SAD

A Semi Automatic Disassembler Tool for 8061 and 8065 microcontrollers
Version 0.2

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 of a particular vehicle works via the algorithms and data revealed. It is not intended as a tool which provides directly for reassembly of modifed code. 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 SAD actually do ?

SAD attempts to automatically disassemble a binary file taken from an EEC-IV or EEC-V engine management system. SAD can analyse both single bank 8061 and multibank 8065 binaries. 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 help with analysis and understanding of the data and code structures.

SAD reports its findings in a warnings file, outputting 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 to help understanding of the how this tool works.

1.2 Binary File format

There are several different file formats in use. Some have 'filler' areas before the code, some do not. Some binaries have different bank ordering. SAD handles these different types by checking the probable beginning points of each bank. It should handle most file formats.

SAD assumes that -

Filesize		Bank Layout	Bank Numbers
Min	Max		
0	64k	8061 binary, 1 bank	8, but not shown
64k	128K	8065 binary, 2 bank	1, 8
128k	256K	8065 binary, 2 bank	0, 1 ,8, 9, or 0,1,9,8)

SAD then maps each bank to start at 0x2000, and continue to a maximum of 0xFFFF.
SAD then checks for fill data at the end.

1.3 Input and Output files

SAD must have a binary EEC file as a minimum.

SAD then opens these files if they exist -
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 file includes a list of commands deduced during the disassembly. This list can be cut and pasted into the dir file to refine the disassembly if necessary.

Example file names if input was A9L.bin 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 Output format

This is a typical section of output (8061)

```
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 printed also -

```
1 ad48: 89,66,06,3c       cmpw  R3c,666
1 ad4c: d6,15             jge   ad63           if (R3c < 666)  {
1 ad4e: 07,3c             incw  R3c            R3c++;
1 ad50: 07,3e             incw  R3e            R3e++;
1 ad52: 05,40             decw  R40            R40--;
1 ad54: c3,37,2a,08,3c    stw   R3c,[R36+82a]  [R36+82a] = R3c;
1 ad59: c3,ea,be,40       stw   R40,[Rea+be]   [Rea+be] = R40;
```

```
1 ad5d: c3,dc,b2,3e       stw   R3e,[Rdc+b2]    [Rdc+b2] = R3e;
1 ad61: 20,03             sjmp  ad66            goto ad66; }
1 ad63: 91,20,46          orb   R46,20          R46 |= 20;
```

1st column is the binary address, in hex (with 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 resolved, loops identified etc.

The last column is designed to provide a 'source code' explanation of what each instruction does. This column has names, bits/flags, labels, resolved in a reasonably simple form. 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.

The numbers appearing in square brackets are pointers. For example, [2d6a] means “the value in address 2d6a”. Several 8061 opcodes support pointer types, both direct and indexed. An indexed pointer looks like [R30+5ed2], which means “the contents of the address made by adding the contents of register 30 and 0x5ed2”

There is also a R26++ type format, which means 'increment R26 after it is used'. This increment is dependent upon mode, so it will increment by 2 for word opcodes, and one for byte opcodes.

All references, including pointer structures are also resolved into names where possible.
Users are recommended to have a copy of the 80c196 user manual for full description of the opcodes and their operation. The 8061 and 8065 are close relative 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 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, 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 with no directives at all, and work continues to make as much as possible fully automatic.

The basic structure of each command is -

command start end bank “name” : options : options

The first command item is compulsory, the other parameters are as required for each command.
Bank ident is supported in all commands except 'opts', and is not allowed for single bank binaries.

where -
command is one of the list below
start is the start address, in hex
end is the end address in hex

bank	is the bank – multibank files only
“name”	is a text string, assigning a name to the start address, in 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.

The valid commands ist is given here, and described in more detail below

opts	SAD decoder options
bank	file bank definition (automatically calcuated if not specified)
rbase	this register is a 'base + offset' type pointer, calibration pointer, etc.
cscan	scan this as a block of code to be analysed
vect	a list of pointers to subroutines
code	opcode instructions
xcode	this block is reserved as data, scans and code are not allowed here
symbol	defines a name for an address (optionally with bit flag)
subr	defines a subroutine, its name, and any parameters/arguments
args	defines a set of arguments at one specific subroutine call
fill	default filler data (0xff)
byte	byte data
text	character data
word	word data
table	a byte data table (2 dimensional, scaled)
func	a byte or word 'function' (1 dimensional, scaled)
struct	a data structure, which can be of variable format
timel	a timer list – early types only at present – under development

2.1 Opt Command

The **opt** command is the only comand which does not 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 : 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 naming or processing allowed.

N Auto 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_Int".

P Auto name new subroutines.
Each new subroutine encountered will be named as 'subn', where n is a number.

S Do a 'signature scan' for certain subroutine types.

X List all numbers in unsigned decimal by default, hex otherwise

Notes -

1. SAD automatically names the base registers 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 subroutines form part of SAD's techniques used to find the table and function data structures. There are subroutines which use embedded arguments and sometimes even variable lists, which can interfere with correct disassembly. 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 X S.

2.2 Command Structure

Most commands (excepting 'opt' and 'bank') conform to the format

command start end bank "name" : options : options

where -

start	is the start address, in hex
end	is the end address in hex
bank	is the bank ident (default = 8)
"name"	is a text string, assigning a name to the start address, in 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.

If name is specified, 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. Bank is not allowed for single bank binaries, and is optional, defaulting to 8.

Options structure

The options consist of letters which modify the basic command. Not all options are valid for every command. The options are specified in 'levels' , separated by a colon, to define a 'level' within the command. This can get quite complex to read, but it gives a lot of flexibility for each comand. More explanation later

Some letter options have number values following them. Values are unsigned hex by default, but can be preceeded by a '+' or '-' to redefine them as signed decimal. The 'V' option (divisor) can also have 2 decimal places in the number.

Meanings of each letter -

Y	Items are byte sized	(default)
S	Items are signed	(default is unsigned)
W	Items are word sized	(default with WORD command)
X	Print value(s) in decimal	(default is hex, decimal if OPT : X used)
R	Item is a pointer to a subroutine	
N	Item is a named pointer, look for a symbol name.	
P <n>	Minimum print width of each item (characters). Use this when items don't line up neatly in tables. The default is 3 chars for byte entries, and 5 chars for word.	
O <n>	Number of Columns in a TABLE, or repeat cOunt	
V <n>	Divisor – for scaling of values. May have up to 2 decimal places.	
T <n>	Bit number for single bit flag names – 'sym' command only	

Special purpose extensions – see later descriptions for more info.

D <n>	A pointer with a fixed offset added. Also used as second bank reference.
E <n> <n>	An encoded address type. (type, base register)
F <n><n><n>	A special subroutine type (type, par1, par2)

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

D This is normally in a subroutine, 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.

F The disassembler automatically detects certain special types of subroutines to aid analysis. This option allows for manual override of these special types

The venerable 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 a zero or 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 3000 3100 8	The data between 0x3000 and 0x3100 bank 8 is all words (16 bit).
code 3000 3100 8	The data between 0x3000 and 0x3100 bank 8 is code instructions.

2.4 Less Simple Comands

These commands are still simple to write, but do more !

cscan 2000 8

This command tells SAD to do a code 'scan' from this address and bank upwards. SAD will

decode each opcode instruction from here, and track jumps, subroutine calls, and data accesses to sort code from data. This is how the automatic analysis process works, and is the heart of the automated disassembly process.

vect 2010 201f (for 8061)
vect 2010 205f 8 (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.

The 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 0x2000. Multibanks have a interrupt vector list in each bank.

vect 3100 3160 1 : D 8

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

bank 0 7fff

This defines a single 8061 file size, as zero to 0x7fff, which maps to program addresses 0x2000 – 0x9fff. For an 8061 binary, the bank id is not required, it defaults to 8. The bank command always specifies FILE offsets, not program addresses

bank 0 dfff 1
bank e000 1bfff 8

This defines a typical 2 bank 8065, bank, with bank1 (0x2000-0xffff) as bank 1 first, followed by bank8 (0x2000–0xffff), neither bank has a front fill block.
NB. Bank definitions do not have to be contiguous.

bank 2000 ffff 0
bank 12000 1ffff 1
bank 22000 2ffff 8
bank 32000 3ffff 9

This is an example of a 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.

WARNING !! Please be careful if you override the bank definitions as incorrect commands may cause unpredictable behaviour and possibly crashes.

xcode 3000 3010 1

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

into this area are to be regarded as illegal, and to be ignored.

rbase 76 4080

Many binaries use defined registers as permanent fixed data '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 the relevant symbol name, if there is one. SAD will normally detect the most commonly coded 'calibration pointers' (0xf0 – 0xfe) and set these automatically.

Many binaries also set registers as pointers into RAM and KAM, but SAD has no way to detect and confirm this reliably, so this command is provided to set them where required.

2.5 Complex commands

Complex comands define more complex data structures or subroutine parameters, and may have multiple levels and letter and number groups. See Chapter 4 for notes and examples on the data structures. These commands do look scary, but provide for some very complex data types, and are made up of simple steps.

A table will have one command level, and can be scaled to make sense of the data values

table 2579 25f1 1 "Ign_Adv" :O +11 Y V +4 P +2

This command defines a table which -
Exists in bank 1 from 2579 to 25f1, and is named “Ign_Adv”.
It has 11 byte size columns (from **O +11 Y**) , and 11 rows. (rows are calculated from end address and columns)
Each datat entry is scaled with a divisor of 4 (**V +4**) and has a minimum print width of at least 2 spaces per item (**P +2**).

A function will have TWO command levels, one for input value, one for output value.

func 28cc 2917 :W V +12800 : W V +12
SYM 28cc "VAF_TFR"

These two commands define -
There is a function from 28cc to 2917 (In bank 8) , which is named “VAF_TFR”.
The first column is a word (**W**) and scaled with a divisor of 12800 (**V +12800** - this turns the values into “Volts-IN” for an A to D sensor convertor) .
The second column is word (**W**), and is scaled with divisor of 12 (**V +12**).
The separate SYM command illustrates an alternate way of adding symbols.

A structure these commands can look truly scary !!

sym 22a6 "InjTTab"

struct 22a6 2355 :R X N:Y X O +6 : W P +1:W X: R X N:Y X O +6 : W P +1

Here's a real example of a data structure. This is the A9L 'injector table', which defines its eight injectors in a neat way, so the code only needs the injector number to do the input/output.

This command defines -
A 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 or address (**R X N**).
Next 6 items are bytes, printed in hex (**Y X O +6**).
Next item is a word, printed in decimal with an extra space (**W P +1**).
Next item is a word, printed in hex (**W X**)
Next item is a a pointer to a subroutine, to be printed as a name or address (**R X N**).
Next 6 items are bytes, printed in hex (**Y X O +6**).
Next item is a word, printed in decimal with an extra space (**W P +1**).

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, etc. Check out the A9L listing to see how this works in detail.

2.6 Symbols.

Sym 2314 8 "Bap_default"
Sym 15 "VAF_fail" : T +3

The SYM command defines a symbol name at the defined address. SAD will then replace each address to its defined name automatically, as makes sense. Symbols can be allocated to any address within the address range(s), so can be a simple label, a subroutine, a data item, or a bit field.

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 useage correctly with a single name specified.

2.7 Subroutine Commands

subr 4326 "Calc_airflow"

This command defines a subroutine at the specified address. If it has no options attached, it's really identical to a 'sym' command.

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

subr 3654 "URolav3" : W N X O +2: W N X E 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. It can be difficult to decode these automatically in all circumstances, especially where the number of argumants can be variable (SAD does do this correctly in many cases, but some complex chained calls defeat it).

This command defines -

A Subroutine is called 'Ufilter3', at 0x3654, and has 3 arguments.
First two arguments are word, and named (**W N X O +2**).
Third argument is an encoded address, type 1 from register e0, and is named.

Here is what this subroutine call then looks like in the A9L listing

```
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 see that these are two rolling average calculations for RPM.

SAD decodes the last argument to a true address (the 'E' option)- d04c maps to 97f4 and d05c maps to 9804 in the A9L.

args 3e28 : W N X O +2: W X

This command functions in the same way as the subroutine command above, except that it defines a set of arguments for a SINGLE subroutine call at the specified address only.
This is used for subroutines which have variable arguments.
This command overrides any subr command for the specified address only.

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 -

<bank> <address> <# or |> <comment text>

where

bank	is optional and defaults to 8.
address	defines the opcode line to which <text> will be added.
# or |	# is a printed delimiter, | means 'print a newline'

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 , 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 repat 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.

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, 0x80-0x7f 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 structures 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 x airflow, and is scaled as degrees*4, so is accarate to one quarter of a degree. The lookup routine also interpolates the answer in 2 dimensions.

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, 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 included 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 generic SAD command to map these into a readable form.

Simple structures are often just address lists (vector 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, and all 8061 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 2 and 4 byte entries (byte addresses) , or 3 and 5 byte (word addresses).
Later 8065 binaries have a shortened form of this list too.

5. Special Subroutine Types

SAD attempts to recognise particluar subroutine types automatically, but sometimes a manual setup may be necessary. This can be useful when binary is tricky to decode.

All subroutine types are implemented via the F <n> <n> <n> option.

List of types (shown as option format)

: F 1 32 1	a lookup subroutine with **function** address in register 31, byte size
: F 1 32 2	a lookup subroutine with **function** addr in register 31, word size
: w : w: F 1 1 2	a lookup subroutine with **function** addr in first embedded param, word size
: F 2 40 32	a lookup subroutine with **table** address in register 40, col size in Reg 32

: F 2 40 -11 a lookup subroutine with **table** address in register 40, fixed col size of 11
: F 3 39 2 a subroutine with variable embedded word params, number in reg 39

Subroutine type 1 is a **function lookup subroutine**. The first parameter is the word register (in hex) where the function's address is loaded before the subroutine gets called. The second parameter is the entry size (1 or 2 for byte or word).
Examples above state that function address is set in register 0x32, for both byte and word funcs.

When this subroutine is called in the code, SAD will check the address to see if the target is a function, and create the correct FUNC command automatically.

Subroutine type 2 is a **table lookup subroutine**. The first parameter is the word register (in hex) where the table's address is loaded, and the second is the byte register loaded with the table row size (number of columns) SAD will check the address and size are valid, and then automatically create the correct TABLE command automatically.Example above states that table address is set in register 0x40, and number of columns in register 0x32. If the second parameter is negative, it is taken as a fixed column size

Both of the above types allow for embedded parameters (i.e subroutine takes values embedded in the ROM code) a shown in the third example. If the parameter value is less than 16, it assumes this is the parameter number (first item is 1, etc)

Subroutine type 3 is a subroutine with variable parameters. First parameter is the byte register which defines the number of parameters, the second is the entry size of the parameters. Example above states that subroutine has the number of word parameters set in register 0x39

6. Encoded address types

There are currently 3 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 d000 then result is d040.

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 3 e0

If the top bit of the word 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.

- - - END - - -