SAD User Manual 5

A Semi Automatic Disassembler Tool for Ford EEC-IV and V EEC controllers
SAD Version 5 onwards

1. Disclaimer

(I have to include this first.)
SAD is not intended for any commercial purpose, and no liability is accepted whatsoever.
SAD works with files which may be copyrighted, use it entirely at your own risk.

1.1 What does this program actually do ?

SAD is intended as a disassembly tool to help understand how the engine tuning of a particular vehicle works via the algorithms and calibration data revealed in the program code. It does not provide for reassembly of modified code for custom tuning. SAD is still under development, and no guarantee is given as to its usefulness for any binary file.

This manual is intended as a quide starting with simple examples to get you started, and then introduing more complex and advanced commands.

For full understanding of EEC_IV and V, more data is available on the web. Note also the Intel 8096 series documentation, as the 806x ‘core’ is based on this CPU. Excellent information available in OpenEEC project at “https://github.com/OpenEEC-Project”.

OK – let's get on with the proper instructions.

SAD attempts to automatically disassemble a binary file taken from an EEC-IV or EEC-V engine management system. SAD can analyse Intel 8061 and 8065 single and multi-bank binary files. It attempts to produce an output listing with code and data correctly analysed. SAD can operate with or without user commands, from fully automatic analysis down to entirely manual in its processing. It is designed with the idea of iterative runs by the user, each run with probably a few more commands, symbols etc., as understanding grows. The output listing is not intended to be compatible with any standard assembler tools, but is designed to be human readable to help with analysis and understanding of the data and code structures, and how they are used to manage a vehicle internal combustion engine. It does print out a ‘psuedo code’ which is similar to many popular languages.

SAD reports errors and relevant information, including the output of the commands it was able to deduce. These commands can then be cut and pasted into a command file and edited, and SAD then run over again.

Examples are included throughout this doc to help understanding of the how this tool works.

1.2 Numbers and symbols used in this guide.

Most descriptions in this manual are plain English. For technical terms and concepts, this guide sticks with the widely recognised standards.

Numbers in this manual are hexadecimal by default, and are written with the prefix '0x', matching the standard for most programming languages. All numbers in SAD, commands, input and output files are hexadecimal by default, unless otherwise specified. '806x' means both 8061 and 8065 CPUs.

1.3 Input and Output files

SAD must have a binary EEC file as a minimum, with a suffix of .bin.
The file name full path length has a maximum of the system it runs on (typically 2048 chars).

sad.ini - this is a config file. It defines the default directories for the various file classes. It is optional. If it does not exist, all files are assumed to be in the same directory as the SAD executable itself (including sad.ini).

SAD itself is a console (i.e. non graphical) command line program, but has a 'wrapper' called SADwin to provide a GUI interface for easy use. (SADwin is Windows only at present) SAD is available for Windows and Linux.

You can run SAD directly via a console terminal window, with
"SAD -c <path> <bin_name>" where
-c <path> is to specify the directory location of SAD.ini (optional)
<bin> is the binary filename (can have .bin suffix or not)
If you run SAD with no extra inpuut, it will look for a local sad.ini and will ask for a filename.

SAD then reads or creates the following files in the directories specified by sad.ini, or if sad.ini does not exist, in the same directory as SAD.exe. "xx" is binary file name without the .bin suffix
All files are plain text (except the .bin file), and can be opened with Notepad or similar.

sad.ini	read (optional)	Config file for directories
xx_dir.txt	read (optional)	Directive file, which has user commands in it.
xx_cmt.txt	read (optional)	Comments file, which has user comments in it.
xx_lst.txt	create&write	Disassembled listing
xx_msg.txt	create&write	Information and messages from the disassembly process.

The xx_msg.txt file includes a list of commands deduced during the disassembly. This list (part or whole) can be cut and pasted into the .dir file to refine and rerun the disassembly as required.

Example of file names for the binary A9L are therefore -

A9L.bin	binary file (input)
A9L_dir.txt	directives file (optional)
A9L_cmt.txt	comments file(optional)
A9L_lst.txt	disassembled listing
A9L_msg.txt	information and messages

1.4 Binary File (.bin) format

There are several different common binary file layouts in common use. These files are produced by a variety of readers. .Most files have unused ('filler') areas, sometimes before the code starts. There can be 1 , 2, or 4 'banks', each of which are a maximum of 64k bytes (0-0xffff). Nearly all bins have 'filler' at the end of each bank. Multibank 8065 files can have differing bank orders. 'Filler' areas are typically '0xff' values, some have other patterns (which may be caused by a reader fault). 8061 binaries are always 1 bank, 8065 binaries may be 1, 2, or 4 bank. There are bins which have a 'dummy' bank, which only has the minimum required by hardware and is otherwise empty. (more below)

The hardware rules for the 806x CPU require certain minimum things be present in each bank. SAD looks for these items to try to find the true start of each bank. This item is a 'jump' instruction followed by a set of interrupt handler pointers. Only bank 8 (where the code starts) has a true jump, the other banks have a 'loopstop' jump, which is a jump which goes to itself. SAD attempts to work out the bank order

from the code in the interrupt handlers. If it cannot, it defaults to one of the common layouts. (see below). 8065 has many more interrupt handlers than 8061.

SAD should be able handle all bin files, including those with a missing or extra byte or two at the front. The maximum number of banks is 4. These banks are numbered 0,1,8,9 by Ford convention. Multibanks typically have their calibration data in bank 1. For multibanks, address 0x2000 in bank 8 is always where the CPU starts its initialisation and therefore where the code starts (or 'boots up' in computer terms).

SAD assumes the following for banks and layouts -

File size, between		Banks	CPU	Bank Order
1k	64k	1	8061 or 8065	8, bank not shown in listing
64k	128K	2	8065	1, 8 or 8,1 (auto detected)
128k	256K	4	8065	See next table.

For 4 bank binaries, SAD tries to determine the true order of the banks, but it only knows for sure where bank 8 is, so it defaults to the following rules, which may not work for all bins. In the case it does not work you must specify the bank and file layout with user command(s).

Bank order for 4 bank binaries.

If bank 8 is in found in	default order is	
First slot	8, 9, 0, 1	
Second slot	9, 8, 0, 1	
Third slot	0, 1, 8, 9	(most common)
Fourth slot	0, 1, 9, 8	

SAD then maps each bank to nominally start at 0x2000, and continue up to a maximum of 0xFFFF. This may vary slightly where there are missing or extra bytes, and smaller files or banks give smaller end addresses.

SAD then checks for 'filler' (=unused) data (typically 0xff) at the end of each bank.

The smallest bin found so far is the 'AA' (UK Ford Granada 1985) which has only 0x2000 to 0x3fff (2kbytes) for all data and code.

1.4 Output format

This is a typical section of output (single bank)

```
20c5: 01,9c             clrw  R9c             R9c = 0;
20c7: c3,01,2a,01,9c    stw   R9c,[12a]       [12a] = R9c;
20cc: a1,00,80,76       ldw   R76,8000        R76 = 8000;
20d0: 05,82             decw  R82             Timer21_mS--;
20d2: b1,85,0c          ldb   Rc,85           HSI_MASK = 85;
20d5: b1,03,03          ldb   R3,3            LIO_PORT = 3;
20d8: a3,01,6a,2d,6c    ldw   R6c,[2d6a]      R6c = [2d6a];
20dd: a0,6c,6a          ldw   R6a,R6c         R6a = R6c;
20e0: 01,12             clrw  R12             R12 = 0;
20e2: a1,ff,7f,8e       ldw   R8e,7fff        R8e = 8000;
20e6: c7,01,c0,01,9e    stb   R9e,[1c0]       [1c0] = R9e;
20eb: a0,06,fe          ldw   Rfe,R6          Rfe = IO_TIMER;
```

```
20ee: a3,01,ae,2c,68    ldw   R68,[2cae]      R68 = [2cae];
20f3: a3,01,b0,2c,fa    ldw   Rfa,[2cb0]      Rfa = [2cb0];
20f8: 36,15,03          jnb   B6,R15,20fe     if (B6_R15) {
20fb: 91,04,b2          orb   Rb2,4           XFail = 1; }
```

Multibanks have the bank number included in the addresses printed to give a single address -

```
1ad48: 89,66,06,3c       cmpw  R3c,666
1ad4c: d6,15             jge   ad63           if (R3c < 666)  {
1ad4e: 07,3c             incw  R3c            R3c++;
1ad50: 07,3e             incw  R3e            R3e++;
1ad52: 05,40             decw  R40            R40--;
1ad54: c3,37,2a,08,3c    stw   R3c,[R36+82a]  [R36+82a] = R3c;
1ad59: c3,ea,be,40       stw   R40,[Rea+be]   [Rea+be] = R40;
1ad5d: c3,dc,b2,3e       stw   R3e,[Rdc+b2]   [Rdc+b2] = R3e;
1ad61: 20,03             sjmp  1ad66          goto 1ad66; }
1ad63: 91,20,46          orb   R46,20         R46 |= 20;
```

1st column is the binary address, in hex (incl. bank number for multibanks)
2nd column is the data bytes, in hex,
3rd column is the opcode instruction
4th column is the instruction operands
5th column is ‘psuedo source’ code, with symbol names and addresses resolved where possible.

The last column is designed to provide a 'fake source code' or ‘psuedo source code’ type explanation of what each instruction does in human readable form. This has symbol names, bits/flags, labels, resolved in a reasonably simple form to help readability. If a 'symbol' command has been specified for that address or field, that symbol name will appear.
By default, register references are preceded by an 'R' , and bit flags by 'Bn_', where n is the bit number.
Mixed size and mixed sign instructions also have extra flags appended to help understanding.

The numbers appearing in square brackets are pointers. 806x opcodes support pointer types of indirect and indexed. Indirect pointers are shown as [R30], which means “the contents of the address pointed to by the value in R30”. An indexed pointer looks like [R30+5ed2], which means “the contents of the address made by adding the contents of register R30 to 0x5ed2”. This gets more complicated with multibanks, as it's not always obvious which bank is being referred to (more on this later), but SAD attempts to sort out which bank is being referred to.

There is a '++' suffix, which means 'increment register after it is used'. This increment is correct by access size, so increments by 2 for word opcodes, and 1 for byte opcodes.

All references, including pointer structures are also resolved into symbol names where possible.
For more detailed information on the CPU ‘engine’ and its various address modes and hardware, refer to the OpenEEC project.

2. Commands (.dir file)

SAD allows a set of commands which specify a wide range of instructions for control of disassembly. These commands can be used to override or steer parts of the automated processing, right through to a fully manual process. These commands all reside in the xx_dir.txt file.

It would be fantastic if SAD could always do its analysis automatically and correctly, but some binary files are very complex and may require user commands to work correctly.

SAD will not override any command made in the file, and will continue to try to do as much as it can automatically. The default rule is that user commands are ‘king’ and are locked/unchangeable, but other related attributes may be changed if not specified directly in the command.

For example, defining a table name as a SYMBOL does not specify anything else (e.g about its data format or size). Specifying a TABLE with its sizes and no name allows an automatic symbol name to be added to it.

The commands therefore define or override only where it is necessary. SAD will work happily with no directives at all, and development work continues to make as much as possible fully automatic. However, you can specify an entirely MANUAL mode which will do no automatic processing if you prefer.

Note – commands have a NEW SYNTAX from prior versions. The version 4 syntax still works for backwards compatibility.

The basic structure of each command is -

command params “name” [$ global] [data item] [data item] ...

key	min	max	description
command	1	1	A text string of at least 3 characters (compulsory)
params	1	4	Hex values, e.g start and end address. Depends on command
“name”	0	1	Optional text string, assigned to a symbol name at the start address
[ ]			A set of options
global	0	1	Options affecting whole command (e.g. layout options) Optional.
data item	0	31	Defines 1 data item (e.g. size, signed/unsigned etc) optional and multiple

The characters ‘$’, ‘[’, ‘]’ , ‘|’ have special meanings

[ ]	Start and end delimiters for a single data item. May have several options inside it
[$]	Defines that this is a global options item.
|	Specifies that a newline should be printed after this item in the listings file (splits long lines)

A full syntax description and allowed options for each command are shown at the end of this document, but here we will give an overview of what the commands do

2.1 setopts and clropts Commands

setopts defines top level disassembly options for SAD. The structure of setops and clropts is -

setopts [name] [name] [name] ...etc.
clropts [name] [name] ..etc.

where name is a text string, and its meaning is -

Name string	Purpose	Default	Note
default	Set (Clear) default options (as per ‘Default’ column)		
sceprt	print ‘pseudo source code’ for opcodes	x	
tabnames	automatically name new tables	x	1
funcnames	automatically name new functions	x	1
ssubnames	automatically name new special function subroutines	x	1
8065	set SAD for 8065 cpu instead of 8061		2

intrnames	Automatically name intterrupt handlers		
labelnames	automatically add label names for all jumps		1
manual	inhibit ALL automatic analysis, just use user commands		
subnames	automatically name subroutines when called	x	1
signatures	look for signatures (special code sequences, eg, table lookup)	x	
acomments	Auto comments	x	4
sympresets	Add preset symbol names for special registers	x	1,3
compact	Use compact layout for arguments and data structures		5
extend	Use extended layout for arguments and data structures	x	5

Notes.

1. All automatic names can be overridden with SYM commands.
2. 8061/8065 CPU type is auto detected.
3. Sad has a preset list of names for special registers , 0-0x10 for 8061, and 0-0x22 for 8065.
4. Automatic comments may be added to help with code analysis.
5. These are a listing wide default setting and can be overriden in a single command

If the .dir command file has no setopts: command, then ‘setopts : default’ is assumed.

2.2 Global Options

The global options group consists of the following possible letter options. See full definition for which options are valid with which commands.

A	Use ‘args’ layout. Each data item is printed on a new line. (as per ‘extended)
C	Use ‘compact’ layout. All data items are printed on the same line, unless a ‘|’ char is used.
Q	This command has terminator byte(s). Optionally followed by number 1-3 for no of bytes
F <text>	Special Subroutine type (e.g. Table or Function lookup)
=	Allows print of a pseudo answer, i.e. “Rx = SUBR();”

Data item options

The data options consist of one or more letters which define a data item. Some letters have a following parameter. Not all options are valid for every command. The options are separated by space in a list. A colon defines a new data item. This allows for a data structure of different types and sizes. This can get quite complex, and can look horrid to read, but it gives a lot of flexibility for the commands.

Letter	Param	Meaning	Notes
Y	none	Item is byte sized	default
S	none	Item is signed	
U	none	Item is unsigned	default
W	none	Item is word sized	
L	none	Item is long (4 bytes)	
X	2,10,16	print radiX and 2nd param for	Sets binary, decimal or hex print format

		float	
R	none	Item is a pointer	e.g. to a subroutine
N	none	Look for a symbol name for item	Print symbol name instead of value, if found.
P	1-31	Minimum print width of item.	To make items line up neatly. Default rules below.
K	0,1,8,9	Bank number	Used where addresses/items span banks
O	1-31	Repeat Count	also number of Columns in a TABLE.
B	0-15, 0-15	Bit field.	Defines a bit field start->end.

Retired options. Still work, but return with a ‘calc’ definition in command

D <n>	1H	A pointer or value with a fixed address offset added.
E <n> <n>	1D 1H	An encoded address type. (type, base register)
V <n>	1D	Devide by n

The A9L code contains examples of many of the above features, and so is a good example and illustration of what these extras are designed to do.

A quick example

func 28cc 2917 [W][W = int (x/128)]

Yes, this looks horrible, but read on, all will be explained.

This command defines a FUNCTION, which is a one dimensional lookup. The function begins at 0x28cc, ands at 0x2917, and the first column (the INPUT column) is unsigned word. The output column is also unsigned word, and is divided by 128, and calculates as an integer.

Parameters and addresses

Addresses and bank number.

Single bank binaries have no bank number. All addresses are 4 digits max, 0-0xffff.

Multibank binaries. CPU registers have no bank (0-0x3ff). All addresses over 0x3ff must have a bank. They are 5 digits, with the bank number being the first digit. Valid banks are 0,1,8,9. Bank 0 is NOT assumed and must be specified with a leading zero. (i.e. ‘2345’ is NOT a valid address for bank 0)

In commands, the end address can be 4 digits, and is then assumed to have the same bank as the start address. Banks may differ if you use 5 digits, depending upon the command details. Where a command requires only a start address (e.g SYM), then 'end' address can be left out entirely.

Note: Internally, SAD treats a single bank binary as if it is bank 8, because this allows exactly the same analysis process and code for all binaries.

After an initialise (= power on) the CPU always starts at 0x2000 in bank 8.

Start and End

For convenience, and for larger data structures, start to end may define a multiple instance of a command, e.g. WORD 2400 241F defines 16 words . Note that any additional data definitions (e.g. a divisor) will then also apply to all 16 words.

For data structures, this technique allows a definition of a number of 'data columns' (with the extra data items) without defining the number of rows, as this can be derived from (end-start)/row size.
For mixed data items (as in a STRUCT) the same applies , where row size is calculated from the list of data items, and number of rows via start and end.

Simple Command Examples

Simple commands have zero extra options and are straightforward.

code 15688 15698 defines only code exists from bank 1 5688 to bank 1 5698
code 15688 5968 is identical, bank 1 is assumed for end address

code 82000 82003 means code exists from bank 8 2000 to bank 8 2003
code 2000 2003 for a single bank binary

If name is specified in the command, this is equivalent to a separate SYM command, which assigns a name to the **start** address.

fill 3000 3100 The data between 0x3000 and 0x3100 is empty/dummy (typically 0xFF)
byte 3000 3100 The data between 0x3000 and 0x3100 is all bytes

word 93000 3100 The data between 0x3000 and 0x3100 bank 9 is all words (16 bit).
code 13000 3100 The data between 0x3000 and 0x3100 bank 1 is code instructions.

2.4 **Simple Comands,** which do a bit more work

scan 82000

This command tells SAD to do a code 'scan' from this address and bank. SAD will decode each opcode instruction from here, and track jumps, subroutine calls, and data accesses to sort code from data. This example is where the automatic analysis process starts, and is the heart of the automated disassembly process.

vect 2010 201f (for 8061)
vect 82010 205f (for 8065)

This command defines this block as a 'pointer list' to subroutines. SAD will then log each pointer as an address to be scanned as a subroutine, and assigns a name to each. If pointer is not valid then it is displayed as a WORD.

These two commands above (scan and vect) are defined automatically by SAD as its 'master start' setting. This start scan and vect pointers are the standard interrupt handling subroutines in all binaries. Note that multibanks have a interrupt vector list in every bank, which SAD also defines automatically.

vect 13100 3160 [K 8]

This command defines a vector list (pointers to subroutines) in bank 1 at 0x3100, but the subroutines pointed to are in bank 8. This 'cross bank pointer' is common for a multibank binary

WARNING !! Please be careful if you override the default bank definitions with a ‘K’ as an incorrect bank may cause unpredictable behaviour (because of faulty pointers) and possibly even crashes.

xcode 13000 4010

Sometimes SAD logs an area as code incorrectly. This is can be as a result of over running a vector list of subroutines, or sometimes the list has a data item embedded in it. It is almost impossible to design a strategy which catches all the real code without EVER getting a false pointer. This command tells SAD that this area (13000 to 14010 is definitely DATA, not code, and any code pointers and jumps into this area are to be regarded as illegal, and therefore ignored.

rbase 76 4080
rbase 76 8740 2230 2350

(Set register 76 as pointer to 4080, and set register 76 as pointer to 8740 between adresses 2230 and 2350)

Many binaries use a set of defined registers as permanent, fixed 'base' address pointers, and then use the indexed address mode of instructions to get at the data. This command allows SAD to decode the index to produce a true absolute address, and add or find a symbol name, if there is one defined. SAD will normally detect the most commonly coded 'calibration pointers' (typically Rf0 – Rfe) and set these automatically. Encoded address types (see later) typically use these rbase registers.

Many binaries also set other registers as ‘base’ pointers into RAM and KAM, some permanent, some temporary. SAD attempts to detect and confirm permanent pointers. The temporary pointers are used for example as an rbase within one subroutine. They are valid as rbase only witihn that subroutine. This is why you can specify an address range.

You can specify an rbase register definition with or without an address range. This defines that within the specified code range, the register points to the address defined.

When no range addresses are specified, the rbase command defaults to the whole binary. This is shown in the second example, where R76 would be redefined between addresses 2230 and 2350, and the default (i.e. first one) would apply everywhere else.

Bank Commands

These are only necessary where SAD cannot auto decode the bank layout. In most cases the bank command is not required, and SAD detects automatically. The commands are printed in the message file for reference, but are commented out. They are described here if you need to change the order in a multi bank binary. The bank command uses FILE OFFSETS,first and program addresses where necessary. Banks are not always contiguous in the file.

bank 8 0 2000 9fff

This defines a single bank (8) starting at file offset zero (the most common single bank layout), and finishing at 0x9fff .

bank 1 0 12000 1ffff
bank 8 e000 82000 8ffff

This defines a typical 2 bank 8065 layout, bank1 first at offset 0, (= 0x12000 - 0x1ffff), followed by bank8 at offset 0xe000 (= 0x82000–0x8ffff).

bank 0 2000 02000 0ffff
bank 1 12000 12000 1ffff
bank 8 22000 82000 8ffff

bank 9 32000 92000 9ffff

This is an example of a ‘full’ 256K 4 bank binary. There is a 0x2000 gap between the bank offsets, which is for the 0x2000 front filler block which is present in each bank for this binary file. Some other utilities require this format. So for this binary -
Bank 0 starts at file offset 0x2000 (= 0x02000 - 0x0ffff)
Bank 1 starts at file offset 0x12000 (= 0x12000 - 0x1ffff)
Bank 8 starts at file offset 0x22000 (= 0x82000 - 0x8ffff)
Bank 9 starts at file offset 0x32000 (= 0x92000 – 0x9ffff)

Bank Commands for non standard layouts

Occasionally, a binary may have a missing byte at the front, or have an unusual layout.

If a binary has a missing byte, then it effectively starts at 0x2001 instead of 0x2000. This can be represented by changing the parameters.
bank 8 0 2001 .9fff

This defines a missing byte at the front of a binary. As the first bytes are typically 0xff, 0xfa (NOP, DI) then disassembly can still work without a problem, and the first address would be 0x2001.

bank 8 2 2000 9fff

This defines a binary with 2 extra bytes at the front, hence the disassembly starts at file offset 2.

bank 0 0 02000 0ffff
bank 1 e000 12000 1dfff
bank 8 28000 82000 8ffff
bank 9 1a000 92000 9ffff

this defines a multibank with a ‘short’ bank 1. This layout is because the hardware has extra RAM at 1e000 which is not included in the binary file

2.5 Complex commands

Complex commands typically define data structures or subroutine parameters, and often have multiple and different items. See Chapter 4 for more explanations and examples on the data structures. These commands do look very scary at first, but are designed to provide for some complex mixed data lists, but all are made up of a series of data items each with options in them.

A **table** (a 2 dimension lookup structure) has one extra command level, which can be scaled to make sense of the data values. e.g.

table 12579 25f1 "Ign_Advance" [O 11 Y P 3 = float (x/4)]

This command defines a 2 Dimension lookup (a ‘Table’), which -
Exists in bank 1 from 2579 to 25f1, and is named “Ign_Advance”.
It has 11 byte size columns (**O 11 Y**).
It has 11 rows. (defined indirectly by end address, e.g END-START+1 = 0x79, decimal 121, = 11 x 11).
Each data item is divided by 4 ‘ = float (x/4)’ Values here are 1/4 of a degree.
It is printed with a minimum of at least 3 spaces per item (**P 3**).

A **function** (a 1 dimension lookup structure) will have TWO command levels, one for input value, one for output value. By definition this structure has 2 columns, IN and OUT.

func 28cc 2917 [W =float(x/12800)] [W = float(x/12)]
SYM 28cc "VAF_Transfer"

These two commands define -
There is a function from 28cc to 2917, which is named “VAF_Transfer”.
The first column is an unsigned word (**W**) and is scaled with a divisor of 12800**.**
NB. This divisor turns a raw A to D sensor value into “Volts in”) .
The second column is an unsigned word (**W**), and is scaled with divisor of 12 .
The separate SYM command illustrates an alternate way of adding symbols.

A complex data structure.

OK, this is where the commands get to look scary, but it is just the same rules, step by step. This is how to define a list of different items for each ‘row’ in a structure. Here is the A9L injection structure ....

struct 22a6 2355 [R N][O3 Y][= add(x +2c5) Y N][O2 Y][O2 W |][R N][O3 Y]
[Y N = add(x+2c5)][O2 Y][W]

This structure defines items to manage eight injectors, to handle on and off events via queues and their locations for those events.

This command defines the structure as -
1. The data structure starts at 22a6 and ends at 2355.

First item	is a (word) pointer to a subroutine, to be printed as a name if found, or address (**R N**).
Next 3 items	are unsigned bytes, printed in hex (**O 3 Y**).
Next item	is an unsigned byte, and byte is an offset from address 2c5 **= address (x + 2c5) Y N)** which points to a queue in RAM N means “look for Symbol name”
Next 2 items	are unsigned bytes (**O2 Y)**
Next 2 items	are words (**O2 W**). A newline is printed after this data item.
Next item	is a a pointer to a subroutine, to be printed as a name or address (**R N**).
Next 3 items	are bytes (**O3 Y**).
Next item	is a byte, and is an index from address 2c5 “ **= address (x + 2c5)”** which points to a queue in RAM
Next 2 items	are unsigned bytes (**O2 Y)**
Next item	is an unsigned word (**W**).

This complete structure definition then repeats until the end address is reached, as a series of ‘rows’. Each ‘row’ here is 22 bytes in size, which means there are 8 rows (from 0x2355 – 0x22a6, which is (decimal) 176 bytes, divided by row size of 22 = 8 rows)

This A9L structure actually consists of an “ON” and an “OFF” part (and queue) for each of 8 injectors, and the printout is split to show the on and the off as two parts on separate lines (via the ‘|’ char). This structure defines a subroutine to calls to set the On and Off times for each injector, the other items are various bit masks and indexes to keep track of those events, There are eight queues in RAM for those events kept in RAM.

So this is an example of a complex structure definition.

2.6 Symbols.

Sym 82314 “Bap_default”
Sym 15 “VAF_fail” [B 3]
SYM 11 “Data_Bank” [B 4 7]
SYM 11 “Stack_Bank” [B 0 3]

sym 30 82234 82256 “Calc_result”

The SYM command defines a symbol name for the defined address. This address can be any valid one, including registers. SAD will then replace each address reference with its defined name, as makes sense. Symbols can be allocated to any address within the binary address range of the bank(s), so can be anything (a subroutine, a data item, or a bit field).

Symbols below 0x2000 are treated as special in that they do not have bank numbers, on the rule that there is only one register block, one RAM and one KAM area. 8065 register block is treated as 0-0x3ff, 8061 is 0-0xff.

If the option B <n> <n> is included , the name refers to a field within the address. A single B<n> is read as a single bit (or flag) within the address. Bit 7 is the most significant bit, Bit 0 the least. The range of valid bit numbers is 0-15. The SYM 11 are for 8065 bank operations and show the example of two 4 bit fields.

Note. Some EEC binary code interchangeably uses word and byte accesses for the same single flag, because Bit 9 of 0x26 is the same as Bit 1 of 0x27. SAD handles this usage correctly with a single symbol name specified as either SYM 26 [B9] or SYM 27 [B 1].

The symbol can be limited to a defined address range within the code, say for a single subroutine, as per the third example. This allows multiple symbols for the same register for example. If the range is not specified then the symbol defaults to the whole binary. It is legal to define a default symbol with no ranges, and a symbol with a range.

2.7 Subroutine Commands

subr 4326 “Calc_airflow”

This command defines a subroutine at the specified address. If it has no options attached,
it is equivalent to a 'sym' and a ‘scan’ command.

Subroutines can have **arguments** attached, and this format matches the data structure definition above.

Extra rules for subroutines.

Because SAD attempts to autodetect subroutines with arguments and special types (e.g. table lookup) there are a few extra rules for subroutines.

sub 8563
Defined this way, SAD can add or change this subroutines name and any arguments and special types. (see below for special types)

sub 8563 “My_name”
SAD can NOT change this subroutine’s name, but can change any arguments and special types

sub 8563 [Y S][W]
SAD can change this subroutines name, but can NOT change any arguments and special types

sub 83654 "URolav3" [W N O3 = enc(x, 1, e0)]

SAD can change this subroutine's special type but cannot change name or arguments.

Subroutines with arguments

sub 83654 "URolav3" [W N O3 = enc(x, 1, e0)]

This is a real example of one of the A9L's subroutines with embedded arguments. An embedded argument is one which exists in the ROM next to the subroutine call code itself. SAD can decode all arguments by 'emulating' the subroutine code, even complex and variable setups, so manual definition will rarely be necessary. The enc is a built in calc function which decodes Ford style encoded addresses. There are 4 types

This command defines -

A Subroutine is called 'Ufilter3', at 0x3654, and has 3 arguments.

Arguments are data items which are 'attached' to a particular subroutine call, by being embedded in the code immediately after the call itself. Arguments are listed separately, making the subroutine easier to read and comments can be added for any individual line (by its address). The examples are rolling average calculations for RPM.

The three arguments are also **encoded** address, type 1 from register e0, and are named with symbols. SAD decodes the arguments automatically, and adds the calc function. e.g. Arg3 = d04c decodes to 0x97f4 and d05c maps to 9804 in A9L. (see encoded address section below)

Here is what this subroutine call then looks like in the A9L listing, with symbol names resolved

```
3e24: c7,74,21,36         stb   R36,[R74+21]     N_byte = R36;
3e28: ef,29,f8            call  3654             Srolav3T (
3e2b: 08,01                     #arg1               RPM_Filt1,
3e2d: ae,00                     #arg2               Rpmx4,
3e2f: 4c,d0                     #arg3               97f4 );
3e31: c3,72,88,3e         stw   R3e,[R72+88]     RPM_Filt1 = R3e;
3e35: ef,1c,f8            call  3654             Srolav3T (
3e38: 7c,02                     #arg1               RPM_Filt2,
3e3a: ae,00                     #arg2               Rpmx4,
3e3c: 5c,d0                     #arg3               9804 );
3e3e: c3,74,fe,3e         stw   R3e,[R74+fe]     RPM_Filt2 = R3e; }
```

Special subroutine types

The F option, in global options section, has a list of strings and parameters to define that a subrotuine has a special type. Currently there are two special types, function (1D) lookup, and table lookup (2D).

The syntax defines sizes and whether signed, along with 1 or 2 parameters for where the address is held, and the column size.

$ F <string> par1 par2

where string is

string	meaning	parameters
"uuyflu"	unsigned in, unsigned out, byte function (1D) lookup	1
"usyflu"	unsigned in, signed out, byte	1
"suyflu"	signed in, unsigned out, byte	1
"ssyflu"	unsigned in, unsigned out, word function (1D)	1

"uuwflu"	Word sized	1
"uswflu"		1
"suwflu"		1
"sswflu"		1
"uytlu"	unsigned out, byte table (2D)	2
"sytlu"	signed out, byte table (2D)	2
"uwtlu"	unsigned out, word table (2D)	2
"swtlu"	signed out, word table (2D)	2

(no word tables found anywhere yet)

First par is the register holding the data structure address(function or table)
Second par is the register holding the number of columns - tables only

sub 82354 “SUfunlu” $F suwflu 36
sub 85674 “SYtablu” $F sytlu 38 34

SAD will normally detect these automatically, so these commands will rarely be necessary, but are provided for user override.

The first subroutine is named “Sufunlu” and ...

F suwflu 36 This is a function lookup. The data structure address is fed in via R36. The subroutine reads functions with signed values in its input column, and unsigned values in its output column.

The second subroutine is

F sytlu 38 34 This is a table lookup. The table address is fed in via R38, and column size is in R34. The table consists of SIGNED bytes

Note that this defines the SUBROUTINES, NOT the data. If these subroutines are defined, SAD will automatically create the table or function definitions for each address called.

args 3e28 3e2b [WN O2] [W]

This command functions in the same way as the others for data structures, but it defines ONE set of arguments for a SINGLE subroutine call at the specified address – (NOT the subroutine address). This is used for subroutines which have variable arguments, and SAD produces these automatically. This command overrides any automatic processing for that specified address.

3. The calc command

Many data items in the binaries are ‘scaled’ in some way, both in structures and single variables. This scaling is often done in a binary scale, for speed. For example RPM is typically stored as RPM*4 giving a resolution of 0.25 rpm. In some binaries RPM is also stored in other scales too. A to D sensors, e.g. engine temperature use 0-5 volts, which the CPU reads in a scale of 0-12800.

SAD provides for 2 methods of embedding calculations in data items. The ‘short form’ is to have an ‘=’ in the data item, in the form “= calctype <formula>. Where a calculation is used often, user can save space by declaring it as a command by a name, and then referring to it in data items.

calc "Encode1" = address (enc (x, 1, e0))
calc "Encode2" = address (enc (x, 3, f0))

calc “topb” = hex(x/128) # top byte

and then a command such as

WORD 1234 1235 [W = topb] is the same as

WORD 1234 1235 [W =hex(x/128)]

The formula can be nested and multiple terms just like standard math formula e.g.

calc "ntest3" = float (((x / 6.0) + 1.041))

4. The Comments file (xx_cmt.dir)

The comments file allows your comments to be added to any line or inserted between code lines.

The comments file consists of a series of entries of the format -

<address> <comment>

where
address defines the opcode line to which <text> will be added.
Comment a text string which is printed after the address opcode or data printout, and may contain special sequences (see below)

Bank number is embedded into the address in the same way as the commands, so address is 4 digits for a single bank, and 5 digits for a multibank binary.

For example -

2037 # Watchdog Timer reset
2039 # Flip CPU OK and back
204a # ROM Checksum fail
2050 # Checksum segments

These commands will add the comment notes at the end of the relevant lines.
ADDRESSES MUST APPEAR IN CORRECT NUMERICAL ORDER (including the bank number) , or the comment will not be correctly printed.

Special sequences

These sequences allow items to be embedded or special behaviour to be defined to aid comment layout and names. All special sequences begin with a ‘\’ character. These are included to minimise effort if changing symbol names etc.

\n

Prints a newline at that point. This allows a comment text block to be inserted after a certain code line by using a format like this …

```
24b6 \n################################################################
24b6 \n# Load - Base Fuel Adjustment
24b6 \n################################################################\n
```

This will print that text block with extra newline above and below for separation. A ‘\n’ can be used at the end of a comment an extra blank line. To save retyping the main address repeatedly, a '1' can be used for repeat lines in a block, so that

```
24b6 \n################################################################
1 \n# Load - Base Fuel Adjustment
1 \n################################################################\n
```

works exactly that same as the previous definition.

\W

This is a ‘wrap’ option. This is like a newline, but will pad out to the comment column, so multiple lines can be added aligned at the end of the opcode line. For example, the lines

```
244c # Flags (when set)\w# B0 (set)  but not used anywhere ?
244c \w# B2 (clear) Force bank 2 = Bank 1 calculated injection time
```

would produce this output (NB spaces removed to fit in this page, wider in true printout)

```
244c: 09  byte 9   FLGS_B3_B0          # Flags (when set)
                                       # B0 (set)  but not used anywhere ?
                                       # B2 (clear) Force bank 2 = Bank 1 calculated injection time
```

\S is an embedded symbol name derived from its address.

For example, if you have a SYM 70 “RPM” in the directives file, anywhere in the comment text a sequence “\S70” occurs it will be replaced by “RPM”.

Bit flag names can also be printed by using the format \S70:4 for bit 4 of address 70

\1 , \2, \3

This will embed an operand in the comment (correctly sized etc, just like the opcode printout).

\\ will print a single ‘\’ char

any other ‘\x’ sequence will ignore the backslash and print from the ‘x’ character.

\P is the same as \S but auto pads the symbol name to allow for neater layout or columns.

4. Notes and details on data structures

EEC binaries have various types of data structures, from single byte values, through to complex structures like the A9L injector table, referred to above.

Ford have the EEC wide concept of particular structures, including '**table**' and '**function**'. These types

have dedicated lookup subroutines to access them, and those routines include interpolation, which calculates answers for input values which fall between defined points in the function.

A **function** is a 1 dimension data lookup, typically used for scaling purposes, for example to convert an input voltage from a temperature sensor into degrees Centigrade, or the BAP (Baromatric Pressure) sensor into air pressure/air density. Functions are used to remove the need for complex calculations, as these conversions are often not linear. Functions are often used to scale inputs into a row or column number for a later table (2 dimension) lookup.

Functions can be byte or word, and have 4 subtypes, which are around signed unsigned values.
The two columns have the same size (both byte or both word)

The 4 types are –
unsigned in, unsigned out (UU), unsigned in, signed out (US),
signed in, unsigned out (SU), signed in, signed out (SS)

A function therefore always consists of 2 columns, and the input column normally includes the full number range possible (i.e. 0x00 to 0xff for unsigned bytes, 0x7f-0x80 for signed) and the output is often scaled into a range which makes it directly useful for binary calculations.

A **table** is a 2 dimensional block of byte or word data, used for lookup of answers against 2 parameters. A typical example is spark advance, which are often 11 rows by 11 columns, RPM and airflow, and is scaled as degrees*4, so is accurate to one quarter of a degree. The lookup routine also interpolates the answer in 2 dimensions. A table may be signed or unsigned output.

A good example of how tables and functions fit together is the spark advance table lookup. The RPM is first scaled via a function to scale it to 0-10, and then the airflow (or load) is fed through a function to convert it to 0-10, and then these values are used as X and Y to lookup in the table.

Note that the function lookups are not linear, as the ignition timing changes much more quickly for low rpm ranges, for example the AA RPM scaler looks like this

rpm	scaled result	actual value in function
0-700	0	0
1000	1	256
1300	2	512
1600	3	768
2000	4	1024
2500	5	1280
3000	6	1536
3500	7	1792
4000	8	2048
5000	9	2304
6000+	10	2560

Note the result is actually stored as 256 times bigger in the function, by storing value in top byte. This means there are at least two decimal places inferred for the table lookup. This solution is far faster than doing complex maths to get a curved map. If the RPM falls between the lookup points, it is linearly interpolated to get the answer. With careful setting of the lookup points this is almost as accurate as a true curve.

Structures

There are all sort of different structures and lists in the EEC code, used for all sorts of purposes, which is why there is a the complex generic SAD ‘struct’ command to map these into a readable form.

Simple structures are often just vector lists (address lists of subroutines, supported by the command VECT), but can go right up to complex constructions of mixes of data, pointers, bit masks, etc. like the A9L injector structure.

All binaries have a **Timer list,** which is a structure defining a list of registers used to time various events in anything from milliseconds to minutes, and consists of a mix of entries. This also is a data structure.

5. Encoded address types

Many binaries use a form of 'encoded' address in their subroutine arguments, in combination with preassigned 'pointer' registers (rbase). These are decoded with the built in 'enc' calculation function.There are currently 4 encoded address types supported.
More types may be found as more binaries are analysed.
Note that these encoded address do not work unless an **rbase** command is in force for the base register. This will probably be detected automatically by SAD, but can be added manually if necessary.

Enc (value to decode, type, register base)

e.g. enc(x, 1, e0)

Type 1 - If the top bit of the word parameter is set, then take the top 4 bits of the word, multiply it by 2, and use this as an offset for base register lookup, then add lower 12 bits as an offset. If top bit is not set, pass value through unchanged.

If parameter is a040 then lookup e0+14 with offset 40 => [Rf4+40]. If Rf4 is set to 9000 then result is 9040.

Type 3
If the top bit of the word parameter is set, then take the top 3 bits of the word parameter, multiply it by 2, and use this as an offset for the base register, then add lower 12 bits as an offset. If top bit is not set, pass value through unchanged.

If parameter is a040 then lookup e0+2 with offset 40 => [Re2+40]. If Re2 is set to 9000 then result is 9040.

Type 2

Always take the top 4 bits of the word parameter, use this as a plain offset for base register lookup, then add lower 12 bits as an offset.

If parameter is a040 then lookup e0+0x10 with offset 40 => [Rea+40]. If Rea is set to b000 then result is b040.

Type 4

Always take the top 3 bits of the word parameter, use this as a plain offset for base register lookup, then add lower 13 bits as an offset.

If parameter is a040 then lookup Re0+2 with offset 40 => [Re2+40]. If Re2 is set to b000 then result is b040.

- - - END - - -

Command Definition

Address parameters by command. All values are hexadecimal 'x' = not allowed
see below for explanations

Command	par 1	par 2	par 3	par 4	Min data defs	Max data defs
args	start	end	x	x	1	16
bank	Bank no	start	pc_start	pc_end	0	0
byte	start	end	x	x	0	0
code	start	end	x	x	0	0
fill	start	end	x	x	0	0
function	start	end	x	x	2	2
pswset	start	from	x	x	0	0
rbase	register	address	Start range	End range	0	0
scan	start		x	x	0	0
structure	start	end	x	x	1	16
subroutine	start		x	x	1	16
symbol	start	Start range	End range	x	0	1
table	start	end	x	x	1	1
text	start	end	x	x	0	0
vector	start	end	x	x	0	1
word	start	end	x	x	0	1
xcode	start	end	x	x	0	0
setopts	x	x	x	x	See below	
clropts	x	x	x	x	See below	
calc	x	x	x	x	x	x

The min and max data-defs are to define the minimum and maximum number of extra data definition options which can be attached to the command.

command	A text string
[Params]	is 1-4 hexadecimal numbers, typically start and end address
"name"	is an optional text string, to assign a symbol name to the start address
{options}	is an optional set of subcommands which define detail items for the command
Global options	Are for options affecting whole command (e.g. layout options)
: options	Define the structure of a data item (e.g. word/byte, signed/unsigned etc) (multiple)
| options	Define the structure of a data item (e.g. word/byte, signed/unsigned etc) (multiple)

For multibanks, the bank number is embedded into the start and end addresses. See detailed description below.

The valid commands list is given here, and described in more detail below A minimum of the first 3 characters of the command string is required.