8695	L7F52
L7F7D	L7F80	L860F	LB3C8	LB3FB	LB486	L0397	LA521	LA516
L0398	L94F4	L961A	L4958	LA67F	LA683	LA627	L4134	LA636
L0233	LA6D4	L083D	LA6ED	LA6F4	LAFB2	L0097	LAFDF	LAFED
L0095	LB001	LB00B	LB01C	L5D23	L019C	LB026	LB02E	LB037
LB054	L009B	LB048	L009D	L0077	LB064	LB06E	LB07D	LB096
LB099	LE6B0	LEAAA	LE5B6	LE5B2	LE5A5	LE5F5	LE5F1	LE5E5
L5B34	L5B2B	LE3A9	LE3AD	LE9DB	LE9D7	LE984	LE98E	L5B87
LE992	LE676	LE672	LE61E	LE96E	LE96A	L016D	LE954	LE96B
LE3E5	L5B3A	L5B31	LE3E9	LE4B5	L5B2F	LE3C3	LE3C7	L0100
LF3F2	LF3FE	LF401	LEPDD	LBADB	LBAB7	L6001	LB870	L6004
LB82B	LB822	LBC0C	LB88E	LBC0C	LB886	LB88E	LB8A6	LF141
LB8D5	LB8D9	LB8E8	LB8E9	LBD3C	LBD3E	LBD2F	LBEE7	LBEE8
LBEB8	LBD99	LBDAL	LBD87	LBD84	LBD2F	LBDCC	LBEE5	LBEBF
L691F	LBEBF6	LBFB0B	LBFB4	LBFB5	L691D	LC0A3	L691E	LC0AB
LC0B4	L50CA	L92FE	L9305	L946E	L9473	L9478	L5E8F	LCE20
LCE26	LE32A	LE33B	LD604	LD5FF	L4015	LCA2F	LCA32	L9A15
L4F80	L9A1F	L4F81	L0857	L9AA1	L9AA4	L9333	L9A6B	L0860
L9A66	L0861	L9A7B	L4F87	L9A82	L4F86	L50C7	LA029	L50C8
L931F	L50CB	L50C9	L4EE4	LA18B	LA1A0	L4EE5	L0893	LA1FA
LA249	LA250	L0866	LA367	L0867	L9F2E	L9F48	L5097	L9F45
L9F60	L9F58	L507D	L9F7B	LDF24	LDF40	LDF44	L4E26	LDF5B
LDF84	LDF76	LDF98	LDF9B	LDFE8	LE01B	LDC16	LDC10	L02AB
L4E41	LDC24	LDC27	LDD14	L4E39	LDBEF	LE224	L48C8	LE272
LE275	LE2DE	L02BC	L91BF	L02BD	L91D5	L91D8	L02B8	L02BA
L920E	L91FD	L920A	L9221	L7F95	L85D8	LAF3C	LAF3F	L5181
L5183	L4D8B	L86C4	L866E	L86BA	L86DA	L86F5	L86F2	L861B
L8720	LD3CA	L862A	L5D1D	LB3F4	L5D1F	L5D21	LB3FA	L00FA
L5D0C	L01C9	L01CA	L01CD	L01CB	L01CE	L5D0E	LB426	LB424
LB40D	L00D5	LB48D	LB4DD	LB4E8	LB4A5	LB4AC	LB4B6	L0181
L00D4	L00D9	L00DA	LB4D3	L018B	LB552	L00D6	LB504	LB549
LA532	L4EAE	L9524	L9532	L9551	L4EB0	L086D	L0879	L087A
L0864	L086E	L95DA	L95D5	L95EB	L962A	L9654	L9632	LA694
LA6B0	LA6AD	L4143	LA644	L02B1	L50DA	LA721	L02AF	L02B0
LA72F	LA731	LA742	LA74D	L0837	LA75B	LA76B	LA77A	L0209
LA77D	LA78E	LA79F	LA7AF	L5D02	LB0B2	LB0D0	LB0D2	L5B1D

LE5B3	LE5F2	L5B85	LE9AF	L5B45	L5B46	L5B47	L5B48	LE66A
LF411	LF414	LF01D	LEFD7	L008D	LBB44	LBAFD	LBB02	L0145
LBB41	LBC14	LBC1C	LBCE8	LBBCA	LBBFD	LBBD6	LBBA1	L5FC9
L018C	LF148	LF14E	LF155	LB8F8	LB8FC	LB8FD	LBE8E	L674A
LBD5D	L674E	LBDEA	LBFD1	L6920	LBF14	LBFAE	L6924	LBFE6
LBFD2	LBFC6	LBFDA	L02CE	L9488	L948D	L948F	LCA3F	L4F82
L4F83	L0868	L3008	L933F	L9340	L9A73	LA273	L4FBC	LA291
LA269	LA2B1	L9F6D	LDF77	LE036	LDC35	LDD0D	LDD8A	LDD7E
L4E4F	LDD41	L024A	L4E55	LDD3B	LDD7B	L4E3A	LE2AA	L48C2
LE293	L48C5	LE2CB	LE2A4	LE31E	L48CC	L48C9	L400E	LE308
L48CB	L48CA	L48CD	L9218	L9205	L922E	L9236	L9244	L7FFA
L01ED	L48B6	L7FAE	L48B5	L48B7	L7FBB	L7FFD	L48B8	L7FD4
L7FE6	L7FF4	L8001	L85EB	L85FA	L8681	L8689	L8694	L86F6
L8627	L8A01	L8745	L872E	L8734	L873F	L493F	L8756	L8758
L876E	L4940	L0290	L4942	L028F	L8789	LD3DD	L0811	LD3D8
LD410	LB43D	LB458	L01D0	LB472	LB47A	LB524	LB51E	LB52D
L95E3	L95E6	L0862	L0871	L087D	L4E87	L9614	L960E	L9611
L9920	L9684	L4F19	L4F1B	L4F1A	L4F1F	L9674	L4F20	L4F21
L4FC1	L4F1E	L4F1D	L4F22	L4F23	L9690	L9696	LA6B7	LA7AA
L083F	LA7B9	L020F	LA7D6	L01E4	LA7EC	LA7FD	L021B	LA810
LA81A	LA830	L4141	L413D	L021C	LA849	L021E	L022E	L413F
LA8A3	LA88D	LA882	L5D13	L5D12	LE3CA	LB0FB	LB0F8	L5B89
L5B8C	LE9D8	LE673	LF42A	LF421	LF024	L0144	LB1D1	LB2D3
LB36E	LB333	LBC5A	LBC33	L0139	L0138	L013A	L5FCF	L5FD0
L018E	L018F	LBBE9	LBBF2	LBBFB	LBC0B	LBBB8	L5FCA	L5FCB
L5FCC	LBBD0	LB905	LB917	LD96D	LB927	LB929	L013E	LB942
L00E2	L00E9	L5FFF	LB94B	L5FFE	LB988	L6923	LBF7E	LBF35
LC0B8	L0110	LBF53	LBFAE	L0185	L0125	LBF62	LBF6A	L00EB
LBF71	L00ED	LC046	LBFF3	LC01C	LC00A	L6747	LC025	L6922
LBFE0	L9A7E	L4FBB	LA287	LA2A3	LA29E	LA2CF	LA2C3	LA2BE
LE010	L023A	LDC3C	LDCFD	LDDC1	L4E57	LDDA0	L4E58	LDDC6
L4E56	L4E59	LDDCB	LDD59	L0248	L4E53	LDD56	LDD44	L48C6
LE2D9	L8010	L48C1	LE29F	L48C4	L48DF	L0261	L804F	L0200
L8014	L801D	L48B9	L8022	L48BB	L803D	L48BA	L804A	L8053
L4969	L8609	L860C	L8708	L871F	L8F97	L8A1B	LA8A5	L48C7
L8A35	L8A28	L8A32	L8A3B	L8A52	L8A47	L0289	L0287	L8A7B
L8790	L4E4E	L028E	L493D	L87A2	L87BB	L87AF	L87B7	L01F7
L87BD	LD405	L080F	L4132	LD402	LB543	LB53B	LB553	LB5FE
LEBB1	L95EE	L0889	L9938	L9963	L994F	L50BE	L96C5	L970F
L4F05	L4F06	L96B4	L4F07	L96BC	L4F09	L4F08	L96FF	L0217
L413C	L413E	L0154	L0220	LA8F1	L083C	LA90C	LA914	LA895
L4854	LE364	LE35F	LE38E	LBCCB	LBC5C	LBCB5	L5FCD	L5FCE
LD97B	LD982	LB954	LB976	L0141	LB9AB	L5FE1	LB9A0	LBA76
LC054	LC10A	L0188	LC0CC	L0186	LC0F8	LC107	LC118	LC05C
LC05F	LC02B	LC034	LC03F	LC017	LA2AC	LA2E0	LA2EB	LA2CC
LDC39	L0242	LDC52	LDC4B	LDC4F	L4E42	L0243	L0244	LDE01
L4E66	LDDFB	L024F	L4E69	LDDFB	L4E52	LDD86	L4E50	L0247
L4E51	LDD74	LDD87	L48C3	L0262	L8069	L8075	L808B	L48CE
L8098	L80B8	L8712	L8FA6	L8FAC	L8A6B	L48E2	L8A71	L48CF
L8AEE	L8AFC	L48D8	L8AE9	L48D4	L8AC9	L8AB0	L8AC1	L0294
L48D5	L8AD9	L8B05	L87CA	L0298	L87D0	LB562	LB574	LB57C
LB586	LB588	LB591	LB597	LB59F	LB5A1	LB5AB	LB5AD	LB5B3
LB5BF	LE414	LB5CE	LB5DB	LB5E3	LB5EB	LB5F1	LB5FD	L0114
LB64D	LB631	LB64C	LEC13	LEBD6	L088B	L50BF	L9664	L96F8
L971A	L9781	L415D	LA91D	LA977	LE37B	LBCD8	LBC84	LBC6E
LBC7E	LBC8E	LBC89	L013F	LB987	L5FE2	LB9BA	LB9BF	LB9CA
LB9C3	L5FE3	LB9F7	L013D	LBA91	LBA8E	L601E	LBA9F	LBA5A
LBAAC	LBABF	LE424	L0112	LBAC8	LA2FF	LA31F	LA313	LA30D
LA30E	L4E47	LDC8E	L4E43	L4E44	L4E45	L4E46	LDC7A	L4E4A
L4E4D	LDD03	LDC90	L4E6E	LDE39	LDE1E	L4E70	LDE41	LDE44
LDE30	LDD83	L80A0	L4905	L80B5	L0254	L4D8A	L025B	L0257
L80ED	L80F0	L4D88	L8112	L0271	L0275	L8138	L813B	L81D1
L817F	L8156	L514B	L8153	L084D	L514C	L514F	L514D	L8170
L514E	L8179	L0075	L50CE	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	L48D6	L8AF9	L8B3F	L8B13	L4952	L48DA	L48DB
L8B39	L082C	L48DC	L8B31	L48E0	L8AD3	L48EE	L8AE4	L8B02
L029F	L87D3	L87E9	L87E6	L885C	LE420	LB665	L00D7	L5FC5
LB67D	LB691	LB69B	LB74E	LEC4E	LEBE1	LEBE5	LECE15	L0132
LEC4F	LEC4B	L4F12	L4F13	L96F0	L4F14	L4F16	L4F15	L973C
L9730	L9756	L088F	L97A5	L932A	L97A2	L4F24	L4F25	L97AF
L97B3	L4159	L415A	LA932	L415B	LA939	LA9CC	LBCAE	LBC9A
LB99D	LBA21	LBA7C	LB9E5	L5FDD	L5FE4	L013C	LBA03	LBA05
LBA18	L0142	LBA2E	LBA3D	LBA5D	LBA4D	L5FDE	LBA54	LE430
LA332	LA32D	LA319	LA340	LDCBB	LDC9C	L48ED	L4E48	LDCAC
LDCB6	L4E6F	LDE28	LDE3C	LDE4E	L4E73	LDE96	LDE6F	L4E72
LDE6C	L4E74	LDE9E	LDE74	L81DB	L818A	L81E9	L8198	L81A0
L81A2	L81C5	L495C	L028D	L495B	L81EC	L8300	L9039	L9033
L9042	L90F3	L9004	L9007	L50D1	L901E	L4909	L8B49	L4E71
L8B4D	L4955	L8B6B	L4957	L8B64	L48E1	L8AFF	L87F1	L885F
L8859	L880D	L8873	L0282	L029D	L89F4	L601D	LB6AC	LB6CE
LEC30	L0133	LEC7E	L974C	L0895	L0891	L977E	L977B	L9775
L9331	L9332	L97BC	L0869	L97C6	L97CA	L415C	LA954	L415F
L46E8	LAA40	L46E9	LA9EC	L0838	LBC9C	L5FDF	L5FE5	L0140
LBA71	L5FE0	LBA63	LA33B	LA352	LA34D	L0245	LDC55	L4E4C
LDC55	L4E4B	LDCD3	LDEA1	L4E76	L4E78	LDE89	L4E79	LDEB6
LDEB3	L0856	L4E7A	LDED7	L087F	L81FE	L48BC	L8218	L8207
L8213	L4938	L0268	L0267	L4939	L8315	L81DF	L83D0	L905A
L9064	L9067	L02AD	L50D3	L9082	L9085	L02B3	L02B5	L90AF
L50D4	L90A2	L90A5	L90DB	L90C2	L50D8	L90C8	L9176	L9120
L9144	L9105	L916E	L8B75	L8D50	L8818	L881C	L8825	L8889
L0291	L8883	L88A1	L89FB	L0292	LB6BA	L00DB	L00D8	LB6FA
LB6E9	LB6F8	L0134	L0897	L97D3	L97E1	L97ED	L4F76	L97FA
L4F77	L9807	L4F78	L9819	L4F75	L9820	L9829	LA95D	LA967
LAA69	L4151	LAA5C	L022B	LAAA3	L083A	LAA18	LAA23	L083B
LAA35	LAA3D	LA35B	LDCBE	LDC60	L4E49	L8222	L822F	L8247
L48BD	L821A	L83EE	L48D1	L48D2	L48D3	L83F4	L006C	L8425
L90E6	L90F0	L917A	L90D5	L90D8	L919A	L918D	L912A	L9148
L50DC	L50DB	L913B	L50DD	L9177	L50D9	L4917	L888D	L4918
L0286	L8B9C	L026E	L8D71	L081A	L8D68	L0818	L48EC	L8DAE

```

L8D96 L8DA0 L8DA6 L8E1D L8E56 L8834 L494B L889B L494C
L02A3 L88F6 L4943 L88C9 L88DB L0285 L88E6 L8893 LB6C1
LB6C7 LB701 LB70B L4F74 L9AA8 L9D03 L9D89 L9840 L984E
L9855 LA973 LA97D LA980 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 L8236 L48C0 L081D
L8258 L8255 L8402 L4510 L840E L8416 L842E L8449 L844F
L8473 L847F L91B5 L91B2 L50CF L9197 L9153 L9161 L9164
L02AA L9173 L4919 L0203 L0204 L0205 L8BCA L490D L8BD2
L6CDF L8E1F L8DCE L48EA L8DCA L8DD9 L8E04 L48EB L8E00
L8DFE L8E1A L8E18 L8E5C L48F2 L8E75 L8E70 L48F0 L8E73
L4901 L4902 L4903 L4904 L8908 L891F L8932 L894C L02A4
L8962 L89B0 L493E L8970 L88F2 LB713 LB72A LB733 L0113
L9AB2 L9ABA L9AD2 L9ADD L9D13 L9D53 L9D95 L9D9F L9863
L0870 L986A LA995 LA9A2 LA9A8 L0211 L4145 LAA9F LAB47
LAB24 L0231 LAB17 L4164 L4161 L493A L8262 L8330 L450F
L8422 L847B L848E L84A0 L859F L8C34 L8BDD L490A L8BE8
L0299 L490F L8D0C L028C L8D15 L48E4 L8E34 L48E5 L48E6
L8E41 L48E7 L48E8 L48E9 L8ED2 L8E99 L8EA2 L8ECC L48EF
L8E7B L89C5 L4944 L89BD L89CC L897B L897C L4EE9 L9AE8
L4EEA L9B10 L9D1A L9D20 L9D26 L4FBF L9D3D L9D4B L9D5E
L9D72 L9D9D L9DDF L9DB7 L9DC2 L9DC8 L9874 L9877 L9891
L988A L450A LA9C0 L450B LA9BC LA9C3 LAB60 LAB59 LAB63
L46D5 L46D6 L46D7 L46D8 L82CA L8284 L828E L82E0 L8341 L8349
L83CA L84DE L84AD L84D3 L85AE L029A L4913 L8CA4 L8C5A
L490E L4914 L490B L8C7C L8C88 L491B L8C9D L8CA3 L4912
L8CC0 L4915 L4916 L491A L8D1B L490C L8C09 L8C31 L8C17
L8D36 L8D49 L0399 L0263 L8EFD L48BE L48BF L8F00 L026C
L8EC9 L8EBB L48FF L48FD L89E9 L4908 L8998 L4907 L89AD
L8995 L9AFE L9B3B L9B2A L9B41 L9CE9 L4FC0 L9D47 L4FC1
L9D56 L9D67 L9DBD L5030 L9DC5 L5031 L9DD5 L9DD8 L9899
L509F L98B5 L509E L98B8 L98C6 LAB91 L45CE LAB72 L45CF
LABA2 LAB7B L4E6D L4935 L82DA L4937 L8325 L828F L493B
L82B8 L493C L82AA L4933 L82E3 L4934 L82C6 L4930 L82E8
L082B L8357 L8362 L492C L836B L492D L837B L8390 L839D
L839F L492F L83B4 L492E L84E9 L854C L84F2 L8507 L8518
L0281 L8535 L027F L027D L0280 L027C L8559 L84B7 L4922
L84DC L496A L84D9 L8555 L85C9 L85D2 L4963 L4965 L8C1E
L8CD6
*** Warning: Branch Ref: 0x1800 is outside of Loaded Source File.
L8F4B L0297 L8F1E L8F89 L0296 L8F2C L48E3 L8F3D L8F4E
L8F57 L4900 L48FE L89A3 L4906 L9B09 L9B50 L9B82 L9B70
L0873 L9D70 L98BE L98C0 L98D6 L9913 LABB2 L414B LABD5
LABD2 LAB87 LAB89 L0269 L82FB L4931 L82F6 L4932 L831E
L4928 L8576 L492A L8599 L858C L4910 L8C3A L4911 L8C36
L491C L8CF0 L8F82 L8F70 L48F1 L8F6B L8F7D L504B L504C
L9B7C L9CBD L9B8D L9BA1 L98FD L50A2 L98EE L98F2 L50A0
LABC9 L4149 LABF9 LABE9 LABF4 L039A L8586 L8CFB L8C39
L8F7A L9CDC L9CCC L9CD3 L9CD9 L9B96 L9BB2 L9BBC L990B
L4147 L0213 LAC0B LAC1D LAC1B LAC24 L180F
*** Warning: Branch Ref: 0x180F is outside of Loaded Source File.
L0224 LAC3F LAC5D L3FE8 L3FE6 LACFC LACCB LACD7 LACE0
LACD1 LACB9 L4E67 LACC8 L8D0B L9CE7 L9B9F L9C31 L50B8
L9BF9 L9BDC L50BC L9BF6 LACFC LAD5E L3FEC L3FE4 L4E68
LACCE L9C49 L9C3E L9C44 L9C5F L9C0A L9C23 L9C1E L9C62
L9C56 L9C5C L9C2E L9C6D L9C94 L9C81 L9C8F L50BB L9CBC

```

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.

```

Disassembly Complete

Due to copyright issues, the source binary file and resulting disassembly file 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 output filename to “AV94BNBH.DIS”, sets 21 user-specified entry points and 16 user-specified labels, and enables the outputting of address information in the output file. 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.

After reading the Control File, the disassembler reads the M6811DIS.OP opcodes file – there are 308 unique opcode definitions for the HC11. Then after reading the opcodes, it loads the source file. 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.

This version of the disassembler allows for only one source binary to be loaded during a run and consequently has only one single load address that can be specified. If more than one file needs to be loaded, they must be externally concatenated into a single unit. Future versions of the disassembler will allow for multiple source files to be loaded, each at their own unique starting address, and will allow for file formats other than binary to be used. In any case, overlaps in the files' are not permitted.

After loading the source file, 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. Note that this version of the disassembler assumes that all indirect values specified are indirect addresses of (or pointers to) code entry points – such as an interrupt table entry. Future versions of the disassembler will also allow the specifying of indirect addresses for pointers that point to data instead of code.

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. 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 bootstrap
- One of many other possible “system” reasons

It can, however, from any of the following “user” reasons:

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

If you see errors of this nature, check your source file. Make sure it is the correct length and that you have specified the load address correctly. Make sure that you have all the pieces of the binary and that they are concatenated and/or padded correctly.

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 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 binary 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 binary. That is what makes this 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 binary 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 and all numeric arguments must be entered as hexadecimal values with no hex-denoting prefix or suffix. That is, you do **not** put a '0x' or '\$' or any other symbol before or after to indicate a hexadecimal value. Future versions of the disassembler will allow for other number bases, but on this version all values are assumed to be in hexadecimal.

Comments can be placed 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.

The order of the commands in the Control File does not matter on this version, as none of the commands have any interdependencies. However, on future versions, there may be some command orderings that will matter. For example, on future versions where the number-base can change, it would be necessary to specify a new default base before using that base. But on this version, order makes no difference.

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

On this version, all commands must be placed into one single Control File. Future versions will allow for combining of Control Files and more extensive Control File parsing options.

A bare minimum Control File can consist of only 2 lines – an “input” statement and an “output” statement, as you’ll see from analyzing the example and command descriptions below. The default load address for the source binary 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 three categories: 1) Switch Commands, 2) Value Commands, and 3) List Entry Commands. Each Switch Command switches a yes/no option in the disassembler. This version supports the following Switch Commands: ASCII, ADDRESSES, and OPCODES. The Value Commands allow you to specify a value for a disassembler option. This version supports the following Value Commands: INPUT, OUTPUT, and LOAD. 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. This version supports the following List Entry Commands: LABEL, INDIRECT, and ENTRY. Each of the supported commands are 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
;

input AV94BNBH.BIN
output AV94BNBH.DIS

load 4000

addresses
ascii

label ffd6 scivect
label ffd8 spivect
label ffda paievect
label ffdc paovect
label ffde tofvvect
label ffe0 ti4o5vect
label ffe2 to4vect
label ffe4 to3vect
label ffe6 to2vect
label ffe8 to1vect
label ffea ti3vect
label ffec ti2vect
label ffee tilvect
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 tovrtn
indirect ffe0 ti4o5rtn
indirect ffe2 to4rtn
indirect ffe4 to3rtn
indirect ffe6 to2rtn
indirect ffe8 to1rtn
indirect ffea ti3rtn
indirect ffec ti2rtn
indirect ffee tilrtn
indirect fff0 rtrtn
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 a .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.

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 www.bpssoft.com). As mentioned earlier, it is **not** good to include parts in the binary source image that isn't actually part of the code or data being disassembled – in this case the 0xFF padding.

In our output, we'd like for the disassembler to include the actual memory address for each instruction in the disassembly output, so we include the "addresses" switch to enable that. And we'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, so we'll include the "ascii" switch.

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'll include 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 binary, 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 get 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 "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.

The only Control File command supported in this version of the disassembler that isn't illustrated in this example is the "opcodes" switch. It works in the same way that "addresses" and "ascii" does, except that it causes the disassembler to output the actual opcode bytes for each disassembled instruction along with the normal disassembly.

Control File Commands

Switch Commands

ADDRESSES

Format: addresses

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      L0177
EBAD      LEBAD:   staa      L0177
EBB0      LEBB0:   rts
EBB1      LEBB1:   brset    *L003B,#0x04,LEC13
EBB5      LEBB5:   ldx      #0x5B00
EBB8      LEBB8:   brclr   0x08,x,#0x04,LEC13
EBBC      LEBBC:   brset    *L0090,#0x40,LEBD6
EBC0      LEBE0:   bset     *L0090,#0x40
EBC3      LEBE3:   brclr   *L001E,#0x04,LEC13
EBC7      LEBE7:   ldab    *L0031
EBC9      LEBE9:   cmpb    0xB8,x
EBCB      LEBEB:   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 deletes and easily delete the addresses, the 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

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. Here is an output example with “ascii” 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@;Rcf_a_[VRONNMKGDB>Rcf_URONMLK'
```

Note that the disassembler, in addition to outputting the ASCII equivalent of the bytes, will also output the byte values themselves. This 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” on or off. If it has a lot of printable text, running with it 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 it on and see what ASCII strings it produces and then decide from there.

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

If “ascii” had not been specified on the above, the output would have appeared as follows:

```
.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” 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).

OPCODES

Format: opcodes

The “`opcodes`” switch causes the disassembler to output the actual byte values for any instruction that it disassembles, as a comment on a line before the disassembled instruction. Here is an example of code that has been disassembled with “`opcodes`” turned on. Note that this is the same code as was used for the “`addresses`” example above:

```
; EBAC: 4F
LEBAC:  clra
; EBAD: B7,01,77
LEBAD:  staa      L0177
; EBB0: 39
LEBB0:  rts
; EBB1: 12,3B,04,5E
LEBB1:  brset    *L003B,#0x04,LEC13
; EBB5: CE,5B,00
        ldx      #0x5B00
; EBB8: 1F,08,04,57
        brclr   0x08,x,#0x04,LEC13
; EBBC: 12,90,40,16
        brset    *L0090,#0x40,LEBD6
; EBC0: 14,90,40
        bset     *L0090,#0x40
; EBC3: 13,1E,04,4C
        brclr   *L001E,#0x04,LEC13
; EBC7: D6,31
        ldab    *L0031
; EBC9: E1,B8
        cmpb   0xB8,x
; EBCB: 24,46
        bcc     LEC13
```

This is 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 default mode if “`opcodes`” is not specified is `OPCODES 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) as it will only make the output file bigger. Note that this option will not affect the reassembly of the file since all of the extra output is done as comments and will be ignored by the assembler. This option is here mainly 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 “`opcodes`” switch left off, please refer to the “`addresses`” command.

Value Commands

INPUT

Format: input <filename>

The “input” command allows you to specify the name of the source 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 binary format, as that is the only format that is supported by this version of the disassembler – future versions will support additional formats and will have additional arguments and options for this command.

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

The source file 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 need not fill the entire memory. If you are disassembling a 1K chunk of code that is originated at 0x0800 (for example), then the binary 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 binary 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.

With this version of the disassembler, one and only one source file can be specified. Any additional “input” commands override previous “input” commands and only the last specified file will be loaded.

Failure to specify the source input file will cause the disassembler to halt with an error.

LOAD

Format: load <addr>

The “load” command lets you specify the relative load offset address of where the source binary file will be loaded. Since binary files, by definition, contain no address structure, this command is needed to specify the absolute address of where in HC11 memory the file should be loaded. A file cannot be loaded outside of the 64K boundary of the HC11.

The <addr> argument is assumed to be in hex, and should not contain a ‘0x’ or ‘\$’ or anything else before it or an ‘h’ or anything else after it. An example, to load a file at 0x4000, would be:

```
load 4000
```

Future versions will allow for other bases.

If no “load” command is specified, 0x0000 is used.

Note that this is a relative offset for any addresses specified in the source file’s address structure. A binary file (which is the only format supported by this version) has no address structure, therefore, the load address will become the absolute address for the file. However, in future versions of the disassembler, when other file types are available, since this is a relative load offset, if the file type of the input file does specify address structure, this will offset the addresses in the file. For example, an Intel Hex file format specifies addresses for each byte or group of bytes in the file. If the hex file specified 0x0080 as the starting address and the load address specified by the “load” command was 0x1000, then the file will be loaded at 0x1080. But that will be in future versions.

OUTPUT

Format: output <filename>

The “output” command allows you to specify the name of the output text file for the disassembler to write. 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.

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 output file. The output file can then be viewed, edited, and/or printed by any favorite text editor. This output file can be re-assembled if need be – see *Reassembling a Disassembly* in this document.

Warning: If the specified output file exists, the disassembler will overwrite it with the new disassembly 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 re-run the disassembler, or even accidentally re-run 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 re-run the disassembler to, perhaps, disassemble some missed portion of the 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. Future versions will allow for saving comments and edits without losing them when re-running the disassembler.

Failure to specify the target output file will cause the disassembler to halt with an error.

List Entry Commands

ENTRY

Format: entry <addr>

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. However, I suggest that if you can represent the entry with an indirect that you do so, as the number of hard entries that can be specified in the control file is fairly limited – see the section *Limitations in This Version* in this document. Since most entries can be done with indirects (as most are jump tables) then in most cases this limitation will not keep you from disassembling something that you need to – I've disassembled some very complex things with hundreds of undetermined jumps and indirect entries and have never had problems with running out of entry points (however, I have run out of labels – see the “label” command – but not entry points). If you do run out of hard entry addresses, the disassembler will warn you. Future versions won't be bound by this limitation.

With respect to the limitations described above, the sample Control File is somewhat of a bad example. Most of the “entry” commands in that file 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”. And even then with a 48K source-file (not much less than the 64K size-limit of the processor), I was able to enter them all without running into the limitation.

The <addr> argument is assumed to be in hex, and should not contain a '0x' or '\$' or anything else before it or an 'h' or anything else after it. An example, to specify a code entry point at 0x7C12, would be:

```
entry 7c12
```

Future versions will allow for other bases.

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.

INDIRECT

Format: indirect <addr> <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 other 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 0xFFFFE, which is the reset vector for the processor. Immediately after power-on, the HC11 reads the 2 bytes starting at 0xFFFFE and uses those 2-bytes as the address for starting the execution of the HC11 startup code.

The <addr> argument is assumed to be in hex, and should not contain a ‘0x’ or ‘\$’ or anything else before it or an ‘h’ or anything else after it. 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 the reset vector located at 0xFFFFE, would be

```
indirect fffe reset
```

Future versions will allow for other bases.

This example is illustrated as follows:

```
RESET:    ....
          ... reset program code ...
          ...
RSTVEC:   .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 0xFFFFE 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 indirectioned 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 “RSTVEC” comes from in the above illustration.

On this version, labels are limited to six (6) characters and there is also a limit to the number of indirect vectors that you can specify – see the section *Limitations in This Version* in this document. However, the number of allowable indirects is sufficiently high enough that most all disassembly applications will have more than enough. This character limit is why the label for the vector appears as “RSTVEC” in the above example rather than as the specified “RSTVECT”. Future versions of the disassembler will get rid of the character limit and the number of entries limit.

Additionally, this version assumes that all indirects are Code-Indirects, or indirect vectors to program code, but it is also possible for source programs to have Data-Indirects, or indirect vectors to data, as well. Future versions will allow you to specify both Data-Indirects and Code-Indirects.

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 <addr> argument is assumed to be in hex, and should not contain a ‘0x’ or ‘\$’ or anything else before it or an ‘h’ or anything else after it. 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 0xFFFFE, would be:

```
label fffe rstvect
```

Future versions will allow for other bases.

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.

On this version, labels are limited to six (6) characters and there is also a limit to the number of labels that you can specify – see the section *Limitations in This Version* in this document. However, the number of allowable labels is sufficiently high enough that most disassembly applications will have more than enough. However, I’ve run into a few cases where there just wasn’t enough label space. Future versions of the disassembler will get rid of the character limit and the number of entries limit. The character limit will cause the label in the above example to appear as “RSTVEC” rather than as the specified “RSTVECT”. But, you can go ahead and specify the full name in preparation for the newer versions, as long as the first 6 characters are unique between labels.

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

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 Opcodes File: <filename>

This indicates that the disassembler had trouble either locating or opening the Opcodes File – which is called “M6811DIS.OP” for this disassembler. Check to make sure that it is in the current directory. While the disassembler can be run anywhere from the system path, the Opcodes File must reside in the current directory at run-time.

*** Error: Opening Control File: <filename>

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: Input and Output files MUST be specified in Control File.

You must specify both the Input File (or Source File) and Output File somewhere within the Control File.

*** Error: Opening Source File: <filename>

This indicates that the disassembler had trouble either locating or opening the Source File specified in the Control File. Check to make sure the file exists and is accessible to the disassembler and that you typed the name and/or path correctly in the Control File, and that you have read-access rights to the file.

*** Error: Opening Output File: <filename>

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: 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.

Not Enough Memory

During the initialization process the DOS memory limit was exceeded and the program isn't able to run. Check to make sure that you are loading DOS in high memory and that no unnecessary TSR (Terminate-and-Stay-Resident) programs such as mouse drivers, drive mappers, etc, are loaded.

Warning Messages

*** Warning: Branch Buffer Full

This indicates that the internal table that stores reference addresses to all branches is full. Branch addresses are added during the code-seeking pass of the disassembler as jumps and branches are encountered. When the limit has been reached, the disassembler will continue to function, however, sections of code that should be disassembled as code may come out as data because the disassembler was unable to add the necessary entries to the branch table. See the section *Limitations in This Version* in this document.

*** Warning: Branch Ref: <addr> 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 <addr> above) that was outside of the area loaded from the source file. 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 the load offset specified in the Control File was incorrect and caused the Source File to be loaded at incorrect memory locations. So check the source and make sure the warning makes sense.

*** Warning: Duplicate Label Definition

This warning is displayed anytime a label that has already been defined in the Control File is redefined again within the Control File, based on address. With this version of the disassembler, only one label can be defined per address and any attempt to label an address with more than one label or name produces this warning and the extra label is ignored. You can, however, use the same name for more than one address. Why you would want to, I don't know, as it will probably cause any assembler that you later try to reassemble with to croak, not to mention confusing yourself. The disassembler doesn't check for ambiguous names, cause it really doesn't care what you call each address.

*** Warning: Entry Point: <addr> is outside of Loaded Source File

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 <addr> is outside of Loaded Source File” for more information. Also, check to make sure that you have typed the entry point correctly in the Control File.

*** Warning: Indirect Buffer Full

This warning indicates that the internal list of indirect code vectors is full. Since this version of the disassembler doesn't know how to add any indirects on its own, other than the ones you specify in the Control File, this means that you've entered more indirects in the Control File than memory allows for. See the section *Limitations in This Version* in this document. Once the table is full, the extra entries are ignored, meaning that some of the code area may be incorrectly outputted as data.

*** Warning: Label Buffer Full

This warning indicates that the internal list that associates a string label with an address is full. This can either be a result from “label” commands in the Control File, or from labels generated during the code-seeking pass of disassembly and/or during indirect resolution. See the section *Limitations in This Version* in this document. Once the table is full, any code areas needing a label will still be disassembled with a “Lxxxx” label, however, the location itself won't have the label labeling the line, and thus probably will not re-assemble without giving an unresolved reference or unknown label error. The output is still very useful, especially if you use the “addresses”

command in the Control File to output the addresses, and if there aren't too many missing labels, it may not be too big of a chore to add them by hand later.

*** Warning: Memory wrap around encountered, Check Load Offset/File Length

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 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 "wrap around" at the end of the 64K boundary of the processor back to an address of $0x0000$. In this particular version, the wrap will proceed back around to address $0x0000$ and the file will continue loading from there. However, future versions may, and probably will, terminate the file loading at the 64K boundary. So it is not a good practice to rely on this wrapping effect in this version. 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. Future versions may allow for emulation of bank switching methods, but you'll still have to reverse engineer the system enough to figure out what those methods are and write a function or interface for the disassembler to emulate it. And, it may not be possible to emulate all methods.

*** Warning: Too many ENTRY Commands

The number of "entry" commands that can be specified in the Control File is limited because of memory constraints. However, usually very few direct entry points have to be specified because most will be done with indirect vectors. If you run out of entry points, and really need more, you can always "beat the system" by specifying them in an unused portion of the source binary and using an "indirect" command in the Control File. There SHOULD be an unused portion in the source, as there has to be some sort of RAM memory used in the MPU. The desired effect will result and you will have "beaten" the entry point limit. However, I've never run into any real world application that needed more entry points than could be specified, since most things already are indirects – like interrupt vectors, jump tables, etc. See the section *Limitations in This Version* in this document.

*** Warning: Unrecognized command in Control File

This means that one or more commands in the Control File 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. All values must be in hexadecimal with **no** '0x' or '\$', etc, before it, and **no** 'h', etc, after it – everything is assumed to be in hexadecimal and may be falsely converted to zero (0) if you do otherwise. Future versions will allow for different bases, but not this one.

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 (OFST=12):

```
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 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. So, this version of the disassembler doesn’t deal with the problem at all. (Sorry). In future versions, I’m contemplating a “fixed length” solution and possibly a solution of object types whereby you can specify certain typings and/or methods that the disassembler can use to figure out lengths on variable inline data.

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 proceeding 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”. 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 and your done.

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):

```
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 output file – Usually, search-and-replace works well for 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 work-arounds. 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.

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. 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 correctly 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.

When this program was developed, the initial goal was reverse engineering, not hacking. Therefore, this version has no “spit” mode. However, because of many requests from hackers that want quick results, future versions will incorporate 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 (www.mot-sps.com).

Mnemonic	Machine Code	Form	Disassembler Action	Discontinue Disassembly
test	00	test		
nop	01	nop		
idiv	02	idiv		
fdiv	03	fdiv		
lsrc	04	lsrc		
lsl	05	lsl		
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		
sbc	18 E2 ff	sbc ff,y		

add	18 E3 ff	add 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	
adbb	18 EB ff	adbb 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 hhl	Add Data Label
sty	18 FF hh ll	sty hhl	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 hhl	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	
belr	1D ff mm	belr 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	
lra	44	lra	
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	
larb	54	larb	
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		
sbc b	C2 ii	sbc b #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
sbc b	D2 dd	sbc b *dd	Add Data Label
add	D3 dd	add *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	
sbc b	E2 ff	sbc b ff,x	
add	E3 ff	add 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 hhl1	Add Data Label
cmpb	F1 hh ll	cmpb hhl1	Add Data Label
sbc b	F2 hh ll	sbc b hhl1	Add Data Label
add	F3 hh ll	add hhl1	Add Data Label
andb	F4 hh ll	andb hhl1	Add Data Label
bitb	F5 hh ll	bitb hhl1	Add Data Label
ldab	F6 hh ll	ldab hhl1	Add Data Label
stab	F7 hh ll	stab hhl1	Add Data Label
eorb	F8 hh ll	eorb hhl1	Add Data Label
adcb	F9 hh ll	adcb hhl1	Add Data Label
orab	FA hh ll	orab hhl1	Add Data Label
addb	FB hh ll	addb hhl1	Add Data Label
ldd	FC hh ll	ldd hhl1	Add Data Label
std	FD hh ll	std hhl1	Add Data Label
ldx	FE hh ll	ldx hhl1	Add Data Label
stx	FF hh ll	stx hhl1	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 following 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 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-saver. The M6811 Code-Seeking Disassembler was designed for the reverse-engineer and so the output was targeted for a specific assembler.

This disassembler targets is 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 versus 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.

Limitations in This Version

This version, since it is compiled as a DOS mode 16-bit program, is limited to the 640K boundary of DOS. Additionally, the compiler used also requires all objects to be no larger than one 64K segment – in other words, it uses only offset (16-bit) addressing for these objects rather than a full 32-bit address. Therefore, all of the internal tables in the program must not exceed 64K. I guess I could have gotten fancy and had multiple parts of the table in separate 64K segments, but that would have greatly complicated the design and would have slowed it down. See the section on *Future Versions* to see how these limitations will be eliminated in upcoming releases.

The worst two limitations are the name size limit for labels and the label table overall size – and yes, these two are related. If the size of the label names were increased, the number of entries must be reduced. In some large applications, it is possible to run out of label entries (as I have experienced), so I decided to limit the name length for the labels as a trade-off between label name length and number of label entries.

The next big limitation is the number of branch table entries, but usually you run out of label table entries first, since all branches will get a label in addition to the labels from the data areas – so obviously more labels are used. Thus far, I've never run out of branch table entries – yet.

Below is a table denoting the limitations within this version:

Item	Limit	Units
Entry Address (Control File Only)	32	Table Entries
Branch Address Table	4096	Table Entries
Label Table	4096	Table Entries
Label Name Size	6	Characters
Indirect Table	512	Table Entries
Indirect Vector Types	Code Only	Vector Type
Source Binary	65536	Bytes
File Format	Binary Only	File Type
Number of Loadable Source Files	1	Input Files
Number of Loadable Control Files	1	Control Files
Number Format (Control File Only)	Hexadecimal	Data Type
Code-Seeking	Seek Only (no spit)	Seek Methods

Bugs

With any software application, it is likely that at least one bug will exist somewhere. Unfortunately, this version of the M6811 Code-Seeking Disassembler is no exception. However, all of the known bugs have benign consequences, can be easily resolved, and do not hinder the disassembly process. Below is a list of known bugs along with how to “fix” them. Obviously, these are only the ones that I know about, meaning that there could be others. If you find additional bugs, or think you have, please contact me and let me know – See the *Support* section for contact information:

Erroneous “*** Error: Opening Source File” messages

This is caused by a missing or corrupt “m6811dis.op” file. Make sure that you have a good copy of the **m6811dis.op** file in the **current directory** at the time you run the disassembler. I failed to reset the IO error status after reading (or attempting to read) the opcodes file. Since the source file is read after the opcodes file, this results in an erroneous error message indicating that the source file has problems.

Program Runs Forever in an Endless Loop and Fills Up Remaining Hard Disk Space

The disassembler determines the end of the file by looking at loaded vs. unloaded parts of the source binary. If you load a full 64K (65536) byte source file, the program will erroneously loop around (because of a 16-bit pointer) and keep looping when writing the output file. To fix this, trim the unused RAM area and processor register area out of your source binary file. You shouldn’t be trying to load these unused portions anyway as they can cause erroneous output in the output file by it thinking that a memory address has a certain value when in reality it is dependent upon what is in the RAM, which will obviously be different. There **will** be some RAM mapped somewhere, as it is needed for stack space.

Mid-Opcode Labels Don’t Get Reported in Output

This isn’t so much of a bug as it is from being either invalid code seeking from things such as Code Inline Data (See *Disassembly Pitfalls* in this document) or incorrectly specified labels in the Control File. What happens is that if a label is specified or generated and the address for that label happens to occur in the middle of an instruction (rather than being the address for the first byte of an instruction), the label doesn’t get assigned in the output file. In reality, this erroneous label should have never gotten generated in the first place and can be caused by you specifying wrong entry points and/or indirect entries in the control file, or by having the code somehow skewed, such as with Code Inline Data, or by just having erroneous screwy code in the first place. This bug (or quirk) is no big deal and will only cause an assembler to later report that a label wasn’t defined. You should examine any such labels to find the cause and if necessary, add equates before re-assembling.

Support

The Disassembler

I will continue to maintain this version and will be enhancing it and releasing future versions. This version, and future M6811 only disassemblers 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. This does not apply to GenREP (the Generic Reverse Engineering Platform) that I'm developing, which will have a M6811 module. That application, when completed, will be a commercial product – or so that is the current plan.

Currently, my ISP only provides dynamic IP support and so I cannot register a domain name for my web server machine. However, I keep a dynamic front-end page on their server that constantly points to the web server on my machine. The front-end page can be found at: <http://home.midsouth.rr.com/dewtronics>

Once on my site, browse for the downloads-section. 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, since Motorola's website is so hideous (and almost impossible) to locate anything – or so it used to be unless they've recently changed its layout.

If you find any bugs, have suggestions or ideas for program enhancement, or have any questions in general, you can email me at: dewtronics@tech-center.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. Their website is at <http://www.mot-sps.com>.

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. Many of the CD-ROM's are online and can probably be found on the web, though I haven't really looked. The version I have tested 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, I have not confirmed the functionality of newer versions with this disassembler. As I come across other versions and resources, I will post them online to be downloaded, but will keep Version 1.50 online as well since I know for certain that it works.

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

As I was writing this document, I checked his anonymous FTP site and it does indeed contain a newer version of the assembler – Version 2.10 to be exact, dated April of 1999. At first glance, there doesn't appear to be any problems with using this newer version. I will test it as soon as possible and update this document accordingly. It does look like this new version has Linux support – hurray!

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 can be found at BreakPoint Software, called Hex Workshop, at www.bpssoft.com.

Future Versions

So what is planned in the future? Well, a whole lot is planned. This version was the first of many 6811 disassemblers to come. This version, Version 1.0, was originally written in Borland Pascal 7.0 and will be the baseline standard for the DOS 16-bit platform. I have ported it to MS Visual C++ 5.0 and created a DOS 32-bit version, which will soon be released as Version 1.2. It eliminates all of the current limitations and is bounded only by available system memory. It uses a DLL class that I call DFC (Data File Converter) to allow support for any source file format – binary, Intel Hex, Motorola Hex, etc. For formats that I happen to not supply a DFC for, you simply need to write a DLL to handle your new format – no recompiling, no rebuilding, just make a DLL and run.

At the time I am writing this, the Version 1.2 works, but I would like to change the internal opcode seeker to be a hash-table of hash-tables rather than a hash-table of arrays. For the 6811 it won't make much difference, but for other processors like the 6816 and 68332, it makes a world of difference in disassembly time. So you say, why not release it and then update it later. The reason is I'm creating a GDC class – Generic Disassembler Class – which will allow disassemblers to be encompassed by a DLL. This will later be used in the GenREP (Generic Reverse Engineering Platform) program that I plan to release as a commercial product. The idea is that the program will not be dependent on any processor, any target assembler, any file format, etc. You can simply load DLL files into the program at run-time.

So with this available, why do I still support Version 1.0? Well, Version 1.0 is a 16-bit application that can run directly in DOS or DOS-Command Prompt mode. Version 1.2, although it is a DOS version, is a 32-bit application and requires MS Windows 95, 98, NT, etc, and has to be run in a DOS Prompt Window. For some, this isn't acceptable.

I am also writing a Version 2.0 that will be a Windows GUI for the disassembler. This will allow for easy graphical entry and manipulation of Control Files, easy source editing, etc. It will basically be a front-end for the 32-bit Version 1.2 of the disassembler.

I have also recently switched my personal machines over to Linux, abandoning Microsoft almost entirely. This means that before long, a Linux version will also be available and may actually beat the work-in-progress Windows version. Currently, this application is a freeware application, but not an open-source application. I am considering making it open-source when releasing the Linux platform, but haven't fully decided yet.

As for keeping up with what version is which, basically the first part of the version number will denote its platform – 1 = DOS, 2 = MS Windows, 3 = Linux Command Prompt, 4 = Linux X Windows GUI, etc. Also, you might have suggestions or ideas for future versions – please send any suggestions/ideas to me. And keep an eye on my website for future releases – see the *Support* section in this document on how to locate my website and how to contact me via email.

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

This version. Somewhat limited, but it runs in DOS on nearly any machine and is a great entry level version.

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

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 of code-seeking. Mixed number bases is supported in the Control File. The disassembler has been converted into a GDC (Generic Disassembly Class) that will later allow easy porting into GenREP (Generic Reverse Engineering Platform) and 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. All that is missing is a graphical front-end. This version is basically finished except for changing the opcode table to be a hash table of hash tables rather than a hash table of arrays. It will be released soon.

Version 2.0 – Windows 32-Bit (Win95, 98, NT, etc)

This is the graphical front-end that is "missing" on Version 1.2. It will allow editing of source data files, text editing of disassembly output files – including keeping comments and user edits separate from disassembly output so that if the disassembler is ever re-run on a file, you don't have to re-edit everything, and a graphical interface for editing and entering the data into the Control File without generating the Control File by hand.

Version 3.0 – Linux (Command Prompt)

This will be a port of Version 1.2 into the Linux environment. It is possible that the disassembler might become open-source at this stage to allow migration to other Unix platforms.

Version 4.0 – Linux with X-Window GUI

This can be thought of as either a port of 2.0 to Linux or as a wrapper for the 3.0 Linux version.

With the advent of GDC, it will be easy in the future for others to create modules for additional processors and easily drop them into the program without having to modify or recompile the main program at all. Once this gets further along, and the specifications and methodology have been defined, I have in mind releasing a "development kit" for those wanting to develop disassemblers for other processors. The kit will be free, but what I ask in exchange is that you submit any additional disassembler modules you create, so they can be provided to the world to use free-of-charge.

Enjoy the disassembler. I hope it proves to be most helpful. Please visit my website (see the *Support* section) and register. I enjoy tracking the progress of my software and like to see how many different countries it ends up in. You can also "vote for" those versions that don't exist yet. Those with the greatest number of requests will receive a higher priority in the programming/debugging process.