SAD User Manual

A Semi Automatic Disassembler Tool for Ford EEC-IV and V EEC controllers
SAD Version 4.0.6 onwards

1. Disclaimer

SAD is not intended for any commercial purpose, and no liability is accepted whatsoever.
SAD works with files which may be copyrighted by other organisations, use it entirely at your own risk.

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

This manual is intended as a quick overview and examples to get you started. For full understanding of EEC_IV and V, consult the various documents around on the web, also Intel 8096, as the 806x is based on this CPU. Also good information in OpenEEC project at “https://github.com/OpenEEC-Project”.

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

1.1 What does this program actually do ?

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, each with perhaps 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.

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 things in this manual are standard 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' just as the standard for most programming languages. All numbers in SAD, commands, input and output files are hexadecimal by default.
'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 has a maximum length of 256 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.

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. (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>” where
-c <path> is to specify the directory location of SAD.ini (optional)
<bin> is the binary filename (can have .bin suffix or not)

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 and 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 can be cut and pasted into the dir file to refine and rerun the disassembly as required.

Example 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. Some have unused ('filler') areas before the code starts, some do not. Nearly all have ‘filler’ areas at the end of each bank. Multibank 8065 files can have differing bank orders. ‘Filler’ areas are typically '0xff' values, but some have other patterns. 8061 binaries are always single bank, 8065 binaries may be 1, 2, and 4 bank.

SAD checks for a certain ‘fingerprint’ to find the true start of each bank, this being a jump followed by the interrupt handler pointers. Only bank 8 has a true jump, the other banks have a ‘loopstop’ jump, which is a jump which goes to itself. If SAD cannot determine the bank order from the interrupt handlers, it defaults to the most likely common orders (see below).

SAD should be able handle most bin files, even those with a missing byte or two at the front. The maximum number of banks is 4, and these banks are numbered 0,1,8,9 by Ford convention. (No obvious reason !) Bank 8 is always where the CPU starts its initialisation.

SAD assumes that -

Min and Max File size		Banks and CPU	Bank Order
Min	Max		
0	64k	8061 or 8065 binary, 1 bank	8 (bank not shown)

64k	128K	8065 binary,	2 bank	1, 8 or 8,1 - auto detected
128k	256K	8065 binary,	4 bank	See next para.

For 4 bank binaries, if SAD cannot determine the true order of the banks, it only knows for sure where bank 8 is, so uses the following default rules, which may not work for all bins. In the case it does not work you must specify the bank and file layout with a user command.

Bank order for 4 bank binaries.

If bank 8 is in found in	assumed order is	
First slot	8, 9, 0, 1	
Second slot	9, 8, 0, 1	
Third slot	0, 1, 8, 9	(this the is 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 where there are missing 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.

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 = 32768;
20d0: 05,82             decw  R82             Timer21_mS--;
20d2: b1,85,0c          ldb   Rc,85           HSI_MASK = 133;
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 = 32767;
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 a psuedo program code, with names and addresses resolved where possible.

The last column is designed to provide a 'fake source code' type explanation of what each instruction does. This column has symbol names, bits/flags, labels, resolved in a reasonably simple form to help user readability. The example above shows a mix of the formats available. If a 'symbol' command has been specified for that address, 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 or 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. An indexed pointer looks like [R30+5ed2], which means “the contents of the address made by adding the contents of register R30 to 0x5ed2”. Indirect pointers are simply [R30], which means “the contents of the address pointed to by the value in R30”. 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 also 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, the 80c196 user manual is useful. The 806x are close relatives of the 8096 CPU range.

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 some extra commands to work correctly.

SAD will not override any command in the file, but will continue to try to do as much as it can automatically. The default rule is that whatever you specify in a command is locked, but other related attributes may be changed if not specified in the command.

For example, defining a table name as a SYMBOL does not specify anything else (e.g its 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.

The basic structure of each command is -

[command text] [Values] [“name”] [$ global] [(: or |) data item] ...

key	min	max	description
command	1	1	A text string of at least 3 characters (compulsory)
[ ]			A group of one or more things
Values	1	5	Hex values, e.g start and end address. Depends on command

“name”	0	1	text string, assigned to a symbol name at the start address
global	0	1	Options affecting whole command (e.g. layout options)
data item	0	16	Defines a data item (e.g. word/byte, signed/unsigned etc)

The characters ‘$’ ‘:’ and ‘|’ have special meaning and can appear ONLY at the beginning of a group of options, and have the following purpose

$	Defines that this is start of global options. $ required as delimiter
:	Defines start of a new data item in the list
|	As colon, but specifies that a newline should be printed before this item to split the list up in the listings file

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	Default	Purpose	Note
default		Set (Clear) default options (as the default column)	
sceprt	x	print ‘fake source code’ for opcodes	
tabnames	x	automatically name new tables	1
funcnames	x	automatically name new functions	1
ssubnames	x	automatically name new special function subroutines	1
8065		set SAD for 8065 cpu instead of 8061	2
D8065		set SAD for later 8065 cpu (Step D) [from SAD 4.0.7]	3
labelnames		automatically add label names for all jumps	1
manual		inhibit ALL automatic analysis, just use user commands	
subnames	x	automatically name subroutines when called	1
signatures	x	look for signatures (special code sequences, eg, table lookup)	
acomments	x	Auto comments	5
sympresets	x	Add preset symbol names for special registers	1,4

Notes.

1. All automatic names can be overridden with SYM commands.
2. 8061/8065 CPU type is auto detected. Default is 8061.
3. Later 8065 CPU has an extra addressing mode.
4. Sad has a preset list of names for special registers , 0-0x10 for 8061, and 0-0x22 for 8065.

5. Auto comments are added to help with code analysis.

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.
C	Use ‘compact’ layout. All data items are printed on the same line, unless a ‘|’ char is used.
Q	This command has a terminator byte. Optionally followed by number 1-3 for size (bytes)
F <text>	Special Subroutine type (e.g. Table or Function lookup)

Data item options

The options consist of one or more letters which define a data item. Some letters have a following parameter, which is DECIMAL. 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 varying 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	
X	none	Flip Print decimal/hex	Flips default format for command
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 3 chars for byte, 5 chars for word.
K	0,1,8,9	Bank number	Used where pointers span banks
O	1-15	Repeat Count	also number of Columns in a TABLE.
V	<float>	Divisor for scaling of values.	Floating point value.
T	0-15	Bit number.	Defines a single bit (e.g symbol command)

Special purpose extensions – see later descriptions for more information on these. Parameters are H for hexadecimal, D for decimal

‘=’	1H	Define an answer from a subroutine in register. Use Y,W,S,U for size.
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)

The D and E options cater for some 'tricks' used in the EEC code -

D	Offset Address (hex)	This is to define a value which is ‘base address’ + this value. Used as an ‘offset pointer’ in subroutine arguments and some structures
E	Encoded address. (Decimal, Hex)	There are several types of encoded address. This again is a type of pointer index which is converted to a real address via a register base pointer.

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 V 12800 : W V 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 unisgned word, and is scaled at 65536 = 5.12, which makes it an Analogue to Digital value as read by the CPU. The output column is also unsigned word, and is scaled at 128, or in other terms, the top byte of the word. This top byte technique is often used as a kind of extra precision calculation in the code.

Parameters and addresses

Addresses and bank number.

For multibank binaries, addresses are 5 digits, with the bank number being the first digit. In commands, the end address is assumed to have the same bank as the start address if not specified (i.e. is 4 digits), but bank MUST match if you include it with 5 digits. Where a command requires only a start address (e.g SYM), then 'end' address can be left out.

A single bank binary uses 4 digits, as there is no concept of banks. Internally, SAD treats a single bank as bank 8, as this allows 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 of a command, for example WORD 2400 241F defines 16 words . Any additional data definitions (e.g. a divisor) will apply to all 16 words.

For tables, this allows a definition of the number of columns (with ‘O’ as repeat value) without defining the number of rows, as this can be done 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.

Simple Command Examples

Simple commands have zero or a few option letters and are straightforward.

code 15688 15698 means code exists from bank 1 5688 to bank 1 5698
code 15688 5968 is equivalent

code 82000 82003 means code exists from bank 8 2000 to bank 8 2003
code 2000 2003 is equivalent for a SINGLE BANK binary only, otherwise **this means Bank 0**

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

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 in bank 1 at 0x3100, but the subroutines pointed to are in bank 8.
This is not unusual for a multibank binary

WARNING !! Please be careful if you override the default bank definitions as incorrect commands may cause unpredictable behaviour and possibly 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 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 2230 and 2350)

Many binaries use a set of defined registers as permanent (and fixed) 'base' pointers, and then use the index mode of instructions to get at the data. This command allows SAD to decode the index to produce a true absolute address, and add 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) also use these rbase registers.

Many binaries also set other registers as permanent and temporary pointers into RAM and KAM. SAD attempts to detect and confirm these too, but is not guaranteed to get them right.

You can specify an rbase register definition with or without an address RANGE. This is for use when a register is used as a temporary pointer within an area of code, say for a single subroutine. This tells SAD that within that code range, the register points to the address defined.

When no 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. The commands are printed in the message file for reference, but are commented out. They are included here if you need to change the order in a multi bank binary. The bank command uses FILE OFFSETS, not program addresses. Banks are not always contiguous in the file.

Bank 8 0

This defines a single bank 8 starting at file offset zero (the most common single bank layout). Sad will automatically calculate the bank's end address from the file size

bank 1 0
bank 8 e000

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
bank 1 12000
bank 8 22000
bank 9 32000

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. 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 adding extra parameters.
Bank 8 0 2001

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. This is typically detected automatically by SAD.

Bank 8 2 2000

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

Bank 8 0 2000 dfff

this defines a bank which is not full size, and stops at 0xdfff

2.5 Complex commands

Complex commands can define data structures or subroutine parameters, and may 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, and are designed to provide for some complex mixed data lists, but all are made up of items separated by ':' or '|' and options to define each item.

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 V 4 P 3

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 by end address, e.g END-START+1 = 0x79, decimal 121, = 11 x 11).
Each data item is scaled with a divisor of 4 (**V 4**), (values here are 1/4 of a degree)
It is printed with 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 V 12800 : W V 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 (**V 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 (**V 12**).
The separate SYM command illustrates an alternate way of adding symbols.

A (complex) data structure.

OK, these commands can look truly scary, but 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 A9L injection structure ....

struct 22a6 2355 : R N : O3 Y : D2c5 Y N : O2 Y : O2 W | R N : O3 Y : D2c5 Y N : O2 Y : W

This is the REAL A9L 'injector structure', which defines its eight injectors in a data structure to handle various on and off events and queue location for those events.

This command defines the structure as -
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 is an index from address 2c5 (**D 2c5 Y N)** which points to a named queue in RAM
Next 2 items	are unsigned bytes (**O2 Y)**
Next 2 items	are words (**O2 W**).
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 (**D 2c5 Y N)** which points to a named 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 kind of 'row'.
Each 'row' 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 for each of 8 injectors, and the printout is split to show the 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 bytes in between are various bit masks and indexes to keep track of those events, and there are two 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 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> is included , the name refers to that single bit (or flag) within the address. Bit 0 is the most significant bit, Bit 7 the least. The range of valid bit numbers is 0-15.

Note. Defining symbols with bit number greater than 7 can cause name overlaps, as two consecutive bytes make up a word. 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 26 :B9 or 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 E 1 e0 O 3
SAD can change this subroutine’s special type but cannot change name or arguments.

Subroutines with arguments

sub 83654 "URolav3" : W N E 1 e0 O 3

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.

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. 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

(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 : 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 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 data, (no word ones found so far at least), 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). 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.

Command = E 1 e0

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.

Command = E 3 e0

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.

Command = E 2 e0

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.

Command = E 4 e0

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	end	x	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
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
timer	start	end	x	x	1	1
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	
option	x	x	x	x	deprecated	

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.

opts	SAD generic options (see below)
rbase	this register is a 'base + offset' type pointer, calibration pointer, etc. (see later)
scan	scan from here as a block of code to be analysed
vect	a list of pointers to subroutines (= “vector list”)
code	this is actual code, opcode instructions
xcode	this block is defined as data only, code and scans are not allowed here
bank	Manually define a bank within the file
symbol	defines a name for an address (optionally with bit flag and ‘address range’)
subr	defines a subroutine, its name, and any parameters/arguments
args	defines a set of arguments for one specific subroutine call
fill	default filler data (typically 0xff)
byte	byte data
text	character data
word	word data
table	a byte data table (2 dimensional, probably scaled)
func	a byte or word 'function' (1 dimensional, probably scaled)
struct	a data structure, which can be of variable format
timel	a timer list – early types only at present – under development