SAD

A Semi Automatic Disassembler Tool for 8061 and 8065 microcontrollers
(Ford EEC-IV and V)
Up to SAD Version 4.0.5

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, therefore use it entirely at your own risk.

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

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 8061 and 8065 single and multi-bank binary files.
It attempts to produce an output listing with code and data correctly analysed. SAD can make automatic choices or can be set to be entirely manual in its processing. The output is not intended to be strictly 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 its findings in a warnings file, which includes the outputting of the commands it was able to deduce. These commands can then be cut and pasted into a directives file and edited/improved by hand, and SAD rerun over again in an iterative process.

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 are standard English, but a few items here explained for clarity.

Numbers in hexadecimal are written with the prefix '0x' in this guide as is the standard for most programming languages. All numbers in 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 and its path have a maximum length of 256.

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 (terminal run) command line program, but has a ‘wrapper’ called SADwin to provide a GUI interface for easy use. (Windows only at present)

Windows
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)

Linux
same equivalent command,
“SAD -c <path> <bin>

SAD then opens these files if they exist, in the paths specified by SAD.ini (or in same directory), where xx is binary file name without the .bin -
xx_dir.txt directive file, which has commands in it.
xx_cmt.txt comments file, which has comments in it.
Both these files are in plain text, and their format is described later in this document.

SAD produces two output files, which it will overwrite if they already exist.
xx_lst.txt the disassembled listing
xx_msg.txt for information and messages from disassembly process.

The messages/warnings file includes a list of commands deduced during the disassembly at its end. This list can be cut and pasted into the dir file to refine the disassembly if necessary.

Example file names for 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 format

There are several different binary file layouts in common use. Some have 'filler' areas before the code, some do not. Nearly all have ‘filler’ (unused) areas at the end of each bank. Some .bin files have different bank ordering. SAD handles these different orders by checking the code and interrupt pointers at the likely beginning points of each bank. ‘Filler’ areas are typically '0xff' values, but some have other patterns.

SAD also checks the front of each bank, as some have a filler block at the front. SAD should be able handle most files, even those with a missing byte or two at the front. The maximum number of banks is 4, and these are numbered 0,1,8,9 (yes, no obvious reason, this is the Ford convention). Bank 8 is ALWAYS the bank where the CPU starts at initialisation.

SAD assumes that -

File size		Layout 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
128k	256K	8065 binary, 4 bank	See next para.

Bank order for 4 bank binaries
SAD cannot easily determine the true order of the banks, it only knows for sure where bank 8 is.

SAD uses the following rules, which may not work for all bins. In that case you can specify the bank order with a user command.

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 checks for a certain 'fingerprint' to finalise the start of each bank, this being a jump followed by the interrupt pointers. Only one bank has a true jump, other banks have a 'loopstop' jump, which is a jump which goes to itself. This is how SAD knows where to put bank 8, and start scanning from there

SAD then maps each bank to start at 0x2000, and continue to a maximum of 0xFFFF.
SAD then checks for fill data (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; }
```

Multi banks 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 'C' like code equivalent, with names and addresses resolved where possible.

The last column is designed to provide a '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 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. For example, [2d6a] means “the value in address 2d6a”. Many 806x opcodes support pointer types, both single 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” 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 for you.

There is also a '++' suffix, which means 'increment R26 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 disassembly. These commands can be used to help and override or guide 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 work automatically and corectly, but the binary files are simply too complex in some cases, requiring some directives to work correctly. SAD will not override any directive specified in the file, and will continue to try to do as much as it can automatically. This way, the directive file can define and override only where necessary to do so. SAD will work happily with no directives at all, and development work continues to make as much as possible fully automatic.

The basic structure of each command is -

command start end “name” : options : options

The first command item is compulsory, the other parameters are as required for each command.

where -

command	is one of the list below
start	is the start address, in hex
end	is the end address in hex
“name”	is a text string, assigning a symbol name to the start address, in double quotes
options	is a set of subcommands which define detail items like size, signed/unsigned etc. and can repeat up to 16 times, separated by colons.

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

2.1 Opts Command

The **opt** command does not quite conform to the standard layout as described above. It consists only of option letters. It typically is the first command in the file

format is **opt : <options>** example **- opt : L N P S X H**

The letters are used to define -

H Use 8065 interrupts, registers, and instruction set (allowed with a single bank binary)

L Auto create and name jump labels.
Each jump destination will be named 'L_n', where n is a number.

M Manual only mode. No automatic scans are attempted
No automatic naming or processing done. Only command in dir file are followed.

N Automatically name the interrupt functions (from vector pointers at 0x2010 onwards)
The names are preset to match the hardware interrupt sets for 8061 and 8065 CPU.
If the subroutine is determined to be a dummy, then its name is replaced by "Ignore".

P Automatically name new subroutines.
Each new subroutine encountered will be named as 'subn', where n is the address.

S Do a 'signature scan' for certain subroutine types. (see later for ‘signatures’)

F Automatically name function and table names.
Names are ‘funcn’ or ‘tabn’ where n is its start address

G Automatically name special subroutine names – at the moment these are table and function lookup routines as detected by SAD

C Print pseudo source code after opcode

D Print operand values in decimal (not addresses) Default is Hex.

Notes -

1. SAD automatically names the base registers (0 – 0x11 for 8061, 0-0x23 for 8065) with the commonly used names.
2. All names (including labels, and subroutines) can be overridden with 'sym' commands.
3. What is a 'signature' ? (Option S)
   Some subroutines and structures have common code over much or all of the EEC range. Examples are the table lookup and function lookup subroutines. These signatures are a kind of 'fingerprint' which can be detected by SAD. The signature scan has a set of pattern matches to help find these subroutines and identify them. These functions are then identified as special types, which can also be done manually via the F option (see later)

The default option setting (i.e. no opt command specified) is P N S C F G.

2.2 Command Structure

All commands conform to the standard format, but some have extra (or fewer) parameters

command start end "name" : options : options

where -

start	is the start address, in hex
end	is the end address in hex
"name"	is a text string, assigning a name to the start address, quotes are optional
options	is a set of subcommands which define a list of items each with details like size, signed/unsigned etc. and can repeat up to 16 times.

Example - **func 28cc 2917 :W V 12800 : W V 12**

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

Addresses and bank number.

For multibank binaries, addresses are 5 digits, with the bank number being the first digit. The end address of a command does not have to have a bank number specified, as it is carried over from the start address, but it must match if you include it.

A single bank binary has 4 digits shown, as there is no concept of a bank, but internally SAD treats this as a single bank 8, as this allows the same analysis process for all binaries.

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

Simple Examples

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. Where a command requires only a start address (e.g SYM), then 'end' address can be left out.

Options structure

The options consist of letters which define a list of data items. Not all options are valid for every command. The options are specified in a list, separated by a colon, to define each item within a structure. This can get quite complex (and horrid!) to read, but it gives a lot of flexibility for each command.

Meanings of each letter -

Y	Item is byte sized	default
S	Item is signed	
U	Item in unsigned	default
W	Item is word sized	default with WORD command
X	Print item in decimal	default is hex, except in data structures
R	Item is a pointer	e.g. to a subroutine
N	Look for a symbol name for item	Show name if found.
P <n>	Minimum print width of each item (in chars).	Use this when items don't line up. Default 3 chars for byte, 5 chars for word.
K <n>	Bank number	Used where pointers change banks
O <n>	Repeat Count	and number of Columns in a TABLE.
V <n>	Divisor for scaling of values. Floating point.	
T <n>	Bit number for single bit flag names	symbol command only
Q	This structure has a ‘terminator’ byte	typically 0 or 0xff

Special purpose extensions – see later descriptions for more information on these.

D <n>	A pointer or value with a fixed address offset added.
E <n> <n>	An encoded address type. (type, base register)
F <n>	Special Subroutine type (e.g. table or Function lookup)

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

D This is typically in a subroutine argument, where a list as accessed by an index value

E Encoded address. There are several forms of encoded address. This again is a type of index used in a subroutine 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.

2.3 Simple commands

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

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 are carried out automatically by SAD, these are the standard interrupt handling subroutines in all binaries, and the code start is at 0x82000. Multibanks have a interrupt vector list in every bank.

vect 13100 3160 : D 8

This command defines a vector list in bank 1 at 0x3100, but the subroutines pointed to are in bank 8.

WARNING !! Please be careful if you override the 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 overunning 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 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

Many binaries use 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.

Many binaries also set registers as pointers into RAM and KAM. SAD attempts to detect and confirm

these too, but is not guaranteed to get them right, as they may only be used as a temporary pointer.

You can also specify an rbase register definition within an address range, say for a single subroutine. 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. They are included here for completeness and if you need to change the order in a multi bank binary. The bank command FILE OFFSETS, not program addresses. Bank definitions do not have to be 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 deduce the end address from the file size

bank 1 0
bank 8 e000

This defines a typical 2 bank 8065 layout, bank1 first (0x2000 - 0xffff), followed by bank8 (0x2000–0xffff).

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.

The complete bank command allows for 4 variables, but rarely would you have to use these.
bank 8 0 dfff de56
where 8 is bank number, 0 is the file offset, dfff is the end of the file slot (which would give adresses 0x2000 to 0xffff) and de56 is the start of the ‘filler’ at the end of the slot.

2.5 Complex commands

Complex commands can define data structures or subroutine parameters, and may have multiple items. See Chapter 4 for more explanations and examples on the data structures. These commands do look scary at first, but are designed to provide for some very complex mixed data lists, but are made up of simple steps and ‘levels’. Each step, or data item, is separated by a colon.

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

table 12579 25f1 "Ign_Advance" :O +11 Y V 4 P 2

This command defines a 2D 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 the end address).
Each data item is scaled with a divisor of 4 (**V 4**), so values here are 1/4 of a degree
It has is printed with at least 2 spaces per item (**P 2**).

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 **data structure.**

OK, Yes, these commands can look truly scary, but just the same rules, step by step.

struct 22a6 2355 "InjTTab" : RN : Y O6 : W : W : RN : Y O6 : W

This is a real data A9L structure. This is the 'injector table', which defines its eight injectors in a structure to handle various events and timing.

This command defines the structure as -
The data structure starts at 22a6 and ends at 2355 (bank 8).
First item is a pointer to a subroutine, to be printed as a name if found, or address (**R N**).
Next 6 items are byte items, printed in hex (**Y O 6**).
Next item is a word, (**W**).
Next item is a word (**W**)
Next item is a a pointer to a subroutine, to be printed as a name or address (**R N**).
Next 6 items are bytes (**Y O 6**).
Next item is a word (**W**).

This complete structure definition then repeats until the end address is reached.

This real A9L structure is actually split into an “ON” and an “OFF” for each injector, the two subroutines schedule on and off events, and the bytes in between are bit masks and indexes to keep track of times, a set for on and a set for off. Check out the A9L listing to see how this works in detail.

2.6 Symbols.

Sym 82314 “Bap_default”
Sym 15 “VAF_fail” : T 3

sym 30 82234 82256 “Calc_result”

The SYM command defines a symbol name or label at the defined address, including registers. SAD will then replace each address to its defined name automatically, as makes sense. Symbols can be allocated to any address within the binary address range(s), so can be a simple label, 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 T <n> is included , the name refers to that single bit (or flag) within the address. Bit 0 is the most signficant 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 opcodes 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 name specified.

The symbol can be limited to a defined address range within the code, say for a single subroutine, as per the third example. If the range is not specified then the symbol defaults to the whole binary.

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 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 now decode all arguments by ‘emulating’ the subroutine code, even complex and variable setups, so will rarely be necessary.

This command defines -

A Subroutine is called 'Ufilter3', at 0x3654, and has 3 arguments.
The three arguments are **encoded** address, type 1 from register e0, and are named.

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

```
3e24: c7,74,21,36       stb   R36,[R74+21]    N_byte = R36;
3e28: ef,29,f8          call  3654            URolav3(RPM_Filt1, Rpmx4, 97f4);
3e2b: 08,01,ae,00,4c,d0 #data
3e31: c3,72,88,3e       stw   R3e,[R72+88]    RPM_Filt1 = R3e;
3e35: ef,1c,f8          call  3654            URolav(RPM_Filt2, Rpmx4, 9804);
3e38: 7c,02,ae,00,5c,d0 #data
```

```
3e3e: c3,74,fe,3e       stw   R3e,[R74+fe]    RPM_Filt2 = R3e;
```

This makes the subroutine far easier to read and show these are rolling average calculations for RPM.

SAD decodes the arguments via (E flag)- d04c maps to 97f4 and d05c maps to 9804 in A9L.

Special subroutine types

sub 82354 “SUfunlu” : F 1 36 : SW : UW :
sub 85674 “SYtablu” : F 2 38 34 : SY

There are only two ‘special type’ subroutines at the moment, for function and table lookup. If these are defined this way by command, then when they are called in the code SAD will automatically define a table or function for you. SAD will normally detect these automatically, so these commands will rarely be necessary, but are provided for user override.

The first subroutine is
F 1 36 This is a function lookup. Its address is fed in via R36.
: SW : UW Subroutine is used for WORD functions, first column (IN) is signed, 2nd (OUT) column is unsigned.

The second subroutine is
F 2 38 34 This is a table lookup. Its address is fed in via R38, and column size is in R34.
: SY Subroutine is used for Byte tables, values (OUT) are signed.

args 3e28 : WN O2: W

This command functions in the same way as the others for data, but it defines ONE set of arguments for a SINGLE subroutine call at the specified address.
This is used for subroutines which have variable arguments, and is typically what SAD produces automatically. This command overrides any others at the 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> <# or |> <comment text>

where
address defines the opcode line to which <text> will be added.
or | # is a printed comment delimiter in listing, | means 'print a newline'

bank number is embedded into the address in the same way as the commands.

The comment is added at the end of the printout's line.

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.

The comment function includes a few short cut/special characters used to aid layout and names.

A 'pipe' character (|) indicates a 'newline' in the text. This allows a text block to be inserted after a certain code line by using a format like this …

```
24b6 ||################################################################
24b6 |# Load - Base Fuel Adjustment
24b6 |################################################################|
```

which will set a block with extra newlines above it for separation, and '|' char can be used at the end also for an extra blank line. To save retyping the main address a '1' can be used for repeat lines in a block, so that

```
24b6 ||################################################################
1 |# Load - Base Fuel Adjustment
1 |################################################################|
```

works just as well.

An @ character is used to indicate a value to decode to a name. For example after a command SYM 70 “RPM” in the directives file, anywhere in the comment text an “@70” occurs it will be replaced by “RPM”. An '@' character causes names to be padded out to 21 characters to allow for neat layouts, An '$' character decodes but does not pad the name. The construct @70:4 or $70:4 can be used for a specific bit name.

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 often 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 also 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 (2D) 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 !), 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.

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 typically not linear, as the ignition timing changes much more quickly for low rpm ranges, for example the AA scale looks like this

rpm	scaled result	actual value stored
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

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 table.

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

There are currently 4 encoded address types supported.
More types may be found as more binaries are analysed.

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 => [f4+40]. If f4 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 => [e2+40]. If e2 is set to 9000 then result is 9040.

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 2 e0

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

If parameter is a040 then lookup e0+0x10 with offset 40 => [ea+40]. If ea 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 an offset for base register lookup, then add lower 13 bits as an offset.

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

- - - END - - -