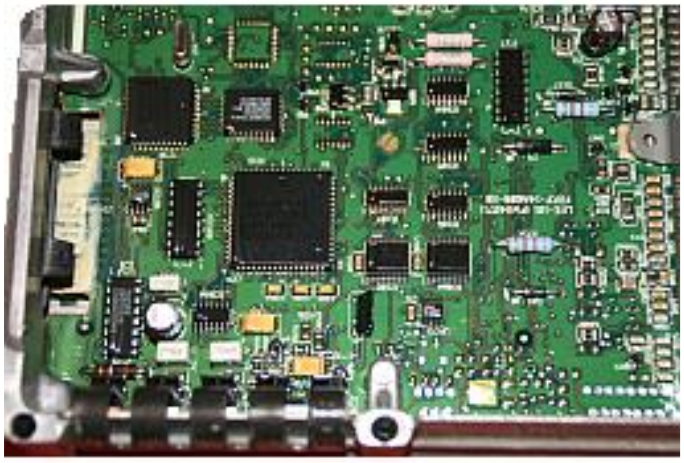


SAD 8061 – 8065 / SAD806X

Version 1.0.0.3

DOCUMENTATION
Version 0.1

By Pym (thepym@free.fr)

Table of contents

Description:

SAD 8061 – 8065 or SAD806x, is a semi-automatic disassembler tool for Intel 8061 or 8065 microcontrollers, specifically dedicated to Ford engine control units EEC-IV and EEC-V.

Its initial purpose was as following:

- To disassemble 8061/8065 roms
- To do it automatically or semi-automatically
- To generate disassembly outputs in multiple formats

Current version gives comparison functions in addition.

SAD806x is still under development, is not a commercial product and so no guarantee can be provided on it.

SAD806x never updates the binary file which is used, other tools are dedicated for this job, but do not melt files extensions, to be sure.

This document contains some kind of glossary, which could help on some meanings, but a certain knowledge about disassembling and ECU tuning will clearly help. A good starting point, would be to read “TECHNICAL NOTES ON THE EEC-IV MCU” (Eectch98.pdf).

Thanks to Andy (tvrfan) for SAD, software used as template for initial output, to Mark Mansur for TunerPro, which permits to continue working generated data.

Installation:

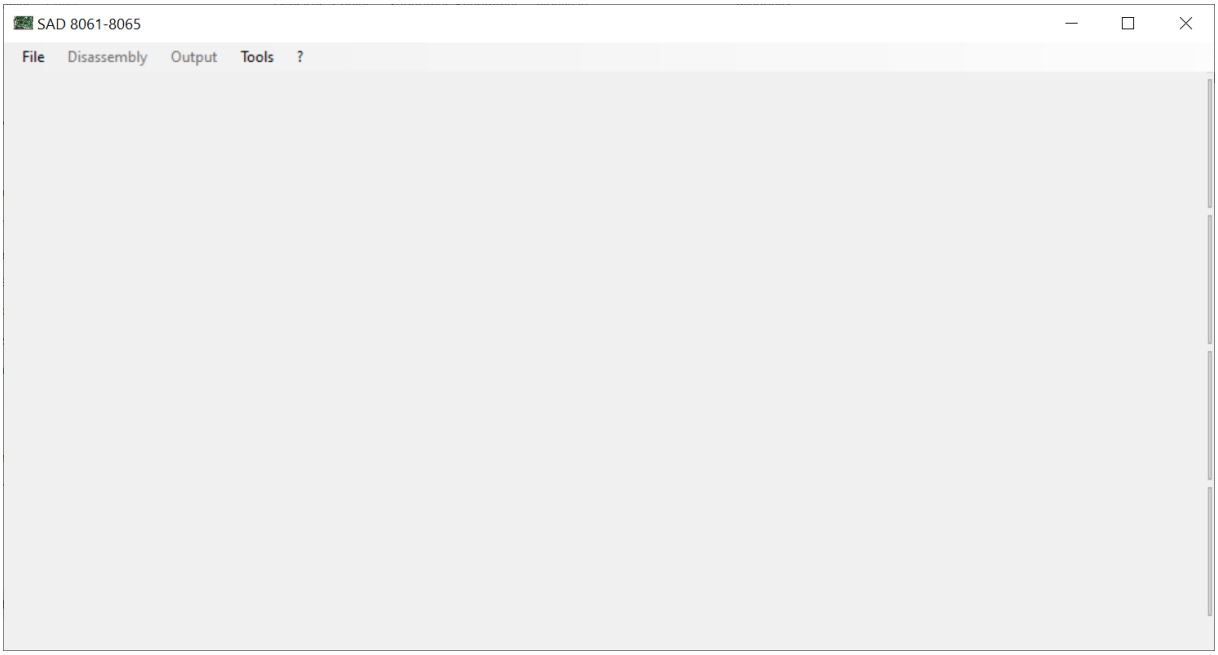
SAD806x can be installed everywhere on a Microsoft Windows system, using Framework 2.0 at least. Following files should be present in its folder to permit it to work properly:

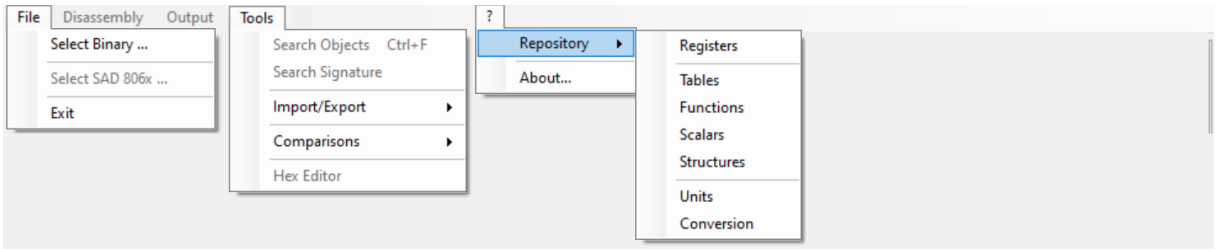
- SAD806x.exe : the executable file.
- NCalc.dll : Mathematical Expressions Evaluator for .NET (https://github.com/sheetsync/NCalc)
- System.Windows.Forms.DataVisualization.dll : Microsoft Charting for .NET
- conversion.xml (optional) : Conversion repository
- units.xml (optional) : Units repository
- registers.xml (optional) : Registers repository
- structures.xml (optional) : Structures repository
- tables.xml (optional) : Tables repository
- functions.xml (optional) : Functions repository
- scalars.xml (optional) : Scalars repository

If you want to update/create repository, just make sure you have enough right on computer and folder.

First start:

You have two ways to start SAD806x, by command line, which will provide additional options (it will be seen later on) or directly and directly by double clicking on its executable.





Menus are activated based on status of worked binary.

It is possible to work on repository directly without loading a binary, but that is the only available ability at this level.

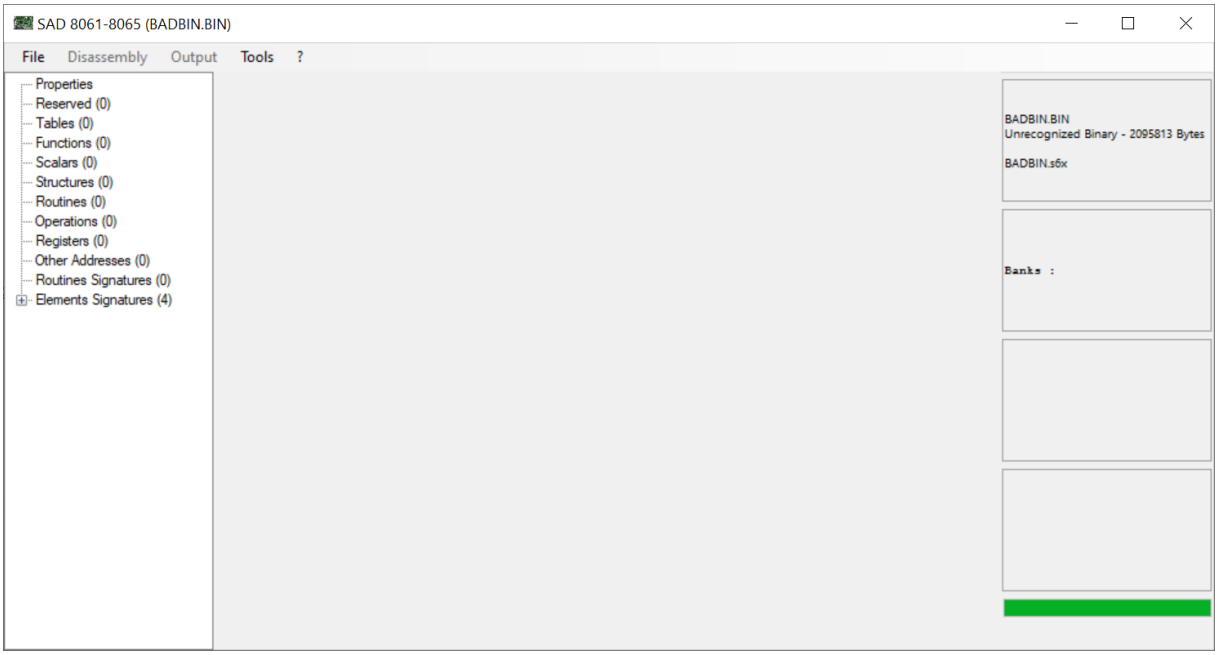
So the next step is to select a binary file, through menu ‘File/Select Binary …’or to drop it on application.

Binary loaded:

When loading a new binary, SAD806x directly tries to analyse it.
By default, SAD806x tries to find S6x definition, in the same folder than the binary, with the same name, but off course with .s6x extensions. If it finds it, it is loaded at the same time.

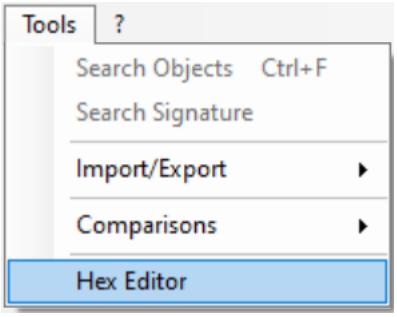
First analyse will show its result in panels on the right.

When a bad binary is loaded, result is as following :



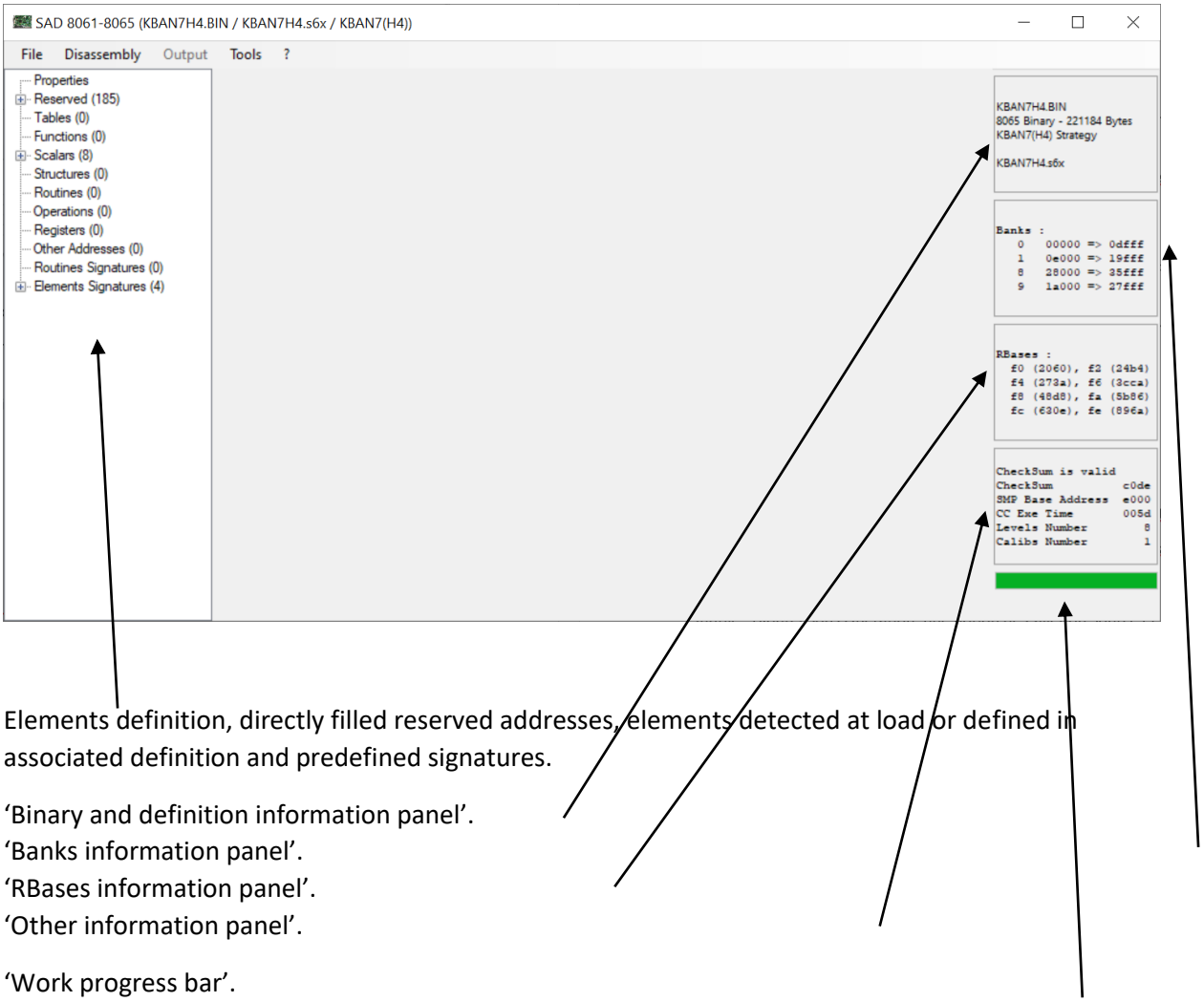
Do not forget, that SAD806x only works with Ford EEC-IV and EEC-V binary, from 32ko to 256ko.

At this level, signle option is to look at hexadecimal code on it, menu ‘Tools/Hex Editor’, to understand the issue and correct it with an external tool or to go to another binary.



It will be detailed later on.

A valid binary file will be loaded like this:

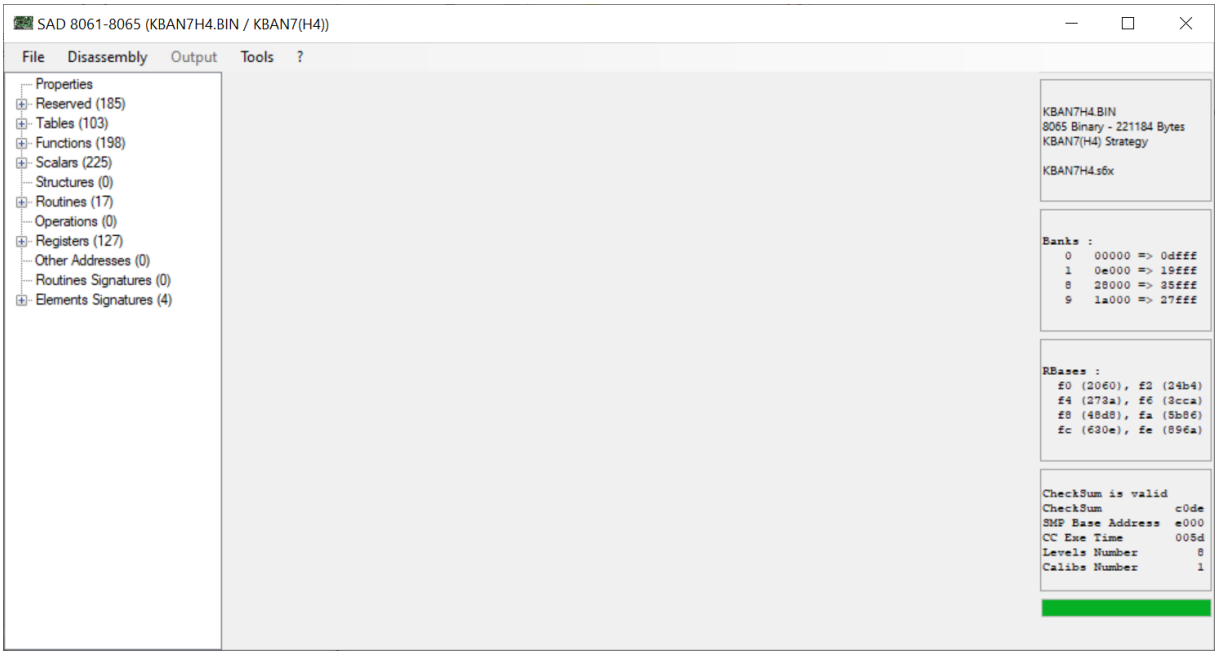


Elements definition, directly filled reserved addresses, elements detected at load or defined in associated definition and predefined signatures.

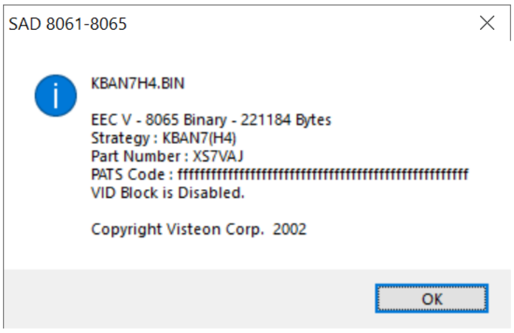
‘Binary and definition information panel’.
‘Banks information panel’.
‘RBases information panel’.
‘Other information panel’.

‘Work progress bar’.

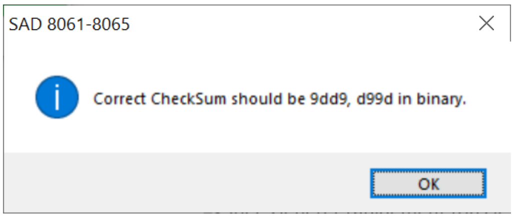
Same thing with an available definition in the folder.


SAD 8061-8065 (KBAN7H4.BIN / KBAN7(H4))
File
Disassembly
Output
Tools
?
Properties
Reserved (185)
Tables (103)
Functions (198)
Scalars (225)
Structures (0)
Routines (17)
Operations (0)
Registers (127)
Other Addresses (0)
Routines Signatures (0)
Elements Signatures (4)
KBAN7H4.BIN
8065 Binary - 221184 Bytes
KBAN7(H4) Strategy
KBAN7H4.s6x
Banks :
0 00000 => 0dfff
1 0e000 => 19fff
8 28000 => 35fff
9 1a000 => 27fff
RBases :
f0 (2060), f2 (24b4)
f4 (273a), f6 (3cca)
f8 (48d8), fa (5b86)
fc (630e), fe (896a)
CheckSum is valid
CheckSum c0de
SMP Base Address e000
CC Exe Time 005d
Levels Number 8
Calibs Number 1

By clicking on ‘Binary and definition information panel’, additional information can be displayed:

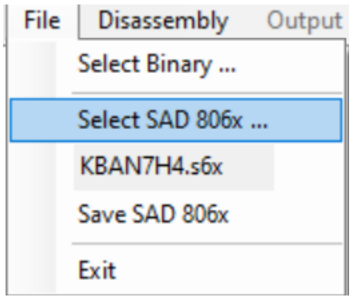


When Checksum is given as invalid in ‘Other information panel’, by clicking on this panel, you can have the right value to use:



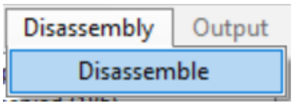
Sometimes Checksum cannot be calculated at all.

At this level some options are now available in menu.

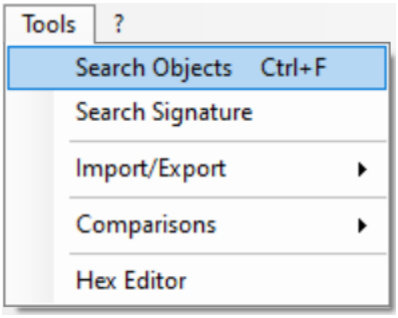


‘File/Select SAD 806x…’ : to select another S6x file. Current name is given just below. From my point of view, the best thing to do is to use the same name than the binary file from the beginning.

‘File/Save SAD 806x’ : available at this level, to create S6x file here.



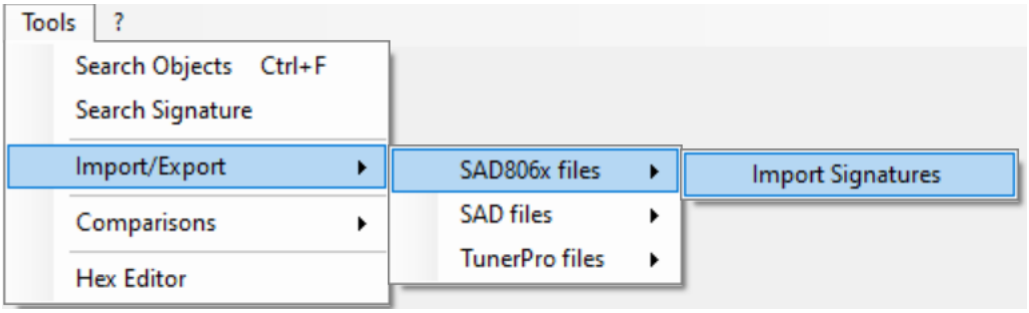
‘Disassembly/Disassemble’ : nothing to say here. It will be detailed later on.



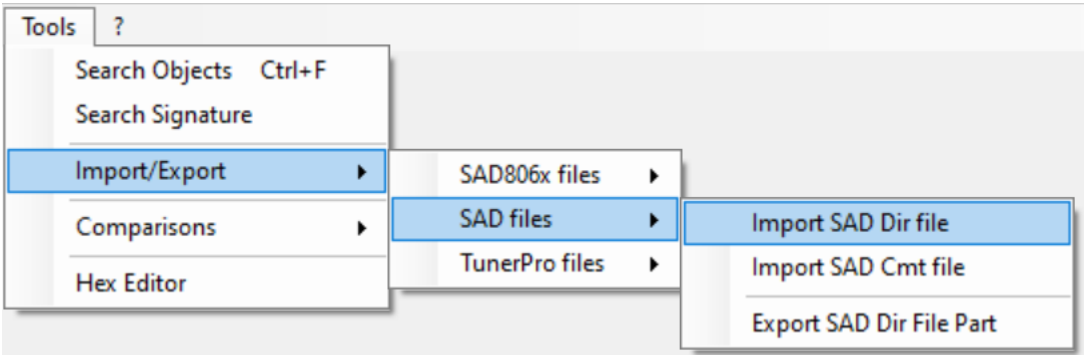
'Tools/Search Objects' : nothing to say here, tools will be described later on.

'Tools/Search Signature'

'Tools/Hex Editor'



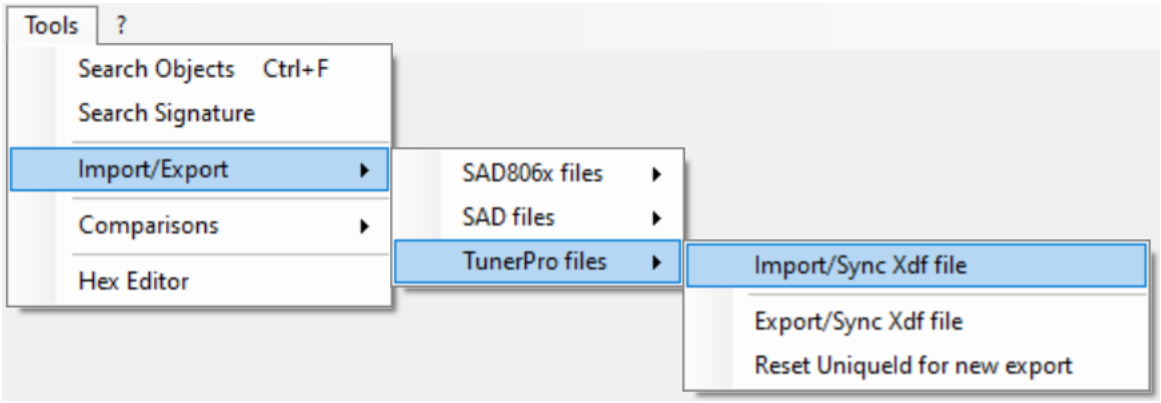
'Tools/Import/Export/SAD806x files/Import Signatures'



'Tools/Import/Export/SAD files/Import SAD Dir file'

'Tools/Import/Export/SAD files/Import SAD Cmt file'

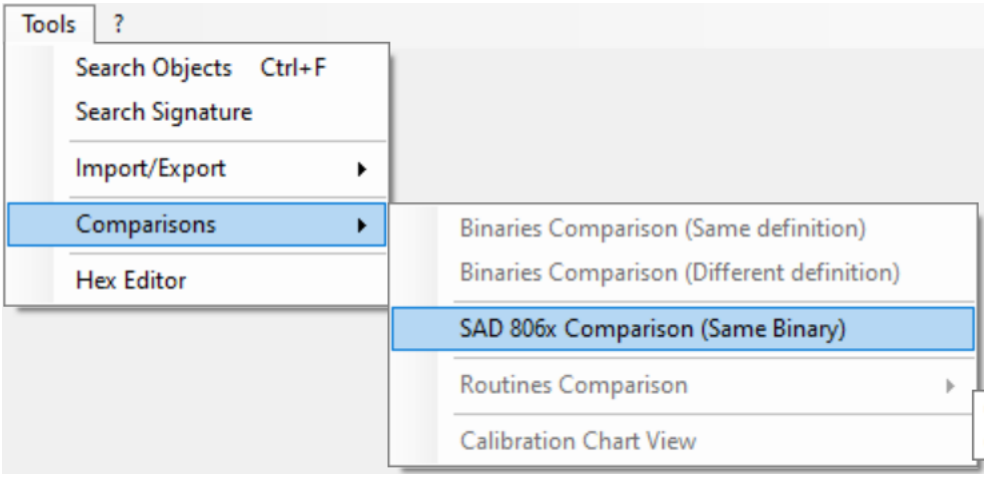
'Tools/Import/Export/SAD files/Export SAD Dir File Part'



'Tools/Import/Export/TunerPro files/Import/Sync Xdf file'

'Tools/Import/Export/TunerPro files/Export/Sync Xdf file'

'Tools/Import/Export/TunerPro files/Reset UniqueId for new export'



'Tools/Comparisons/SAD 806x Comparison (Same Binary)'

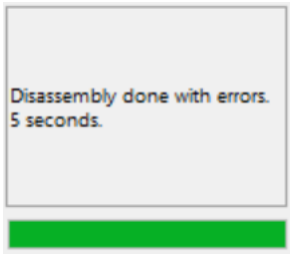
So the classical next step will be to disassemble the binary, with a definition or without, through menu 'Disassembly/Disassemble'.

Binary disassembled:

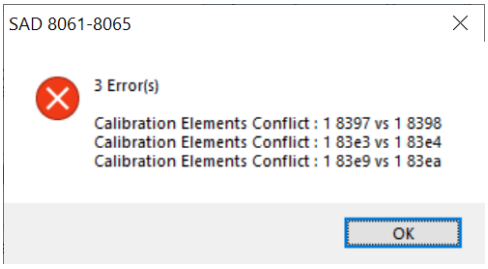
When disassembling a binary, it will take some resources on computer and based on its speed, it could take some seconds, 'Work progress bar' will help and at the end result is always the same, with a definition provided or not, this information will appear in 'Other information panel'.

When result is 'Disassembly done', SAD806x has not detected any error in operations or calibration elements, but it could have some, which appear later on output.

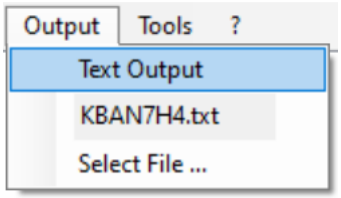
When definition contains errors or with some binaries, result could be 'Disassembly done with errors'.



It does not signify that disassembly has failed, but that some operations or calibration elements are wrong, in fact with addresses shared with others. By clicking on 'Other information panel', details will be provided. Identification and/or correction of these errors will be detailed later on.

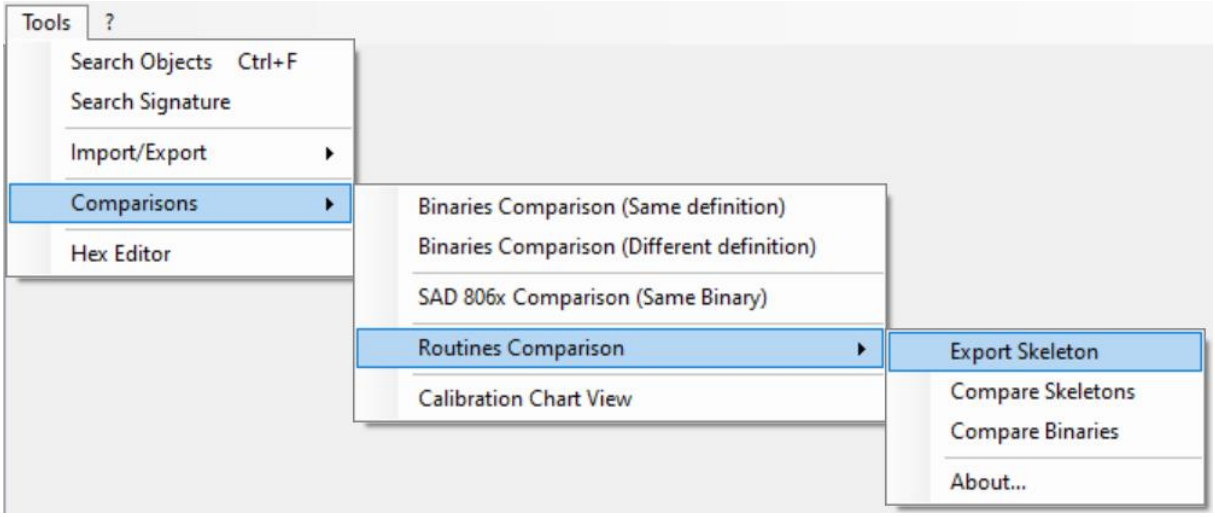


At this level, new options are available.



'Output/Text Output' : to generate the disassembled text output. It will be detailed later on.

'Output/Select File …' : you can notice that the output file name is shown just before. You can just select another output file. But if shown file already exits, by double clicking on the file name, you can open in in the default editor.



'Tools/Comparisons/Binaries Comparison (Same definition)' : nothing to say here, comparison tools will be described later on.

'Tools/Comparisons/Binaries Comparison (Different definition)'

'Tools/Comparisons/Calibration Chart View'

'Tools/Comparisons/Routines Comparison/Export Skeleton' : nothing to say here, Routines comparison tools will be described later on.

'Tools/Comparisons/Routines Comparison/Compare Skeletons'

'Tools/Comparisons/Routines Comparison/Compare Binaries'

'Tools/Comparisons/Routines Comparison/About…' : This one is information about this menu.

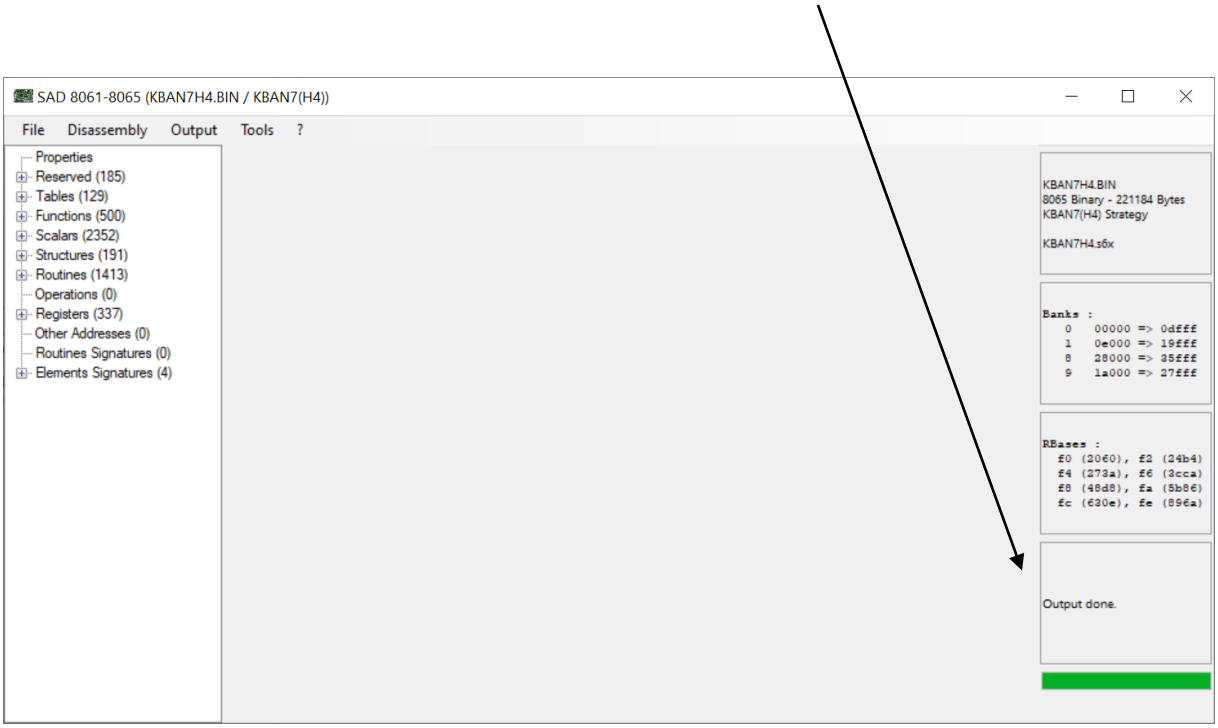
When binary is disassembled, SAD806x memory contains auto-detected and already defined operations, calibration elements and other elements, which have been separated. In addition routine grouping has been created and is available, same thing for useful registers. Everything is available in 'Elements definition' tree, except non provided operations, which have no interest here.

Everything related with ‘Elements definition’ tree with be seen later on, type by type.

For the next step that can be done with this memory, without talking about tools, it is the text output. So just use menu ‘Output/Text Output’.

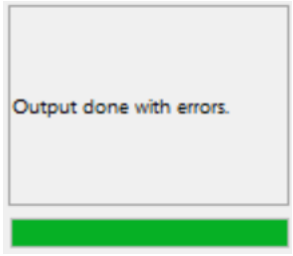
Disassembled Binary Outputted:

To output the disassembled binary, it will take some resources on computer and based on its speed, it could take some seconds, because generated file can have many lines, 'Work progress bar' will help to know the status. This information will appear in 'Other information panel'.

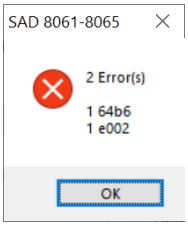


Like for disassembly, when result is 'Output done', SAD806x has not detected any error in operations or calibration elements outputting, but it still could have some.

When definition contains errors or with some binaries, result could be 'Disassembly done with errors'.



Like for disassembly, it does not signify that output has failed, but that some operations or calibration elements are wrong, in fact with addresses shared with others. By clicking on 'Other information panel', details will be provided. Identification and/or correction of these errors will be detailed later on.



At this level, options are the same than at the disassembled level.

The disassembly text file will be explained later on.

SAD 806x definition:

Working without a proper definition, is a required starting point in many cases.

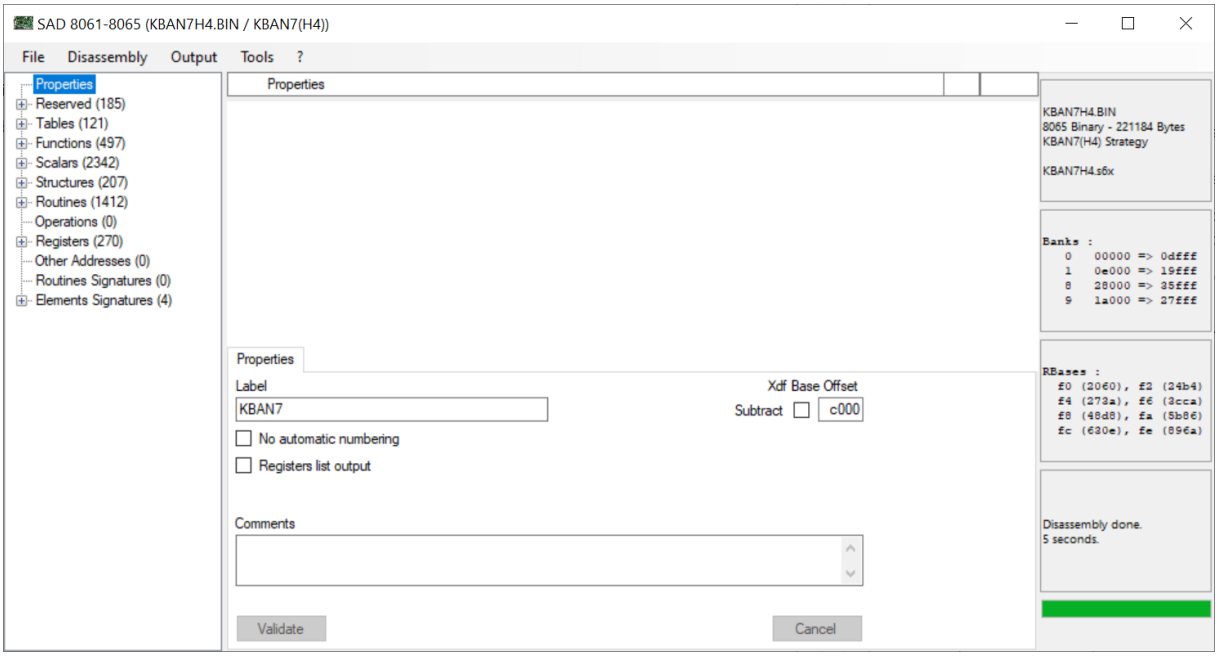
SAD 806x will do its first job, to disassembly the binary and as a result, most of the calibration elements will be identified and the code will be translated, grouped and separated from the elements. Doing a text output will show this, but it will be seen later on.

SAD 806x will also do its second job, to show all these elements and to permit to update them, to create a proper definition, which could be saved, exported and so on.

Everything is accessible through ‘Elements Definition’.

Let’s take this from the beginning.

Properties:



'Properties' give a generic setup for the SAD 806x definition, which will be mainly used for output and Xdf file export. Following items are available:

- 'Label' : The label of your definition, initialized with detected strategy name, if found.
- 'No automatic numbering' : It indicates, if auto-detected elements will use generated count or directly their address, in their generated labels, short labels. Checked means, it will use addresses.
- 'Registers list output' : It indicates, if the list of user defined registers, will be outputted at the beginning of text output or not.
- 'Comments ' : Always useful.
- 'Xdf Base Offset' : 'Subtract' checkbox and address text box, permits to provide to TunerPro, the right position for Calibration Bank, when exporting to Xdf file. When using auto detected (default) address, only elements in Bank 1 and after can be addressed, but addresses become really clear in TunerPro. For other elements or patches, you will have to do some tries.

'Label', 'Comments' and 'XDF Base Offset', will be reused for Xdf Export.

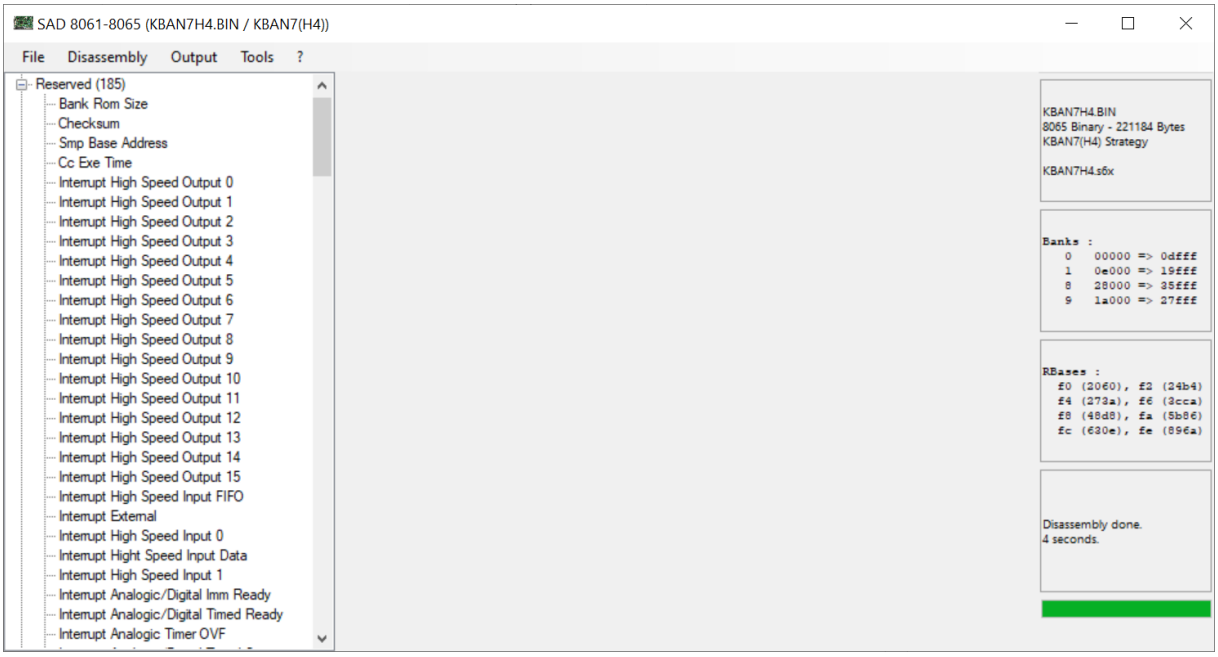
You will see everywhere, the following buttons:

They will be enabled based on current status.

‘Validate’ : It will save into memory, updates done at this level, here on properties. You will see a color change when done, but it will be required to save Sad 806x file (.s6x file), to save things definitively. Do not forget this button, before opening another thing.

‘Cancel’ : It will just cancel updates done at this level, since last load or Validate, here on properties.

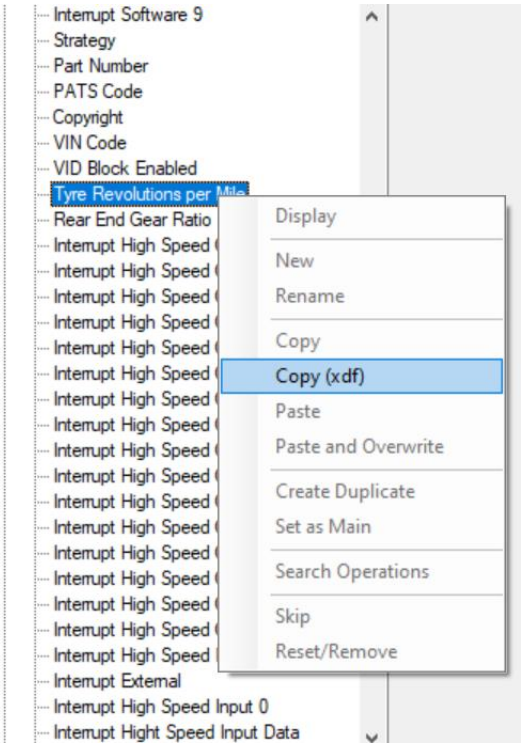
Reserved:



'Reserved' part includes fixed addresses elements or other items that should not be updated at their definition level.

So nothing can be modified on them, available information is displayed when mouse is over the item.

On some of them, through a right click you can have access to the context menu. But here, the only option is 'Copy (xdf)', which is is useful for them, not for the others. Options will be detailled later on.



Everything related with reserved elements will be present in text output, but will not be automatically exported to Xdf or other formats.

Scalars:

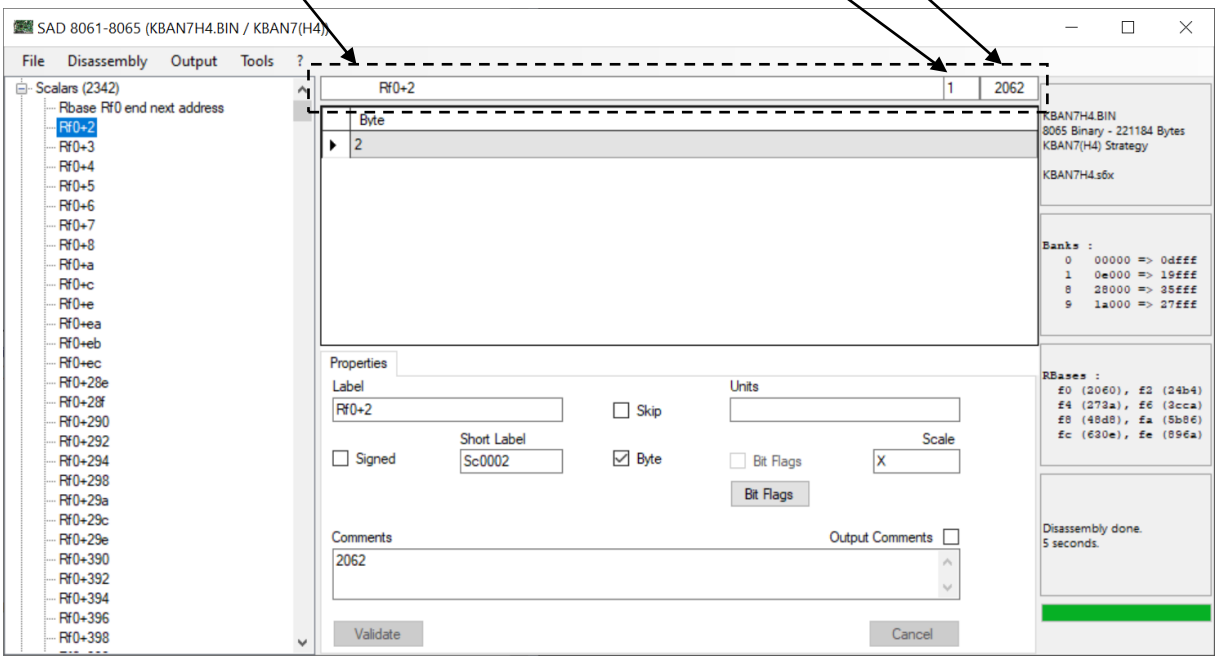
Scalars are the first calibration elements described and the simplest ones, therefore, all available actions on the element will be detailed at this place, but their description can be reused for other calibration elements.

For everything related with assembly, everything related with an address, you have these elements:

‘Descriptor’ : Descriptor of the element, which is read only here, this one appears in the list on the left too.

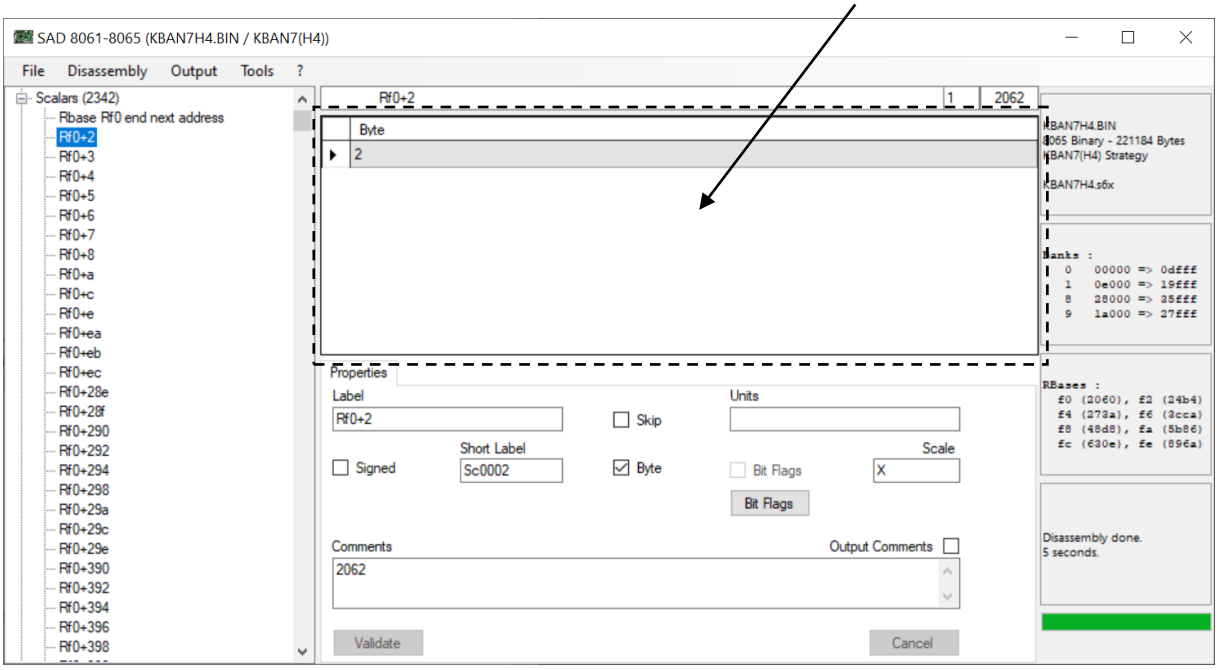
‘Bank’ : The bank number for the element.

‘Address’ : The address of the element in the related bank.



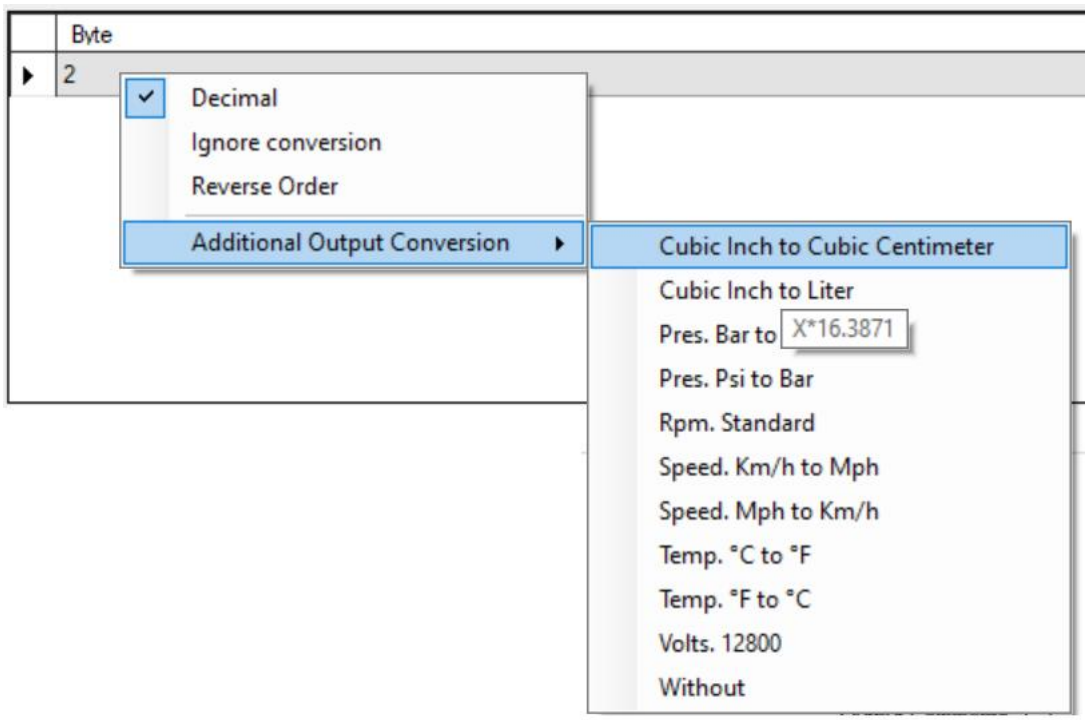
Color changes based on cases. Purple for updated elements, red for new one to be reviewed.

For everything related with calibration elements, you will have an ‘Element Data’ part:



‘Element Data’ will directly display scaled value(s) for the element.

By right clicking on this part you will have access to some options:



‘Decimal’ : Checked, data is displayed as decimal values, unchecked as hexadecimal values (not scaled anymore, when hexadecimal).

‘Ignore conversion’ : Does not scale values anymore when checked.

‘Reverse Order’ : It has no interest for Scalars, but for Functions or Tables, it starts from the last row, when checked, which makes data esaier to read.

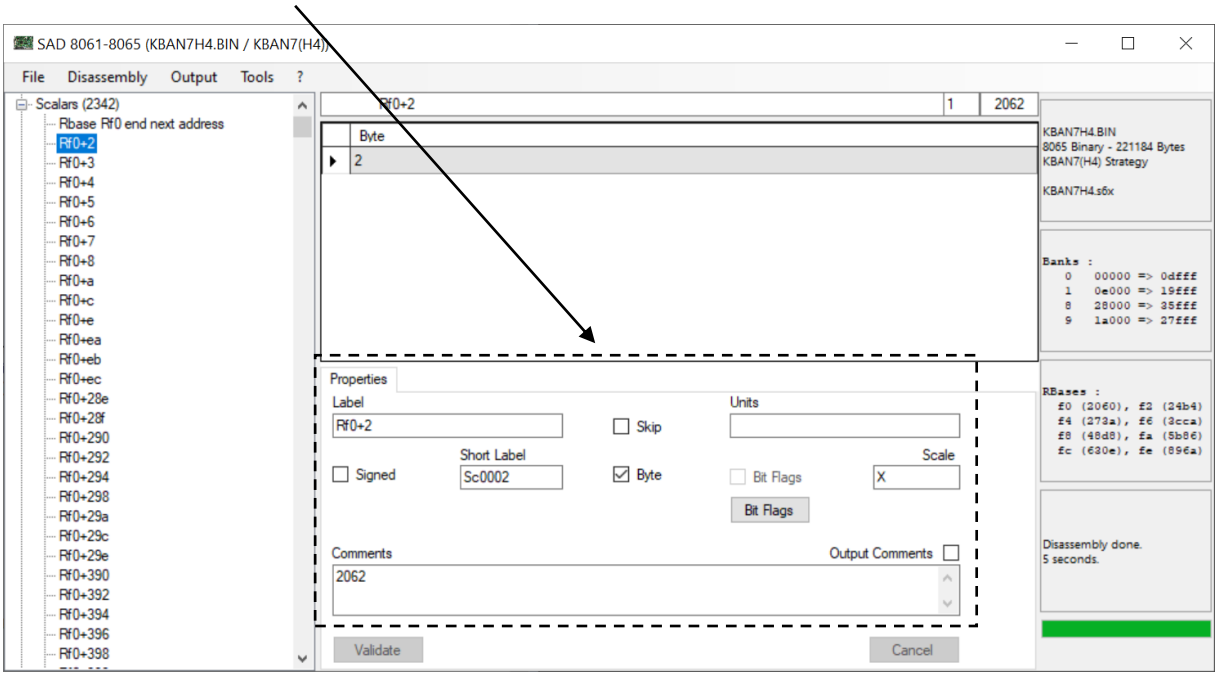
‘Additional Output Conversion’ : It permits to add, only in this place a second scale level, after the first one (if one is defined), to display data converted, to validate a new scaling formula or to identify classical types of values. Options present in the list are coming from the conversion repository, when mouse is over you can see the used formula. It does not apply with ‘Ignore conversion’ or outside ‘Decimal’ range.
A specificity exists for Functions, another option ‘Additional Input Conversion’ will be present, it is the same thing, but a function has an Input value and an Ouput one, therefore, it is necessary to have a specific conversion for each of them.

A Scalar is a byte (8 bits) or a word (16 bits) value. Because we are disassembling based on Intel instructions, word values are store low byte first (LSB in TunerPro) in assembly. This is the case for everything, including, functions, tables and structures.

A Scalar can be used as signed, based on related instructions.

You have 2 main types of scalars, the ones which are related with RBases, they will appear by default with their related RBase and the value added and the others, outside calibration part which will appear differently.

Let's describe 'Element Properties' part:



'Element Properties' part contains generic properties, which you will globally find on all elements, specific properties, only related with this type of element and specificities, which are more complex properties dedicated to this specific type of element.

For something like all text fields, by using shortcut 'Ctrl-Shift-U' shortcut on selected text, text will be upper cased, with 'Ctrl-U' it will be lower cased.

Generic properties are like following:

'Label' : Auto generated by default, based on auto numbering. It will be visible at the element address in the output.
It will be exported as main description, and inside comment in TunerPro.
By using shortcut 'Ctrl-R' shortcut in this place, the related repository will be searched for a matching based on 'Short Label'. In our case 'Scalars Repository'.

'Short Label' : Auto generated by default, based on auto numbering. It will be visible in code when element is used, and for sure at the element address too.
It will be exported with 'Label' inside comment in TunerPro, because it has no equivalent in TunerPro.
By using shortcut 'Ctrl-R' shortcut in this place, the related repository will be searched for a matching based on 'Short Label'. In our case 'Scalars Repository'.

‘Skip’ : When skipped, user defined definition for element is ignored at disassembly. Auto detection comes back to override the defined element.

‘Comments’ : Auto generated by default, with address in this case. It will be visible at the element address in the output only if ‘Output Comments’ is checked.
It will be exported preceded with ‘Label’ and ‘Short Label’ in TunerPro, to keep trace of everything.
By using shortcut ‘Ctrl-R’ shortcut in this place, the related repository will be searched for a matching based on ‘Short Label’. In our case ‘Scalars Repository’.

Scalars specific properties are like following:

‘Byte’ : Checked, scalar is declared as byte (8 bits), otherwise it is declared as word (16 bits). Detection is based on related instructions.

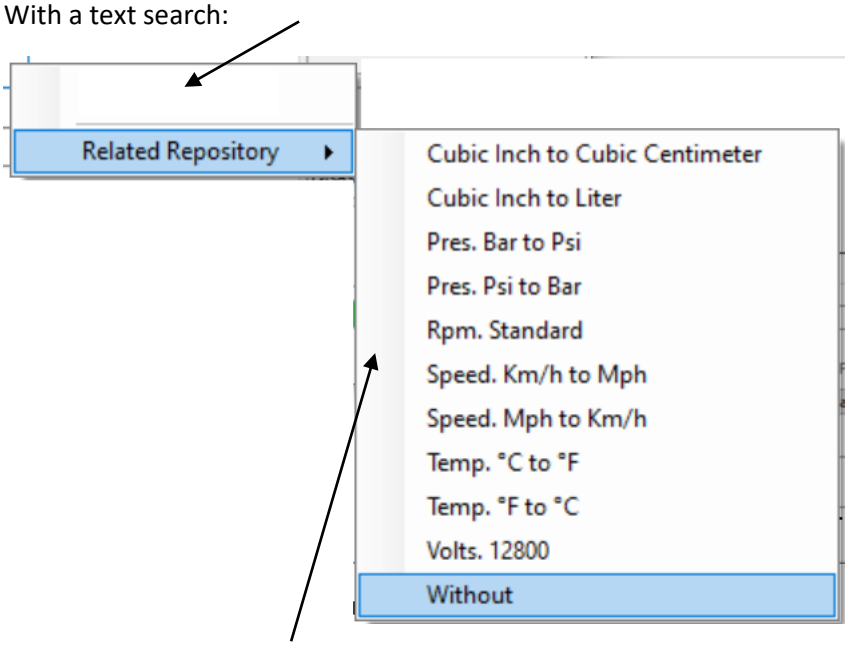
‘Signed’ : Checked, scalar is declared as signed, otherwise it is declared as unsigned. Detection is based on related instructions.

‘Scale’ : Formula to obtain the right scaled value. Scaled value will appear in the output.
By using shortcut ‘Ctrl-R’ shortcut in this place, and on all related ‘Scale’ fields, the ‘Conversion Repository’ will be searched entirely.

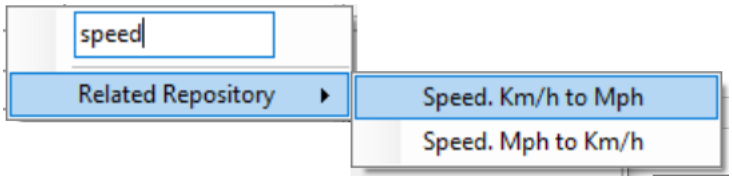
‘Units’ : This is the data unit for the related element, which is only used for information and for TunerPro.
By using shortcut ‘Ctrl-R’ shortcut in this place, and on all related ‘Units’ fields, the ‘Units Repository’ will be searched entirely.

How works a repository search?

With a text search:

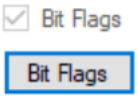


Which provides a search result, which will be used to fill in current field or element.

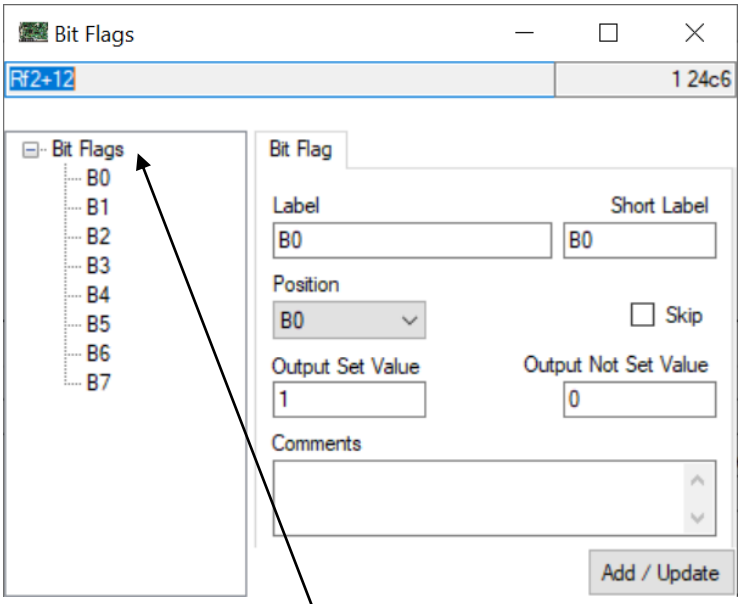


Scalar specificities:

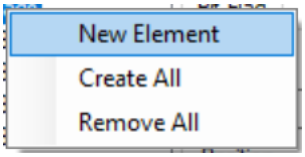
Scalar can be detected or declared as a set of bit flags.



A byte scalar set as bit flags can also contains 8 values of 0 (not set) or 1 (set) flags.
It is not possible to directly check the Bit Flags box, you have to go through the Bit Flags forms, by clicking on the button.



A scalar is autodetected as a bit flags, when it is used in a bit condition (and not only for its sign), only related bit position is set as bit flag. You can do it manually through the related form and by right clicking on 'Bit Flags header'.

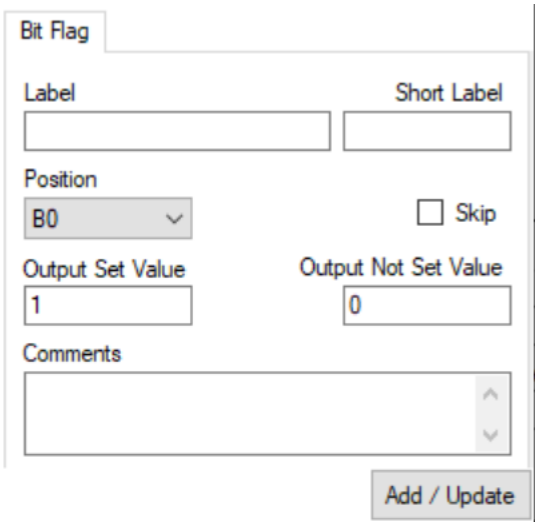


'New Element' : It creates one bit flag in the list, if a position is available (8 positions for bytes, 16 for words).

'Create All' : It creates all remaining bit flags.

'Remove All' : It deletes all declared bit flags.

The ‘Bit Flag’ properties part, permits to detail each flag.



‘Label’ : It is working like for other elements. Result will be seen for text output.

‘Short Label’ : No additional meaning. Result will be seen for text output.

‘Skip’ : No additional meaning.

‘Comments’ : Visible only in this place or in export, not in the output, no way to output it properly.

‘Position’ : This is the bit position inside the scalar, 0 to 7 for bytes, 0 to 15 for words. Bytes and words, bit order has to be known.

‘Output Set Value’ : 1 by default, but you can invert it if necessary, based on the meaning of your label.

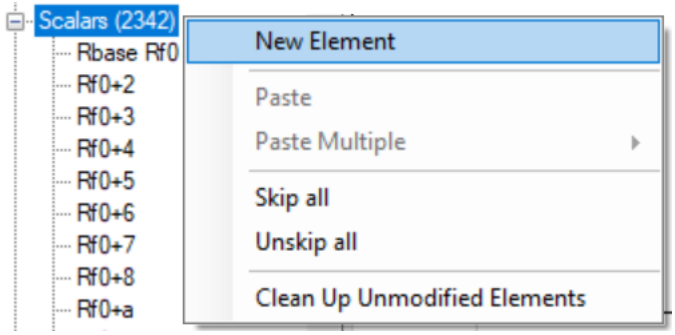
‘Output Not Set Value’ : 0 by default, but you can invert it if necessary, based on the meaning of your label.

‘Add / Update’ button : Permit to validate creation when it is a newly added bit flag or an update, when it was already created. Do not forget it for each bit flag.

When everything is done, just close the form, through the cross, to update Scalar properties.

‘Scalars’ category menu:

By right clicking on a category, you can, in major part of cases access, to options. In case of ‘Scalars’ category, you will obtain this result (based on current status of memory and/or disassembly).



Following options are available here:

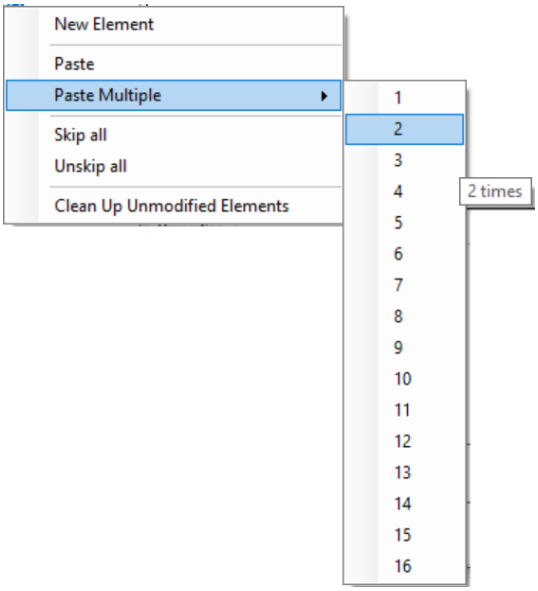
‘New Element’ : It displays creation part for an element in the related category, a scalar here.

‘Skip all’ : It will set ‘Skip’ to true on all elements in the category, scalars here. The danger is that all autodetected elements will be updated, and stored after a save in the S6x file.

‘Unskip all’ : It will set ‘Skip’ to false on all elements in the category, scalars here. The danger is that all autodetected elements will be updated, and stored after a save in the S6x file.

‘Clean Up Unmodified Elements’ : It permits to remove/reset all autodetected elements, to permit to exclude them from S6x file, when they were already saved in it, with their default values. It permits to reduce S6x file size, generated by ‘Skip/Unskip all’ option and by the TunerPro export, that will associate Ids to all exported elements.

To activate ‘Paste’ and ‘Paste Multiple’ options, it is required to copy an element in memory (from SAD 806x or TunerPro, here normally, it should be a scalar.

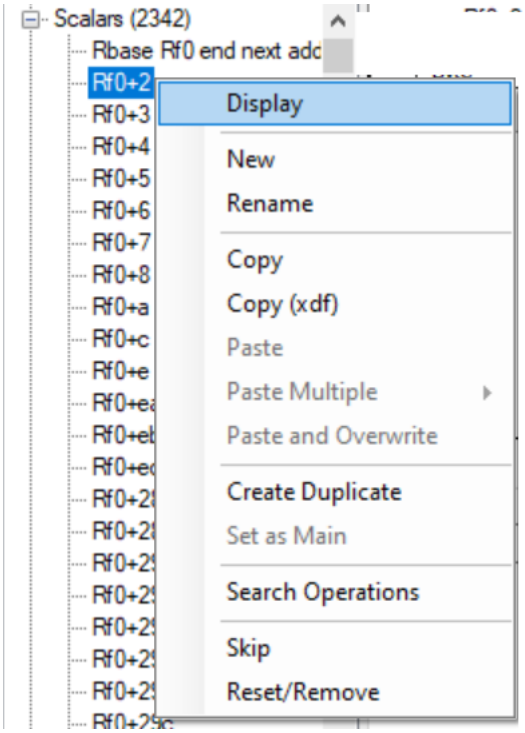


‘Paste’ : It will create/update the element, with all provided properties, based on its category.
When copy was done from SAD 806x, a default available address will be used and element is created. It will appear in red in list to be corrected at address level.
When copy was done from TunerPro, TunerPro address is used mixed with ‘XDF Base Offset’ defined in SAD 806x properties. If an element exists at this address, is will be overwritten and will appear in purple in the list, otherwise, element is created and will appear in red in list to be checked.

‘Paste Multiple’ : It exists only for scalars and works only with SAD 806x data. It permits to do a classical ‘Paste’, with an increment in address, but n times (1 time to 16 times). For example, just take a byte scalar copied at address 0x2000. A ‘Paste Multiple’ 1 time, will create a copy at address 0x2001. If it is a word scalar, it will be created at address 0x2002. For 3 times, you will have 3 byte scalars created at 0x2001, 0x2002, 0x2003 or 3 word scalars created at 0x2002, 0x2004, 0x2006 and so on. If an address is already used, the related address will be ignored, nothing will be created at this address.

‘Scalar’ element menu:

By right clicking on an element, you can, in major part of cases, access to options. In case of ‘Scalar’ element, you will obtain this result (based on current status of memory and/or disassembly).



Following options are available here:

‘Display’ : Equivalent to the left click on the element, it will display the properties and data of the selected element.

‘New’ : It displays creation part for an element with the same category, a scalar here.

‘Rename’ : It put the element in the list in edit mode, to be renamed at descriptor level. The same thing is possible with a short left click on the element in the list. After the descriptor is changed, it is applied to the related value on the properties of the updated element.

‘Copy’ : It copies the current element into the clipboard, to be reused in current SAD 806x session or in another one.

‘Copy (xdf)’ : It copies the current element into the clipboard, with TunerPro format, to be reused in TunerPro.

‘Paste’ : It will create a new element or update an element (if address matches and only when it is coming from TunerPro).
It is the same functionnality than the one on the category.

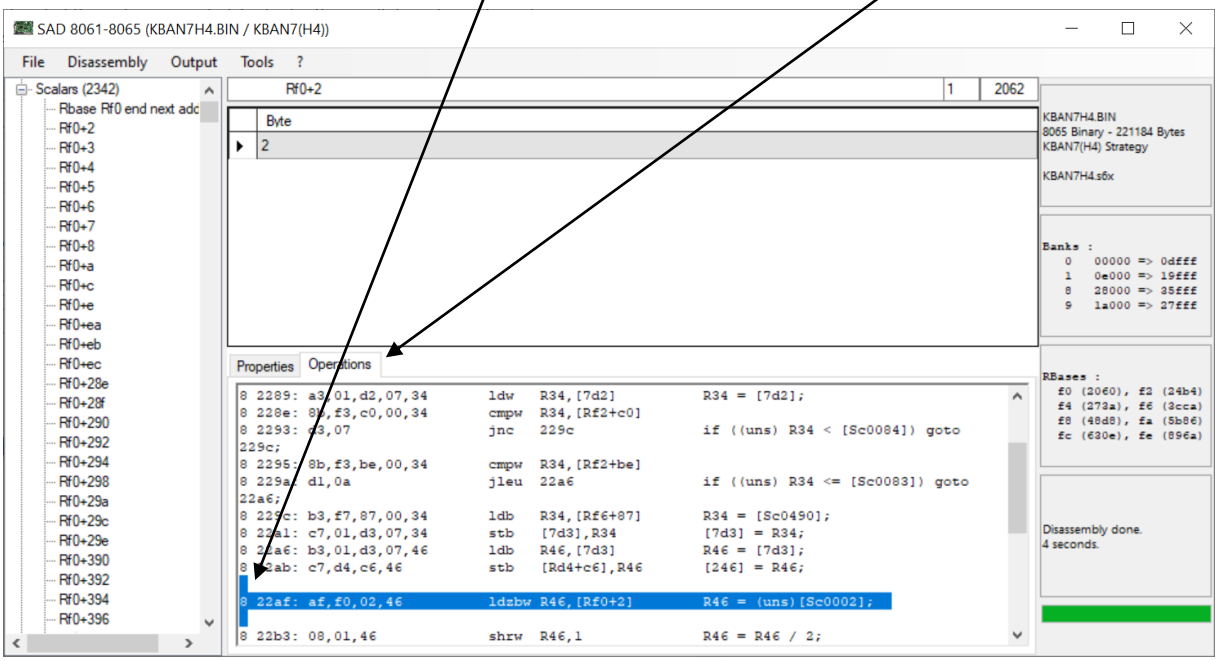
‘Paste Multiple’ : It exists only for scalars and works only with SAD 806x data.
It permits to do a classical ‘Paste’, with an increment in address, but n times (1 time to 16 times).
It is the same functionnality than the one on the category.

‘Paste and Overwrite’ : It is the same functionality than ‘Paste’, with one major exception, it will apply on the address of the selected element, properties coming from clipboard, will in fact overwrite current element.

‘Create Duplicate’ : It is now possible to have multiple elements (of same category) at the same address. It is just to be TunerPro compliant and for some strategies, re-using scalers in a strange way. With this option, you can create a new Duplicate element at the same address. In the output only the main one will be displayed, so we have main element for an adress and its duplicates. Removing the element, will set its first duplicate, if it exists as the main one.

‘Set as Main’ : This option is available when the element is a duplicate one. It permits to switch the main element with the current one to set it as main, with all related consequences.

‘Search Operations’ : The goal of this option is to display, where the element is used in code. So a short part of the code is generated, to display this result. Sometimes it is not possible, but when it is working, it really helps. Result appears in a related ‘Operations’ tab, and it will display where element it used (firstly):



‘Skip’ : It will directly set ‘Skip’ as true on selected element.

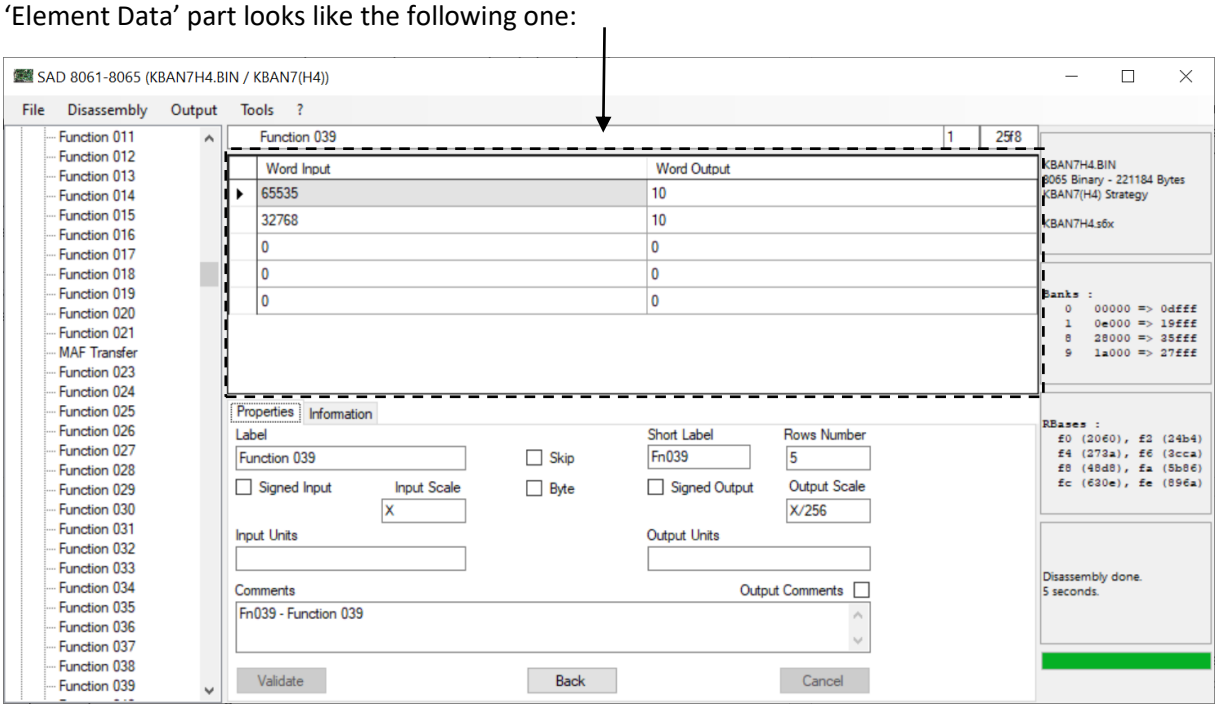
‘Reset/Remove’ : It will delete everything set by user, on the selected element, so it is like a remove for a user created element (or before the disassembly) and like a reset for an auto-detected element, which has been updated by the user, after disassembly.
Really removed element, will disappear after this option is executed, a reset on element, will keep it visible and accessible.

Functions:

A Function is a two columns table, using an input value to get an output one.
Input values can be bytes (8 bits) or words (16 bits) and Output values will have the same size. Input values can be signed or unsigned and Output ones can also be. Setup on one row applies to the whole function.
Because we are disassembling based on Intel instructions, word values are store low byte first (LSB in TunerPro) in assembly.
Number of rows in function is never known or provided to related routine giving the result, this is why it can be dangerous to update function values, in a bad way.
Also, auto detection of rows number, is based on minimum and maximum values, with other things, it is not an exact science and it can be wrong, exactly like the routine would be.

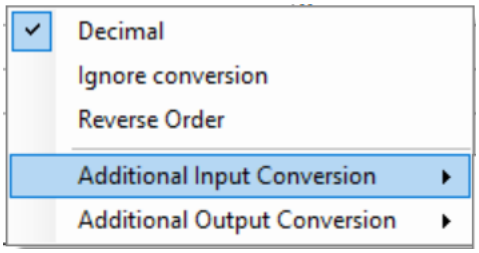
It exists a specific type of function, which we will call ‘Scalers’. They are used to scale table’s inputs. Auto detection tries to detect them, and to auto set their ‘Output Scale’, often to X/16 for byte output and X/256 for word output. They are essential to work with tables.

'Element Data' part looks like the following one:

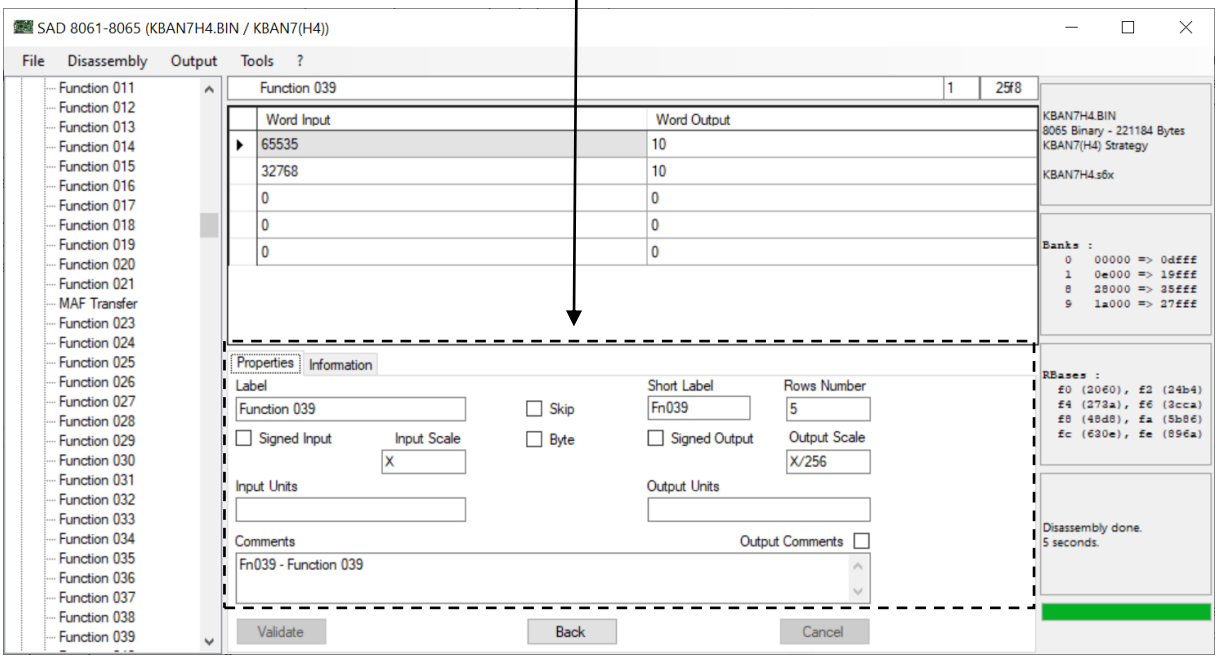


As you can see it is a Word Input, Word Output function, with two columns and their labels are clear enough.

Only specificity for functions, two conversion options in menu, one for Input, the other for Outpout.



‘Element Properties’ part is the following one:



Another time, for something like all text fields, by using shortcut ‘Ctrl-Shift-U’ shortcut on selected text, text will be upper cased, with ‘Ctrl-U’ it will be lower cased.

Generic properties are like following:

‘Label’ : Auto generated by default, based on auto numbering. It will be visible at the element address in the output.
It will be exported as main description, and inside comment in TunerPro.
By using shortcut ‘Ctrl-R’ shortcut in this place, the related repository will be searched for a matching based on ‘Short Label’. In our case ‘Functions Repository’.

‘Short Label’ : Auto generated by default, based on auto numbering. It will be visible in code when element is used, and for sure at the element address too.
It will be exported with ‘Label’ inside comment in TunerPro, because it has no equivalent in TunerPro.
By using shortcut ‘Ctrl-R’ shortcut in this place, the related repository will be searched for a matching based on ‘Short Label’. In our case ‘Functions Repository’.

‘Skip’ : When skipped, user defined definition for element is ignored at disassembly. Auto detection comes back to override the defined element.

‘Comments’ : Auto generated by default, with address in this case. It will be visible at the element address in the output only if ‘Output Comments’ is checked.
It will be exported preceded with ‘Label’ and ‘Short Label’ in TunerPro, to keep trace of everything.
By using shortcut ‘Ctrl-R’ shortcut in this place, the related repository will be searched for a matching based on ‘Short Label’. In our case ‘Functions Repository’.

Functions specific properties are like following:

‘Byte’ : Checked, function is declared as byte one, byte input and output, otherwise it is declared as ‘Word’. Detection is based on related routines.

‘Rows Number’ : Auto detected, rows number is one of the main information for function.

‘Signed Input’ : Checked, input is declared as signed, otherwise it is declared as unsigned. Detection is based on related routines.

‘Signed Output’ : Checked, output is declared as signed, otherwise it is declared as unsigned. Detection is based on related routines.

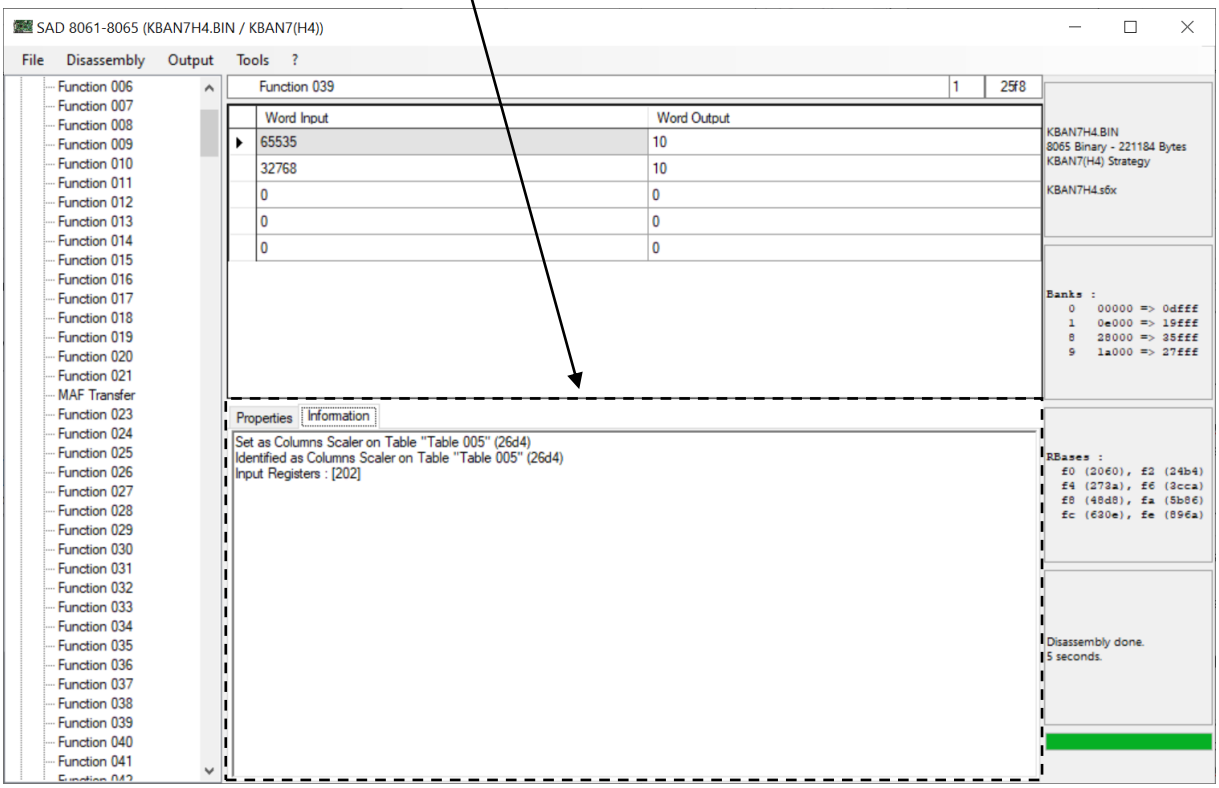
‘Input Scale’ : Formula to obtain the right scaled input value. Scaled value will appear in the output.
By using shortcut ‘Ctrl-R’ shortcut in this place, and on all related ‘Scale’ fields, the ‘Conversion Repository’ will be searched entirely.

‘Output Scale’ : Formula to obtain the right scaled output value. Scaled value will appear in the output.
By using shortcut ‘Ctrl-R’ shortcut in this place, and on all related ‘Scale’ fields, the ‘Conversion Repository’ will be searched entirely.

‘Input Units’ : This is the data unit for the related input, which is only used in this place and for TunerPro.
By using shortcut ‘Ctrl-R’ shortcut in this place, and on all related ‘Units’ fields, the ‘Units Repository’ will be searched entirely.

‘Output Units’ : This is the data unit for the related input, which is only used in this place and for TunerPro.
By using shortcut ‘Ctrl-R’ shortcut in this place, and on all related ‘Units’ fields, the ‘Units Repository’ will be searched entirely.

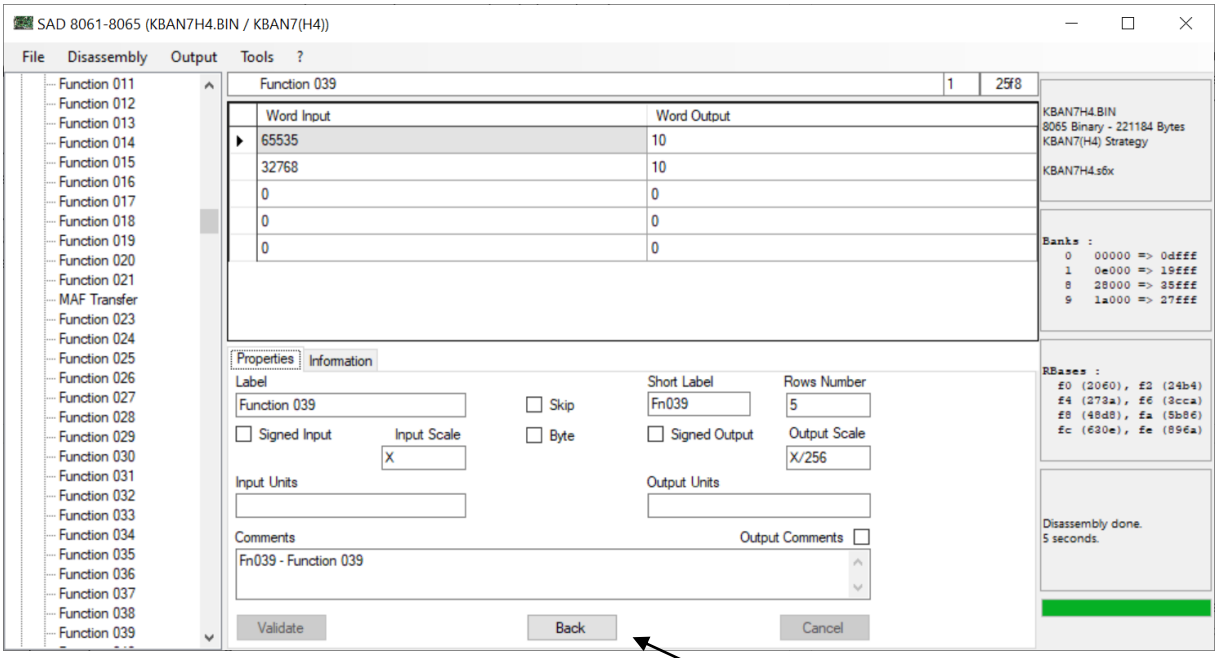
'Element Information' for Function:



Functions possess an additional 'Element Information' tab, which includes additional details grabbed during disassembly and interessant to be known.

In this case, we discover, it has been auto detected as scaler for one table and set like that, because no doubt was possible. We discover which Register is used as Input value too. For sure, when labels are redefined, elements appear translated here.

Function specificities:



When function is opened when clicking on a table sclarer, ‘Back’ button appears to permit to come back to table.

‘Functions’ category menu:

No specificity at all.

‘Function’ element menu:

No specificity at all.

Tables:

Tables are essentially n columns multiplied by n rows containing scalars, often bytes (8 bits), sometimes words (16 bits). But a table is always a fully bytes table or fully words table.

Input values can be bytes (8 bits) or words (16 bits) and Output values will have the same size.

One Input value for column position, another for row position. Output values can be signed or unsigned. Setup on one row applies to the whole function.

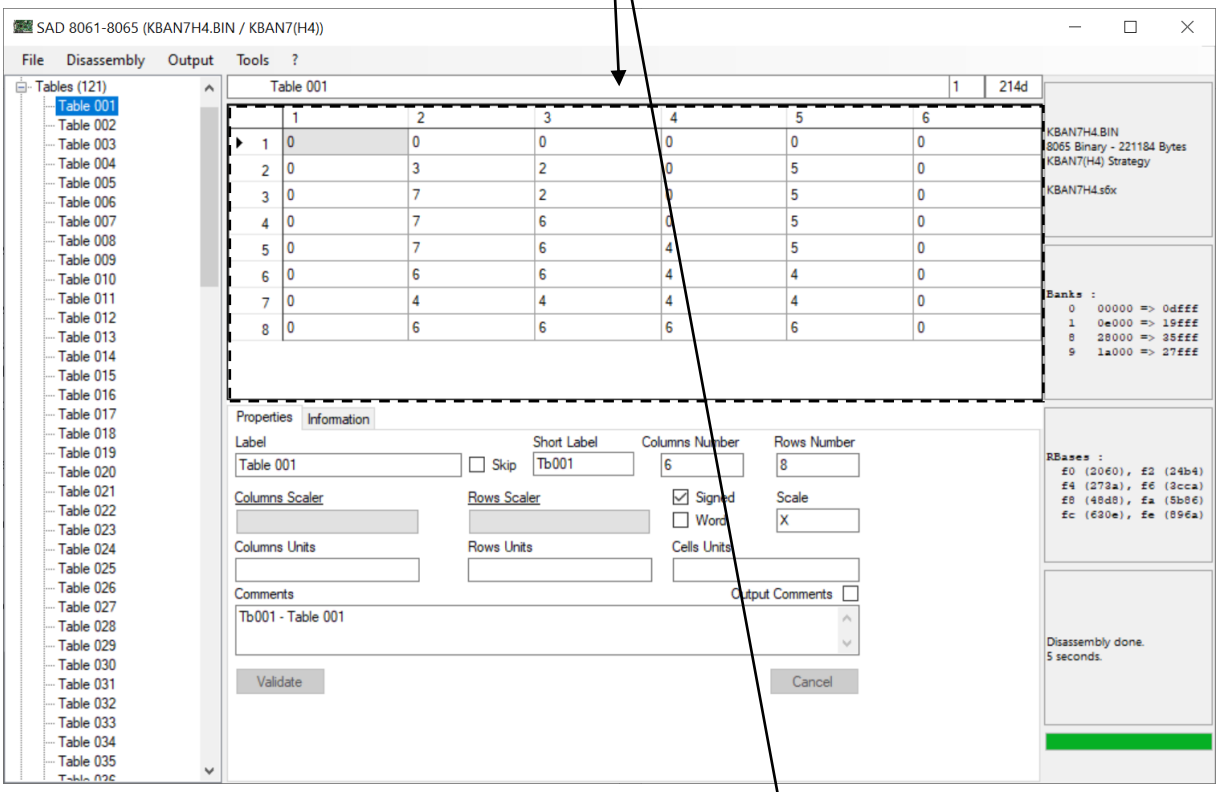
Because we are disassembling based on Intel instructions, word values are store low byte first (LSB in TunerPro) in assembly.

Number of rows in table is never known or provided to related routine giving the result, routine uses 3 input values, the columns number, the column position and the row position.

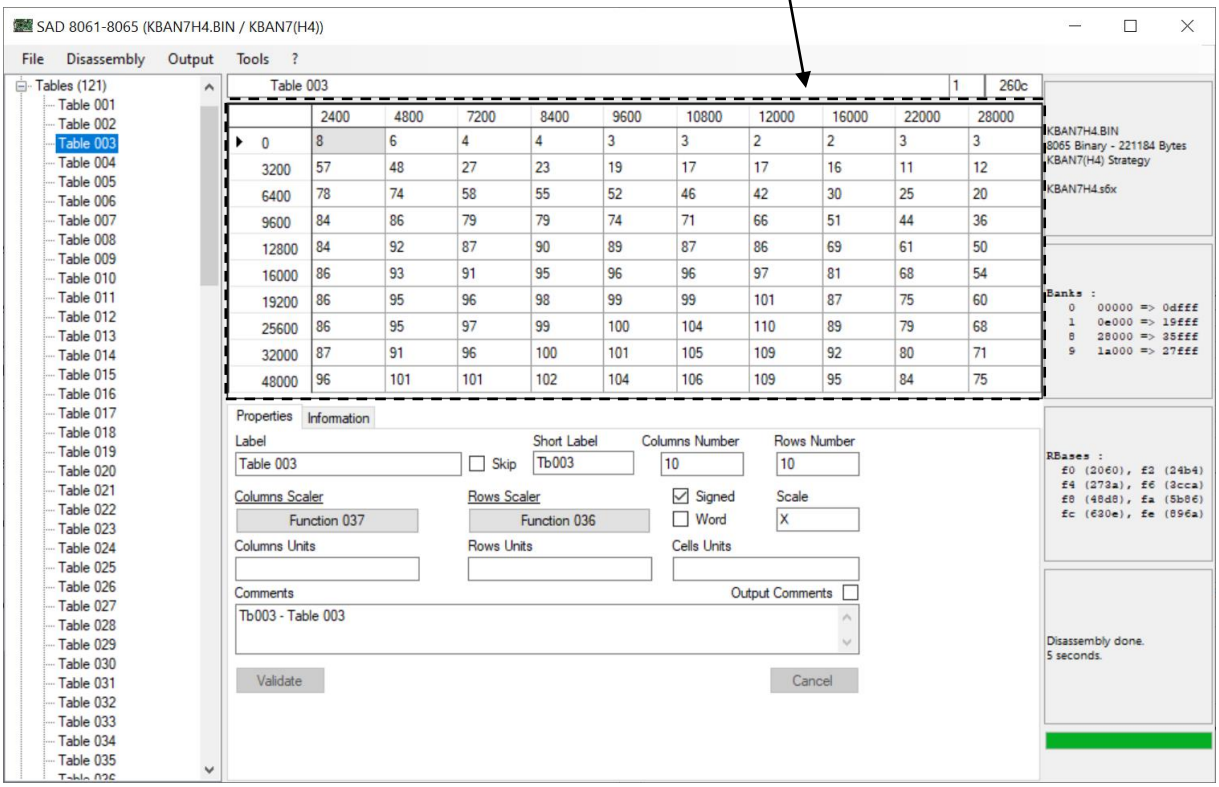
Column or row position are essentially coming from functions, with scaler type, having a scaled output for position from 0 to n-1 column number or row number.

Also, auto detection of rows number, is based on possible sizes, with other things, it is not an exact science and it can be wrong, even if related scalers are not detected or are not rights.

'Element Data' part looks like the following one:

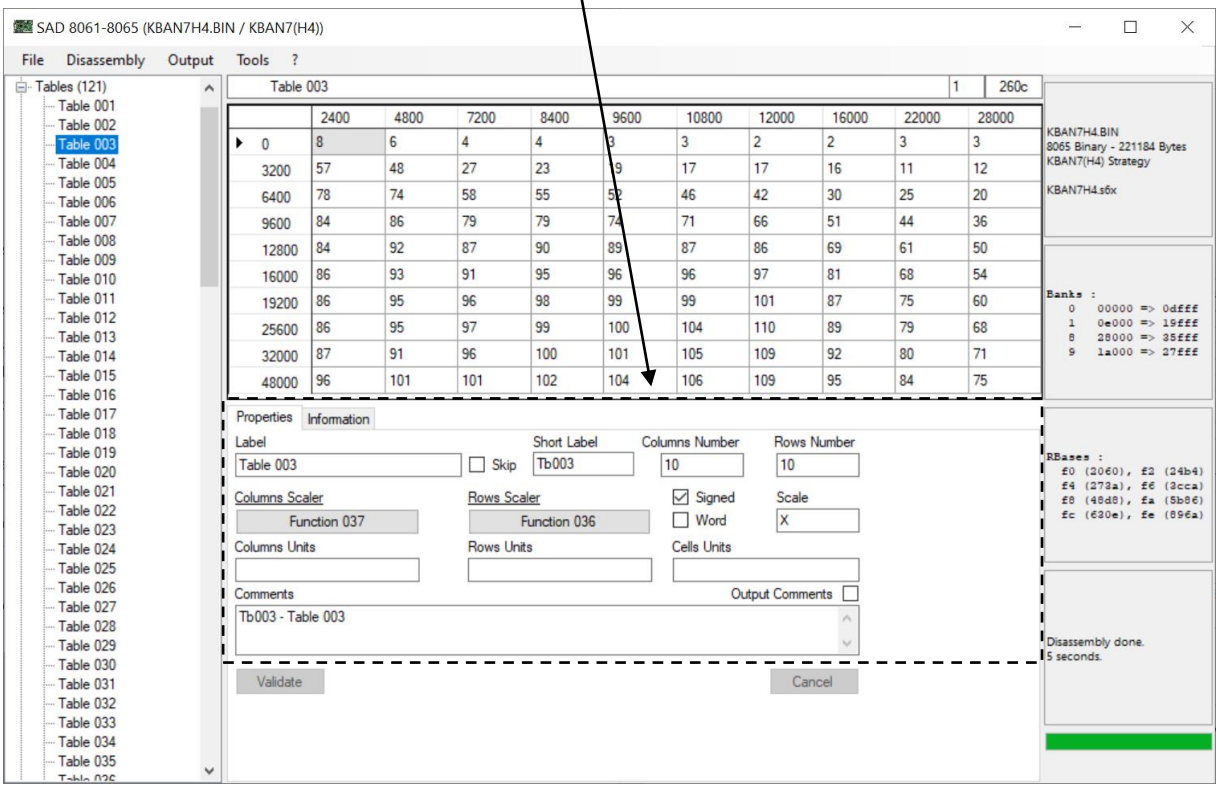


This is a result where scalers are not detected. Columns and rows label are defaulted.



And a result with labelled rows and columns, based on scalers and their input values (Function 037 and Function 036).
Not other specificity exists, but you can see the interest to have the right scaler set at this level.

'Element Properties' part is the following one:



Another time, for something like all text fields, by using shortcut 'Ctrl-Shift-U' shortcut on selected text, text will be upper cased, with 'Ctrl-U' it will be lower cased.

Generic properties are like following:

'Label' : Auto generated by default, based on auto numbering. It will be visible at the element address in the output.
It will be exported as main description, and inside comment in TunerPro.
By using shortcut 'Ctrl-R' shortcut in this place, the related repository will be searched for a matching based on 'Short Label'. In our case 'Tables Repository'.

'Short Label' : Auto generated by default, based on auto numbering. It will be visible in code when element is used, and for sure at the element address too.
It will be exported with 'Label' inside comment in TunerPro, because it has no equivalent in TunerPro.
By using shortcut 'Ctrl-R' shortcut in this place, the related repository will be searched for a matching based on 'Short Label'. In our case 'Tables Repository'.

'Skip' : When skipped, user defined definition for element is ignored at disassembly. Auto detection comes back to override the defined element.

'Comments' : Auto generated by default, with address in this case. It will be visible at the element address in the output only if 'Output Comments' is checked.
It will be exported preceded with 'Label' and 'Short Label' in TunerPro, to keep trace of everything.
By using shortcut 'Ctrl-R' shortcut in this place, the related repository will be searched for a matching based on 'Short Label'. In our case 'Tables Repository'.

Table's specific properties are like following:

'Columns Number' : Auto detected, by direct read in code, columns number is one of the main information for table.

'Rows Number' : Auto detected, rows number is one of the main information for table.

'Word' : Checked, table is declared as word one, word output, otherwise it is declared as byte output. Detection is based on related routines.

'Signed' : Checked, output is declared as signed, otherwise it is declared as unsigned. Detection is based on related routines.

'Scale' : Formula to obtain the right scaled output value. Scaled value will appear in the output.
By using shortcut 'Ctrl-R' shortcut in this place, and on all related 'Scale' fields, the 'Conversion Repository' will be searched entirely.

'Columns Units' : This is the data unit for the related column input, which is only used in this place and for TunerPro.
By using shortcut 'Ctrl-R' shortcut in this place, and on all related 'Units' fields, the 'Units Repository' will be searched entirely.

'Rows Units' : This is the data unit for the related row input, which is only used in this place and for TunerPro.
By using shortcut 'Ctrl-R' shortcut in this place, and on all related 'Units' fields, the 'Units Repository' will be searched entirely.

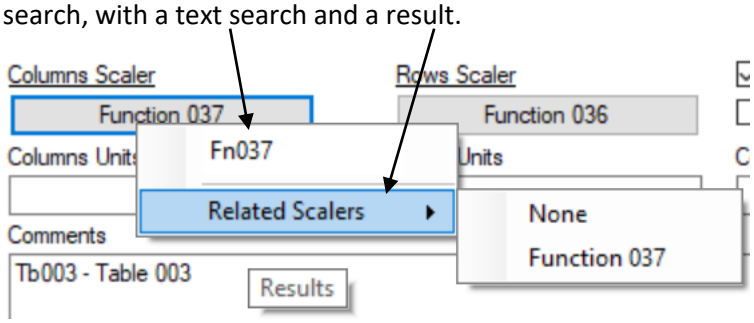
'Cells Units' : This is the data unit for the output, for cells, which is only used in this place and for TunerPro.
By using shortcut 'Ctrl-R' shortcut in this place, and on all related 'Units' fields, the 'Units Repository' will be searched entirely.

'Columns Scaler' : Auto detected, this is the scaler function for columns. Specific format is used here, a clickable label and a clickable button. To see the complete description of the scaler, just move the move over the button. To open the related function, just click on the label (when a scaler is in place). To select a new scaler (function should already exist), just click on the button to access to the 'Scaler Search'.

'Rows Scaler' : Auto detected, this is the scaler function for columns. Everything described for 'Columns Scaler' applies to 'Rows Scaler'.

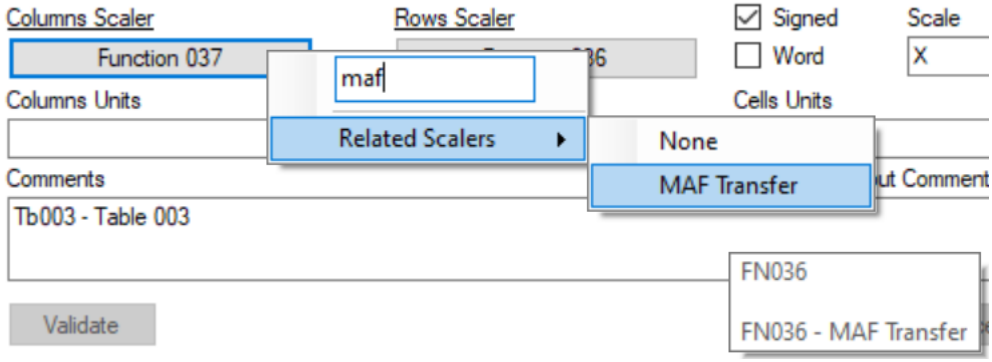
'Scaler Search':

When clicking on the scaler button, you can access to its search, which is something like a repository search, with a text search and a result.



By default, selected function 'Short Label' is used for searching, because the number of results is limited for performance reasons.

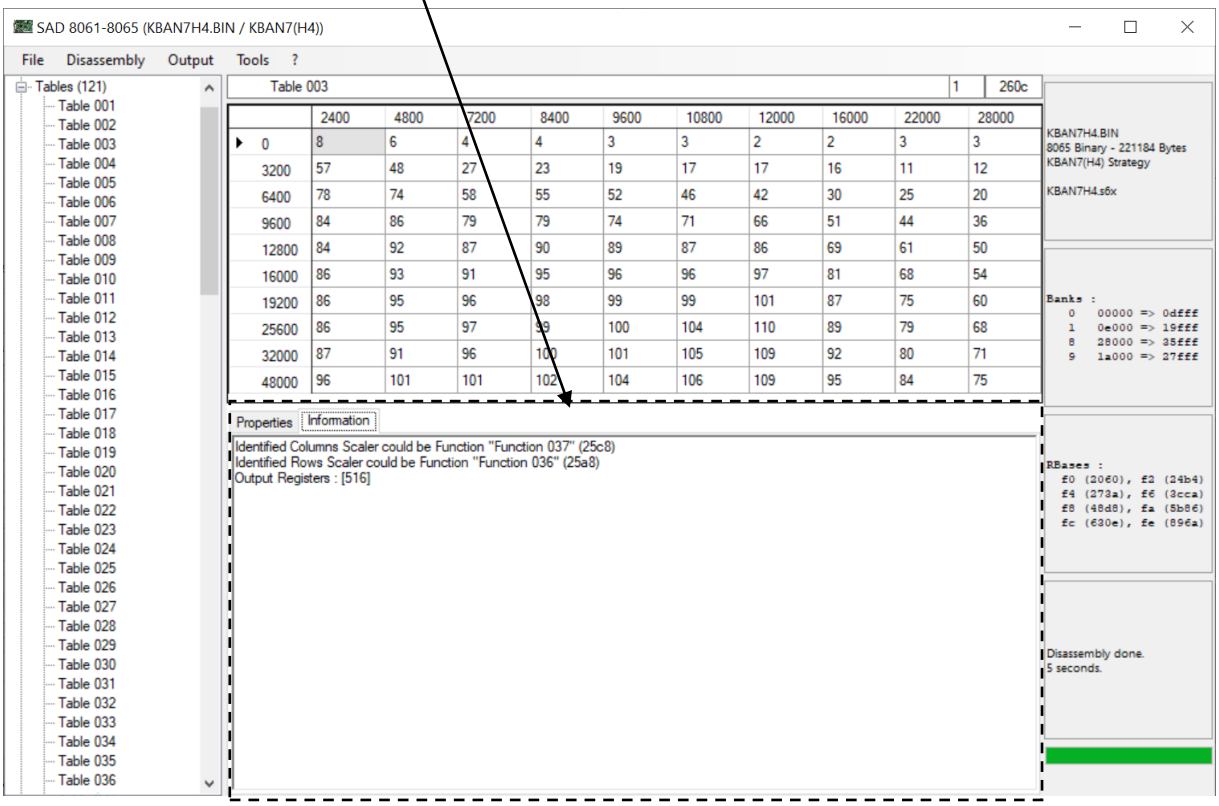
To search another function, just update the text search part and then, open the 'Related Scalers'. Search is done on many properties of the function.



You will say that 'MAF Transfer' function FN036 is not a scaler, and you are right, search is done on all available functions. But like this, you can see what appears, when the mouse is over an element in the list.

The 'None' element, permits to remove the scaler on the table.

‘Element Information’ for Table:



Tables possess an additional ‘Element Information’ tab too, which includes additional details grabbed during disassembly and interessant to be known.

In this case, we discover, it has been auto detected with both scalers. We discover which register is used as output value too. For sure, when labels are redefined, elements appear translated here.

Table specificities:

No specificity at all, except scalers.

‘Tables’ category menu:

No specificity at all.

'Table' element menu:

No specificity at all.

Structures:

Structures are non-generic elements, in fact neither a scalar, nor a function, nor a table.
Scalars, functions and tables can be described by a structure, but the opposite is not always true.

Structures are a set of scalars assembled in different ways, sometimes with rules, sometimes not.

Routines using them are specific too. Even if they can be identified, they have so many different types, than it is really difficult to manage them properly, compared to routines used for functions and tables.

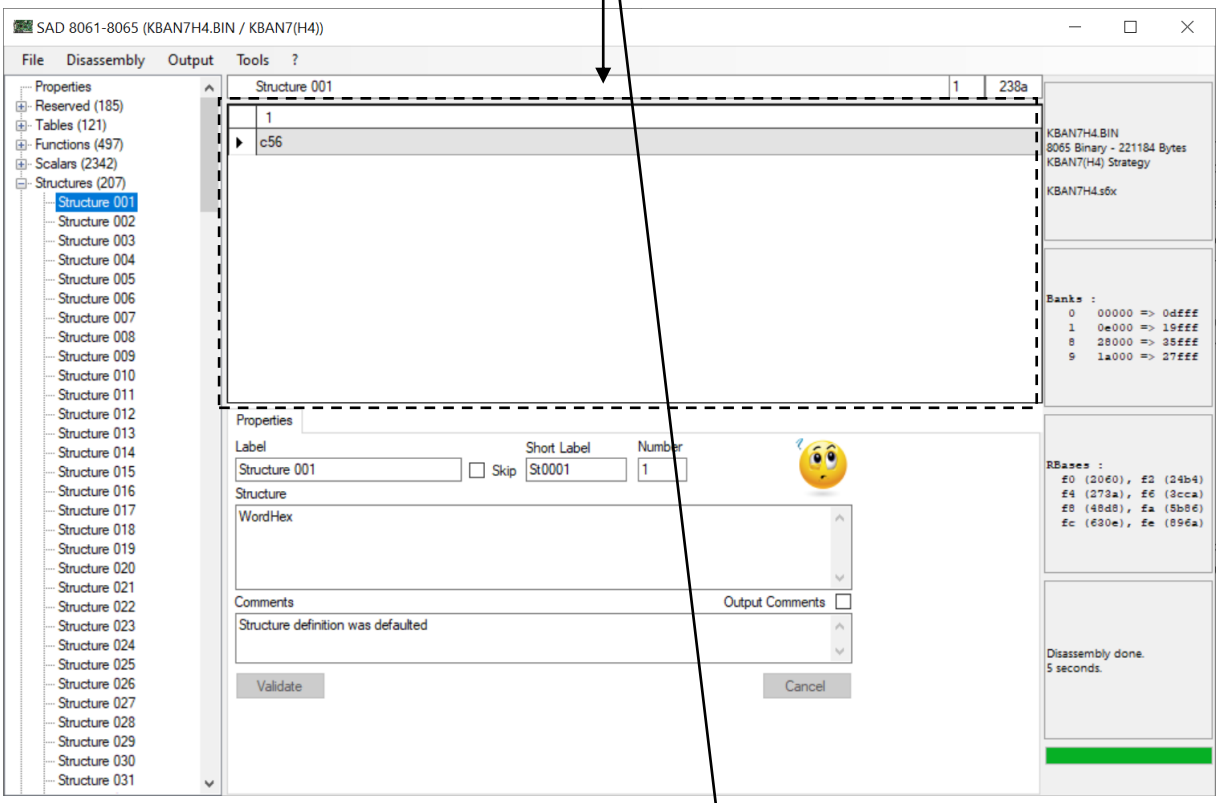
Main information for structures are its definition, just called ‘Structure’, which describes what is where and with which rules and its occurrence number ‘Number’, the number of times it repeats to give the whole structure.

Because we are disassembling based on Intel instructions, word values are store low byte first (LSB in TunerPro) in assembly.

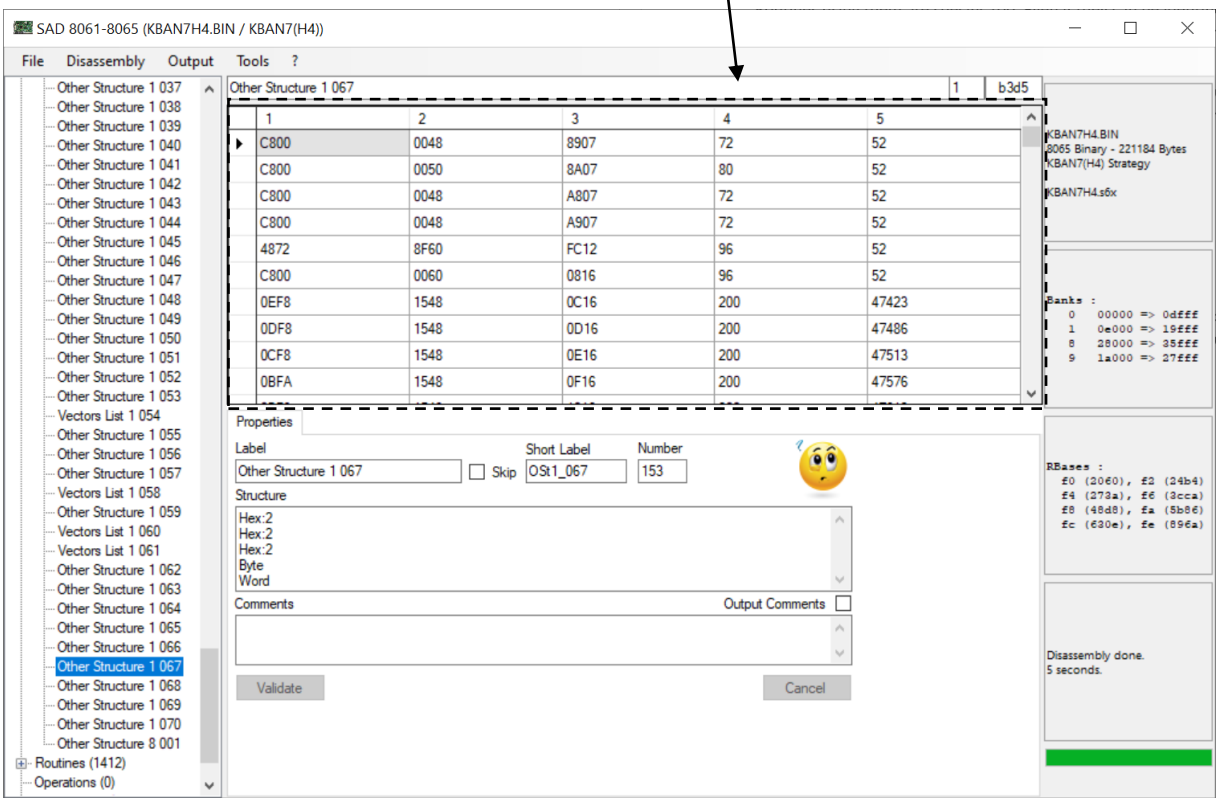
Structures auto detection is globally basic, but Structures definitions auto detection is something much more complex, because it requires to fully understand the routine(s), which are using the structure.

This is why some tries are done to complete these definition and number, but they often finish with a default definition (1 byte or 1 word) and a default number (1), but it remains a good start to analyze related routine and to prepare future signatures.

'Element Data' part looks like the following one:

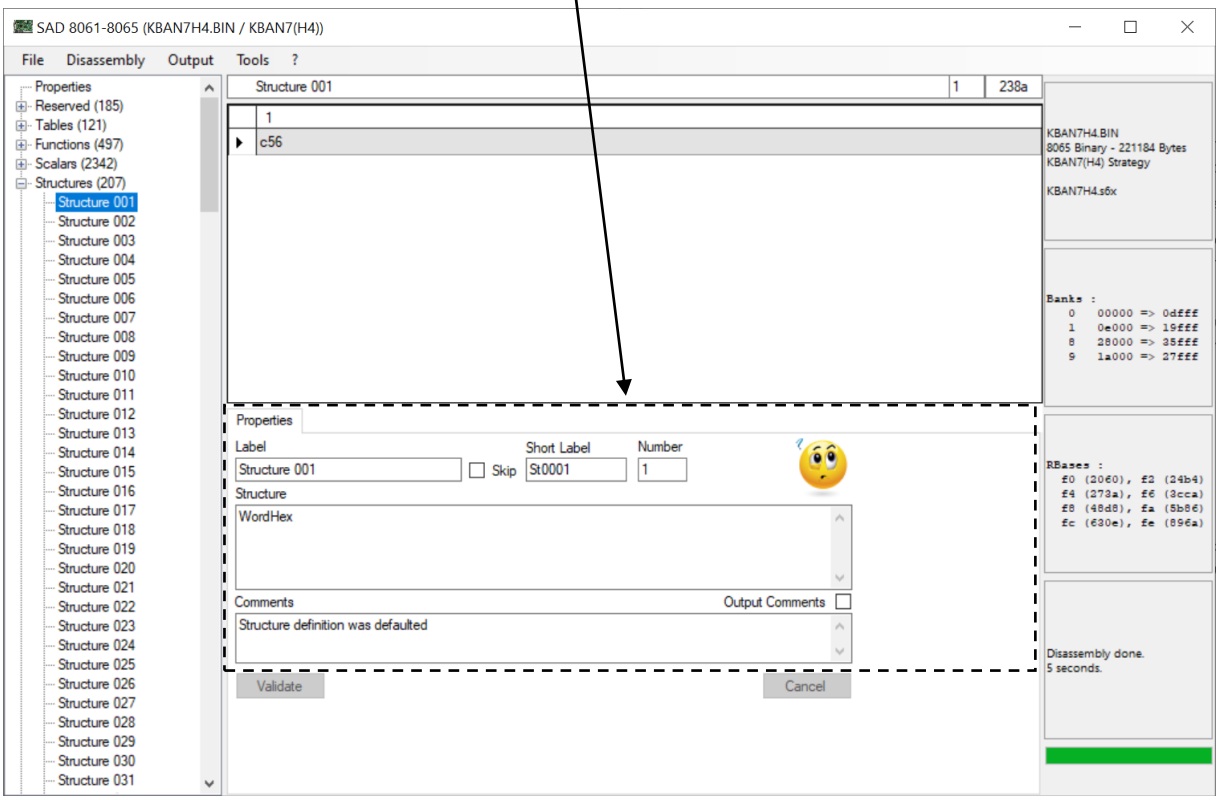


This is a basic data output and as you can see, when 'Comments' indicates that, it not fully recognized.



This one is much better, you can see that data output evolves based on structure definition.
With conditional rules, it can give other more complex things.

'Element Properties' part is the following one:



Another time, for something like all text fields, by using shortcut 'Ctrl-Shift-U' shortcut on selected text, text will be upper cased, with 'Ctrl-U' it will be lower cased.

Generic properties are like following:

'Label' : Auto generated by default, based on auto numbering. It will be visible at the element address in the output.
By using shortcut 'Ctrl-R' shortcut in this place, the related repository will be searched for a matching based on 'Short Label'. In our case 'Structures Repository'.

'Short Label' : Auto generated by default, based on auto numbering. It will be visible in code when element is used, and for sure at the element address too.
By using shortcut 'Ctrl-R' shortcut in this place, the related repository will be searched for a matching based on 'Short Label'. In our case 'Structures Repository'.

'Skip' : When skipped, user defined definition for element is ignored at disassembly. Auto detection comes back to override the defined element.

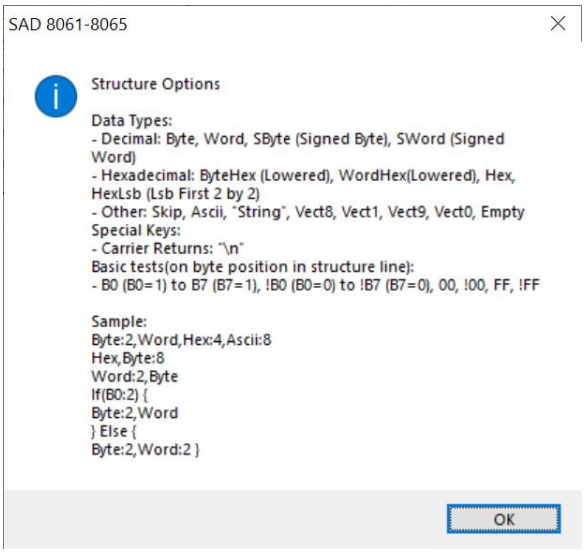
'Comments' : Empty by default, it indicates that definition detection has not been a success, when its value is 'Structure definition was defaulted'. It will be visible at the element address in the output only if 'Output Comments' is checked.
By using shortcut 'Ctrl-R' shortcut in this place, the related repository will be searched for a matching based on 'Short Label'. In our case 'Structures Repository'.

Structures specific properties are like following:

‘Structure’ : Auto detected if possible, otherwise it is indicated with ‘Comments’. It is the definition of the structure, which describes what is where and with which rules, in a comprehensible dedicated text format.

‘Number’ : Auto detected if possible, but not always used, based on type of definition. It is the number of times it repeats to give the whole structure.

‘Yellow Smiley’ image : With mouse over this image, you have some information, about how to write the structure definition. When clicking on it, you have a window with the same information. It is a good starting point.



Structure Definition format:

Following item keywords are available (do not use ‘ character):

- Decimal ones, with a decimal output:
  ‘Byte’ : Byte Item (1 Byte),
  ‘Word’ : Word Item (2 Bytes),
  ‘SByte’ : Signed Byte Item (1 Byte),
  ‘SWord’ : Signed Word Item (2 Bytes)

- Hexadecimal ones, with an hexadecimal output:
  ‘ByteHex’ : Byte Item with lowered output (1 Byte),
  ‘WordHex’ : Word Item with lowered output (2 Bytes),
  ‘Hex’ : Hexadecimal Item (1 Byte),
  ‘HexLsb’ : Hexadecimal Item with Lsb First 2 by 2 (1 Byte)

- Other ones, with a specific meaning:
  ‘Ascii’ : 1 Byte gives an Ascii output,
  ‘Skip’ : 1 Byte Item is ignored,
  ‘Empty’ : 1 Empty Item, not related with data, only for formatting,
  ‘"STRING"’ : 1 string as output, not related with data (STRING here),
  ‘"\n"’ : 1 carrier return, not related with data,
  ‘Vect8’ : 2 Bytes for a vector address. First operation will be on bank 8,
  ‘Vect1’ : 2 Bytes for a vector address. First operation will be on bank 1,
  ‘Vect9’ : 2 Bytes for a vector address. First operation will be on bank 9,
  ‘Vect0’ : 2 Bytes for a vector address. First operation will be on bank 0

VectX keywords, will permit to extend disassembly on the detailed vectors (or routines), if they were not found before. You will see that a specific type of structure, which is ‘Vector List’, is generated, it is a main part of disassembly on works on this base.

Following separators keywords are available, they permit to keep code more clear (do not use ‘ character):

- ‘ ’ : Space one,
- ‘,’ : Comma one,
- Carrier return : Carrier return one

Multiplying items principle:

Instead of writing ‘Hex,Hex,Hex,Hex’, you can write ‘Hex:4’. This is the case for all types of items, by adding after the item keyword ‘:N’ where N is the number of desired items.

Conditional rules principle:

To use condtional rules, you have to use following architecture:
'If (<CONDITION>:#BytePosition) { Items list (when <CONDITION>:#BytePosition) is true) } Else { Items list (when <CONDITION>:#BytePosition is false) }

You can include new conditions inside items lists, because a conditional rule is managed like an item itself.

Condiction definition for <CONDITION> and #BytePosition:

<CONDITION> can have limited number of values, based what is really required for a structure. It is essentially a bit check or a value comparison and it can be negated, like following:

- 'B0' / '!B0' : B0 = 1 / B0 = 0
- 'B1' to 'B7' : B1 = 1 to B7 = 1, so B2, B3, B4, B5, B6 too
- '!B1' to '!B7' : B1 = 0 to B7 = 0, so B2, B3, B4, B5, B6 too
- '00' / '!00' : Related Byte = 0x00 / Related Byte <> 0x00
- 'FF' / '!FF' : Related Byte = 0xFF / Related Byte <> 0xFF
- '01' to 'FE' : It works also for other values, except in range B0 to B7.
- '!01' to '!FE' : It works also for other values, except in range B0 to B7.

#BytePosition is the Byte position to test inside the current structrue occurrence.
If structure defition is 2 Bytes with a number of 3, you can only set a test for Byte 1 or Byte 2.
It can become a bit complicated with variable size occurences, but in fact routine using structure is working like that.

For sure for each occurrence, the right value will be checked.

Some example of conditional rule:

If (B0:2) { Byte:2,Word } Else { Byte:2,Word:2 }

which can be written

```
If (B0:2) {
Byte:2,Word
} Else {
Byte:2,Word:2
}
```

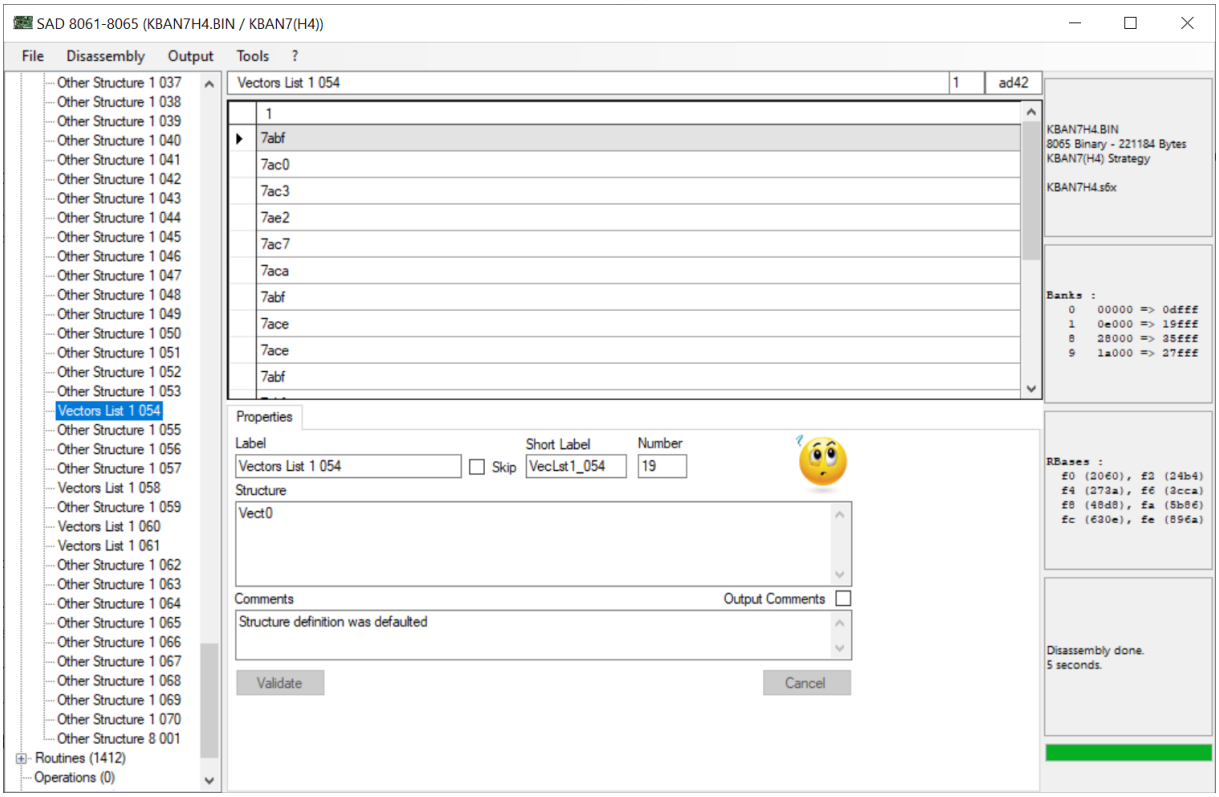
or

If (!B0:2) { Byte:2,Word:2 } Else { Byte:2,Word }

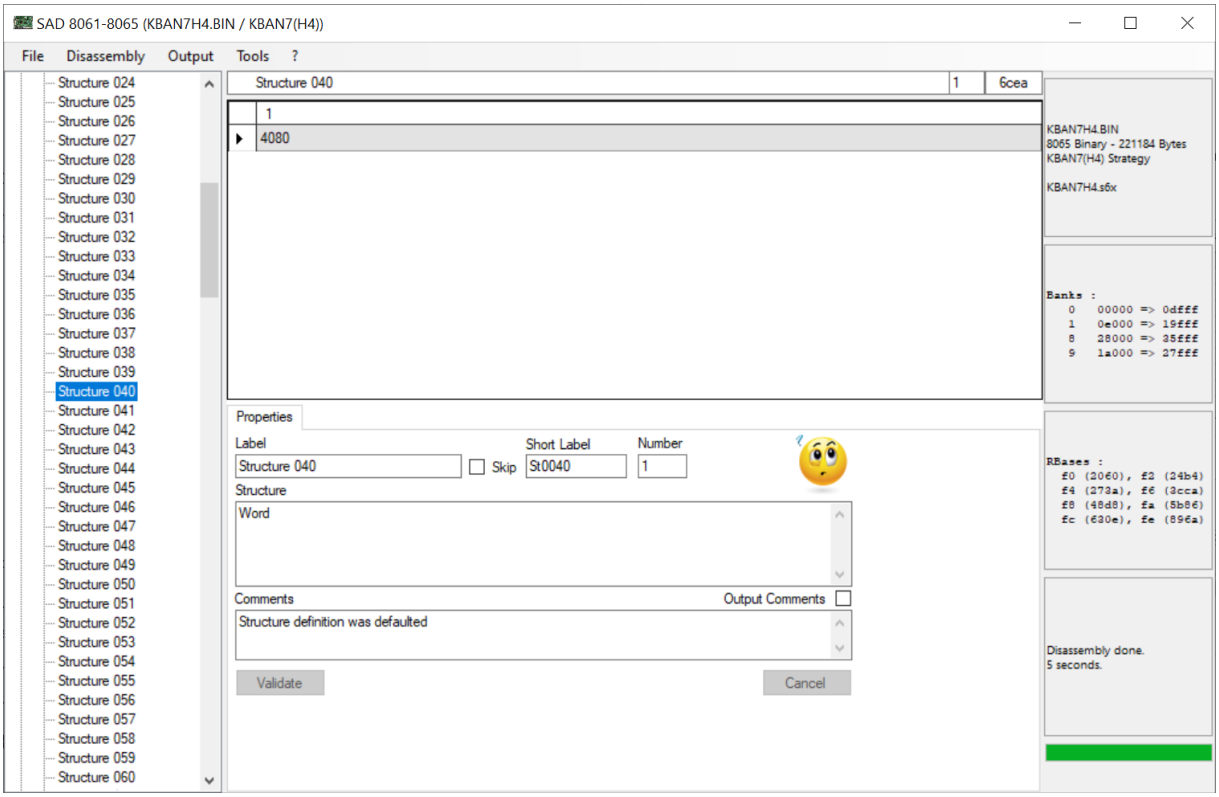
With a bit of practicing, things will become more clear.

Structure specificities:

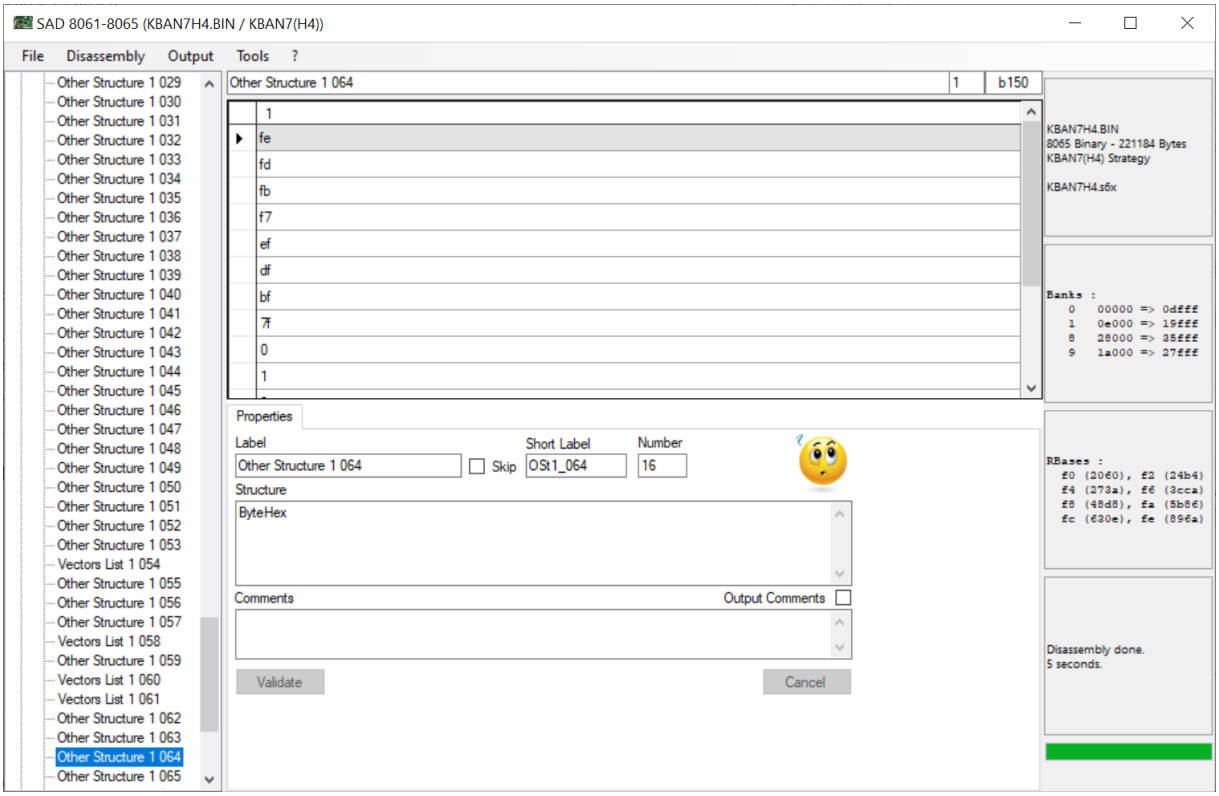
A described previously, you have different types of structures, including ‘Vectors List’.



But also classical calibration structure, related with RBases.



And main part of them, not related with RBases.



As you can see they use by default specific ‘Label’ and ‘Short Label’ based on their type, to help differenciating them.

‘Structures’ category menu:

No specificity at all.

‘Structure’ element menu:

No specificity at all.

Routines:

Routines do not exist in a real way in assembled code. An assembly is just a huge quantity of operations, not written like it would be with a modern tool, grouped inside routines. SAD 806x tries to recreate a kind of operations architecture, were it appears to be nothing. Based on calls, gotos and return instructions, something begins to appear and 'Routines' that come to this definition are some kind of best of.

A real routine would be a part of code, called many time and we have some routines like this. For example, Tables or Functions are detected based on dedicated routines, previously detected.

Basically, a routine is defined by the address of its first operation.

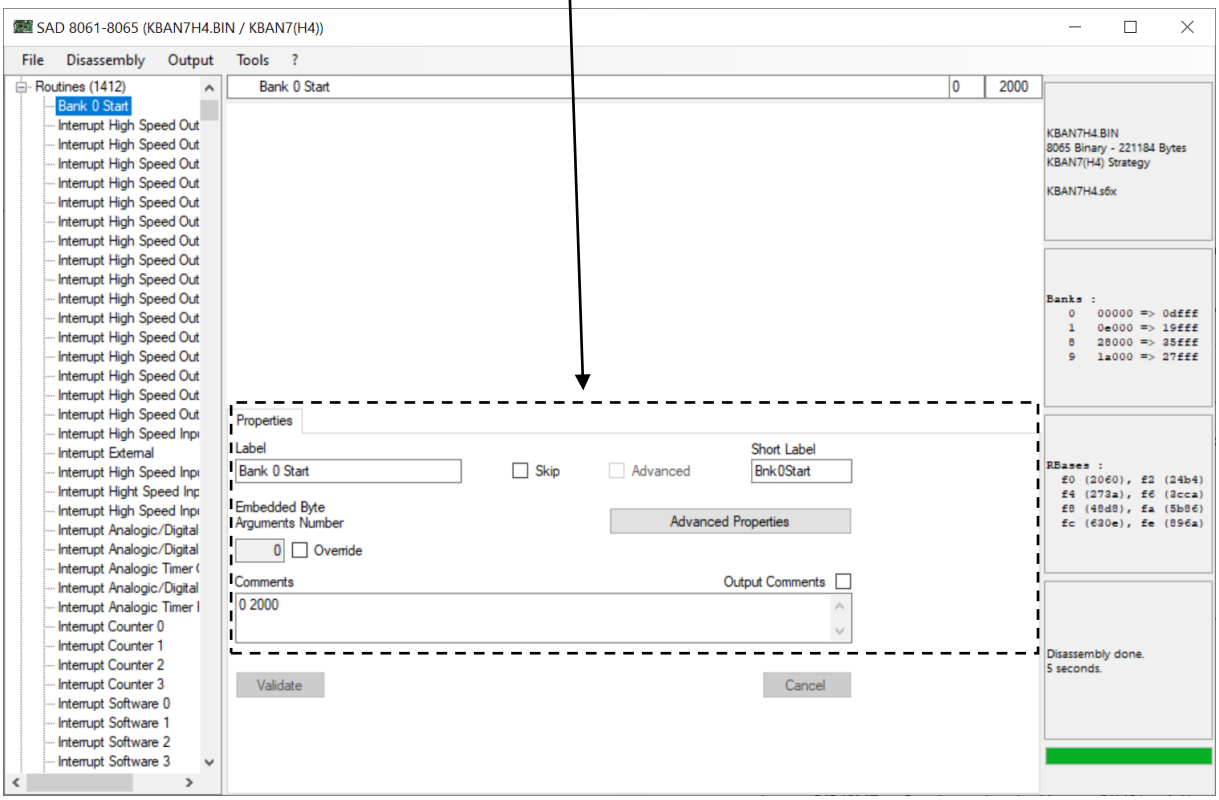
In SAD 806x routines are used to add some labels in output, as far as obtaining a better disassembly by adding new routines or setting up their parameters, if they have some.

Another use of them is at comparison level, because between strategies, if some routines are similar, you have chances to use the same calibration elements in them. So they are an excellent base to compare strategies.

Some known routines are fixed at their address level vs bank number vs EEC version, so they will be directly named and given as result.

Another way to create them is to work with signatures. Tables or Functions routines are detected through hard coded signatures, but you can create new signatures, inside definition to auto identify known routines shared between strategies.

‘Routine Properties’ part is the following one:



Generic properties are like following:

‘Label’ : Auto generated by default, based on auto numbering or based on known addresses. It will be visible at the routine address in the output.

‘Short Label’ : Auto generated by default, based on auto numbering or based on known addresses. It will be visible in code when routine is called, and for sure at the routine address too.

‘Skip’ : When skipped, user defined definition for routine is ignored at disassembly. Auto detection comes back to override the defined routine.

‘Comments’ : Auto generated by default, with address in this case. It will be visible at the element address in the output only if ‘Output Comments’ is checked.

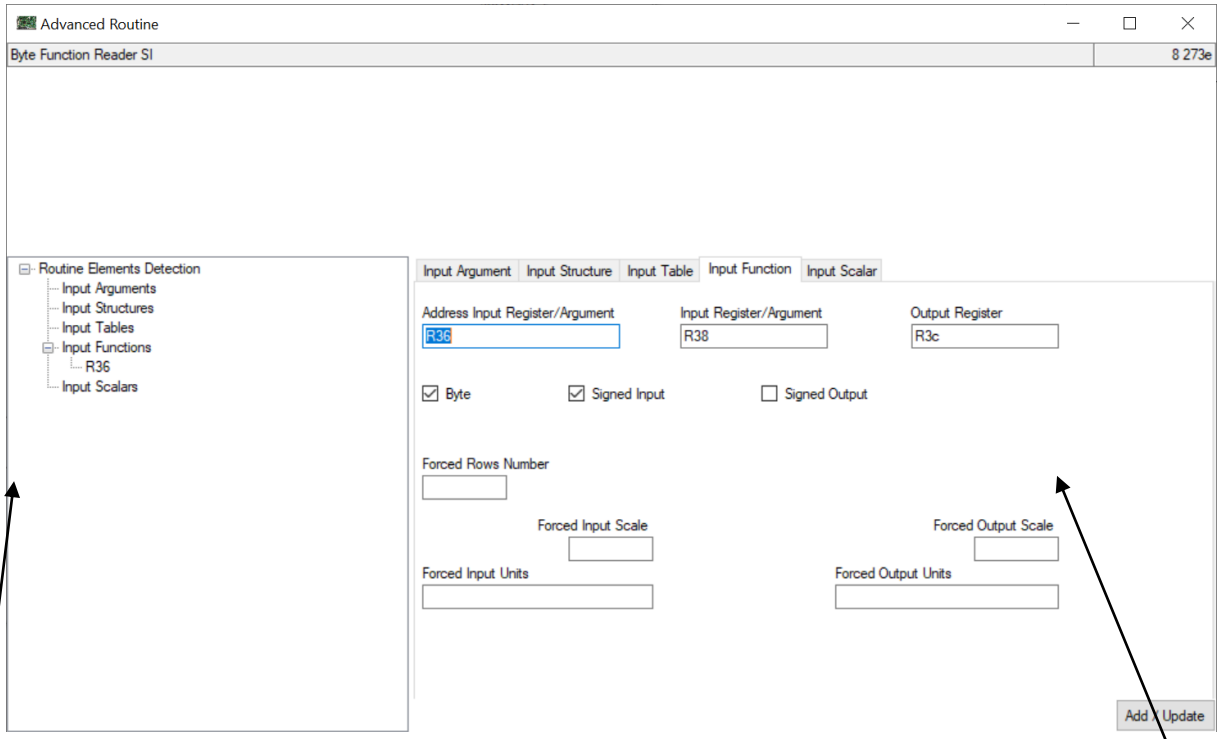
Routines specific properties are like following:

‘Advanced’ : Checked, it indicates that routine is an advanced one, that it is related with special scalars or structures, function or tables, or that it basically has arguments when called. It is a read only information, which can be managed only with ‘Advanced Properties’ button. It is auto detected or can be generated by signatures.

‘Embedded Byte Arguments Number’ : This is the number of arguments, provided when routine is called. It is a read only information, which can be managed only with ‘Advances Properties’ button. It is auto detected or can be generated by signatures. ‘Override’ checkbox permit to order SAD 806x to prefer updated number (built through ‘Advanced Properties’ form), instead of detected one.

Because it is a sensible information, which can broke the disassembly, by putting bad operations at bad addresses, ‘Override’ option should be managed properly.

'Routine Advanced Properties' form is the following one:

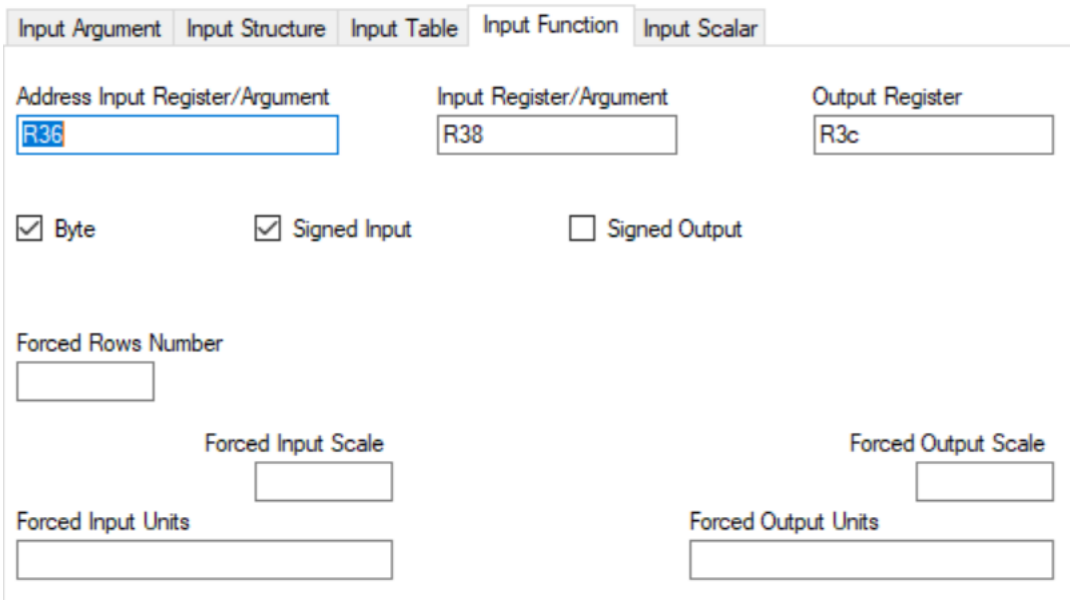


It is accessible, through the 'Advanced Properties' button. In our cases we will start from a Byte Function Reader, which is a good basis for description.

The left part of the screen is a list of possible Inputs for the routine. The right one will detail properties for these inputs. To add a new Input, just right click on the right place on the list to access the 'New Element' menu option for Inputs.

Basically known functions routines have only 1 input function (its address in fact) and can have arguments, known table routines have only 1 input table (its address in fact) and no arguments.
It is possible to setup a routine with everything in multiple samples, but for now realistic things will be easier to explain.

Principle is simple, with a routine like this one, we are able to detect, for each call, a function and its parameters, so it is interesting to setup this, when possible.

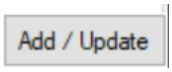


'Input Function' properties can be described like following:

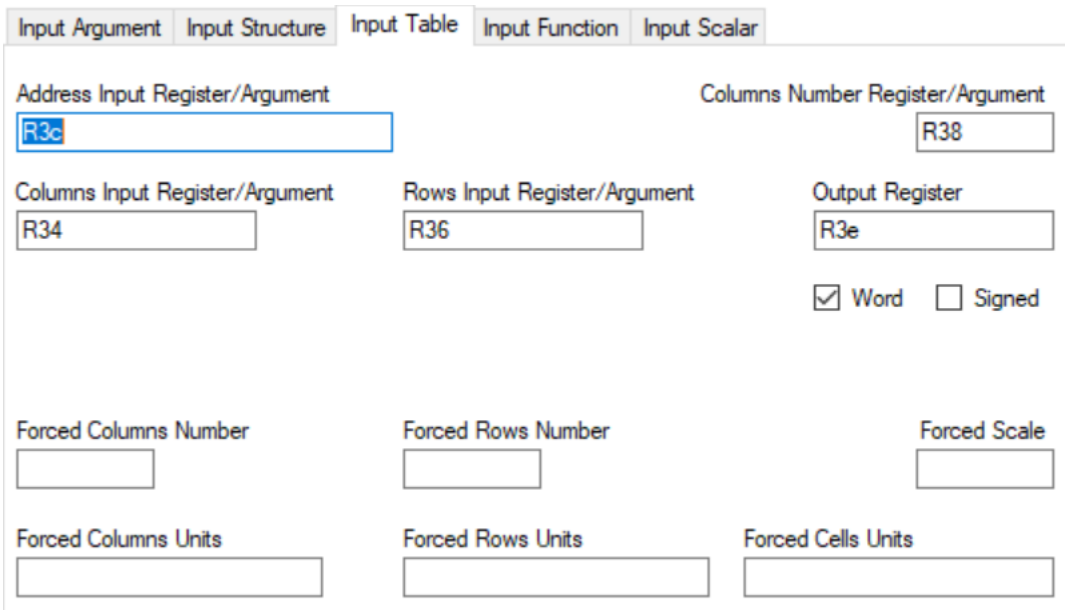
- 'Address Input Register/Argument' : This is the register containing function address when routine is called. It is always a generic register (erased and rewritten for generic puprose). Here 'R36' is one of them for this strategy. It could also be an Argument, when routine uses arguments and in this case is should be written 'Ar01' to 'Ar99', you will understand why.
- 'Input Register/Argument' : This is the register containing the Input value of the function, still a generic one. It is interesting for SAD 806x for tracing source of data for function, as far for proper recognizing. Argument can be used too.
- 'Output Register' : This is the register that will receive the Output result of the function, still a generic one. SAD 806x uses it to follow data, to detect scalers, as far for proper recognizing.
- 'Byte' : Checked, it will define if the provided function is a byte one. Otherwise, it will be a word one.
- 'Signed Input' : Checked, it will define if the provided function has a signed input. Otherwise, it will have an unsigned one.
- 'Signed Output' : Checked, it will define if the provided function has a signed output. Otherwise, it will have an unsigned one.
- 'Forced Rows Number' : Empty, SAD 806x will detect rows number, filled, this rows number will be used, this is the case when rows number is hard coded in routine.
- 'Forced Input Scale' : When filled, it will apply to all related functions.
- 'Forced Output Scale' : When filled, it will apply to all related functions.
- 'Forced Input Units' : When filled, it will apply to all related functions.

- ‘Forced Output Units’ : When filled, it will apply to all related functions.

As reminder, a routine with only 1 ‘Input Function’, with or without ‘Input Arguments’, will be managed as a Function routine, to detect functions.



‘Add / Update’ button permits to validate creation or update. Do not forget it between inputs or before closing ‘Advanced Properties’ form.

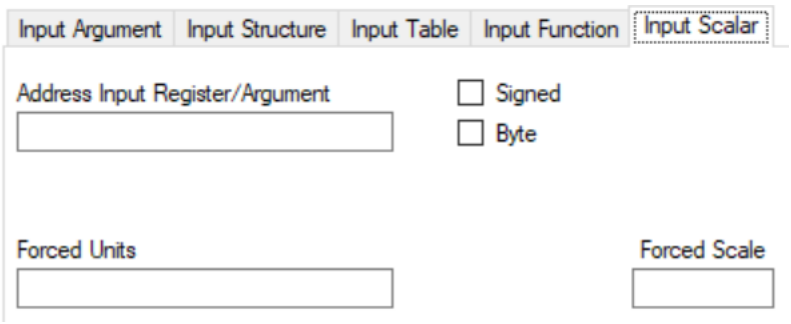


'Input Table' properties can be described like following:

- 'Address Input Register/Argument' : This is the register containing table address when routine is called. It is always a generic register (erased and rewritten for generic puprose). Here 'R3c' is one of them for this strategy. It could also be an Argument, when routine uses arguments and in this case is should be written 'Ar01' to 'Ar99', you will understand why.
- 'Columns Number Register/Argument' : This is the register containing the columns number of the table, still a generic one. It is required for SAD 806x for table recognizing. Argument can be used too.
- 'Columns Input Register/Argument' : This is the register containing the column Input value of the table, still a generic one. It is interesting for SAD 806x for tracing source of data to find scaling functions, as far as for proper recognizing. Argument can be used too.
- 'Rows Input Register/Argument' : This is the register containing the row Input value of the table, still a generic one. It is interesting for SAD 806x for tracing source of data to find scaling functions, as far as for proper recognizing. Argument can be used too.
- 'Output Register' : This is the register that will receive the Output result of the table, still a generic one. SAD 806x uses it to follow data, as far as for proper recognizing.
- 'Word' : Checked, it will define if the provided table is a word one. Otherwise, it will be a byte one.
- 'Signed' : Checked, it will define if the provided table has a signed output. Otherwise, it will have an unsigned one.

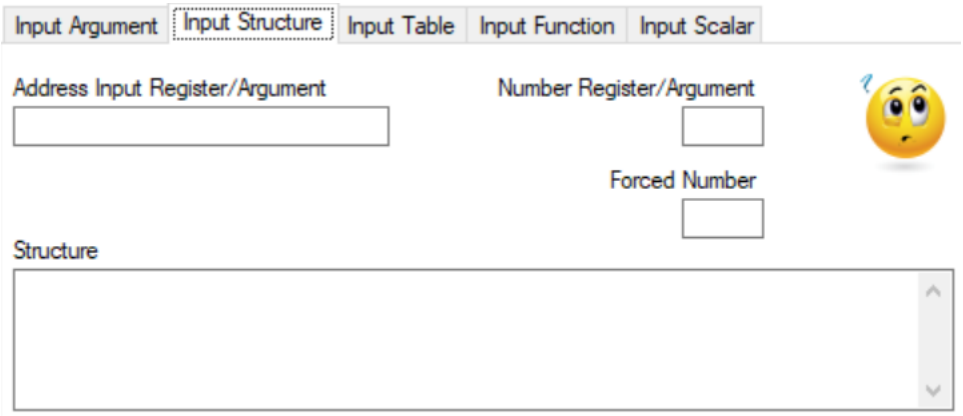
- 'Forced Columns Number' : Empty, SAD 806x will search for columns number, filled, this columns number will be used, this is the case when columns number is hard coded in routine.
- 'Forced Rows Number' : Empty, SAD 806x will detect rows number, filled, this rows number will be used, this is the case when rows number is hard coded in routine.
- 'Forced Scale' : When filled, it will apply to all related tables.
- 'Forced Columns Units' : When filled, it will apply to all related tables.
- 'Forced Rows Units' : When filled, it will apply to all related tables.
- 'Forced Cells Units' : When filled, it will apply to all related tables.

As reminder, a routine with only 1 'Input Table', with or without 'Input Arguments', will be managed as a Table routine, to detect tables.



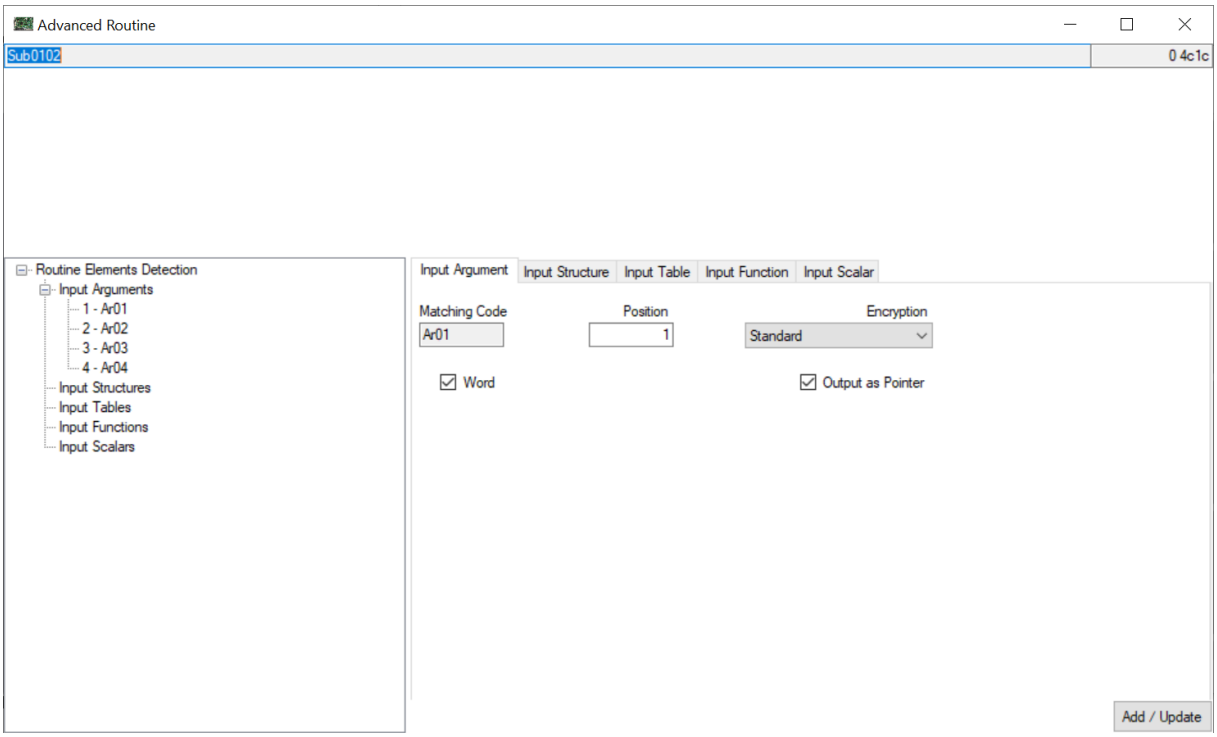
'Input Scalar' properties can be described like following:

- 'Address Input Register/Argument' : This is the register containing scalar address when routine is called. It is always a generic register (erased and rewritten for generic puprose). It could also be an Argument, when routine uses arguments and in this case is should be written 'Ar01' to 'Ar99', you will understand why.
- 'Signed' : Checked, it will define if the provided scalar has a signed output. Otherwise, it will have an unsigned one.
- 'Byte' : Checked, it will define if the provided scalar is a byte one. Otherwise, it will be a word one.
- 'Forced Units' : When filled, it will apply to all related scalars.
- 'Forced Scale' : When filled, it will apply to all related scalars.



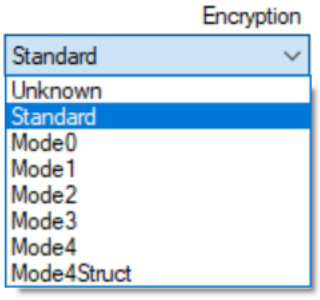
'Input Structure' properties can be described like following:

- 'Address Input Register/Argument' : This is the register containing structure address when routine is called. It is always a generic register (erased and rewritten for generic puprose). It could also be an Argument, when routine uses arguments and in this case is should be written 'Ar01' to 'Ar99', you will understand why.
- 'Number Register/Argument' : This is the register containing the structure repeat number, still a generic one. It is for SAD 806x for structure recognizing. Argument can be used too.
- 'Forced Number' : This is the value for the structure repeat number. It can be hard coded to prevent SAD 806x to detect it. Then it overrides previous 'Number Register/Argument' value.
- 'Structure' : This is the structure definition, that will be used. SAD 806x does not attach a structure definition to a routine automatically, so it is required here.
- 'Yellow Smiley' image : With mouse over this image, you have some information about, how to write the structure definition. When clicking on it, you have a window with the same information. It is a good starting point.



'Input Argument' properties can be described like following:

- 'Position' : This is the position of the argument for the call. First one is 1, second one 2 and so on…
- 'Word' : Checked, argument will be a 2 bytes one.
- 'Output as Pointer' : On autodetection default is true. It permits to be managed as a pointer, which gives a different text output.
- 'Matching Code' : This is the generated code, to be reused in other inputs, it is read only and based on 'Position'. Now, you understand 'Ar01' to 'Ar99'.



- 'Encryption' : Sometimes arguments are provided to be used directly, without operation on their value, sometimes not, arguments are encrypted in this case.
  'Unknown' : With 'Unknown' Encryption, SAD 806x tries to detect it. If it is not possible, it will be managed as 'Standard'.
  'Standard' : It is the case when argument is not encrypted.
  'Mode0' : This mode is not for an encryption, but to set that parameter is a calibration element, which is not using an RBase.

'Mode1' : Is not managed for now and will be processed as 'Standard', because I have never seen it until now.
'Mode2' : Is not managed for now and will be processed as 'Standard', because I have never seen it until now.
'Mode3' : Just an example.

```
f03a (3a,f0) => F[0+((f - 8) * 10) / 8] => F[0+70/8] => F[0+E] => FE + 03a
010c (0c,01) => Not compatible => [10c]
```

'Mode4' : Just an example.

```
44,22 => 2244 => 2 + 244 => f2(f(0+2)) + 244 => RBase f2 + 244
44,32 => 3244 => 3 + 244 => f2(f(0+3-1)) + 1244 => RBase f2 + 1244
```

'Mode4Struct' : Extension of 'Mode4', to provide address in structure .

```
44,22 => 2244 => [2244],[2246]   => Values to read in a structure
     [2244] => 47,26 => 2647 => 2 + 647 => f2(f(0+2)) + 647 => RBase f2 + 647
     [2246] => 12,01 => [112] => Input Register
```

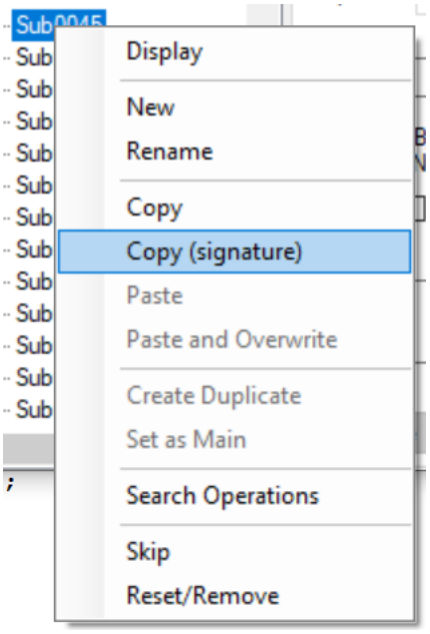
'Encryption' is correctly autodetected, normally no need to manage it.

Do not forget that arguments can be reused, with 'ArXX' in other inputs.

‘Routines’ category menu:

No specificity at all.

'Routine' element menu:



Even if it not possible to 'Copy (Xdf)' a routine, because this type of object is not managed through TunerPro, another option has been added for routines, it is 'Copy (signature)'.

Signature is not in another tool, it can be a part for SAD 806x definition. This is the 'Routines Signatures' category, which will be seen later on.

Signature of a routine is the code, which will be common for all strategies, for the same routine or type of routine.

'Copy (signature)' permits to paste on 'Routines Signatures', an exact copy of the related routine, including its advanced parameters and in addition, the hexadecimal code at the beginning of the routine, as base of the signature that will have to be reviewed.

When using this 'Copy (signature)' you will understand its interest for working on signatures.

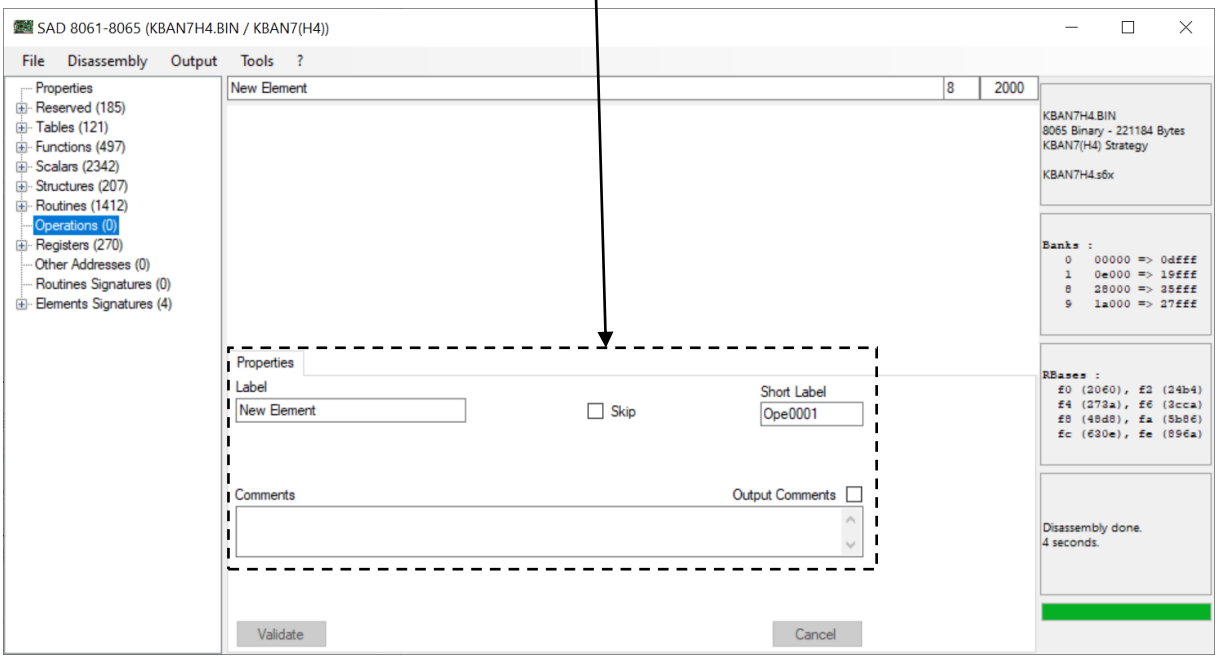
Operations:

An operation is an instruction + its parameters (if available) + its arguments (if available).
Routines are a set of operations.
For sure an operation has an address in the binary, this is in fact the only thing that is interesting in this part.

After disassembly, unlike routines, operations are not loaded in SAD 806x elements tree. This is because of the huge number of operations which are detected and because nothing specific can be setup on them, except basic information.

But it can be interesting to declare existing operations, to set their labels or comments and to declare new operations, when they were missing.
If you see a block of undisassembled code, it can be a structure or code. On my side, I create a basic routine at this address and after redisassembly, I check if disassembly has be well managed around this. So no need to create operation here. But sometimes, you can see a skip or goto, original or patched, that will ignore 1 or 2 operations, and there, it interesting to see what was ignored, there I will create the missing operation, because, SAD 806x, which follows the code, can not arrive at this place, except if we declare to do it.

‘Operation Properties’ part is the following one:



Generic properties are like following:

‘Label’ : Auto generated by default, based on auto numbering or based on known addresses. It will be visible at the operation address in the output, like a header.

‘Short Label’ : Auto generated by default, based on auto numbering or based on known addresses. It will be not be visible in the output.

‘Skip’ : When skipped, user defined definition for operation is ignored at disassembly. Auto detection comes back to override the defined operation.

‘Comments’ : Empty by default. It will be visible at the element address in the output only if ‘Output Comments’ is checked.

‘Operations’ category menu:

No specificity at all.

‘Operation’ element menu:

No specificity at all.

Registers:

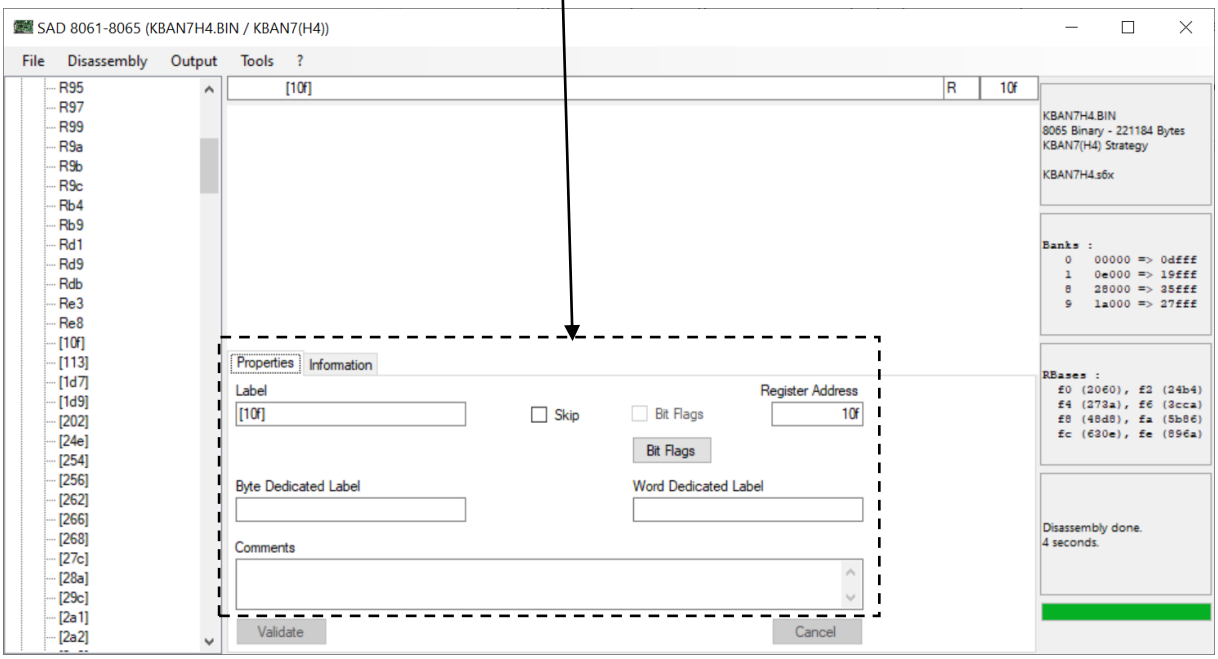
A register is an EEC memory address, not related with rom, used to share data or information inside program. Main part of registers have one unique purpose in our case, which makes them interesting to identify. The other part are generic or temporary registers (erased and rewritten), which have to be managed but, only to find the other ones.

I will not detail for now the related addresses, but globally on EECV addresses start at 0x0000 to go to 0x1FFF and another part can be used from 0xF000 to 0xFFFF.

A part of these addresses are reserved and detected like this. But that is the other part which will help us to understand disassembly.

After disassembly, like routines, only a small part of detected registers are loaded in SAD 806x elements tree. Functions inputs, outputs, like Tables inputs, outputs ar kept to be inserted in registers list.

‘Register Properties’ part is the following one:



Generic properties are like following:

‘Label’ : By default it is what will appear in output. It will be visible each time register is used in output, except if specific ‘Byte Dedicated Label’ or ‘Word Dedicated Label’ are defined.

‘Register Address’ : Register are working a bit differently, it is not possible to update their address on the top of the screen and they have no bank, but a specific code. So you have to set it or update it directly in this place.
As I have described, range for addresses is checked, based on what was described previously, but another use can be done here for addresses, you can use this type of setup : ‘XX+YY’, for example For example [Ra3+12] has to use address a3+12. ‘Ra3’ is some kind of constant (RConst) and ‘12’ is its gap to the defined register. Gap can have all values inside registers addresses range.

‘Skip’ : When skipped, user defined definition for register is ignored at disassembly. Auto detection comes back to override the defined register.

‘Comments’ : Empty by default. It will be visible only in registers lists, if option ‘Register list output’ was chosen in definition global properties.

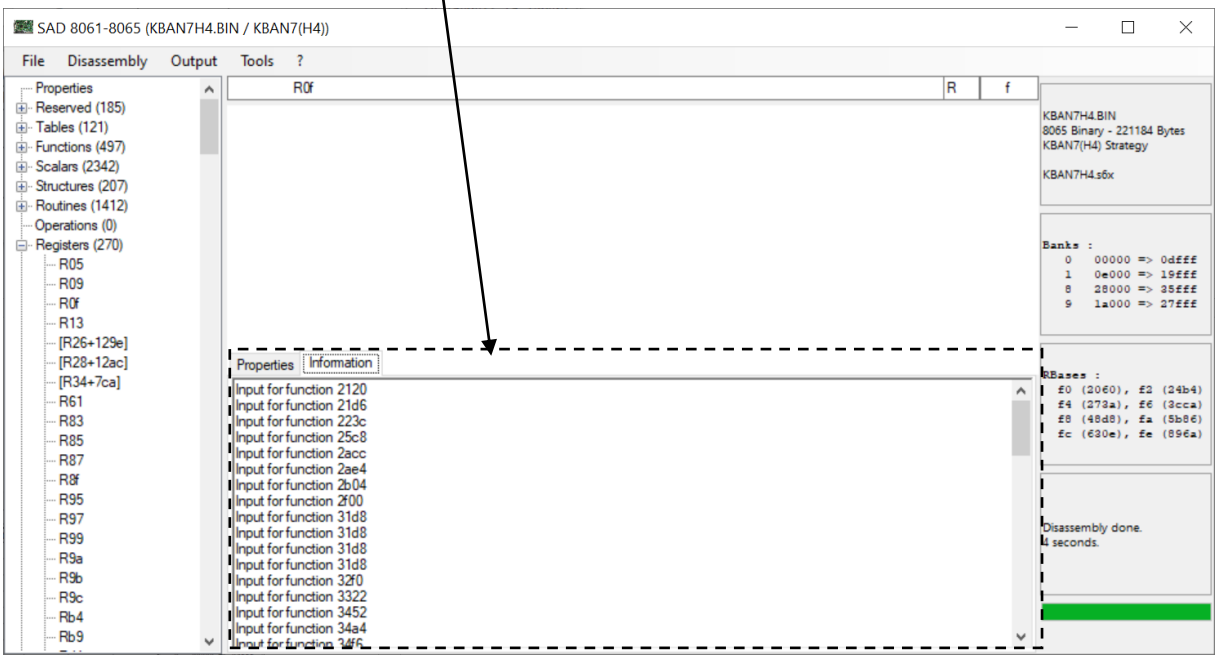
Specific properties are like following:

‘Bit Flags’ : Exactly the same setup than for scalars, but there, it permits to manage bit flags and displays their labels in output instead of ‘Label’, ‘Byte Dedicated Label’ or ‘Word Dedicated Label’, when register is used in bit operations.

‘Byte Dedicated Label’ : In some strategies, registers have not the same meaning when they are used in byte operations versus word operations. This is why additional labels were added. If ‘Byte Dedicated Label’ is set, it will be used in output for byte operations, otherwise ‘Label’ will be used.

‘Word Dedicated Label’ : If it is set, it will be used in output for word operations, otherwise ‘Label’ will be used.

‘Element Information’ for Register:



Registers possess an additional ‘Element Information’ tab too, which includes additional details grabbed during disassembly and interessant to be known.

In this case, we discover, register is used many times as input for functions, no surprise here, R0f is RPM, it can be seen quickly by looking at these functions. If a register is used directly as input for tables, you can be sure it is a dedicated scaler register. For sure, when labels are redefined, elements appear translated here.

‘Registers’ category menu:

No specificity at all.

‘Register’ element menu:

No specificity at all.

Other addresses:

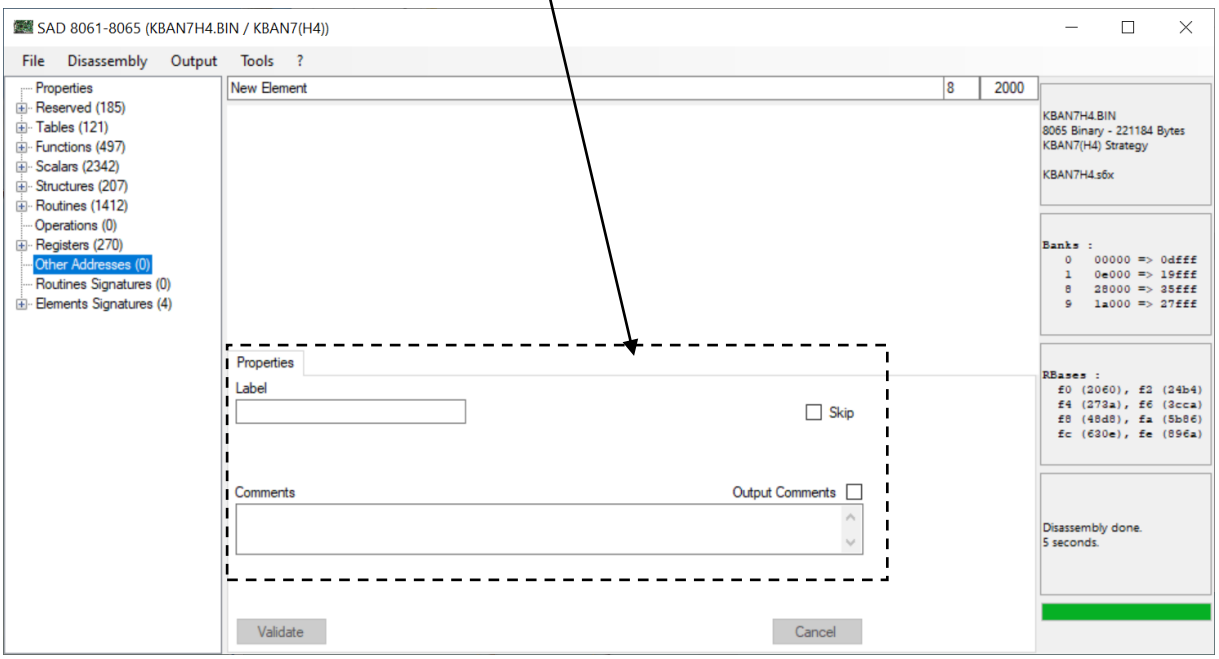
Sometimes, we know that something is present at an address, but we do not know exactly, what it is at the moment, or we want to keep trace of an address without entering more details now, the other address is the perfect place to do it.

It is massively used after a SAD directive file import, for unrecognized addresses or elements.

At disassembly, when SAD 806x detect something, element, operation and when it was not properly defined, it looks at this place, to search for and existing address and to enrich its element.

Definition of this type of thing, is really limited, but totally necessary for some cases.

‘Other Addresses’ part is the following one:



Generic properties are like following:

‘Label’ : It will appear in output at specified address, if nothing else was declared.

‘Skip’ : When skipped, it is ignored at disassembly.

‘Comments’ : It will appear in output at specified address, if nothing else was declared and if ‘Output Comments’ is checked.

‘Other Addresses’ category menu:

No specificity at all.

‘Other Address’ element menu:

No specificity at all.

Routines Signatures:

'Routines Signatures' part is the most complicated part in this application, because it is related with a definition and because it should be possible to duplicate it on others.

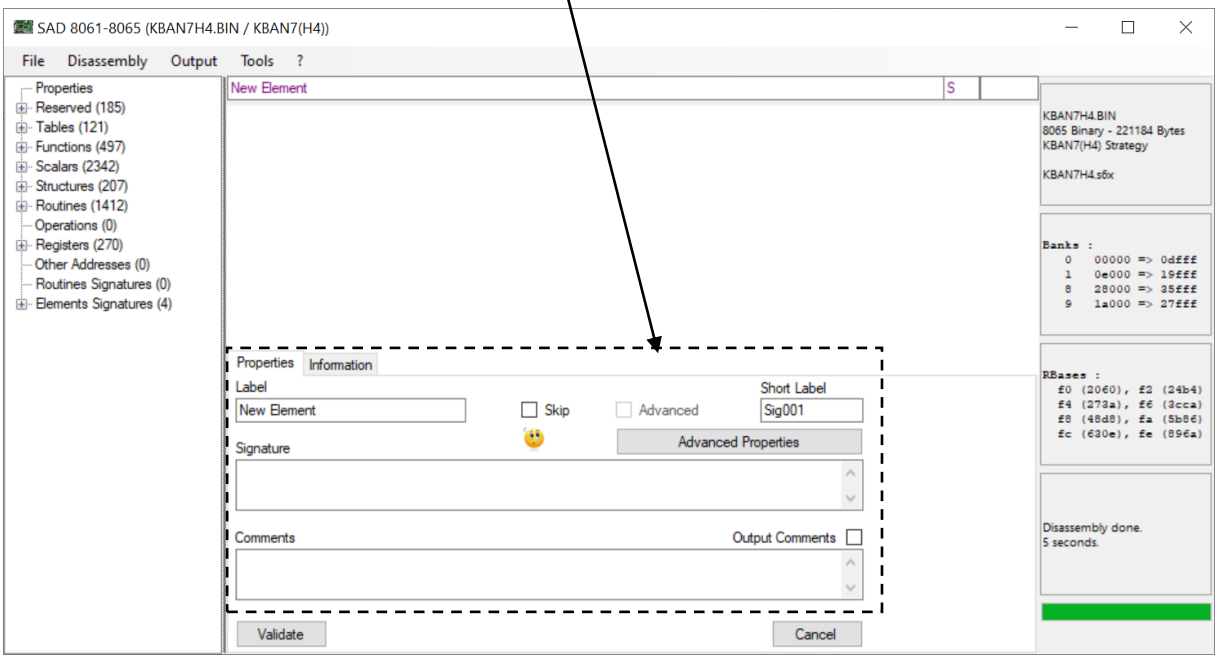
Purpose of a routine signature, is to detect a routine on disassembly, based on some kind of hexadecimal signature and to give it a name and a meaning. So it should be able based on a well written signature to automatically create a routine, in the 'Routines' part, with a well-defined and pre-defined 'Label', 'Short Label' and 'Comments'.

Because a routine can be advanced, like it was seen in 'Routines' part, detected routine should get all required advanced parameters directly, so it should be possible to pre-define them, even if it should stay an option.

But why not in this case, being able to attach elements to this routine, too, because this routine can contain use of elements and because it is strange to detect a known routine and to let its elements unknown. So it should be possible to pre-define elements inside this routine.

'Routines Signatures' part shall be able to do this, but as you will understand, creating a proper signature, which could be shared between different definitions and different technologies, is not so easy. In a perfect world, with the perfect list of 'Routines Signatures', a definition template could be able to disassemble everything properly without any additional human action. Just send me this template when you have finished it ;)

‘Routines Signature’ part is the following one:



Generic properties are like following:

‘Label’ : Auto detected routine, will be created with this ‘Label’. Output will also work like for a classical routine.

‘Short Label’ : Auto detected routine, will be created with this ‘Short Label’. Output will also work like for a classical routine.

‘Skip’ : When skipped, signature will not be searched for.

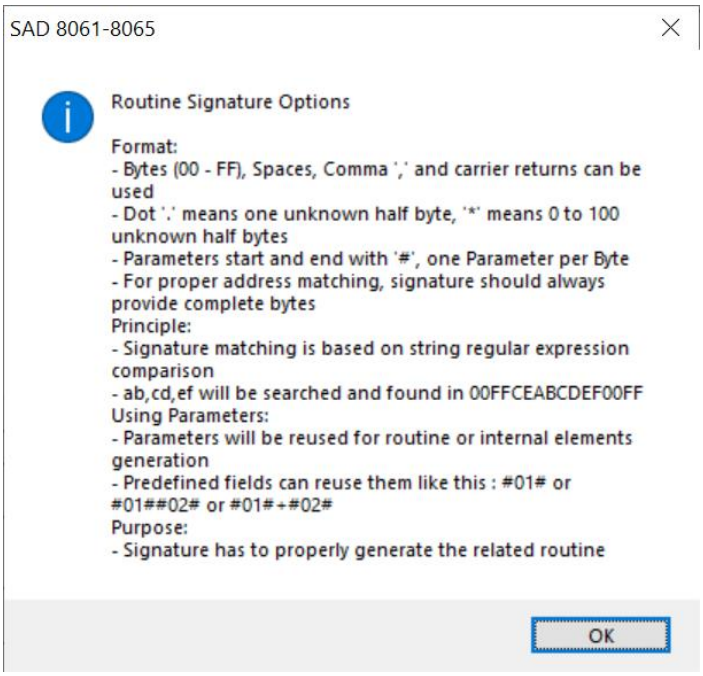
‘Comments’ : Auto detected routine, will be created with this ‘Comments’. Output will also work like for a classical routine, when ‘Output Comments’ duplicated on auto detected routine too, is checked.

Routines signature specific properties are like following:

‘Advanced’ : Checked, it indicates that auto detected routine will be an advanced one, that it is related with special scalars or structures, functions or tables, or that it basically has arguments when called. It is a read only information, which can be managed only with ‘Advanced Properties’ button.

‘Signature’ : This is the hexadecimal signature, which will permit to detect a routine. It will be described later on. Code written here should be unique in binary, to permit to detect only one routine and not another number of identical routines with same code.

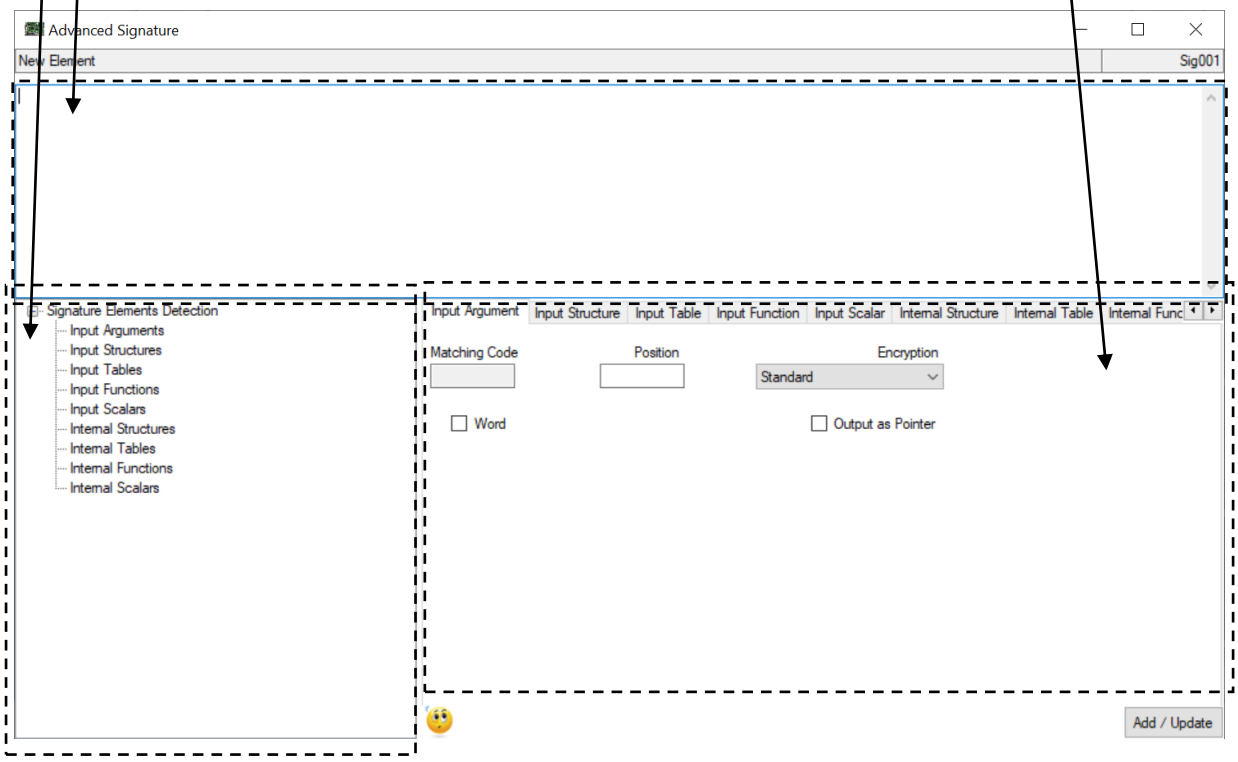
‘Yellow Smiley’ image : With mouse over this image, you have some information about, how to write the signature. When clicking on it, you have a window with the same information. It is a good starting point.


SAD 8061-8065
Routine Signature Options
Format:
- Bytes (00 - FF), Spaces, Comma ',' and carrier returns can be used
- Dot '.' means one unknown half byte, '*' means 0 to 100 unknown half bytes
- Parameters start and end with '#', one Parameter per Byte
- For proper address matching, signature should always provide complete bytes
Principle:
- Signature matching is based on string regular expression comparison
- ab,cd,ef will be searched and found in 00FFCEABCDEF00FF
Using Parameters:
- Parameters will be reused for routine or internal elements generation
- Predefined fields can reuse them like this : #01# or #01##02# or #01#+#02#
Purpose:
- Signature has to properly generate the related routine
OK

'Routines Signature' advanced properties:

By using button 'Advanced Properties', like for routines, it permits to access to the related form. This form is composed, with the following elements:

- Extended signature text box : to fill in Signature in this place.
- Signature Elements Detection List : list of all related elements, which will be created on detected routine or associated with it.
- Related elements properties : properties for all related properties.
- 'Yellow Smiley' image : still the same meaning.
- 'Add / Update' button : to validate element creation / update.



Just before describing possible elements which can be added, I will just describe an interesting specificity in signature. An example is better. This is what will output after disassembly:

```
8 b1f4: f2                pushp                   push(PSW);
8 b1f5: a1,00,24,2a       ldw   R2a,2400          R2a = 2400;
8 b1f9: c3,da,50,2a       stw   [Rda+50],R2a      [6d0] = R2a;
8 b1fd: c3,da,70,2a       stw   [Rda+70],R2a      [6f0] = R2a;
8 b201: c3,da,56,00       stw   [Rda+56],0        [6d6] = 0;
8 b205: c3,da,76,00       stw   [Rda+76],0        [6f6] = 0;
8 b209: f3                popp                    pop(PSW);
8 b20a: f0                ret                     return;
```

Hexadecimal code for this part is the following one :

F2A100242AC3DA502AC3DA702AC3DA5600C3DA7600F3F0

Not really usable, so yes like this it is better:

```
f2
a1,00,24,2a
c3,da,50,2a
c3,da,70,2a
c3,da,56,00
c3,da,76,00
f3
f0
```

If you remember well, this code and everything in fact, after basic disassembly, can come from the auto detected routine, with the 'Copy (signature)' from the related routine menu.

Just by using this code in signature I should be able to identify this routine, but only in that binary, but I want more, I want to create a scalar, at address '0x2400', and I want to identify 2 registers [6d0] and [6f0], to be reused in routine definition.

For this I will update the signature:

```
f2
a1,#02#,#01#,#03#
c3,#06#,#04#,#03#
c3,#06#,#05#,#03#
c3,#06#,..,00
c3,#06#,..,00
f3
f0
```

#XX# things are 'Signature parameters', one for each byte, they can have any value and '..' things are like one byte that can have any value too.

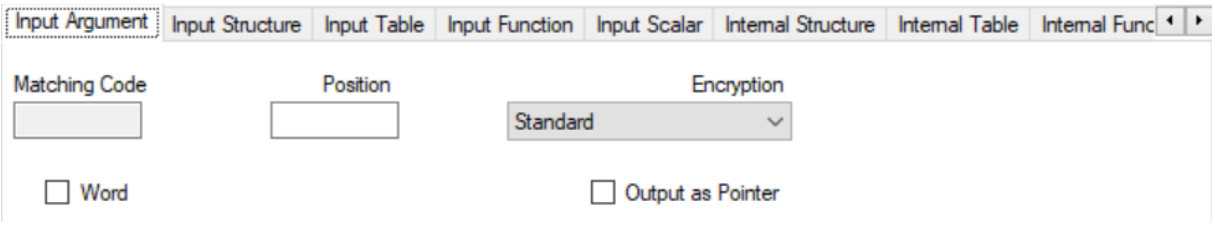
With that, my signature becomes much more generic (I hope not too much, this is the danger) and it can probably being reused in other strategies.

'Signature parameters' can now be reused in signature elements definition.
So my scalar address 0x2400 will be '#01##02#'.
Because SAD806x seems to know value for 'Rda', [6d0] will be '#06#+#04#' and [6f0] will be '#06#+#05#'.

Complicated, yes, but necessary.

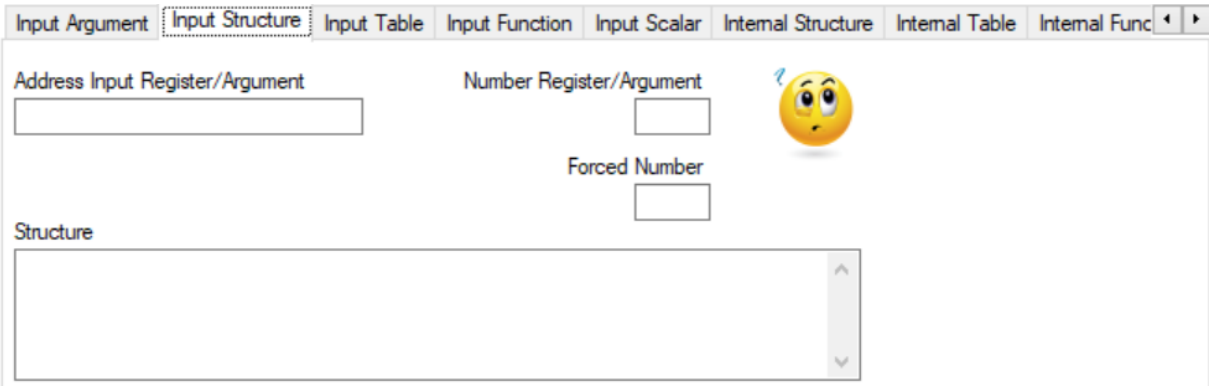
Like routines, added input elements can be like following:

- Input Argument



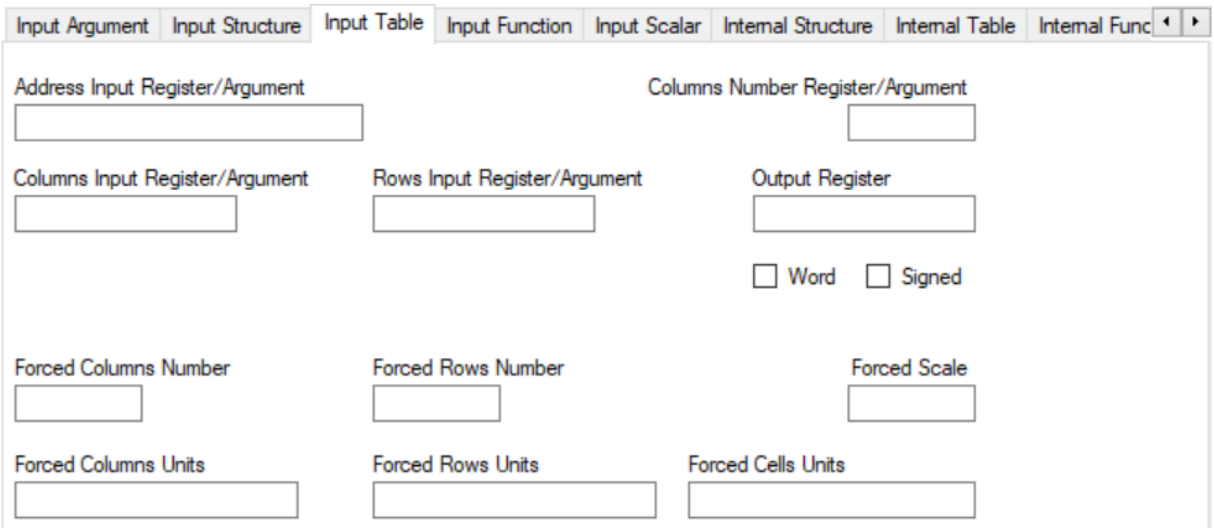
Nothing different here compared to the setup on 'Routines' and now real way to reuse 'Signature parameters' in this place.

- Input Structure



Nothing different here compared to the setup on 'Routines', using Argument ('ArXX') is still possible, but in addition, 'Signature parameters' can be used to fill in automatically 'Address Input Register', 'Number Register' or 'Forced Number'.

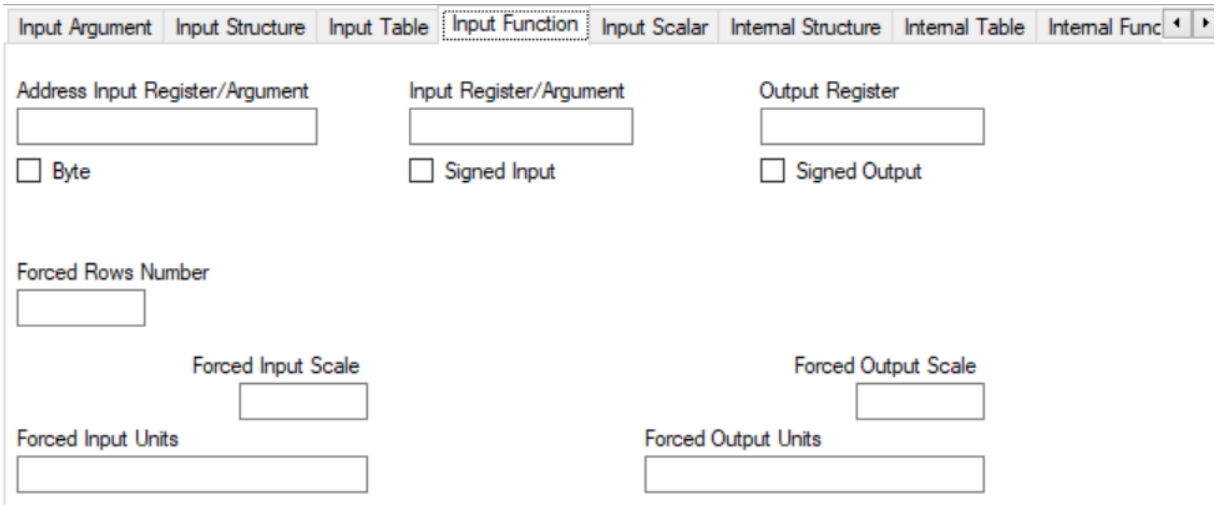
- Input Table



Nothing different here compared to the setup on 'Routines', using Argument ('ArXX') is still possible, but in additon, 'Signature parameters' can be used to fill in automatically 'Address Input

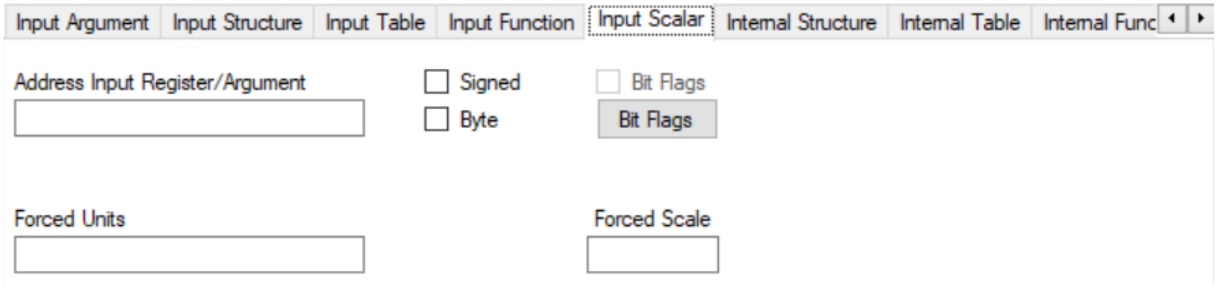
Register', 'Columns Number Register', 'Columns Input Register', 'Rows Input Register', 'Output Register', 'Forced Columns Number' or 'Forced Rows Number'.

- Input Function



Nothing different here compared to the setup on 'Routines', using Argument ('ArXX') is still possible, but in addition, 'Signature parameters' can be used to fill in automatically 'Address Input Register', 'Input Register', 'Output Register' and 'Forced Rows Number'.

- Input Scalar

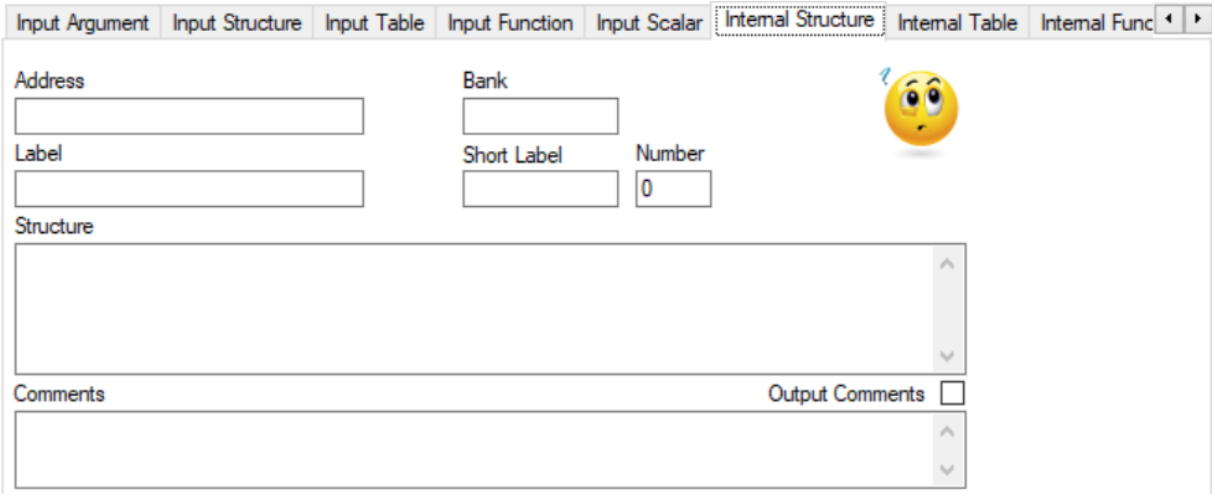


Nothing different here compared to the setup on 'Routines', using Argument ('ArXX') is still possible, but in addition, 'Signature parameters' can be used to fill in automatically 'Address Input Register'.

But unlike routines, it is now possible to setup directly calibration elements, which will be directly created where it is appropriated, when the signature is detected.

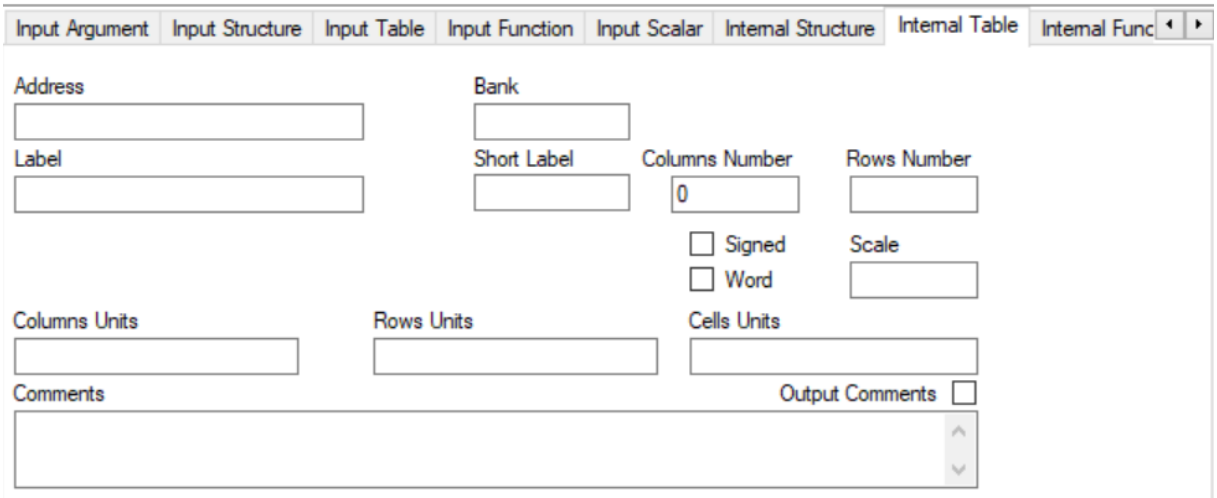
These calibration elements, will be generated based on this setup, so their setup will be the same than for the related category:

- Internal Structure



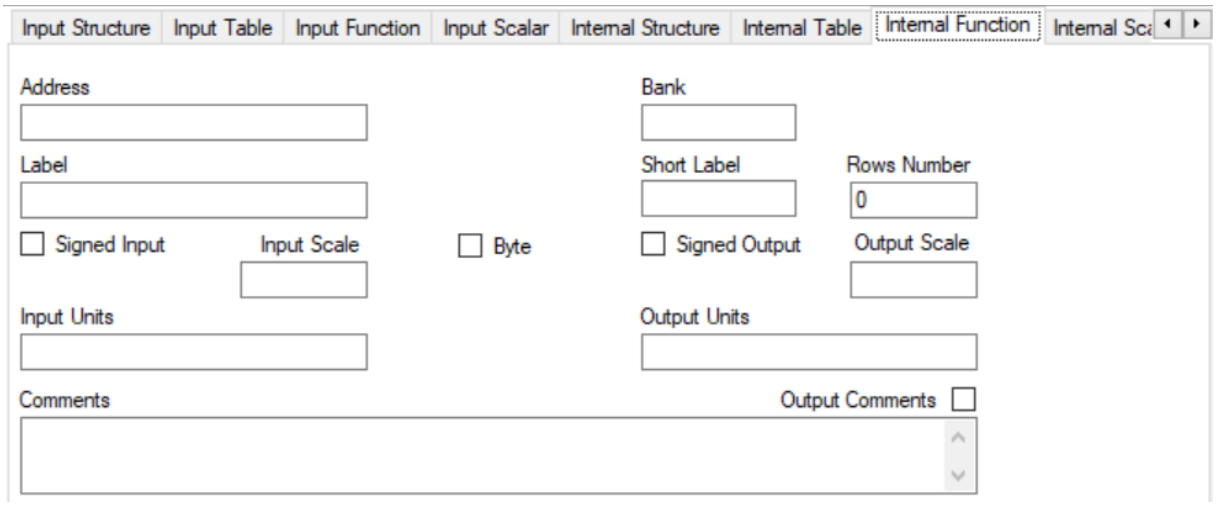
Nothing different here compared to the setup on 'Structures', but 'Signature parameters' can be used to fill in automatically 'Address', 'Number' and the new property 'Bank'. On Signature detection, it will generate automatically the defined structure, in the current definition.

- Internal Table



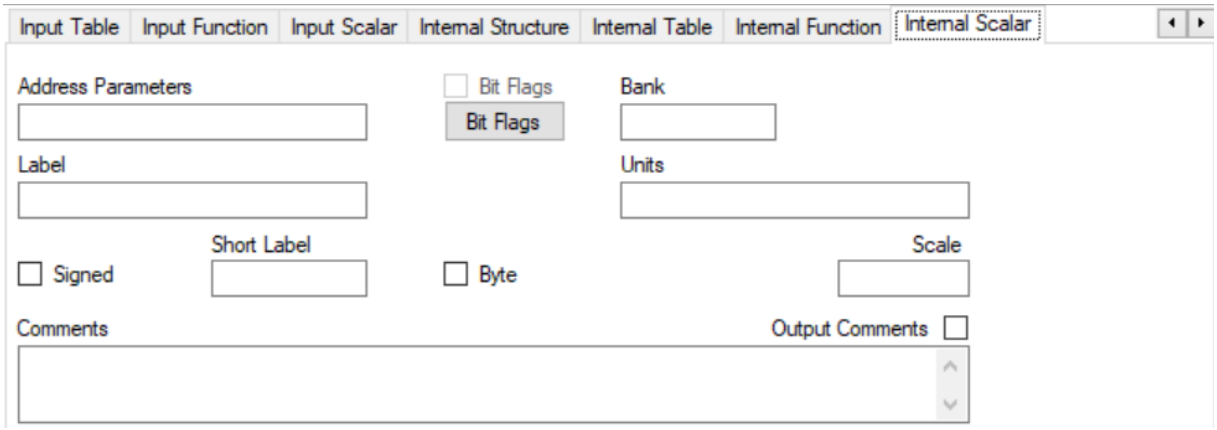
Nothing different here compared to the setup on 'Tables', but 'Signature parameters' can be used to fill in automatically 'Address', 'Columns Number', 'Rows Number' and the new property 'Bank'. On Signature detection, it will generate automatically the defined table, in the current definition.

- Internal Function



Nothing different here compared to the setup on 'Functions', but 'Signature parameters' can be used to fill in automatically 'Address', 'Rows Number' and the new property 'Bank'. On Signature detection, it will generate automatically the defined function, in the current definition.

- Internal Scalar



Nothing different here compared to the setup on 'Scalars', but 'Signature parameters' can be used to fill in automatically 'Address Parameters' and the new property 'Bank'. On Signature detection, it will generate automatically the defined scalar, in the current definition.

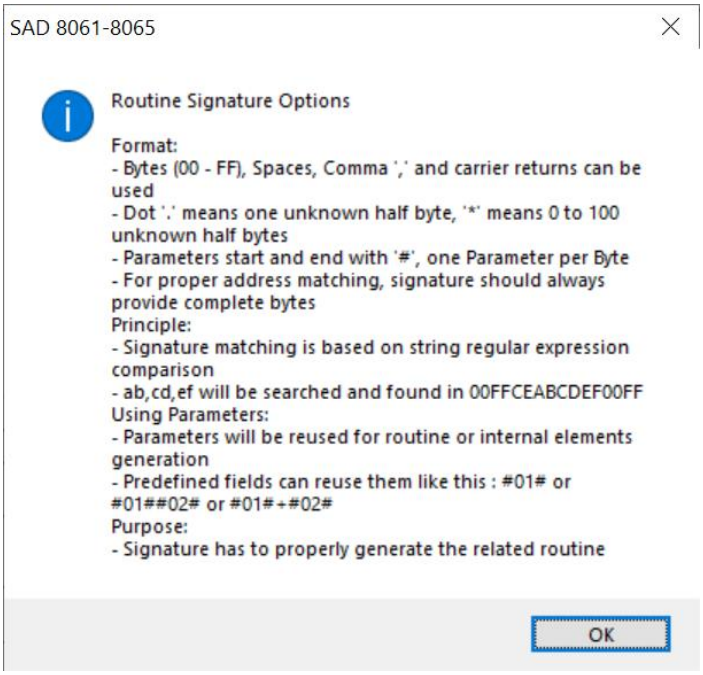
‘Element Information’ for ‘Routines Signatures’:

‘Routines Signatures’ possess an additional ‘Element Information’ tab too, which includes additional details grabbed during disassembly and interessant to be known.

In this case, it permits to see if signature was detected one time or more and if it was one time, which routine was generated by it in definition.

Writing a signature:

Starting from the basis.



As you can see, signature detection is based on a regular expression search, therefore signature and compared binary code should be managed as text string. For sure the best way to do it is through hexadecimal code.

Signature will also mainly be composed with bytes written in hexadecimal from 00 to FF. To have a code more clear to read or to understand, spaces ' ', commas ',' and carrier returns can be inserted between bytes, they will be removed on comparison.

Like for the previous example,

```
8 b1f4: f2                  pushp                   push(PSW);
8 b1f5: a1,00,24,2a         ldw   R2a,2400          R2a = 2400;
8 b1f9: c3,da,50,2a         stw   [Rda+50],R2a      [6d0] = R2a;
8 b1fd: c3,da,70,2a         stw   [Rda+70],R2a      [6f0] = R2a;
8 b201: c3,da,56,00         stw   [Rda+56],0        [6d6] = 0;
8 b205: c3,da,76,00         stw   [Rda+76],0        [6f6] = 0;
8 b209: f3                  popp                    pop(PSW);
8 b20a: f0                  ret                     return;
```

a signature, which will match at 100% here, is the whole code itself.

```
f2
a1,00,24,2a
c3,da,50,2a
c3,da,70,2a
c3,da,56,00
c3,da,76,00
f3
f0
```

But it has no real interest, because it is not a generic code between strategies, 2400 which could be an address, can be different in another strategy and registers will certainly be at different addresses. I am not talking about instructions, which can change or other new instructions which could be added, for bank change or other things.

So the first thing, is to be able to use generic values. Double dot '..' can also be used to replace a complete byte. '24' can be replaced with '..'. But if you are sure, it will always start with '2', you can use '2.' to replace '24' too. Single dot '.' means any character 1 time. Just be sure, your complete signature has the right numbers of half bytes, at the end. The following signature will match here, but probably with other routines too, no ?

```
f2
a1,..,2.,..
c3,..,..,..
c3,..,..,..
c3,..,..,00
c3,..,..,00
f3
f0
```

The second thing is to be able to ignore some values. For example, you can imagine that in some strategies, [6d6] and [6f6] are not reset (set to 0), like here, and the star '*' will help you, it means 0 to 100 unknown half bytes. The following signature will match here, even if operations at '8 b201' and '8 b205' are not existing, but it becomes a bit too much generic, no ?

```
f2
a1,..,2.,..
c3,..,..,..
c3,..,..,..
*
f3
f0
```

As it has already been described, signatures parameters can also be matched with values in binary to be reused in elements. I will just remember you the syntax, inside the signature, still with one parameter per byte, '#XX#' with 'XX' as a decimal number. When matching, it will act exactly like if the double dot '..' was used a single time.

```
f2
a1,#02#,#01#,#03#
c3,#06#,#04#,#03#
c3,#06#,#05#,#03#
c3,#06#,..,00
c3,#06#,..,00
f3
f0

f2
a1,#02#,#01#,#03#
c3,#06#,#04#,..
c3,..,#05#,..
c3,..,..,00
c3,..,..,00
f3
f0
```

Both signatures will match for sure and all parameters will be filled with value, so it is not required when you have the same value in code, to reuse the parameter another time, '#06#' or '#03#' in this case, but it permits to get a closer matching. The second signature could match with more code, which is not expected.

If you still remember well, this code and everything in fact, after basic disassembly, can come from the auto detected routine, with the 'Copy (signature)' from the related routine menu.

Another interesting tool which will be described later on, can be found in main menu 'Tools/Search Signature'. It permits to directly search in binary a provided signature and in fact, to validate if the written one is working or not.

‘Routines Signatures’ category menu:

No specificity at all.

‘Routines Signature’ element menu:

No specificity at all.

Elements Signatures:

'Elements Signatures' part is a bit simpler to use than 'Routines Signatures'. Like its name says, it permits to automatically detect calibration elements and to create directly their complete definition when signature is detected on disassembly. But for sure, it is not the signature of the calibration element itself, it is the signature of the code that is using it.

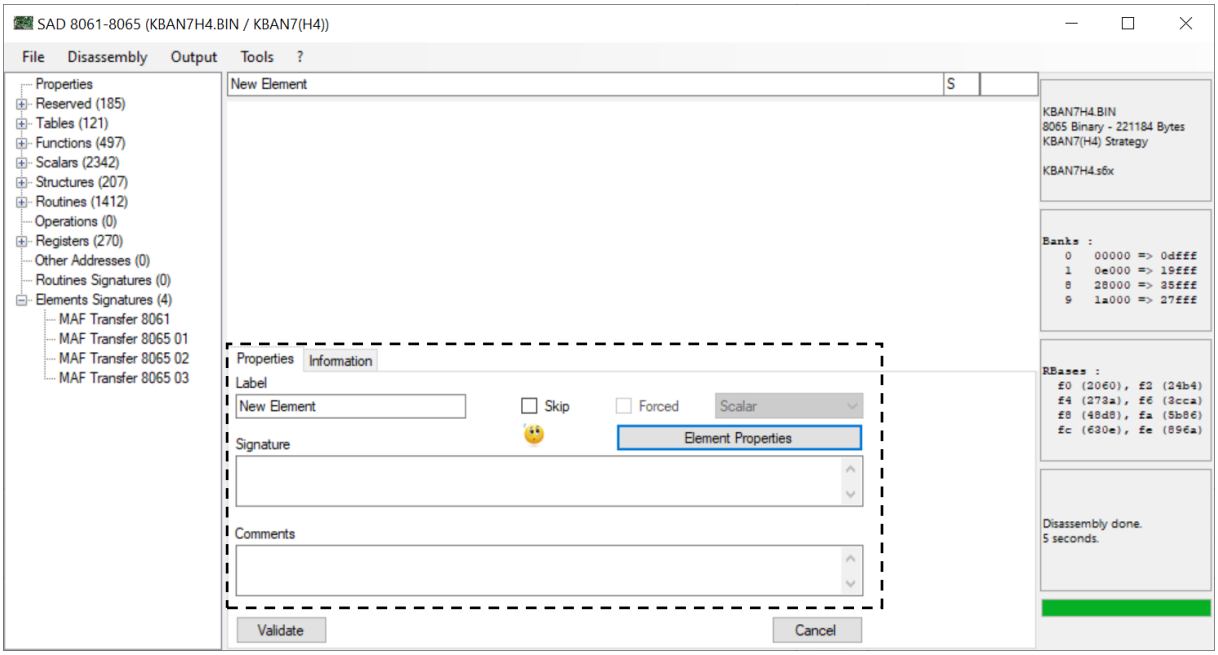
It is still based on some kind of hexadecimal signature, with same principle than for 'Routines Signatures', which still remains to be unique for binary and if possible for other strategies, because the goal is to duplicate them on definition templates, to better automatize disassembly.

The 'Routines', 'Copy (signature)' is not working here, it is dedicated to 'Routines Signatures', but it is still possible to validate a signature through 'Tools/Search Signature'.

Some elements are directly hardcoded in SAD 806x and will be added to new definition, for now 'MAF Transfer'.
This hardcoded signatures can not be removed or updated, they are marked as 'Forced'. But it is a good base to create new ones.

So setup for an 'Element Signature', is basically, the signature itself and the definition of one calibration element.

‘Element Signature’ part is the following one:



Generic properties are like following:

‘Label’ : The label of the signature, it will appear nowhere except here and in ‘Elements Signatures’ list. It will not be duplicated on generated element.

‘Skip’ : When skipped, signature will not be searched for.

‘Comments’ : The comment of the signature, it will appear nowhere except here and in ‘Elements Signatures’ list. It will not be duplicated on generated element.

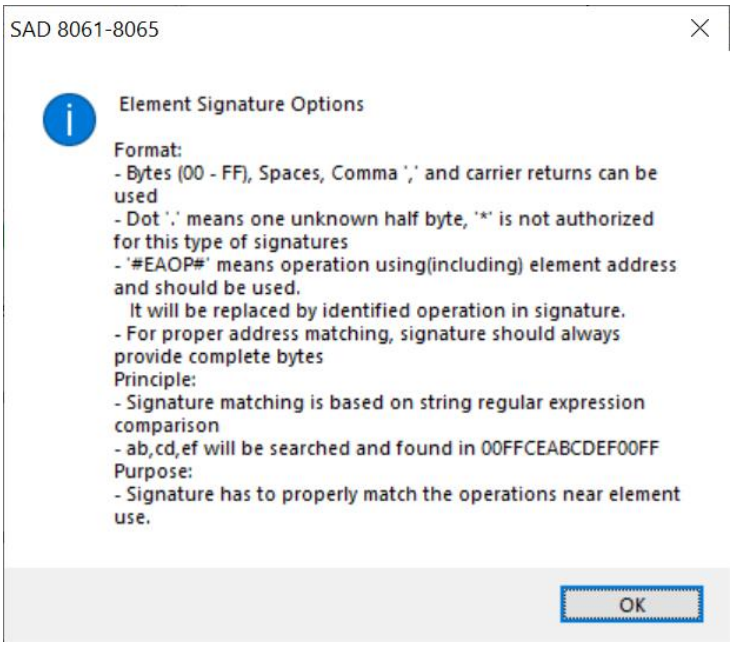
Element signature specific properties are like following:

‘Forced’ : Checked, it indicates that is a provided and hardcoded signature. It can not be removed or updated.

‘Element type’ combo box : On the right of ‘Forced’ checkbox, it indicates, which type of element is detected. It is read only and based on the element setup done through button ‘Element Properties’.

‘Signature’ : This is the hexadecimal signature, which will permit to detect an element based on the code using it. Code written here should be unique in binary, to permit to detect only one element.

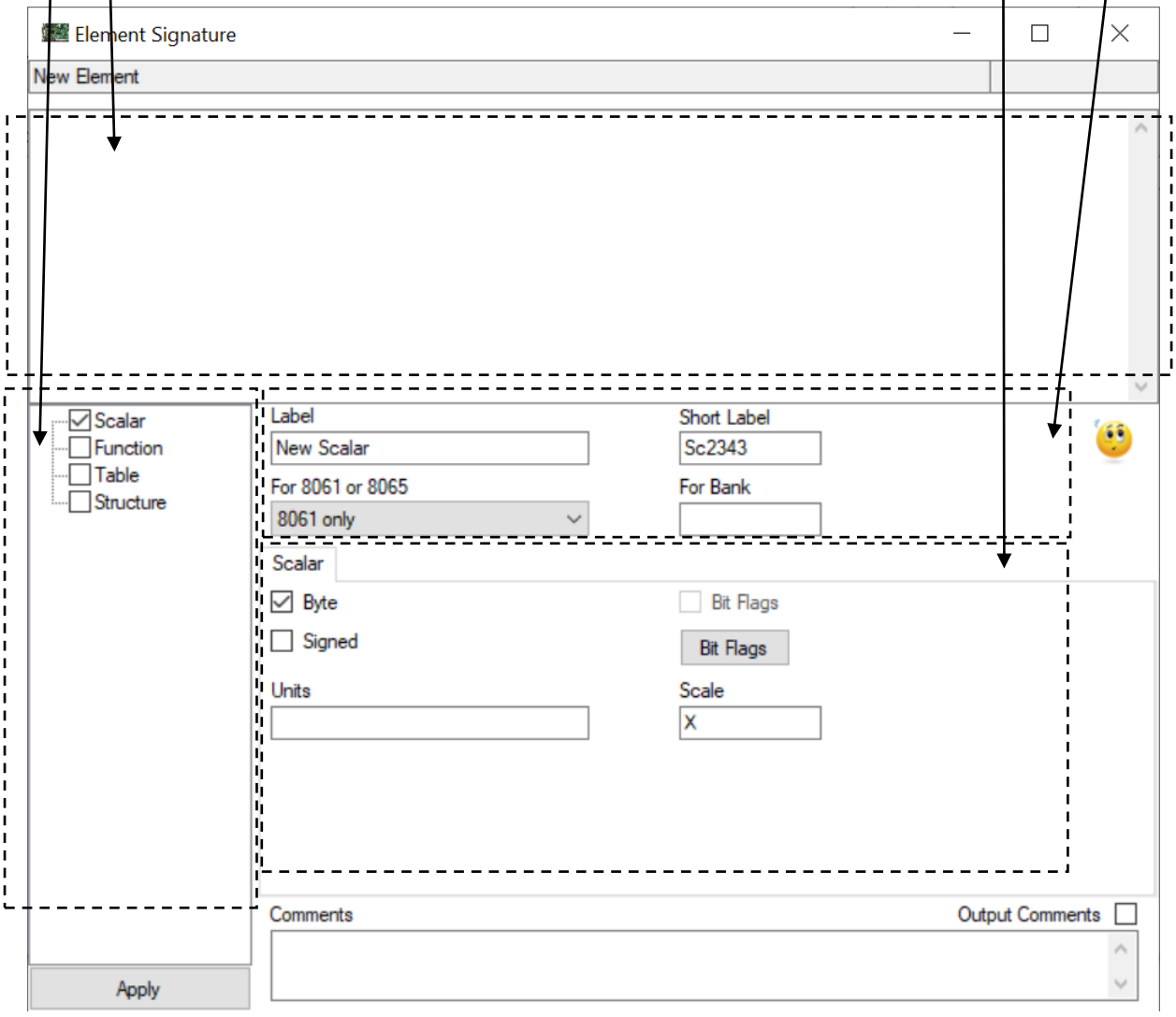
‘Yellow Smiley’ image : With mouse over this image, you have some information about, how to write the signature. When clicking on it, you have a window with the same information. It is a good starting point.


SAD 8061-8065
Element Signature Options
Format:
- Bytes (00 - FF), Spaces, Comma ',' and carrier returns can be used
- Dot '.' means one unknown half byte, '*' is not authorized for this type of signatures
- '#EAOP#' means operation using(including) element address and should be used.
It will be replaced by identified operation in signature.
- For proper address matching, signature should always provide complete bytes
Principle:
- Signature matching is based on string regular expression comparison
- ab,cd,ef will be searched and found in 00FFCEABCDEF00FF
Purpose:
- Signature has to properly match the operations near element use.
OK

'Element Signature' Element Properties:

By using button 'Element Properties', it permit to access to the related form.
This form is composed, with the following elements:

- Extended signature text box : to fill in Signature in this place.
- Element type selection list : list of all possible elements, which will be detected.
- Generic properties : 'Label', 'Short Label', 'For 8061 or 8065' and 'For Bank' fields are generic ones, shared between all element types. 'Label', 'Short Label' will be duplicated on detected element, other will be described.
- Selected element type properties : properties for selected element type.
- 'Comments' : Like 'Label' or 'Short Label', this text box is shared between all types and will be duplicated on detected element like 'Output Comments'. It will do the same job after this on the element, like it was set up directly in definition.
- 'Yellow Smiley' image : still the same meaning.
- 'Apply' button : to validate creation / modification.



Jus before describing setup type by type, I will just describe what is main difference between the ‘Element Signature’ coding, compared to ‘Routine Signature’ coding.

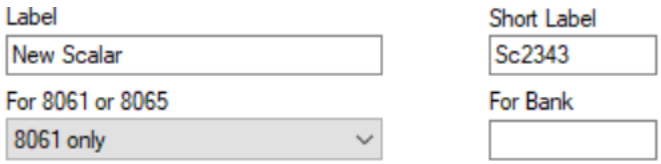
‘Routine Signature’ coding uses signature parameters (‘#XX#’), which are values found in matching code, but ‘Element Signature’ does not need them, it needs the address of the operation using or including the related element. Keyword ‘#EAOP#’ will be used to do this and in fact it will replace the whole operation inside the signature, it is something which is not possible at ‘Routine Signature’ level.

The couterpart for this, is that ‘Routine Signature’ permits to match signature with variable code sizes, through ‘*’ keyword, but it is not possible with ‘Element Signature’, because for now finding ‘#EAOP#’ requires the same number of characters inside rest of signature and code.

I hope it will be possible in the next versions to erase this difference, to extend ‘Routine Signature’ interest and to simplify ‘Element Signature’ code.

For everything else, signature is working exactly in the same way.

Like for calibration elements (Scalars, Functions, Tables and Structures) selected type provides globally same options than related element properties, but some properties are shared between all element types:



and

No address parameter here, '#EAOP#' which should be present in signature, will permit to calculate element address automatically.

'Label', 'Short Label', 'Comments' and 'Output Comments' will be directly duplicated on detected element and they will do the same job after this on the element, like if they were set up directly in definition, for Scalar, Function, Table and Structure. 'Label' and 'Short Label' take default values based on selected element type.

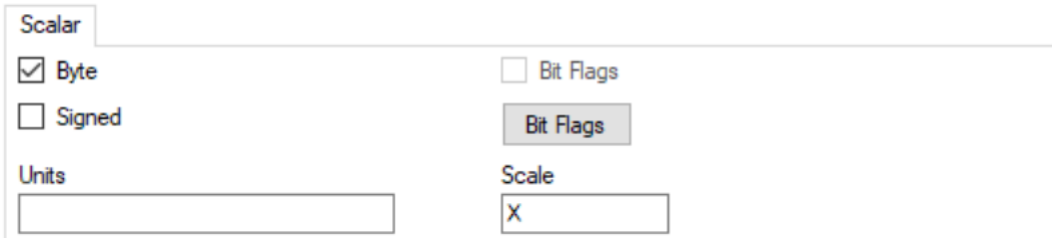
'For 8061 or 8065' combo box is dedicated to structure detection, like 'For Bank' number. It permits to obtain closer signature based on rom hardware and bank number. Otherwise, it is really difficult to work simply with this type of signatures.

With '8061 only' option, singature will be searched only in 8061 roms, thus from EEC IV management. With '8065 only' option, singature will be searched only in 8065 roms, yes from EEC V management.

If 'For Bank' stays empty, signature will be searched on all banks in rom, otherwise it will be searched on specified bank only, one bank only here. Valid banks are the one in the rom, at maximum, you can have banks 8, 1, 9 and 0.

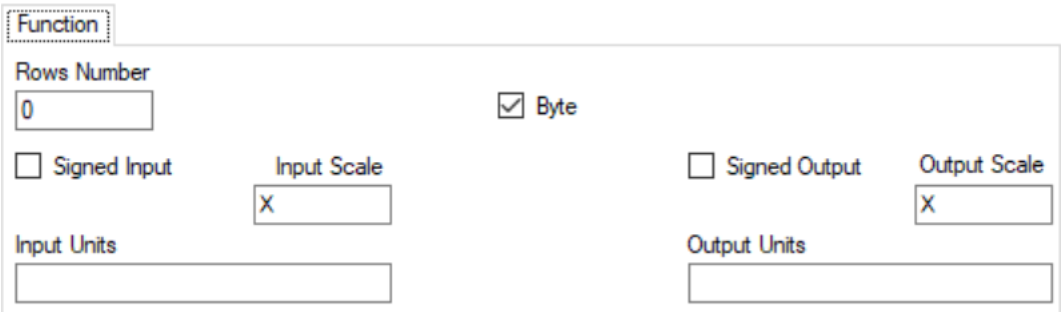
Now let's see dedicated type properties:

- Scalar



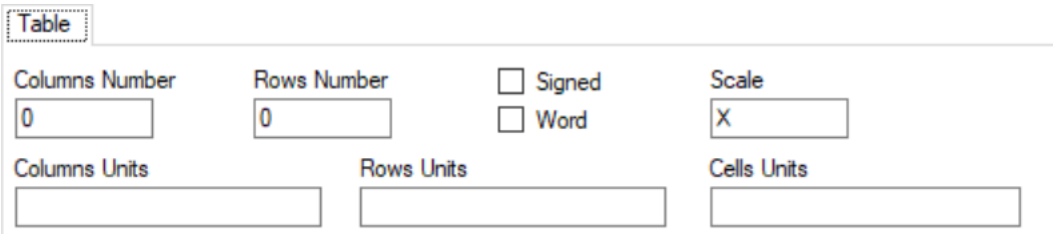
Nothing different here compared to the setup on 'Scalars' for remaining properties, 'Byte', 'Signed', 'Bit Flags', 'Units' and 'Scale' properties will be directly duplicated to detected element on disassembly and will also be applied at the same time, exactly like if you had created this element in definition.

- Function



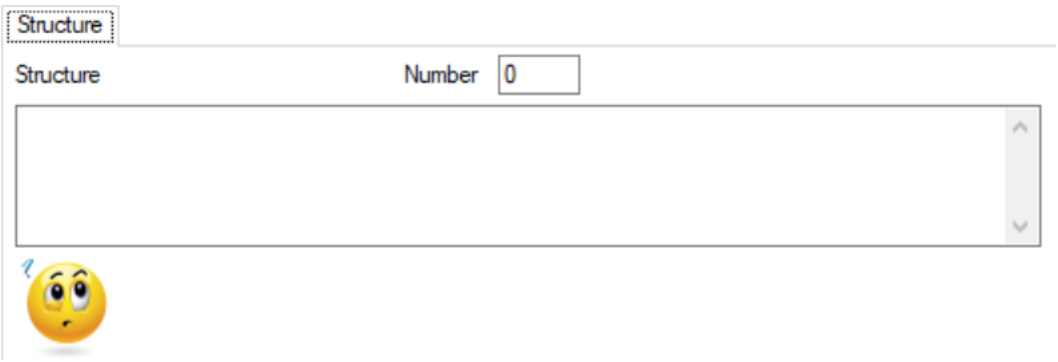
Nothing different here compared to the setup on 'Functions' for remaining properties, 'Rows Number', 'Byte', 'Signed Input', 'Signed Output', 'Input Scale', 'Output Scale', 'Input Units' and 'Output Units' properties will be directly duplicated to detected element on disassembly and will also be applied at the same time, exactly like if you had created this element in definition. If 'Rows Number' stays at 0, autodetection will apply for it.

- Table



Nothing different here compared to the setup on 'Tables' for remaining properties, 'Columns Number', 'Rows Number', 'Word', 'Signed', 'Scale', 'Columns Units', 'Rows Units' and 'Cells Units' properties will be directly duplicated to detected element on disassembly and will also be applied at the same time, exactly like if you had created this element in definition. If 'Columns Number' or 'Rows Number' stays at 0, autodetection will apply for it.

- Structure



Nothing different here compared to the setup on ‘Structures’ for remaining properties, ‘Number’, and ‘Structure’ properties will be directly duplicated to detected element on disassembly and will also be applied at the same time, exactly like if you had created this element in definition. You can notice our ‘Yellow Smiley’ image, present for some help on structure writing. If ‘Number’ stays at 0, autodetection will apply for it.

Do not forget to use the apply button after updates and before quitting this form.

‘Element Information’ for ‘Elements Signatures’:

‘Elements Signatures’ possess an additional ‘Element Information’ tab too, which includes additional details grabbed during disassembly and interessant to be known.

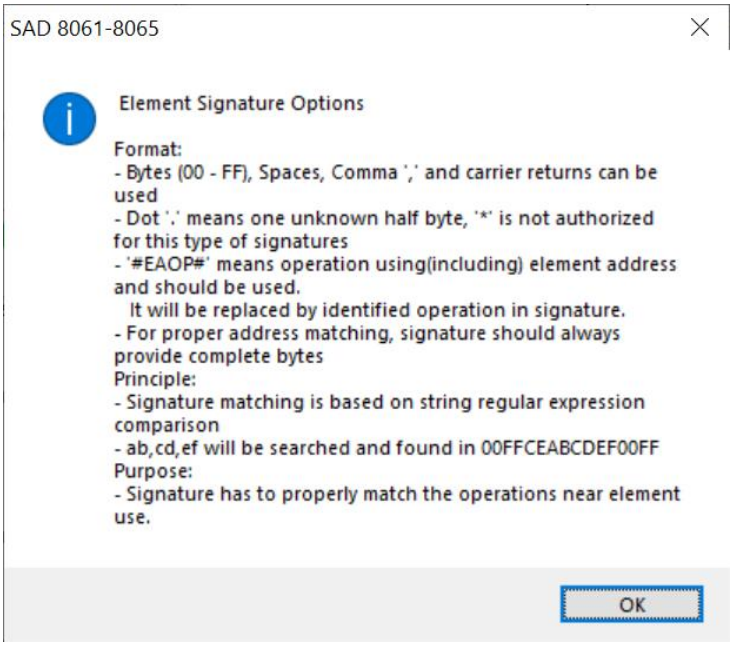
In this case, it permits to see, if signature was detected one time or more and if it was one time, which element was generated by it in definition.

Writing an element signature, using '#EAOP#':

I will not described how to write a signature from the beginning, so please refer to 'Routines Signatures' part for that.

As described previously, an element signature has no signature parameters keywords, except '#EAOP#' for the operation related with use of element to detect. '#EAOP#' is mandatory, for this type of signature, but '*' keyword is not possible due to some limitation for now (probably mines or on my time).

Still the basis, dedicated to element signature:



MAF Transfer function (FN036) signature is the perfect (and the first) example.
This is one of the possible codes which are using it.

```
8 249d: fa                  di                  disable ints;
8 249e: c4,08,d3            stb   R08,Rd3       INT_MASK = Rd3;
8 24a1: a1,3f,00,12         ldw   R12,3f        HSO_MASK1 = 3f;
8 24a5: a1,40,80,16         ldw   R16,8040      HSO_MASK2 = 8040;
8 24a9: f3                  popp                pop(PSW);
8 24aa: 45,18,03,f0,46      ad3w  R46,Rf0,318   R46 = FN036;
8 24af: a3,e4,30,36         ldw   R36,[Re4+30]  R36 = [11b0];
8 24b3: 88,36,46            cmpw  R46,R36
8 24b6: d9,12               jgtu  24ca          if ((uns) R46 > R36) goto 24ca;
8 24b8: 45,78,00,46,34      ad3w  R34,R46,78    R34 = R46 + 78;
8 24bd: 88,36,34            cmpw  R34,R36
8 24c0: d3,08               jnc   24ca          if ((uns) R34 < R36) goto 24ca;
8 24c2: 94,46,36            xrb   R36,R46       R36 ^= R46;
8 24c5: 71,03,36            an2b  R36,3         R36 &= 3;
8 24c8: df,04               je    24ce          if (R36 == 3) goto 24ce;
8 24ca: c3,e4,30,46         stw   [Re4+30],R46  [11b0] = R46;
8 24ce: 71,ef,54            an2b  R54,ef        R54 &= ef;
8 24d1: f0                  ret                 return;
```

This signature will match:

```
#EAOP#
A3,..,..,..,..
88,..,..
D9,14
45,78,00,..,..
8B,..,..,..,..
D3,08
94,..,..
71,03,..
DF,05
C3,..,..,..,..
71,..,..
F0
```

You can see that '#EAOP#', is replacing operation at '45,18,03,f0,46' and it could be any operation, but just before the signature which is following.

It is not necessary to start signature with '#EAOP#', it could be in the middle of the signature or at its end, but it should always replace complete operation, related with a calibration element.

Signature should be as generic as possible, but in some cases, it has to be duplicated.

```
#EAOP#
A3,..,..,..
88,..,..
D9,13
45,78,00,..,..
8B,..,..,..
D3,08
94,..,..
71,03,..
DF,04
C3,..,..,..
F0
```

This one is for MAF Transfer function (FN036) too, but it will not work on provided code and it is required to create some duplicated signature, because of small differences and because of number of operations between both. This is still related with '*' keyword which is missing.

‘Elements Signatures’ category menu:

No specificity at all.

‘Element Signature’ element menu:

No specificity at all.

Disassembly Text Output:

As it was already said, SAD 806x text output is largely inspired from SAD disassembler (created by Andy, tvrfan), so I let you read its documentation to discover what was in place. I will not try to explain, what should be a good text output for a disassembly, but I will show you things, which are a bit different from what is existing or which can be a bit complicated to understand.

Disassembly operations basis:

I will try to quickly describe, what the main items are in the disassembled code.
This is an operation:

```
8 df87: 71,7f,64              an2b  R64,7f              R64 &= 7f;
  1       2                       3                        4
```

1. '8 df87' is the complete address of the operation, '8' is the bank where the operation is stored in the rom, 'df87' is the address in the bank, knowing these addresses have 0x2000 added, which becomes their minimum address.
2. '71,7f,64' is the hexadecimal code for the operation. First byte '71' is the instruction, '7f,64' are the parameters.
3. 'an2b R64, 7f' is the assembler code for the operation. 'an2b' is the instruction, 'R64, 7f' are the parameters.
4. 'R64 &= 7f;' is the C like code for the operation. '&=' is the instruction, 'R64' and '7f' are the parameters. Known parameters are only translated in this place.

'R64' is a register, with address 0x64.
'[112]' would be a register too, with address 0x112, but because it is on more than 2 characters, it is written without 'R'.
'[R64]' would be a pointer to the value which is in register 'R64'.

Here we have 2 operations.

```
8 df8f: 9b,fe,08,00          cmpb  0,[Rfe+8]
8 df93: df,0f                je    dfa4              if (0 == [Sc1640]) goto dfa4;
```

Comparison operations (cmp) are used only to fill in stack with result. Next operations can use this result, like here. This is why last operation, C like code, contains more information, than the current operation hexadecimal code, because it includes comparison operation detail.

'[Rfe+8]' is an address in rom, but 'Rfe' is a RBase register, so address is in this case is on bank 1. So '[Rfe+8]' is the Rfe RBase address with 0x8 added ('1 8972' finally), but real value does not appear in our case, because a scalar '[Sc1640]' has been created at this address, so C like code shows the translated value, including this scalar.

Here we have 2 calls to routines.

```
8 df65: 28,e1                scall e048              Sub0890();
8 df6e: ef,87,48             call  27f8              UUByteLU();
```

When C like code contains parenthesis like this, it is a call to a routine. The first line is for an unknown routine, translated based on the auto numbering, the second one is for a known routine. Principle stays the same.

```
8 7c45: ef,d4,cf            call  4c1c              Sub0102(24,54,0,ff,10,Rff);
8 7c48: 24,00,54,10,ff,ff,00                        #args
```

In this case, it is still a call but using arguments. Arguments are on the second line, but will appear in C like code on the first line, to be clearer. Line at address '8 7c48' is not an operation, it is only arguments for previous operation or you can consider it is part or previous operation.

Disassembly elements basis:

Scalar examples:

```
1 23f6: 60,00               Rf0+396 Sc0027      word        60                  96

1 23f8: 00                  Rf0+398 Sc0028      byte         0                   0
```

1 2 3 4 5 6 7

1. The complete address for the element.
2. The hexadecimal value for the element. Do not forget this is an Intel Rom, word values are inverted.
3. The RBase equivalence, if it exists.
4. The 'Short Label' for the scalar. If 'Label' has been specified, it will output over the element.
5. The type of the element.
6. The combined hexadecimal value.
7. The decimal value, using the defined scale, if it exists.

Scalars detected or defined as bit flags:

```
                                                    B7 B6 B5 B4 B3 B2 B1 B0
                                                     0  0  0  0  1  1  1  0
1 6fb3: 0e                  Rfc+ca5 Sc1249      byte         e                  14

TQ_MODULE_SW - TQ_MODULE_SW - (TQM SW):
                                                                           B0
                                                                            1
1 89a8: 01                  Rfe+3e TQ_MODULE_SW     byte         1                   1

OBDII_TST_SW - OBDII Monitor Tests SW:
                                                    B7 B6 B5 B4 B3 B2 B1 B0
                                                     1  1  1  1  0  0  1  0
1 8d4e: f2                  Rfe+3e4 OBDII_TST_SW    byte        f2                 242
```

Functions examples:

```
Fn137 - Function 137:
1 3f40: ff,00             func                ff,        0                255,        0
1 3f42: 60,0d             func                60,        d                 96,       13
1 3f44: 46,20             func                46,       20                 70,       32
1 3f46: 3c,33             func                3c,       33                 60,       51
1 3f48: 3a,40             func                3a,       40                 58,       64
1 3f4a: 34,40             func                34,       40                 52,       64
1 3f4c: 1c,ff             func                1c,       ff                 28,      255
1 3f4e: 00,ff             func                 0,       ff                  0,      255

Fn138 - Function 138:
1 3f50: ff,ff,00,05       func              ffff,      500              65535,     1280
1 3f54: 50,00,00,05       func                50,      500                 80,     1280
1 3f58: 14,00,00,03       func                14,      300                 20,      768
1 3f5c: 0a,00,00,02       func                 a,      200                 10,      512
1 3f60: 00,00,00,00       func                 0,        0                  0,        0
1 3f64: 00,00,00,00       func                 0,        0                  0,        0
1 3f68: 00,00,00,00       func                 0,        0                  0,        0
1 3f6c: 00,00,00,00       func                 0,        0                  0,        0
```

1 2 3 4 5 6 7

'Short Label' and 'Label' are found as header of the function.

1. The complete address for the function row.
2. The hexadecimal value for the row, including input and output columns. Do not forget this is an Intel Rom, word values are inverted.
3. The type of the element.
4. The combined hexadecimal value for the input column.
5. The combined hexadecimal value for the output column.
6. The decimal value, scaled if set up, for the input column.
7. The decimal value, scaled if set up, for the output column.

Tables examples:

A byte one:

```
Tb123 - Table 123:
1 953c: 03,03,03,03       table          3,   3,   3,   3           3,    3,    3,    3
1 9540: 03,03,03,03       table          3,   3,   3,   3           3,    3,    3,    3
1 9544: 03,03,03,03       table          3,   3,   3,   3           3,    3,    3,    3
1 9548: 03,03,03,03       table          3,   3,   3,   3           3,    3,    3,    3
1 954c: 03,03,03,03       table          3,   3,   3,   3           3,    3,    3,    3
1 9550: 03,03,03,03       table          3,   3,   3,   3           3,    3,    3,    3
```

1 2 3 4 5

A word one:

```
Tb122 - Table 122:
1 951c: e0,1f,e0,1f,e0,1f,e0,1f,e0,1f,e0,1f,e0,1f      table      1fe0, 1fe0, 1fe0, 1fe0, 1fe0, 1fe0, 1fe0      8160,  8160,  8160,  8160,  8160,  8160,  8160
1 952a: e0,1f,e0,1f,e0,1f,e0,1f,e0,1f,e0,1f,e0,1f      table      1fe0, 1fe0, 1fe0, 1fe0, 1fe0, 1fe0, 1fe0      8160,  8160,  8160,  8160,  8160,  8160,  8160
```

1 2 3 4 5

'Short Label' and 'Label' are found as header of the table.

1. The complete address for the table row.
2. The hexadecimal value for the row, including all columns. Do not forget this is an Intel Rom, word values are inverted.
3. The type of the element.
4. The combined hexadecimal value for the all cells in the table.
5. The decimal value, scaled if set up, for the all cells in the table.

Structures examples:

```
ADCHANSt - AD Channels Structure:
8 4072: 32,00,3c,00,6a,01,7c,7c,00,00,39,80,f0,85      ostruct   Rbase+32, Rbase+3c, 16a, R7c, R7c, 0, 8039, 85f0
8 4080: 0b,00,00,00,66,01,8a,00,00,00,00,00,00,00      ostruct   Rbase+ b, Rbase+ 0, 166, R8a, R 0, 0,    0,    0
8 408e: 33,00,00,00,6c,01,7c,00,00,00,4a,80,f7,85      ostruct   Rbase+33, Rbase+ 0, 16c, R7c, R 0, 0, 804a, 85f7
8 409c: 36,00,00,00,6e,01,7c,00,00,00,00,00,e1,85      ostruct   Rbase+36, Rbase+ 0, 16e, R7c, R 0, 0,    0, 85e1
8 40aa: 38,00,00,00,70,01,7c,00,00,00,00,00,00,00      ostruct   Rbase+38, Rbase+ 0, 170, R7c, R 0, 0,    0,    0
8 40b8: 39,00,00,00,72,01,7c,00,00,00,00,00,00,00      ostruct   Rbase+39, Rbase+ 0, 172, R7c, R 0, 0,    0,    0
8 40c6: 3b,00,00,00,f0,02,7c,00,00,00,00,00,00,00      ostruct   Rbase+3b, Rbase+ 0, 2f0, R7c, R 0, 0,    0,    0
8 40d4: 3a,00,00,00,ee,02,7c,00,00,00,1d,80,00,00      ostruct   Rbase+3a, Rbase+ 0, 2ee, R7c, R 0, 0, 801d,    0
8 40e2: 35,00,00,00,74,01,7c,00,00,00,00,00,00,00      ostruct   Rbase+35, Rbase+ 0, 174, R7c, R 0, 0,    0,    0
8 40f0: 0c,00,00,00,bc,02,8a,00,00,00,00,00,00,00      ostruct   Rbase+ c, Rbase+ 0, 2bc, R8a, R 0, 0,    0,    0


IAECTACTLevSt - Ignition Advance ECT ACT Levels Structure:
8 8c2c: 74,00,84,00,07,04,05,04,04,04,06,04,0a,04      ostruct   74, 84, 407, 405, 404, 406, 40a
8 8c3a: 88,00,84,00,08,04,00,00,09,04,00,00,0b,04      ostruct   88, 84, 408,   0, 409,   0, 40b
```

1 2 3 4

'Short Label' and 'Label' are found as header of the structure, properly defined here.

1. The complete address for the structure row/occurrence.
2. The hexadecimal value for the row/occurrence. Based on structure definition, all rows/occurrences, have not always the same size.
3. The type of the element ('struct' is inside calibration part, 'ostruct' melted between operations)
4. The output defined by structure definition itself, so it can be very variable from one structure to another or from one row/occurrence to another.

Disassembly unknown parts:

Some parts are not disassembled, because it was not possible for SAD 806x to reach the code at this moment, when it is an operation part or because calibration element was not detected, based on its use in an operation.

```
Sub0097:
0 49dd: f2                      pushp                   push(PSW);
0 49de: ad,07,94                ldzbw R94,7             R94 = (uns)7;
0 49e1: 20,0a                   sjmp  49ed              goto 49ed;


0 49e3: f2,ad,08,94,20,04,f2,ad                        Unknown Operation/Structure
0 49eb: 09,94                                          Unknown Operation/Structure

0 49ed: 2e,23                   scall 4812              Sub0088();
0 49ef: f3                      popp                    pop(PSW);
0 49f0: f1                      reti                    return;
```

In operations part, it will be marked as ‘Unknown Operation/Structure’, because it could be a structure too. You will find the address, the hexadecimal code (8 bytes by 8 bytes) and the mark.

```
0 6990: f0                    ret                     return;


0 6991: f2,af,3a,03,40,b3,3a,04                       Unknown Operation/Structure
0 6999: 42,b3,3a,05,43,11,38,3b                       Unknown Operation/Structure
0 69a1: 40,17,3c,40,0e,35,40,16                       Unknown Operation/Structure
0 69a9: b3,42,03,37,b3,42,02,36                       Unknown Operation/Structure
0 69b1: b1,02,38,b3,42,01,35,17                       Unknown Operation/Structure
0 69b9: 38,b2,42,34,17,38,f3,f0                       Unknown Operation/Structure


Sub0138:
0 69c1: f2                    pushp                   push(PSW);
```

In this case it could be interesting to create a routine at address ‘0 6991’ in definition. A part starting with ‘f2’ (‘push(PWD);’) and ending with ‘f0’ (‘return;’) should probably be a routine in EEC V rom. SAD 806x has not processed it, because until now, no ‘call’ or ‘goto’ was done to this address. This address is probably present in an undetected vector list or in an unknown structure.

```
9 ff9a: ff,ff                 0                       Tyre Revolutions per Mile
9 ff9c: ff,ff                 0                       Rear End Gear Ratio


9 ff9e -> fffe                   fill                 ff

9 ffff: 91                                            Unknown Operation/Structure
```

Here you can some reserved addresses at ‘9 ff9a’ and ‘9 ff9c’, but just after that, you can see 2 types of ‘Unknown Operation/Structure’. The last line that we have already seen and the line which begins with ‘9 ff9e -> fffe’. When unknown bytes are repeated 8 times or more, they are grouped like this and marked as ‘fill’. So from address ‘9 ff9e’ to address ‘9 fffe’, bytes are filled with ‘ff’ and it is unrecognized/undefined bytes.

```
Fn016 - Function 016:
1 2258: ff,ff,00,07       func              ffff,      700             65535,     1792
1 225c: 66,66,00,07       func              6666,      700             26214,     1792
1 2260: cd,0c,00,00       func               ccd,        0              3277,        0
1 2264: 00,00,00,00       func                 0,        0                 0,        0
1 2268: 00,00,00,00       func                 0,        0                 0,        0
1 226c: 00,00,00,00       func                 0,        0                 0,        0
1 2270: 00,00,00,00       func                 0,        0                 0,        0

1 2274: ff,ff,cd,c8,9a,a2,58,80           Unknown Calibration       ff, ff, cd, c8, 9a, a2, 58, 80       255, 255, 205, 200, 154, 162,  88, 128
1 227c: 00,80,ff,00,58,00,33,22           Unknown Calibration        0, 80, ff,  0, 58,  0, 33, 22         0, 128, 255,   0,  88,   0,  51,  34
1 2284: 1a,40,0a,59,05,65,00,80           Unknown Calibration       1a, 40,  a, 59,  5, 65,  0, 80        26,  64,  10,  89,   5, 101,   0, 128
1 228c: ff,5b,e6,5b,9a,6e,58,80           Unknown Calibration       ff, 5b, e6, 5b, 9a, 6e, 58, 80       255,  91, 230,  91, 154, 110,  88, 128
1 2294: 00,80                             Unknown Calibration                   0, 80                                                      0, 128

Fn017 - Function 017:
1 2296: ff,7f,cd,00       func              7fff,       cd              32767,      205
1 229a: 00,80,cd,00       func              8000,       cd             -32768,      205
1 229e: 00,80,cd,00       func              8000,       cd             -32768,      205
1 22a2: 00,80,cd,00       func              8000,       cd             -32768,      205
1 22a6: 00,80,cd,00       func              8000,       cd             -32768,      205
1 22aa: 00,80,cd,00       func              8000,       cd             -32768,      205
1 22ae: 00,80,cd,00       func              8000,       cd             -32768,      205
1 22b2: 00,80,cd,00       func              8000,       cd             -32768,      205
```

In calibration part (part related with RBase addresses), which contains all tables, functions and classical scalars, an unknown part is marked as ‘Unknown Calibration’. You will find the address, the hexadecimal code (8 bytes by 8 bytes), the mark and the values, hexadecimal and decimal, (8 bytes by 8 bytes). Yes in this case, it is easy to understand, that it is a function (they always start with ff, 7f or ffff or ff7f), but same thing SAD 806x has not found the code using this address, so nothing was disassembled.

SAD vs SAD 806x differences:

SAD version for a routine:

```
  Sub663:
8 df57: f2                pushp                 push(PSW);
8 df58: a3,fe,06,30       ldw   R30,[Rfe+6]     R30 = [8970];
8 df5c: b3,fe,04,32       ldb   R32,[Rfe+4]     R32 = [896e];
8 df60: 11,25             clrb  R25             R25 = 0;
8 df62: ad,97,28          ldzbw R28,97          R28 = (uns)97;
8 df65: 28,e1             scall e048            Sub660();
8 df67: a1,88,89,36       ldw   R36,8988        R36 = 8988;
8 df6b: b0,2c,38          ldb   R38,R2c         R38 = R2c;
8 df6e: ef,87,48          call  27f8            UUByteLU();
8 df71: 9b,fe,03,3c       cmpb  R3c,[Rfe+3]
8 df75: d7,0c             jne   df83            if (R3c == [896d])  {
8 df77: b3,d4,6f,8f       ldb   R8f,[Rd4+6f]    R8f = [Rd4+6f];
8 df7b: 3c,8f,05          jb    B4,R8f,df83     if (!B4_R8f)  {
8 df7e: 91,80,64          orb   R64,80          R64 |= 80;
8 df81: 20,07             sjmp  df8a            goto df8a; } }

8 df83: b3,fe,02,3c       ldb   R3c,[Rfe+2]     R3c = [896c];
8 df87: 71,7f,64          an2b  R64,7f          R64 &= 7f;
8 df8a: c7,01,d7,01,3c    stb   R3c,[1d7]       [1d7] = R3c;
8 df8f: 9b,fe,08,00       cmpb  0,[Rfe+8]
8 df93: df,0f             je    dfa4            if (0 != [8972])  {
8 df95: b3,fe,02,3c       ldb   R3c,[Rfe+2]     R3c = [896c];
8 df99: 9b,fe,03,3c       cmpb  R3c,[Rfe+3]
8 df9d: df,05             je    dfa4            if (R3c != [896d])  {
8 df9f: 91,10,7a          orb   R7a,10          R7a |= 10;
8 dfa2: 20,03             sjmp  dfa7            goto dfa7; } }

8 dfa4: 71,ef,7a          an2b  R7a,ef          R7a &= ef;
8 dfa7: f3                popp                  pop(PSW);
8 dfa8: f0                ret                   return;
```

SAD 806x version for the same routine:

```
Sub0887:
8 df57: f2                  pushp                   push(PSW);
8 df58: a3,fe,06,30         ldw   R30,[Rfe+6]       R30 = [Sc1639];
8 df5c: b3,fe,04,32         ldb   R32,[Rfe+4]       R32 = [Sc1638];
8 df60: 11,25               clrb  R25               R25 = 0;
8 df62: ad,97,28            ldzbw R28,97            R28 = (uns)97;
8 df65: 28,e1               scall e048              Sub0890();
8 df67: a1,88,89,36         ldw   R36,8988          R36 = Fn418;
8 df6b: b0,2c,38            ldb   R38,R2c           R38 = R2c;
8 df6e: ef,87,48            call  27f8              UUByteLU();
8 df71: 9b,fe,03,3c         cmpb  R3c,[Rfe+3]
8 df75: d7,0c               jne   df83              if (R3c != [Sc1637]) goto df83;
8 df77: b3,d4,6f,8f         ldb   R8f,[Rd4+6f]      R8f = [2ef];
8 df7b: 3c,8f,05            jb    B4,R8f,df83       if (B4_R8f) goto df83;
8 df7e: 91,80,64            orrb  R64,80            R64 |= 80;
8 df81: 20,07               sjmp  df8a              goto df8a;

8 df83: b3,fe,02,3c         ldb   R3c,[Rfe+2]       R3c = [Sc1636];
8 df87: 71,7f,64            an2b  R64,7f            R64 &= 7f;
8 df8a: c7,01,d7,01,3c      stb   [1d7],R3c         [1d7] = R3c;
8 df8f: 9b,fe,08,00         cmpb  0,[Rfe+8]
8 df93: df,0f               je    dfa4              if (0 == [Sc1640]) goto dfa4;
8 df95: b3,fe,02,3c         ldb   R3c,[Rfe+2]       R3c = [Sc1636];
8 df99: 9b,fe,03,3c         cmpb  R3c,[Rfe+3]
8 df9d: df,05               je    dfa4              if (R3c == [Sc1637]) goto dfa4;
8 df9f: 91,10,7a            orrb  R7a,10            R7a |= 10;
8 dfa2: 20,03               sjmp  dfa7              goto dfa7;

8 dfa4: 71,ef,7a            an2b  R7a,ef            R7a &= ef;
8 dfa7: f3                  popp                    pop(PSW);
8 dfa8: f0                  ret                     return;
```

As you can see, it is like the same thing, yes routine number is not the same on one side compared to the other, some addresses are not recognized, but it is not the difference.

With SAD 806x layout is for now fixed, for operations or calibration elements, no way to add additional spaces between parts. Other important thing, conditional ‘gotos’ keep their original meaning with SAD 806x and no ‘{’ or ‘}’ is used to group the code.

SAD basic version for scalars:

```
1 23f6: 60,00               word  60

1 23f8: 00                  byte  0
```

SAD 806x one:

```
1 23f6: 60,00             Rf0+396 Sc0027      word        60                  96

1 23f8: 00                Rf0+398 Sc0028      byte         0                   0
```

SAD 806x layout is fixed and the one for SAD has to be defined, so the result is different.

I will not continue to detail differences, because the others are really related with the layout setup which is possible in SAD, not in SAD 806x.

Examples of advanced text outputs:

The header and the famous register list (which remains optional).

```
                                8065 Disassembly

Binary File :                                   S6x file :
                KBAN7H4 224.BIN                                 KBAN7H4 224.s6x
            229376 (38000) bytes
              KBAN7(H4) Strategy                              Part Number XS7VAJ
                                                  PATS ffffffffffffffffffffffffffffffff
                     VID Disabled

Options :
                  Default options

CheckSum :                                      SMP Base Address :         e000
                             c0de               CC Exe Time :              005d
                            Valid               Levels Number :               8
                                                Calibrations Number :         1
Banks :
              0   00000   0dfff                           1   0e000   1bfff
              8   1c000   29fff                           9   2a000   37fff

RBases :
             f0    2060    24b3                        f2    24b4    2739
             f4    273a    3cc9                        f6    3cca    48d7
             f8    48d8    5b85                        fa    5b86    630d
             fc    630e    8969                        fe    896a    a267
```

```
                                Registers List

R09 B     9                     R09 W     wAM_AIRMASS
R0f B     f                     R0f W     RPM
R13 B     13                    R13 W     wTP_REL
R5b       fEGR_MON_FLG2         R61       wLBMF_INJ1
R71       bP0406FLGS            R7e B     bOBDII_RDY
R7e W     wR7e                  R87       bOBDII_ENA
R9b       bECT                  R9c       bACT
Ra0       bDT12SH               Rb4       bATMR1
Rd9       wLoad                 Rdb       wPERLOAD
[10f]     bRPM                  [240]     wTSLAMU1
[242]     wTSLAMU2              [250]     wTOTLDST
[252]     wAIR_LD_CT            [254]     wPCT_LOAD
[256]     wCHT                  [259]     bISCFLG_LST
[25e]     wLAMMUL               [266]     scRPM_016
[268]     scLOAD_012            [270]     wSPK_FFS
[272]     wSPK_MBT_FFS          [274]     wSPK_BDL_FSC
[276]     wMFAMUL               [282]     wSPK_M_B_T
[284]     wSPK_MBT_LAST         [29c]     bSPK_MAX_TRET
[2a0]     bSPK_TIPSTATE         [2a1]     bSPK_TIPSLOPE
[2ad]     scLOAD_013            [2af]     scLOAD_013B
[2b0]     scECT_014             [2b1]     scECT_014A
[2b2]     scSPK_LAMBSE_015      [2b3]     scSPK_LAMBSE_015A
[2b7]     scACT_024F            [2b9]     scRPM_070X
[2ba]     scLOAD_013X           [2bb]     scRPM_017
[2c4]     wSPK_SAF_HOLD         [2c8]     wSPK_BDL
[2d2]     bSPK_LAMBSE           [2e0]     wDEBYMA
[2f8]     wMAFERR               [2fe]     wPERLOAD_ISC
[304]     bISCFLG               [33e]     bP1408FLGS
[376]     wBG_TMR               [517]     bAIR_LD_WOT
[522]     wPG_AIR               [586]     wTQ_BRAKE_S
[59c]     wINJ_ACTUAL           [59e]     bTQ_SOURCE
[5a4]     wTQ_LOSS              [5ba]     wTQ_NET_LED
[5bc]     wLOAD_TQ              [5c6]     bTR_LIM_OSC
[5d1]     bOSC_MULT             [5e2]     wTQDRV_DNDT
[63f]     bVSBAR                [642]     wTQ_BARL
[7d2]     wBP_WORD              [7d3]     bBP
[875]     bTQ_NORM_KAM          [878]     wINFAMB_KAM
[d06]     wDSDRPM_WORD          [d14]     wIDCI
[d16]     wIDC_CL               [dc4]     wMIS_TQ_THRES
[dc6]     wMIS_TQ_DELTA         [dc8]     wMIS_TQ_LAST
```

Some scalars.

```
AHISL - Injector High Slope AHISL:
1 3258: 22,37              Rf4+b30 AHISL        word       3722                  0,01

ALOSL - Injector Low Slope ALOSL:
1 325a: 22,37              Rf4+b32 ALOSL        word       3722                  0,01

FUEL_BKPT - Injector Breakpoint FUEL_BKPT:
1 325c: 50,01              Rf4+b34 FUEL_BKPT     word        150                   0,00

MINPW - Injector Min PW Clip MINPW:
1 325e: 4f,00              Rf4+b36 MINPW        word        4f                    0,30

GASOHOL_AFR - (non fuel correcting):
1 3260: 8f,3a              Rf4+b38 GASOHOL_AFR     word       3a8f                  14,64

NOMINAL_AFR - (Stoich_AFR):
1 3262: 8f,3a              Rf4+b3a NOMINAL_AFR     word       3a8f                  14,64
```

Some functions and tables.

```
FN044 - Load Scaling - FN044:
1 25a8: ff,ff,00,09        func              ffff,      900             2,00,     9,00
1 25ac: 80,bb,00,09        func              bb80,      900             1,46,     9,00
1 25b0: 00,7d,00,08        func              7d00,      800             0,98,     8,00
1 25b4: 00,4b,00,06        func              4b00,      600             0,59,     6,00
1 25b8: 00,00,00,00        func                 0,        0             0,00,     0,00
1 25bc: 00,00,00,00        func                 0,        0             0,00,     0,00
1 25c0: 00,00,00,00        func                 0,        0             0,00,     0,00
1 25c4: 00,00,00,00        func                 0,        0             0,00,     0,00

FN070E - RPM Scaling - FN070E:
1 25c8: ff,ff,00,09        func              ffff,      900         16383,75,     9,00
1 25cc: 60,6d,00,09        func              6d60,      900          7000,00,     9,00
1 25d0: 80,3e,00,07        func              3e80,      700          4000,00,     7,00
1 25d4: e0,2e,00,06        func              2ee0,      600          3000,00,     6,00
1 25d8: 20,1c,00,02        func              1c20,      200          1800,00,     2,00
1 25dc: 60,09,00,00        func               960,        0           600,00,     0,00
1 25e0: 00,00,00,00        func                 0,        0             0,00,     0,00

FN077 - PCT_LOAD Scaling (FN1039) - FN077:
1 25e4: ff,ff,00,08        func              ffff,      800             2,00,     8,00
1 25e8: 00,80,00,08        func              8000,      800             1,00,     8,00
1 25ec: 9a,19,00,00        func              199a,        0             0,20,     0,00
1 25f0: 00,00,00,00        func                 0,        0             0,00,     0,00
1 25f4: 00,00,00,00        func                 0,        0             0,00,     0,00

FN078 - ISCDTY Scaling (FN1039) - FN078:
1 25f8: ff,ff,00,0a        func              ffff,      a00             2,00,    10,00
1 25fc: 00,80,00,0a        func              8000,      a00             1,00,    10,00
1 2600: 00,00,00,00        func                 0,        0             0,00,     0,00
1 2604: 00,00,00,00        func                 0,        0             0,00,     0,00
1 2608: 00,00,00,00        func                 0,        0             0,00,     0,00

FN1036A - Load at Sealevel (LWFM):
1 260c: 08,06,04,04,03,03,02,02,03,03     table      8,  6,  4,  4,  3,  3,  2,  2,  3,  3     0,06,  0,05,  0,03,  0,03,  0,02,  0,02,  0,02,  0,02,  0,02,  0,02
1 2616: 39,30,1b,17,13,11,11,10,0b,0c     table     39, 30, 1b, 17, 13, 11, 11, 10,  b,  c     0,45,  0,38,  0,21,  0,18,  0,15,  0,13,  0,13,  0,13,  0,09,  0,09
1 2620: 4e,4a,3a,37,34,2e,2a,1e,19,14     table     4e, 4a, 3a, 37, 34, 2e, 2a, 1e, 19, 14     0,61,  0,58,  0,45,  0,43,  0,41,  0,36,  0,33,  0,23,  0,20,  0,16
1 262a: 54,56,4f,4f,4a,47,42,33,2c,24     table     54, 56, 4f, 4f, 4a, 47, 42, 33, 2c, 24     0,66,  0,67,  0,62,  0,62,  0,58,  0,55,  0,52,  0,40,  0,34,  0,28
1 2634: 54,5c,57,5a,59,57,56,45,3d,32     table     54, 5c, 57, 5a, 59, 57, 56, 45, 3d, 32     0,66,  0,72,  0,68,  0,70,  0,70,  0,68,  0,67,  0,54,  0,48,  0,39
1 263e: 56,5d,5b,5f,60,60,61,51,44,36     table     56, 5d, 5b, 5f, 60, 60, 61, 51, 44, 36     0,67,  0,73,  0,71,  0,74,  0,75,  0,75,  0,76,  0,63,  0,53,  0,42
1 2648: 56,5f,60,62,63,63,65,57,4b,3c     table     56, 5f, 60, 62, 63, 63, 65, 57, 4b, 3c     0,67,  0,74,  0,75,  0,77,  0,77,  0,77,  0,79,  0,68,  0,59,  0,47
1 2652: 56,5f,61,63,64,68,6e,59,4f,44     table     56, 5f, 61, 63, 64, 68, 6e, 59, 4f, 44     0,67,  0,74,  0,76,  0,77,  0,78,  0,81,  0,86,  0,70,  0,62,  0,53
1 265c: 57,5b,60,64,65,69,6d,5c,50,47     table     57, 5b, 60, 64, 65, 69, 6d, 5c, 50, 47     0,68,  0,71,  0,75,  0,78,  0,79,  0,82,  0,85,  0,72,  0,63,  0,55
1 2666: 60,65,65,66,68,6a,6d,5f,54,4b     table     60, 65, 65, 66, 68, 6a, 6d, 5f, 54, 4b     0,75,  0,79,  0,79,  0,80,  0,81,  0,83,  0,85,  0,74,  0,66,  0,59

FN1037 - Inferred BP Load:
1 2670: 00,00,00,00,00,00,00,00,00,00     table      0,  0,  0,  0,  0,  0,  0,  0,  0,  0     0,00,  0,00,  0,00,  0,00,  0,00,  0,00,  0,00,  0,00,  0,00,  0,00
1 267a: 05,00,00,01,01,01,00,00,00,00     table      5,  0,  0,  1,  1,  1,  1,  0,  0,  0     0,04,  0,00,  0,00,  0,01,  0,01,  0,01,  0,01,  0,00,  0,00,  0,00
1 2684: 0a,05,02,02,02,02,00,00,00,00     table      a,  5,  2,  2,  2,  2,  2,  0,  0,  0     0,08,  0,04,  0,02,  0,02,  0,02,  0,02,  0,02,  0,00,  0,00,  0,00
1 268e: 0b,08,06,06,05,03,02,02,04,03     table      b,  8,  6,  6,  5,  3,  2,  2,  4,  3     0,09,  0,06,  0,05,  0,05,  0,04,  0,02,  0,02,  0,02,  0,03,  0,02
1 2698: 0c,0a,09,13,13,0e,0d,04,08,05     table      c,  a,  9, 13, 13,  e,  d,  4,  8,  5     0,09,  0,08,  0,07,  0,15,  0,15,  0,11,  0,10,  0,03,  0,06,  0,04
1 26a2: 0c,0a,0a,18,16,13,11,06,0b,06     table      c,  a,  a, 18, 16, 13, 11,  6,  b,  6     0,09,  0,08,  0,08,  0,19,  0,17,  0,15,  0,13,  0,05,  0,09,  0,05
1 26ac: 0c,0b,0b,18,16,13,14,08,0c,07     table      c,  b,  b, 18, 16, 13, 14,  8,  c,  7     0,09,  0,09,  0,09,  0,19,  0,17,  0,15,  0,16,  0,06,  0,09,  0,05
1 26b6: 0c,0c,0c,18,16,13,14,09,0d,07     table      c,  c,  c, 18, 16, 13, 14,  9,  d,  7     0,09,  0,09,  0,09,  0,19,  0,17,  0,15,  0,16,  0,07,  0,10,  0,05
1 26c0: 0c,0c,0c,18,16,13,14,0a,0f,0a     table      c,  c,  c, 18, 16, 13, 14,  a,  f,  a     0,09,  0,09,  0,09,  0,19,  0,17,  0,15,  0,16,  0,08,  0,12,  0,08
1 26ca: 0d,0d,0d,18,16,13,14,0b,10,0a     table      d,  d,  d, 18, 16, 13, 14,  b, 10,  a     0,10,  0,10,  0,10,  0,19,  0,17,  0,15,  0,16,  0,09,  0,13,  0,08

FN1039 - Sealevel ISC Flow Inferred BP:
1 26d4: 00,00,04,18,20,3d,4f,59,60,73,73      table      0,  0,  4, 18, 20, 3d, 4f, 59, 60, 73, 73          0,00,  0,00,  0,06,  0,38,  0,50,  0,95,  1,23,  1,39,  1,50,  1,80,  1,80
1 26df: 00,00,04,14,1e,39,45,53,5a,60,73      table      0,  0,  4, 14, 1e, 39, 45, 53, 5a, 60, 73          0,00,  0,00,  0,06,  0,31,  0,47,  0,89,  1,08,  1,30,  1,41,  1,50,  1,80
1 26ea: 00,00,04,13,1b,32,44,50,56,5a,71      table      0,  0,  4, 13, 1b, 32, 44, 50, 56, 5a, 71          0,00,  0,00,  0,06,  0,30,  0,42,  0,78,  1,06,  1,25,  1,34,  1,41,  1,77
1 26f5: 00,00,04,0f,19,2e,3c,4d,4d,4e,6d      table      0,  0,  4,  f, 19, 2e, 3c, 4d, 4d, 4e, 6d          0,00,  0,00,  0,06,  0,23,  0,39,  0,72,  0,94,  1,20,  1,20,  1,22,  1,70
1 2700: 00,00,03,09,11,2a,35,3c,43,45,58      table      0,  0,  3,  9, 11, 2a, 35, 3c, 43, 45, 58          0,00,  0,00,  0,05,  0,14,  0,27,  0,66,  0,83,  0,94,  1,05,  1,08,  1,38
1 270b: 00,00,04,08,0c,14,25,2f,2f,30,40      table      0,  0,  4,  8,  c, 14, 25, 2f, 2f, 30, 40          0,00,  0,00,  0,06,  0,13,  0,19,  0,31,  0,58,  0,73,  0,73,  0,75,  1,00
1 2716: 00,00,02,04,08,0e,10,12,14,16,24      table      0,  0,  2,  4,  8,  e, 10, 12, 14, 16, 24          0,00,  0,00,  0,03,  0,06,  0,13,  0,22,  0,25,  0,28,  0,31,  0,34,  0,56
1 2721: 00,00,01,02,05,06,09,0b,0c,0d,02      table      0,  0,  1,  2,  5,  6,  9,  b,  c,  d,  2          0,00,  0,00,  0,02,  0,03,  0,08,  0,09,  0,14,  0,17,  0,19,  0,20,  0,03
1 272c: 00,00,00,00,00,00,00,00,00,00,00      table      0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0          0,00,  0,00,  0,00,  0,00,  0,00,  0,00,  0,00,  0,00,  0,00,  0,00
```

A classical routine.

```
OBDII_OSC_SUB:
9 2436: f2                  pushp                   push(PSW);
9 2437: c7,01,bd,17,00      stb   [17bd],0          [bOSC_SUB] = 0;
9 243c: c7,01,bc,17,00      stb   [17bc],0          [bOSC_BYTOP] = 0;
9 2441: c3,01,ba,17,00      stw   [17ba],0          [wOSC_ADDR] = 0;
9 2446: c3,01,b8,17,00      stw   [17b8],0          [bOSC_OVAL] = 0;
9 244b: b0,6e,46            ldb   R46,R6e           R46 = [fSCP_EXT_FG1];
9 244e: 3c,46,02            jb    B4,R46,2453       if (B4_R46) goto 2453;
9 2451: 20,de               sjmp  2531              goto 2531;

9 2453: b3,01,79,17,46      ldb   R46,[1779]        R46 = [bOSC_STATE];
9 2458: 99,02,46            cmpb  R46,2
9 245b: df,02               je    245f              if (R46 == 2) goto 245f;
9 245d: 20,d2               sjmp  2531              goto 2531;

9 245f: ef,c9,04            call  292b              Sub1063();
9 2462: a2,20,30            ldw   R30,[R20]         R30 = [STACK];
9 2465: f2                  pushp                   push(PSW);
9 2466: fa                  di                      disable ints;
9 2467: 18,02,31            shrb  R31,2             R31 = R31 / 4;
9 246a: c4,11,31            stb   R11,R31           BANK_SEL = R31;
9 246d: a3,20,04,26         ldw   R26,[R20+4]       R26 = [STACK+4];
9 2471: b2,27,36            ldb   R36,[R26++]       R36 = [R26++];
9 2474: b2,27,37            ldb   R37,[R26++]       R37 = [R26++];
9 2477: b2,27,38            ldb   R38,[R26++]       R38 = [R26++];
9 247a: b2,27,39            ldb   R39,[R26++]       R39 = [R26++];
9 247d: b2,27,3a            ldb   R3a,[R26++]       R3a = [R26++];
9 2480: b2,27,3b            ldb   R3b,[R26++]       R3b = [R26++];
9 2483: ae,27,40            ldzbw R40,[R26++]       R40 = (uns)[R26++];
9 2486: b1,11,11            ldb   R11,11            BANK_SEL = 11;
9 2489: f3                  popp                    pop(PSW);
9 248a: c3,20,02,26         stw   [R20+2],R26       [STACK+2] = R26;
9 248e: 99,ff,3a            cmpb  R3a,ff
9 2491: df,08               je    249b              if (R3a == ff) goto 249b;
9 2493: b1,01,34            ldb   R34,1             R34 = 1;
9 2496: c7,01,bc,17,34      stb   [17bc],R34        [bOSC_BYTOP] = R34;
9 249b: c3,01,ba,17,36      stw   [17ba],R36        [wOSC_ADDR] = R36;
9 24a0: 9b,01,bc,17,00      cmpb  0,[17bc]
9 24a5: d7,0a               jne   24b1              if (0 != [bOSC_BYTOP]) goto 24b1;
9 24a7: a2,36,46            ldw   R46,[R36]         R46 = [R36];
9 24aa: c3,01,b8,17,46      stw   [17b8],R46        [bOSC_OVAL] = R46;
9 24af: 20,08               sjmp  24b9              goto 24b9;
```

Registers are not automatically set with ‘b’ or ‘w’, this is what I have in my own definition, because EEC V strategies often share registers, and meaning at byte level is not the same than the one at word level.

A binary end to finish.

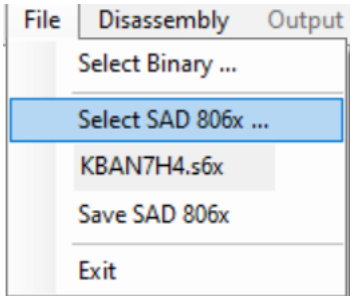
```
Strategy:
9 ff06: 4b,42,41,4e,37,48,34
KBAN7H4


9 ff0d: 2e,48,45,58,2a,ff,00                          Unknown Operation/Structure


Part Number:
9 ff14: 58,53,37,56,41,4a,20
XS7VAJ


9 ff1b: 2a,ff                                         Unknown Operation/Structure


PATS Code:
9 ff1d: ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff


9 ff37 -> ff62                    fill                ff


Copyright:
9 ff63: 43,6f,70,79,72,69,67,68,74,20,56,69,73,74,65,6f,6e,20,43,6f,72,70,2e,20,20,32,30,30,32
Copyright Visteon Corp.  2002

VIN Code:
9 ff80: ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff


9 ff91: ff,ff,ff,ff                                   Unknown Operation/Structure


9 ff95: ff                        0                   VID Block Enabled


9 ff96: ff,ff,ff,ff                                   Unknown Operation/Structure


9 ff9a: ff,ff                     0                   Tyre Revolutions per Mile
9 ff9c: ff,ff                     0                   Rear End Gear Ratio


9 ff9e -> fffe                    fill                ff

9 ffff: 91                                            Unknown Operation/Structure
```

End of Disassembly

SAD 806x menu:

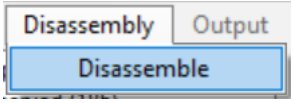
As described at the beginning of the document, all options are not available all the time. I will not come back on these details, I will try to describe which actions are executed by each option, because now, you probably better understand SAD 806x and how it is built.

File menu:



- 'Select Binary …' option permits to show the open file dialog, to select the appropriate rom to be disassembled. By default, SAD 806x will show '.bin' files, but you can use any file you want, at your own risks.
  Selected rom will never be updated by SAD 806x, it will use it as a read only file.
  When file is selected it is directly loaded and its related SAD 806x definition file ('.s6x') too. A status appears to give the result.
- 'Select SAD 806x …' option permits to show the open file dialog, to select another SAD 806x definition file ('.s6x'), which will replace the default one. By default, SAD 806x will show '.s6x' files, but you can use any file you want, at your own risks. Name of the file appears below the option, to be sure.
- 'Save SAD 806x' option permits to save the current definition, into the SAD 806x definition file ('.s6x'). If the file is not existing, it creates it with its default name (same as the rom one). Do not forget to save your file before closing application or before switching to another rom.
- 'Exit' option will close the application.

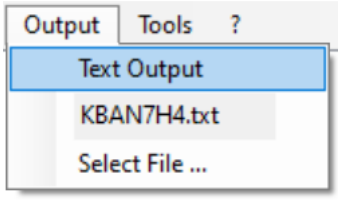
Disassembly menu:



Nothing more to say on this menu, it has only one option and is available only for properly loaded binaries.

- 'Disassemble' option will start disassembly process. Everything is done in memory, no output is done at this level. A status appears to give the result, when disassembly has finished. This process can take some time.

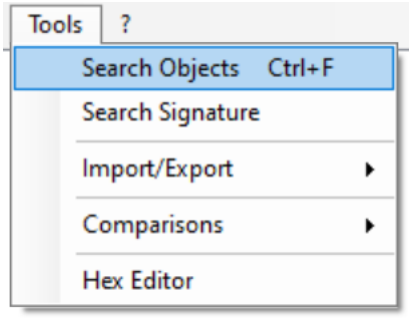
Output menu:



Nothing more to say on this menu too, it has only one option and is available only after disassembly. If you update the SAD 806x definition, you need to disassemble you binary with the new definition, to be able to generate output another time.

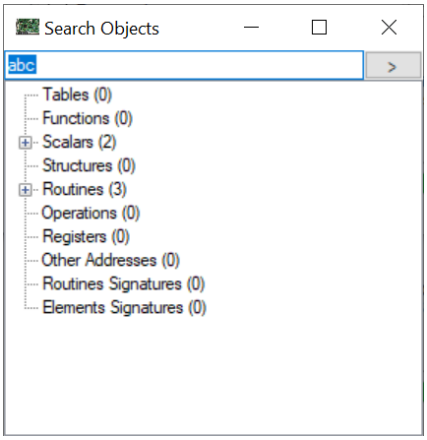
- 'Text Output' option will create or overwrite a text file, with the disassembled code. Everything is done, by the disassembled elements which are in memory. By default outputted file will be in the same folder, with the same name than the binary. This process can take some time.
- File name text box is found below 'Text Output' option. It permit to see what the name of the destination file is. When output is done, by double clicking on this text box, computer default text editor will open the file.
- 'Select File …' option permits to show the open file dialog, to select another destination file ('.txt'), which will replace the default one. By default, SAD 806x will show '.txt' files, but you can use any file you want, at your own risks. Name of the file appears before the option, to be sure.

Tools search menus:



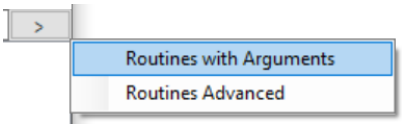
Search options permit to search elements in definition or signatures in binaries.

- 'Search Objects' is available at any time and from anywhere through 'Ctrl+F' shortcut. It permits to search in definition for anything, through a basic text search. This form will be displayed.



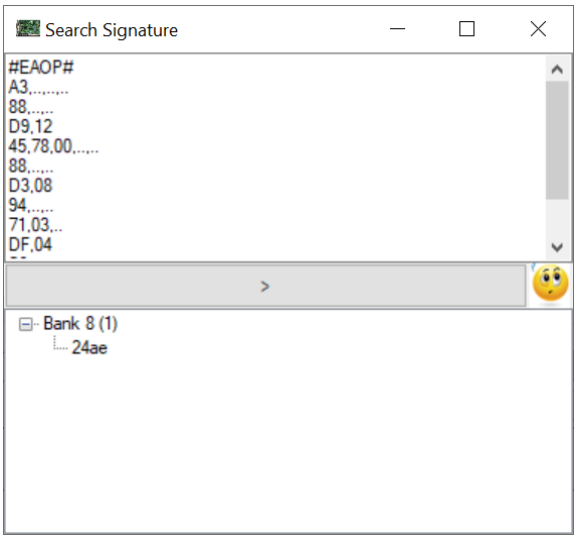
All text fields present on elements will be parsed, including addresses, based on provided search word which shoould be contained somewhere. The result will appear in list and by selecting an item, it will be opened in main application.

Please notice, that when right clicking on the button, on the right, you can exectute special searches.



'Routines with Arguments' will provide all routines which use input arguments and 'Routines Advanced', all routines set as advanced ones.

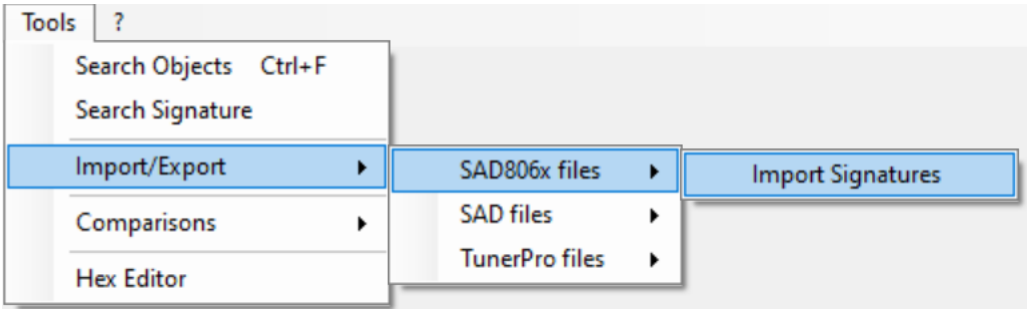
- 'Search Signature' is available when binary is properly loaded. It permits to search a signature directly in binary (its hexadecimal text version), exactly like it is done on disassembly for 'Routines Signatures' and 'Elements Signatures'. Its second, but main in fact, purpose is to validate a signature when writing one in signatures parts. This form will be displayed.



Text box on the top is for the signature. Button in the middle is for searching and yellow smiley, to help on writing signature (this is the 'Elements Signatures' help here) and the list on the bottom, for the result.

Here you can see that the signature has matched at address 0x24ae on bank 8. Multiple matches will appear, if this is the case.

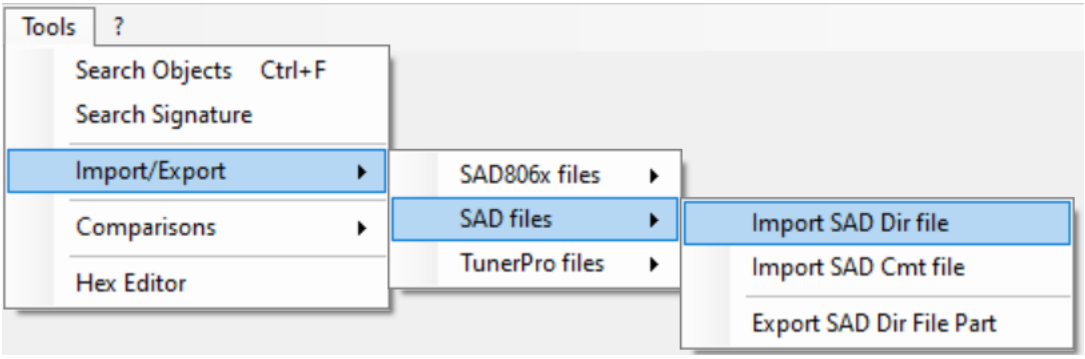
Tools Import/Export menus:



First Import/Export format is SAD 806x definition itself. For now I only see one thing interesting to be imported massively from one SAD 806x definition to another, I am talking about signatures, which are really shared between strategies. For other things, Copy and Paste work well because it is working element by element and for everything else, the repository is perfect.

'Import Signatures' option permits to show the open file dialog, to select another SAD 806x definition file ('.s6x'). By default, SAD 806x will show '.s6x' files, but you can use any file you want, at your own risks. Then it will add or update all signatures from selected definition file to the current one.

For 'Routines Signatures', matching will be done on the 'Short Label'.
For 'Elements Signatures', matching will be done on the 'Short Label' of the defined element.



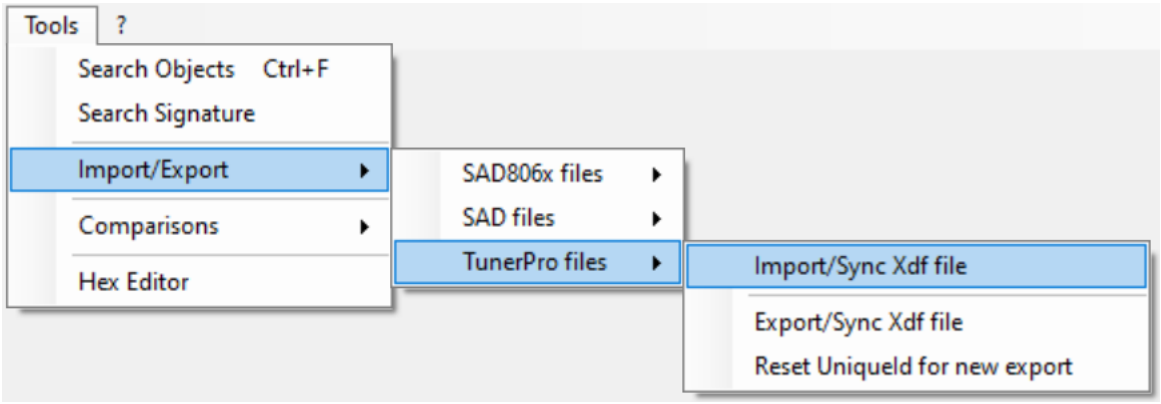
Another Import/Export format is for SAD itself. It will manage definitions (.dir files) and comments (.cmt files). The goal is to import as many things as possible, when definition or comments were written for SAD.

'Import SAD Dir file' option permits to show the open file dialog, to select a SAD definition file ('.dir'). By default, SAD 806x will show '.dir' files, but you can use any file you want, at your own risks. Then it will add or update elements based on the address declared in SAD definition.
If nothing exists at the related address and if element is properly declared in SAD definition, SAD 806x will have no issue to create it at the right place, otherwise it will try to identify it based on address and finally, if it has no correspondence, all interesting details will be put in an 'Other Address'. If something is already declared at the address, it will be overwritten, if it has the same type, otherwise it will be ignored.
Only calibration elements (scalars, functions, tables, and structures), operations, routines and registers are imported. Vectors are not managed.

It is a huge text processing, so prefer to backup you SAD 806x definition before doing it, to permit to go back if required. The process works better, if binary is disassembled before.

'Import SAD Cmt file' option permits to show the open file dialog, to select a SAD comments file ('.cmt'). By default, SAD 806x will show '.cmt' files, but you can use any file you want, at your own risks. Then it will add or update elements based on the address declared in SAD comments.
If nothing exists at the related address, SAD 806x has no way to know what is the type of the element, so it will be put in an 'Other Address'. If something is already declared at the address, comments will be overwritten.
It is a huge text processing too, so prefer to backup you SAD 806x definition before doing it, to permit to go back if required. The process works better, if binary is disassembled before and SAD definition file has been imported before comments.

'Export SAD Dir File Part' option permits to show the save file dialog, to select a SAD definition file ('.dir'). By default, SAD 806x will show '.dir' files, but you can use any file you want, at your own risks. Then it will create or overwrite file with compatible elements coming from SAD 806x. This is not a synchronization process, SAD 806x will try to generate all compatible elements in a SAD definition file which is initialized by default with classical SAD definition header. It has to be reviewed properly before being used by SAD.
Only calibration elements (scalars, functions, tables, and basic structures), operations and routines are exported.
Just backup you SAD definition before doing it, otherwise elements not managed by SAD 806x will be lost.



The most important Import/Export format is TunerPro definition itself. Where SAD 806x permits to quickly and properly prepare a definition, TunerPro can use one to update a binary file. So it is essential to be able to synchronize SAD 806x and TunerPro definition, at least for Ford EEC managements.

- 'Import/Sync Xdf file' option permits to show the open file dialog, to select a TunerPro definition file ('.xdf'). By default, SAD 806x will show '.xdf' files, but you can use any file you want, at your own risks. Please check that TunerPro definition is not locked/crypted before using it.
  Then it will add or update all compatible elements (definition properties, scalars, bit flags, functions and tables). When SAD 806x definition has always been synchronized, XDF UniqueIds are stored in SAD 806x definition and it will try to match on it. When

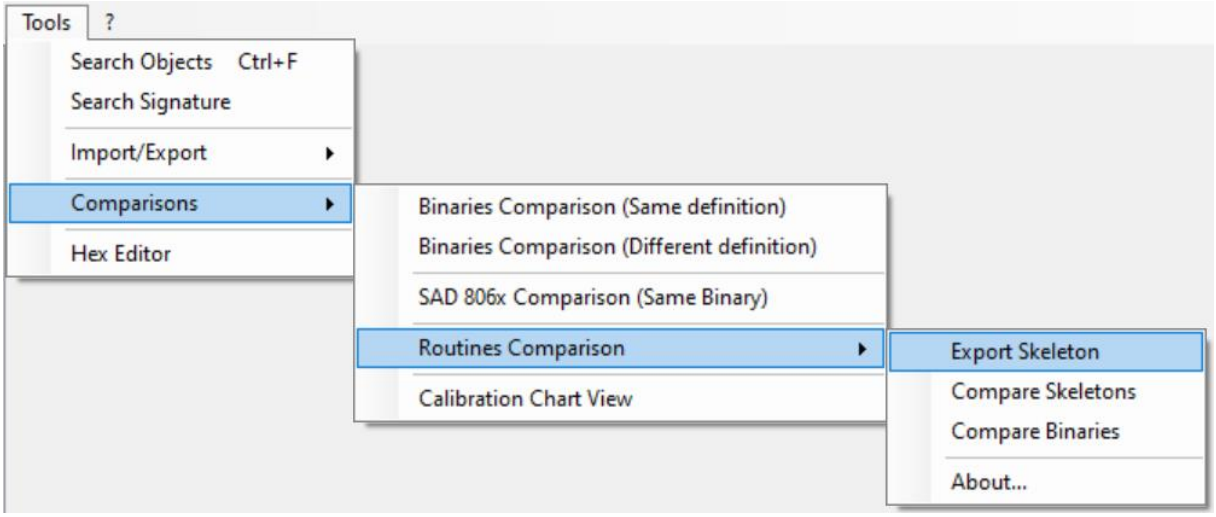
UniqueId does not exist or is not found, it tries to match on 'Short Label'. But as you probably know, TunerPro does not use a 'Short Label', so SAD 806x tries to decompose TunerPro label into <'Label' – 'Short Label' *> or TunerPro description into <'Short Label' – 'Label' *> and then matching is done on found 'Short Label' if one. As you will see the TunerPro description <'Short Label' – 'Label' *> will be the best compatibility basis, with SAD 806x comments <'Short Label' – 'Label' *>. Matching elements will be overwritten in SAD 806x definition, so it can be a good idea to backup it before processing.
Process can take some time. Some elements will not match because of their type, so a message will give their list. Duplicated addresses are now managed in SAD 806x, so they will be processed on import.

- 'Export/Sync Xdf file' option permits to show the save file dialog, to select a TunerPro definition file ('.xdf'). By default, SAD 806x will show '.xdf' files, but you can use any file you want, at your own risks. Please check that TunerPro definition is not locked/crypted before using it. A backup is done before processing for TunerPro definition, but it can be a good idea to do it for SAD 806x definition, you will understand why.
  Then it will firstly match elements (by XDF UniqueId or by 'Short Label') inside SAD806x definition and finally, it will add or update all compatible elements in TunerPro definition (definition properties, scalars, bit flags, functions and tables), but all non-compatible elements (categories, patches, …) will stay intact in definition. Matching elements will be overwritten in TunerPro definition, new elements will be created and their XDF UniqueIds will be set in related SAD 806x elements, so you can understand, that it could be a good idea to backup SAD 806x before processing.
  Process can take some time. Duplicated addresses are now managed in SAD 806x, so they will be processed on export.

- 'Reset UniqueId for new export' option permits to empty this XDF UniqueId on all elements in SAD 806x definition. It is interesting to do it, when it has already been synchronized and when exporting to a new TunerPro definition. Then new XDF UniqueId will restart from the beginning, taking the address order, so TunerPro definition, will be sorted by address. Same thing when importing from a different TunerPro definition, it permits to prevent mismatching, but I do not think it has to be done from different TunerPro definitions, it is not a good idea.

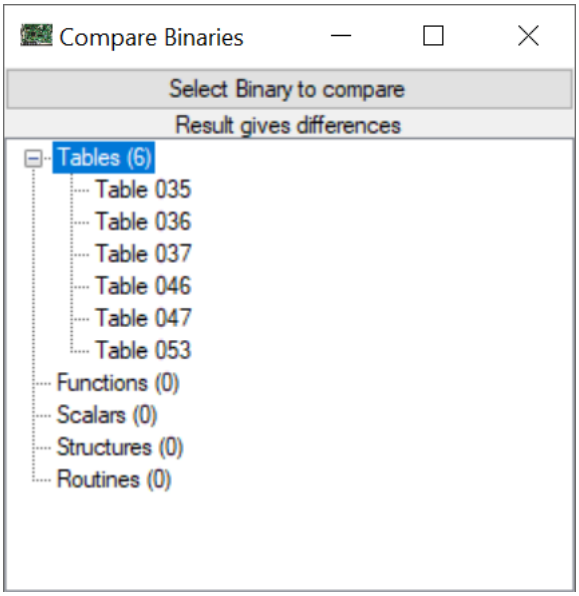
From my experience, I can say, that my referential for a definition is now a SAD 806x one. TunerPro is at the end of the loop, when it is required to update the binary and when definition is advanced enough to do it. All my work is done directly in SAD 806x for definition part and I create a new TunerPro definition when needed. Patches and other things are duplicated from previous TunerPro definition version if necessary.

Tools Comparisons menus:



Comparisons menu contains all required tools to compare binaries between them and definitions between them. It can be done through different ways, which I will try to explain.

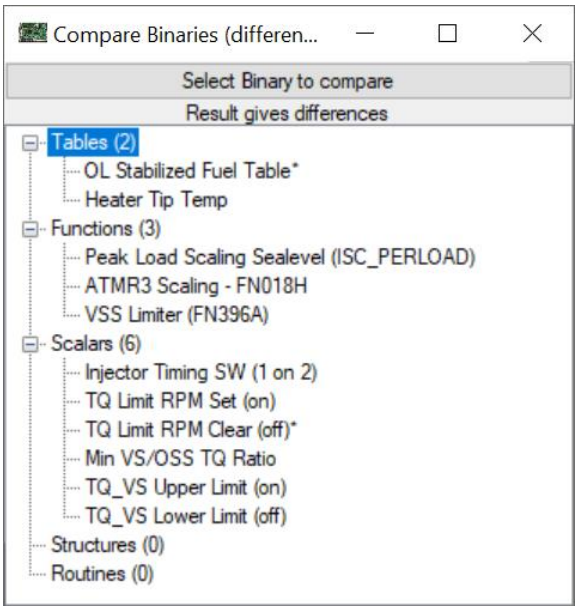
- 'Binaries Comparison (Same definition)' option permits to compare 2 binaries (including the current disassembled one) and to see which elements have been modified between them. This comparison has to be used when both binaries are using the same definition, in fact the same strategy, even if strategy version is different. Differences are only detected for known elements at their known addresses. Current definition has not to be really advanced to do it. If you want more it has to be done with a hexadecimal editor or with the text output inside a text editor.
  It opens this form:



  The 'Select Binary to compare' button will show the open file dialog, to select a binary file. By default, SAD 806x will show '.bin' files, but you can use any file you want, at your own risks. Then it will directly compare current binary, with the selected one and it will output known elements detected in difference, in its result list. With mouse over an element, you will have some information, when selecting it, you will open it in main

application. Differences are not detailed, for now it is required to open binary in another SAD 806x session.

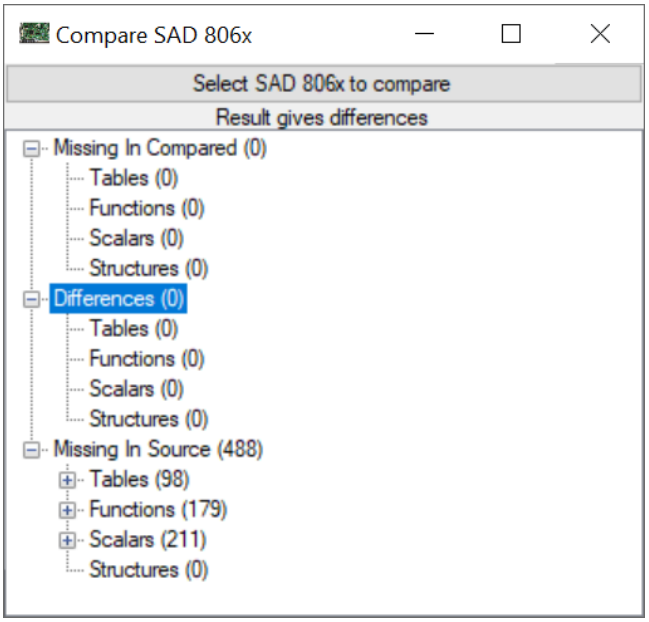
- 'Binaries Comparison (Different definition)' option permits to compare 2 binaries (including the current disassembled one) and to see which elements have been modified between them. This comparison has to be used when both binaries are not using the same definition, not the same strategy, but when they are somehow identical, like for example the same engine on 2 different strategies or a Ford EEC update which has changed the strategy code. Differences are only detected for known elements, based on their 'Short Label', so which should exist in both definitions. Both definitions have to be a bit advanced to do it.
  It opens this form:



  The 'Select Binary to compare' button will show the open file dialog, to select a binary file. By default, SAD 806x will show '.bin' files, but you can use any file you want, at your own risks. Definition related with selected binary should be in the same folder and should have the same name, with '.s6x' extension. Then, it will load selected binary, disassemble it (it takes some time), based on its linked definition and it will compare current binary, with the selected one and it will output known elements, with same 'Short Label' detected in difference, in its result list. With mouse over an element, you will have some information, when selecting it, you will open it in main application. Differences are not detailed, for now it is required to open binary in another SAD 806x session.

- 'SAD 806x Comparison (Same Binary)' option permits to compare 2 SAD 806x definitions (including the current opened one) and to see which user defined elements have a different definition or are not defined. It is useful to see what has changed between 2 versions. On my side I use it to see what has changed between 2 TunerPro definitions, it requires to create a new SAD 806x definition, import the new TunerPro definition and then to compare. This comparison has to use a common basis, which is the binary or a compatible one with same strategy. Differences are only detected for defined elements in one definition or in the other, based on their addresses. At least one definition has to be a bit advanced to do it.

It opens this form:



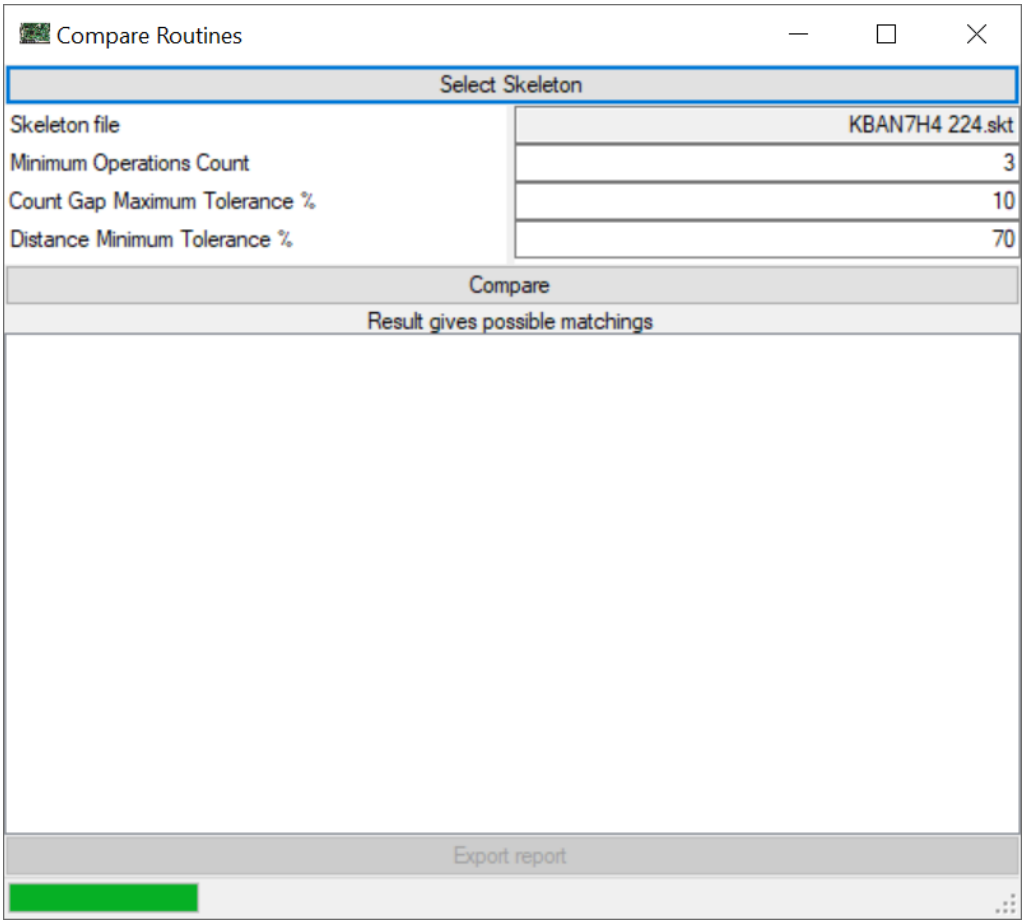
The 'Select SAD 806x to compare' button will show the open file dialog, to select a SAD 806x definition file. By default, SAD 806x will show '.s6x' files, but you can use any file you want, at your own risks. It will compare current definition, with the selected one and it will output each element in difference, in its result list. First part 'Missing In Compared' is for elements which are existing in current definition, but not in selected one, the last one 'Missing in Source' is the opposite and 'Differences' part shows differences when elements exist on both sides and are a bit different (based on a defined set of properties for each type of element). Managed elements are 'Scalars', 'Functions', 'Tables' and 'Structures'. With mouse over an element, you will have some information, when selecting it, you will open it in main application, if it exists in current definition. Differences are not detailed, for now it is required to open definition in another SAD 806x session. In this case current definition was empty, compared to a well advanced one.

- 'Routines Comparison' menu permits to access some interesting options. For now comparisons tools have permitted to compare relatively closed things, which is for sure necessary, but it does not help to advance on a proper disassembly which is globally unknown at its start.
  'Routines Comparison' will permit to compare code from routines between current binary (and its definition) and another one (and its definition too), but without real link between them, it is for example possible to compare EEC IV binaries and EEC V binaries. As you probably know it, the more near in time the binaries are, the more near will be their routines.
  If you are able to match one routine from one binary, where you have identified used elements and/or register, with another routine from another binary, you will be able to match used elements and registers too. This is the goal here.
  To compare routine quickly and properly, the best way, another time was to use hexadecimal code. But no signature to write here, it is somehow automatic. The complete code inside the routine is not used, it is a skeleton, which is used. This skeleton is composed with instructions only, sometimes modified to get better results, so it is

some kind of signature, with only instructions. You can see an example, because you can export one through 'Export Skeleton' into a text file.
Then skeletons from one binary are compared to the other, routine by routine. The method used is to calculate the proximity between routines skeletons, through the Damerau-Levenshtein distance algorithm. Below a number of operations, some routines are ignored, over a certain distance routines are managed as different and when everything is inside values routines are managed as matching.

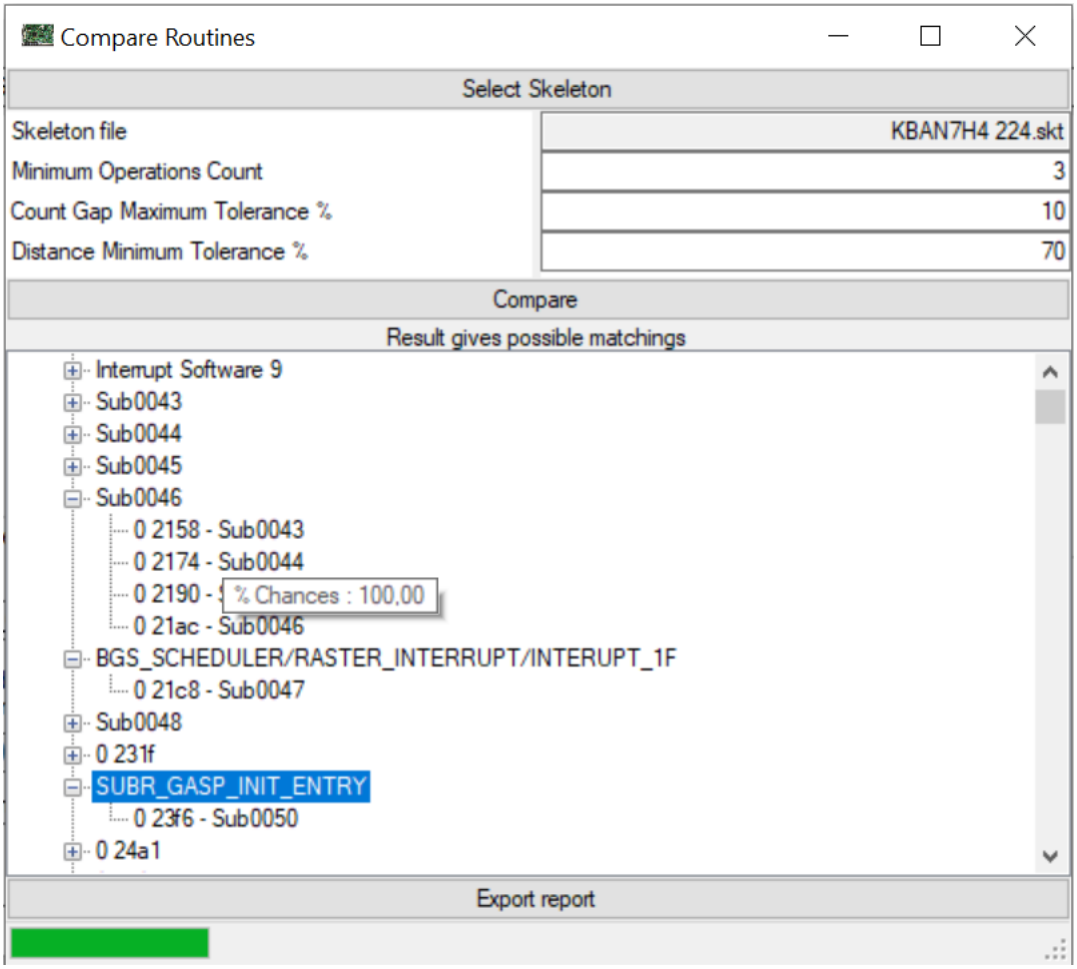
- 'Export Skeleton' option permits to show the save file dialog, to select a skeleton file ('.skt'). By default, SAD 806x will show '.skt' files, but you can use any file you want, it is a text file. Skeleton will be generated from current disassembled binary, but will only store routines, their details, their code, but not their elements.
This skeleton file can be reused at any time with the next option.

- 'Compare Skeleton' option permits to compare routines skeleton, based on current disassembled binary and another one, which was saved previously from another disassembled binary, through 'Export Skeleton' option.
It opens this form, which is the 'Compare Routines' form:



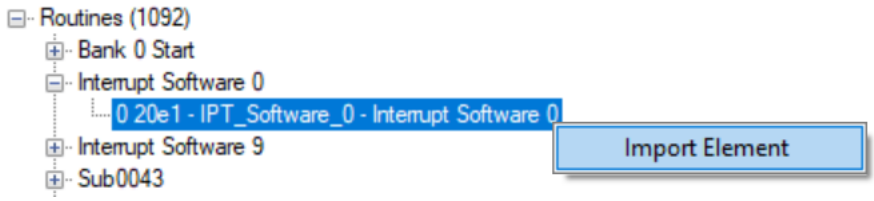
'Select Skeleton' button will show the open file dialog, to select a previously saved skeleton file ('.skt'). By default, SAD 806x will show '.skt' files.
'Skeleton file' text box will show you name of the selected file.
'Minimum Operations Count' number, defaulted, is the minimum number of operations in a routine to permit to compare it. Below this number, routine will be ignore.
'Count Gap Maximum Tolerance %' percent, defaulted, is the maximum gap, for

operations number in routines, presented as percent, between 2 routines to be compared. Over this percent, routines will not be compared to each other. At 10%, a routine with 90 operations will be compared to another with 100 operations, but same routine will not be compared to a routine with 110 operations.
'Distance Minimum Tolerance %' percent, defaulted, is the Damerau-Levenshtein distance, presented as percent, between 2 compared routines. Let say that 100% is for fully identical routines and 0% for nothing similar between both routines. Below given value, routines are considered as different and over they are considered as matching.
'Compare' button, will start the process, it will generate routines skeletons for current disassembled binary and then it will compare it to provided skeleton file, based on given parameters.



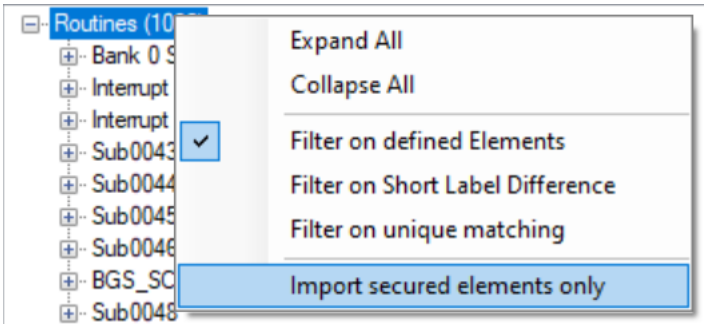
'Result' appears in results list part. With 'Compare Skeleton' option, it is only possible to give routines as result and then to analyze them one by one in disassembly. The more you have matching routines, the nearer are you binaries or strategies. If you put your mouse over a routine or a matching routine, you will see additional details, like '% Chances' which is the opposite of the Damerau-Levenshtein distance, presented as percent (100% is the best proximity). By clicking on a routine, it will be shown (if declared) in main application. Result could give routines which are not visible in one definition or another, because it is not exactly the main routines which are used for comparison, so in this case routine will appear with its address only, without a 'Short Label'. When you see a multiple matching, often for small routines, it is a bit more complicated to choose one.

'Matching Element' menu is available by right clicking on a routine or its matching equivalent.



Only one option is available, 'Import Element', which permits to copy values from the matching equivalent to the one on the current definition. For 'Routines', only 'Short Label' and 'Label' are copied, for security reasons. If menu was shown from current definition routine, but with multiple matchings, it will do nothing, it works only when it is a single matching.

'Elements Category' menu is available by right clicking on a category, here we have only 'Routines' one which is available.



'Expand All' and 'Collapse All' options are easy to understand at this level.

'Filter on defined Elements' is a checkbox, which will reduce number of elements in list, on fact that they are user defined (something was updated on them by someone, and saved in definition, it is not automatically generated). It permits to remove from list non interesting elements.

'Filter on Short Label Difference' is a checkbox, which will reduce number of elements in list, on fact that the 'Short Label' has to be different between current definition routine and its matching equivalent. It permits to remove from list, already copied elements.

'Filter on unique matching' is a checkbox, which will reduce number of elements in list, on fact that they have only one matching equivalent. It permits to remove from list, non-sure elements.

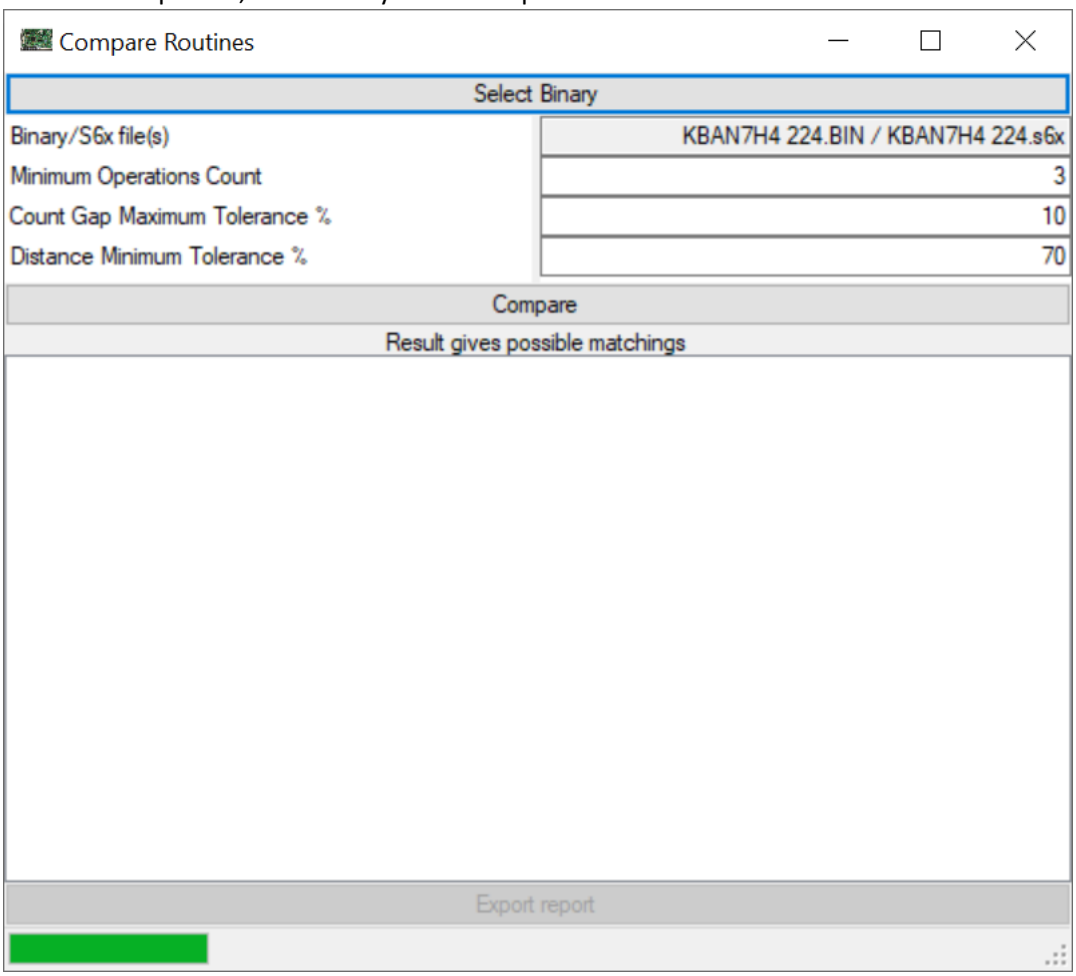
'Import secured elements only' option will do the same thing than 'Import Element' at element level, but here on the whole category, 'Routines' in this case. It will apply on all elements compatible with selected filters, but 'secured' means, that in all cases, it applies on defined elements and unique matching only, with or without these filters checked.

'Export report' button will show the save file dialog, to select an output file for the text report, which will contains the same thing than the shown result. It permits to easily switch between text report, disassembled text outputs and SAD 806x when updating definition and it permits to keep a trace too.

'Compare Skeleton' is a good starting, point, but is not automatic enough, compared with the next option.

- 'Compare Binaries' option is a kind of all in one process, which cumulates disassembly, skeleton export and skeleton comparison, but with the whole range of analyzed elements, because everything from both disassembled binaries is in memory, in the same

session. Except that, use is really near ‘Compare Skeleton’.

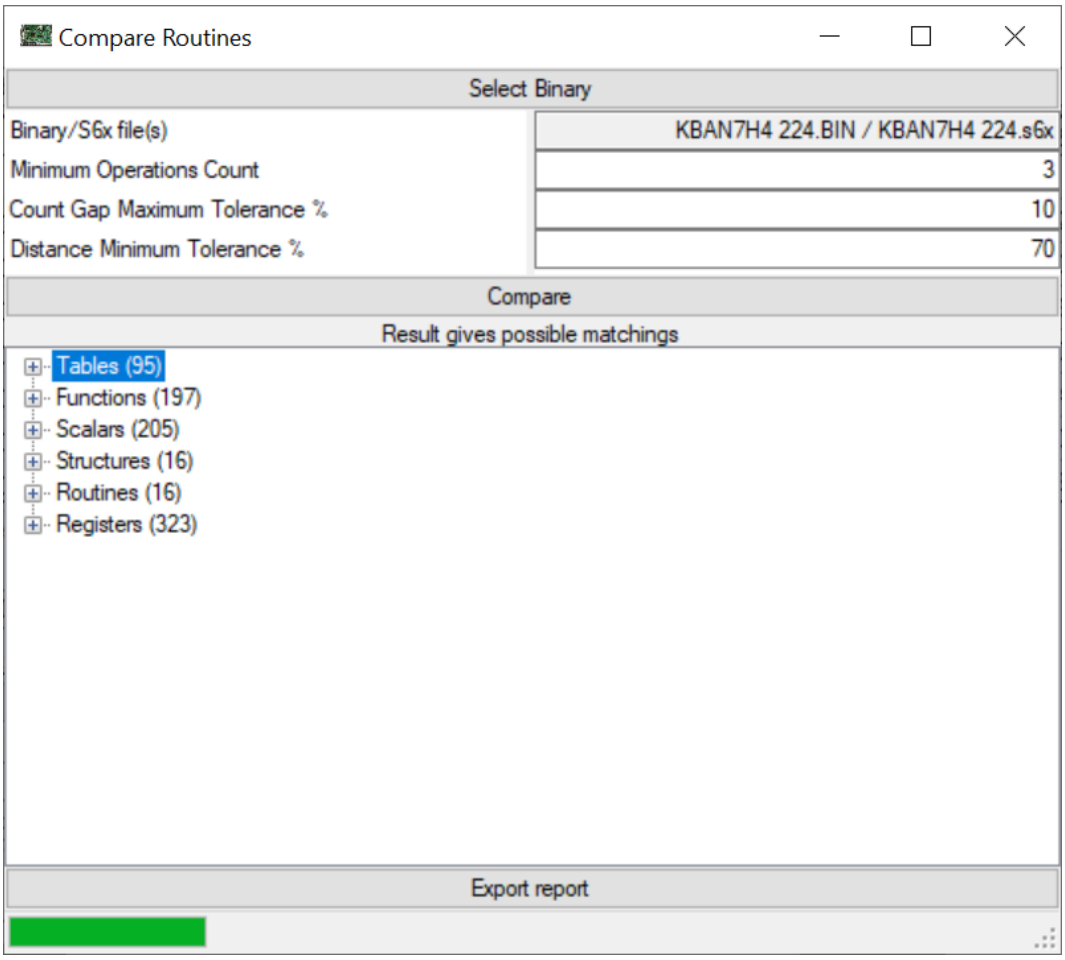


‘Select Binary’ button will show the open file dialog, to select a binary file. By default, SAD 806x will show ‘.bin’ files.

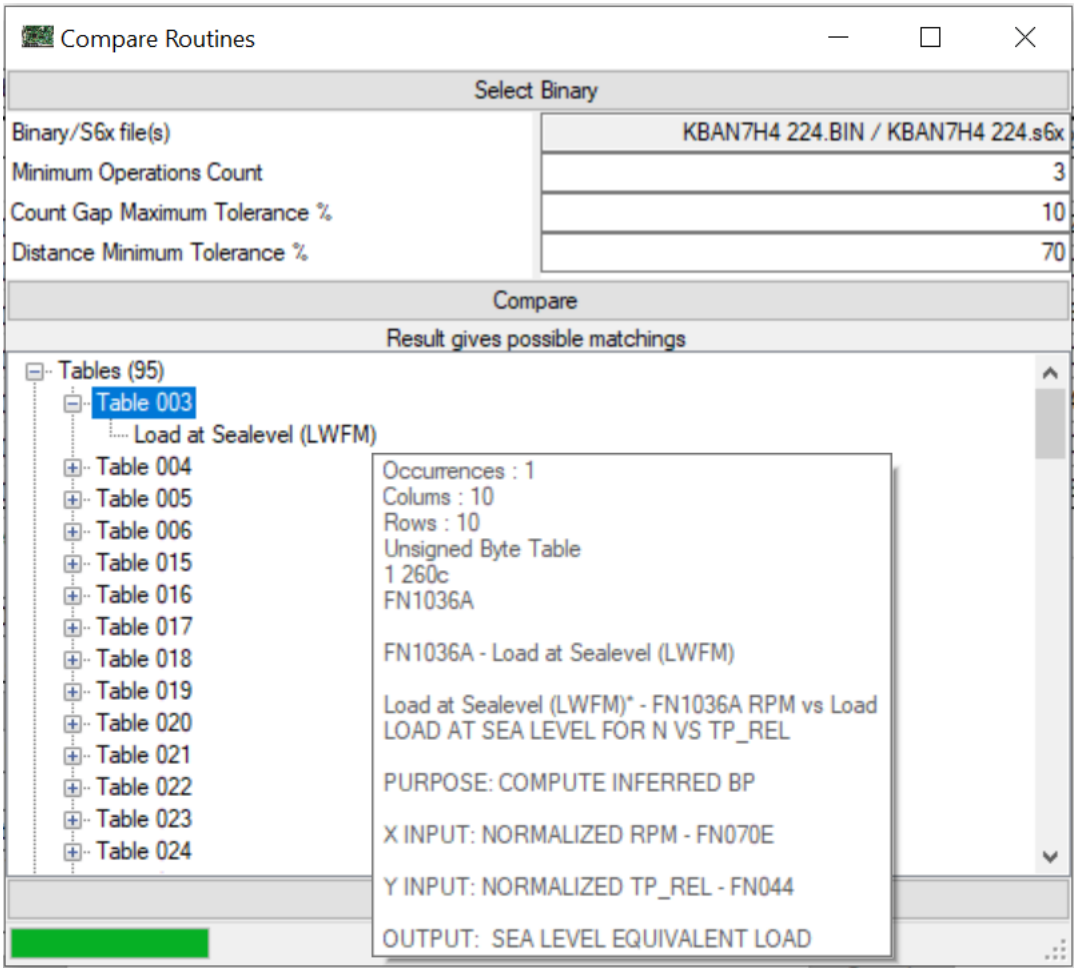
‘Binary/S6x file(s)’ text box will show you name of the selected binary file and if it has an available SAD 806x definition (.s6x) file, with the same name in the same folder.

‘Minimum Operations Count’, ‘Count GAP Maximum Tolerance %’ and ‘Distance Minimum Tolerance %’ are exactly working in the same way than with ‘Compare Skeleton’.

‘Compare’ button, will start the process, but in this case, the first step is to disassemble selected binary, which will take some time, then it will generate routines skeletons for both disassembled binaries and then it will compare them, based on given parameters. At this moment, in memory we have matching routines between one binary and the other, like it was the case with ‘Compare Skeleton’, but the process will now continue. For surely matched routines (unique matching only), it will try to find matching elements (scalars, functions, tables, structures) and matching registers, at the same place or with the same tolerance and everything will be proposed as result.

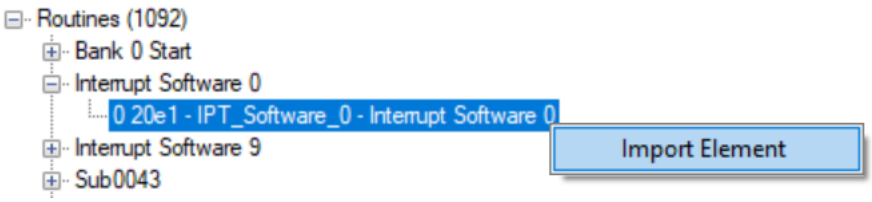


'Result' appears in results list part too. With 'Compare Binaries' option, result can now contain routines, scalars, functions, tables, structures and registers. The more you have matching routines, the nearer are you binaries or strategies.
For 'Routines', if you put your mouse over a routine or a matching routine, you will see additional details, like '% Chances' which is the opposite of the Damerau-Levenshtein distance, presented as percent (100% is the best proximity). By clicking on a routine, it will be shown (if declared) in main application. Result could give routines which are not visible in one definition or another, because it is not exactly the main routines which are used for comparison, so in this case routine will appear with its address only, without a 'Short Label'. When you see a multiple matching, often for small routines, it is a bit more complicated to choose one.



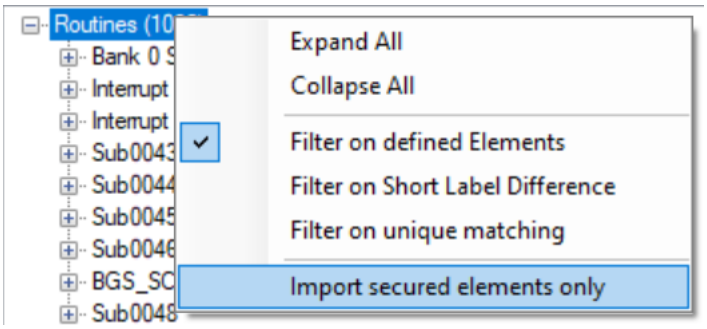
For other elements and registers, if you put your mouse over an element or a matching element, you will see additional details, like 'Occurrences' which tells you how many times, this matching was detected in all routines. By clicking on an element, it will be shown in main application.

'Matching Element' menu is available by right clicking on a element or its matching equivalent.



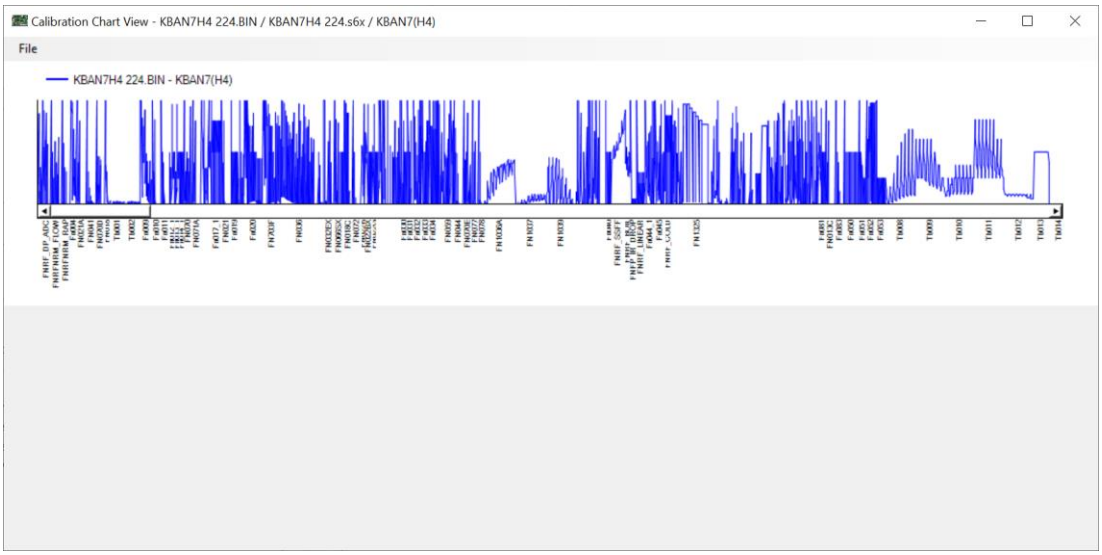
Only one option is available, 'Import Element', which permits to copy values from the matching equivalent to the one on the current definition. For 'Routines', only 'Short Label' and 'Label' are copied, for security reasons, for other elements and registers, all properties are copied. If menu was shown from current definition element, but with multiple matchings, it will do nothing, it works only when it is a single matching. A message could appear, when something is not clear, like a different type or a different number of rows or columns, to validate or cancel copy.

'Elements Category' menu is available by right clicking on a category.

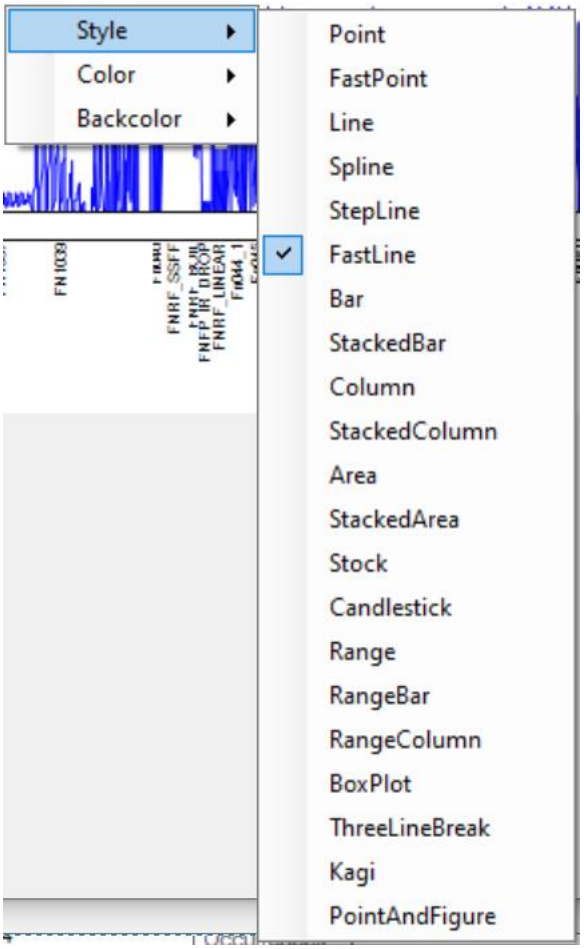


'Expand All' and 'Collapse All' options are easy to understand at this level.
'Filter on defined Elements' is a checkbox, which will reduce number of elements in list, on fact that they are user defined (something was update on them by someone, and saved in definition, it is not automatically generated). It permits to remove from list non interesting elements.
'Filter on Short Label Difference' is a checkbox, which will reduce number of elements in list, on fact that the 'Short Label' has to be different between current definition routine and its matching equivalent. It permits to remove from list, already copied elements.
'Filter on unique matching' is a checkbox, which will reduce number of elements in list, on fact that they have only one matching equivalent. It permits to remove from list, non-sure elements.
'Import secured elements only' option will do the same thing than 'Import Element' at element level, but here on the whole category. It will apply on all elements compatible with selected filters, but 'secured' means, that in all cases, it applies on defined elements and unique matching only, with or without these filters checked. In addition when something is not clear, like a different type or a different number of rows or columns or a register with different byte/word meaning, it is managed as unsecured and ignored too.
'Export report' button will show the save file dialog, to select an output file for the text report, which will contains the same thing than the shown result. It permits to easily switch between text report, disassembled text outputs and SAD 806x when updating definition and it permits to keep a trace too.
'Compare Binaries' is a great add on, to quickly identify elements between binaries and import their definitions, but do not try to go too fast.

- 'Calibration Chart View' ' option permits to compare 2 binaries (including the current disassembled one) visually on a 2D chart. It permits also to see only current binary and to visually identify its elements, which is possible for some advanced people. It will open its related form:

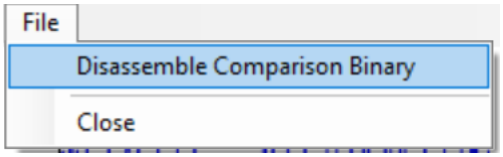


As you can see here, it is a basic 2D chart reflecting the hexadecimal values. It is locked to the real calibration addresses, related with RBases, as it should be. For sure you can zoom and unzoom, by using the mouse wheel.
Interesting thing here, is that elements present in disassembled binary are printed as legend. With mouse over a known element, you will see other details and by clicking on it, it will be shown in main application.
By right clicking on the chart, you can access to some options.

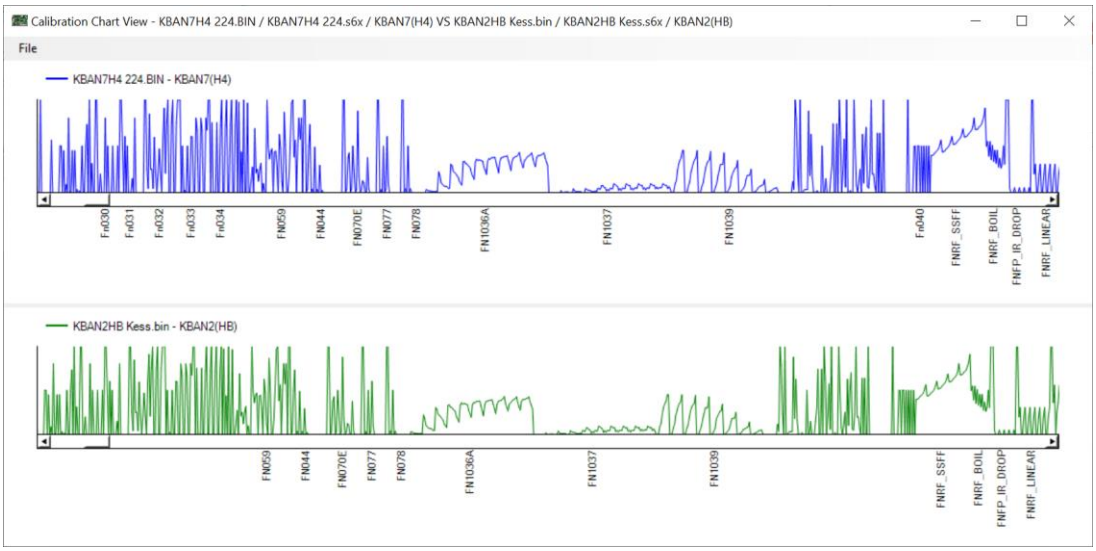


I will not detail them, because it is related with everything possible here and your own

habits, but you can change style (some styles are a bit slow), main color and back color. Now by using the menu, you have access to main things:

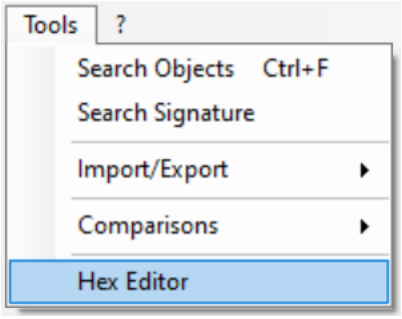


For sur you can close this form, with 'Close' option, but the interesting one is 'Disassemble Comparison Binary'. It will show the open file dialog, to select a binary file. By default, SAD 806x will show '.bin' files. Then it will load it with its default SAD 806x definition (.s6x, with the same name in the same folder), disassemble it in memory (which can take a bit time) and finally it will show the result.



Now you can compare both binaries and for sure you change style and colors too.

Tools Hex Editor menu:



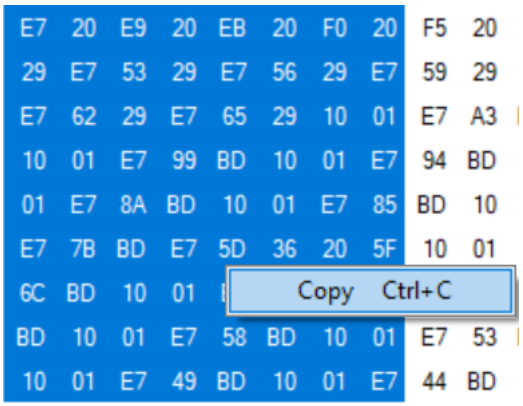
Hex Editor is more a hexadecimal viewer than anything else, because like other things with SAD 806x, nothing is done to modify the opened binary and you, like me, probably know excellent hexadecimal editors, which permit to really edit binaries. So it opens this form:

Hex Editor

Offset	Bank Offset	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Ascii
00000	0 2000	FF	FA	27	FE	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿú'þÿÿÿÿÿÿÿÿÿÿ
00010	0 2010	60	20	63	20	66	20	69	20	6C	20	6F	20	72	20	75	20	` c f i l o r u
00020	0 2020	78	20	7D	20	82	20	87	20	8C	20	91	20	96	20	9B	20	x }
00030	0 2030	A0	20	A5	20	A8	20	AA	20	AF	20	B4	20	B9	20	BE	20	¥ ¨ ª ¯ ´ ¹ ¾
00040	0 2040	C3	20	C8	20	CD	20	D2	20	D7	20	DC	20	E1	20	E3	20	Ã È Í Ò × Ü á ã
00050	0 2050	E5	20	E7	20	E9	20	EB	20	F0	20	F5	20	FA	20	FF	20	å ç é ë ð õ ú ÿ
00060	0 2060	E7	51	29	E7	53	29	E7	56	29	E7	59	29	E7	5C	29	E7	çQ)çS)çV)çY)ç\)ç
00070	0 2070	5F	29	E7	62	29	E7	65	29	10	01	E7	A3	BD	10	01	E7	_)çb)çe)ç£½ç
00080	0 2080	9E	BD	10	01	E7	99	BD	10	01	E7	94	BD	10	01	E7	8F	½ç½ç½ç
00090	0 2090	BD	10	01	E7	8A	BD	10	01	E7	85	BD	10	01	E7	80	BD	½ç½ç½ç½
000A0	0 20A0	10	01	E7	7B	BD	E7	5D	36	20	5F	10	01	E7	71	BD	10	ç{½ç]6 _çq½
000B0	0 20B0	01	E7	6C	BD	10	01	E7	67	BD	10	01	E7	62	BD	10	01	çl½çg½çb½
000C0	0 20C0	E7	5D	BD	10	01	E7	58	BD	10	01	E7	53	BD	10	01	E7	ç]½çX½çS½ç
000D0	0 20D0	4E	BD	10	01	E7	49	BD	10	01	E7	44	BD	10	01	E7	3F	N½çI½çD½ç?
000E0	0 20E0	BD	21	2D	20	73	20	8D	20	A7	20	C1	10	01	E7	30	BD	½!- s § Áçð½
000F0	0 20F0	10	01	E7	2B	BD	10	01	E7	26	BD	10	01	E7	21	BD	20	ç+½ç&½ç!½
00100	0 2100	C7	FF	FF	FF	3A	0A	06	F3	F1	F2	32	0A	F4	C9	EA	2B	Çÿÿÿ:óñò2ôÉê+
00110	0 2110	C4	4A	13	C0	48	06	98	4A	13	D7	F5	B0	4A	CF	48	0E	ÄJÀHJ×õ°JÏH
00120	0 2120	48	CB	48	CB	48	C9	B8	00	CF	B0	D0	D1	B0	0D	D0	94	HËHËHÉ¸Ï°ÐÑ°Ð
00130	0 2130	D0	D1	70	0C	D1	30	D1	03	EF	76	0B	37	D1	05	10	08	ÐÑp♠Ñ0Ñ♦ïv♂7Ñ♣♦
00140	0 2140	EF	6B	6B	36	D1	0C	EF	D4	0A	35	72	06	37	72	03	91	ïkk6Ñ♀ïÔ5r♠7r♥

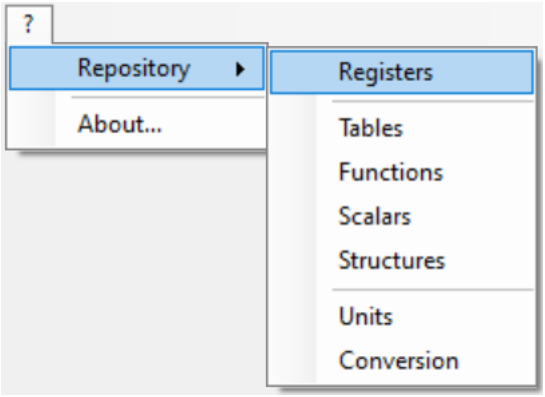
It is useful for some reasons, it permits to see hexadecimal code, for bad binaries and to understand what is wrong (often on the first 16 bytes), but it is also the only editor able to give a bank address. 'Offset' is the address inside the binary, 'Bank Offset' is the address inside the bank, beginning with the bank number itself.

Another interesting thing is the ability to copy hexadecimal code (for signatures or other things), by right clicking on selected part or using 'Ctrl-C' shortcut.



Just a required tool.

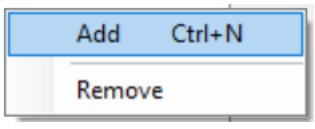
Help Repository menus:



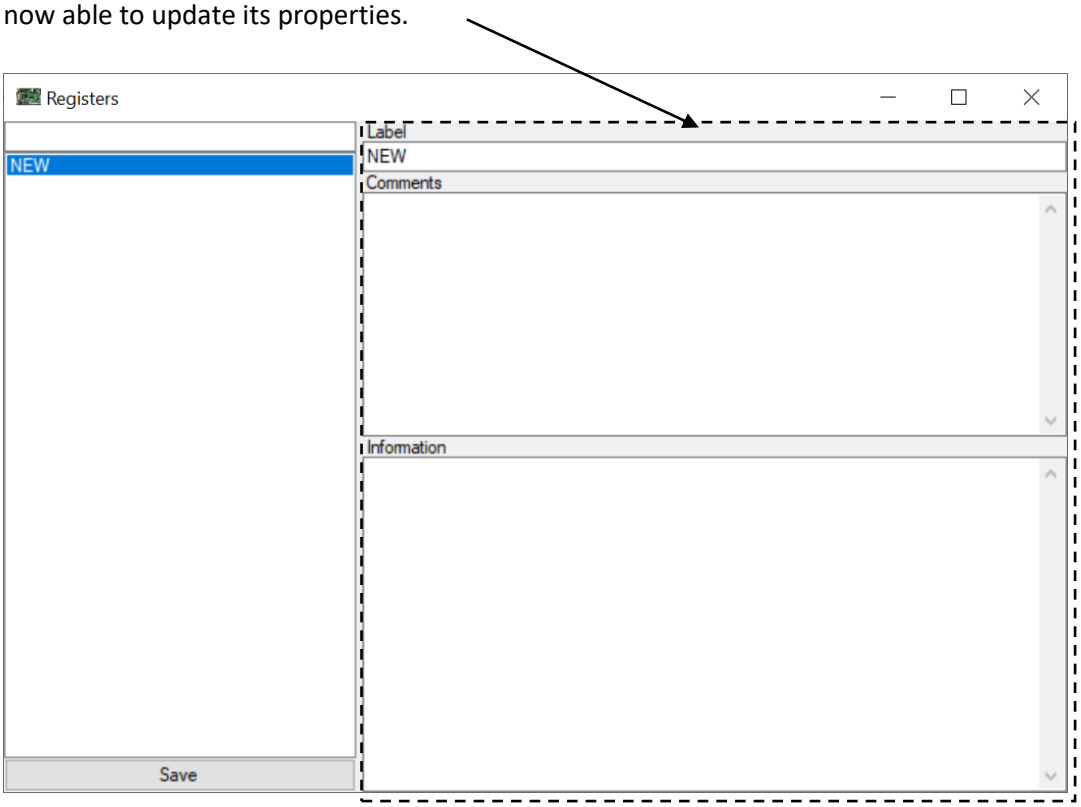
The 'Repository' menu permits to create or update the available repositories. Everywhere when creating definitions for elements, you can get information coming from the repository, based on where you are at this moment, through the 'Ctrl-R' shortcut, to enrich you definition. Repository is composed with xml files, in the SAD 806x folder. If they are not present, you can create them from here, globally all repositories are working in the same way, with small differences. By clicking on the related option, the right form will open.

- 'Registers' repository:
  'Repository search', to find an item, just use a word and press enter.

'Repository list', to show the list of items in repository.
This repository is for now empty, to add a new item, use 'Ctrl-N' shortcut or simply right click on 'Repository list', to display this menu:



As result, you will have a new item created and defaulted, here a register and you are now able to update its properties.

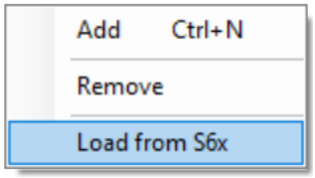


Registers in repository have only a 'Label', 'Comments' and 'Information'. As you have understood, when used from main application, this repository item will publish its 'Label' and 'Comments' on the register worked in application. 'Information' is only details inside repository. Name which appears in list is a syntetic version of interesting details on these properties.

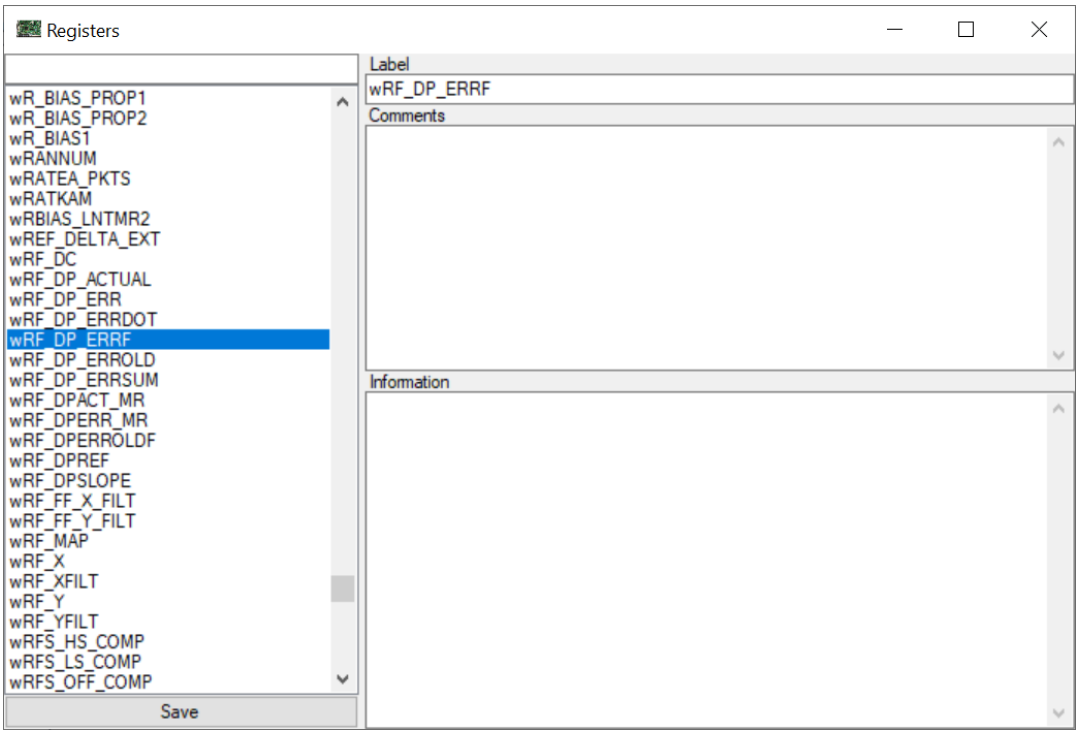
To save the new or updated item, simply use the 'Save' button.

As you have seen in small menu, you can remove an item with 'Remove' option.

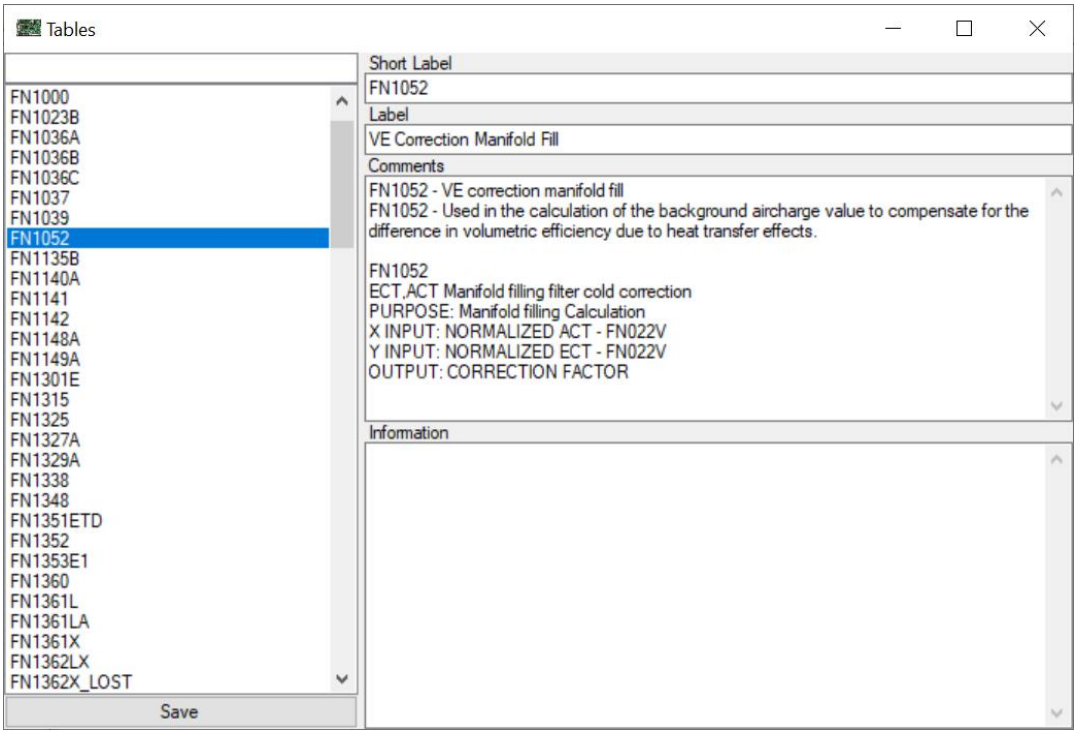
When a proper SAD 806x definition is loaded, a new option is available in this menu.



'Load from S6x' option permits to directly enrich repository, based on current SAD 806x definition. Do no forget to save after this.

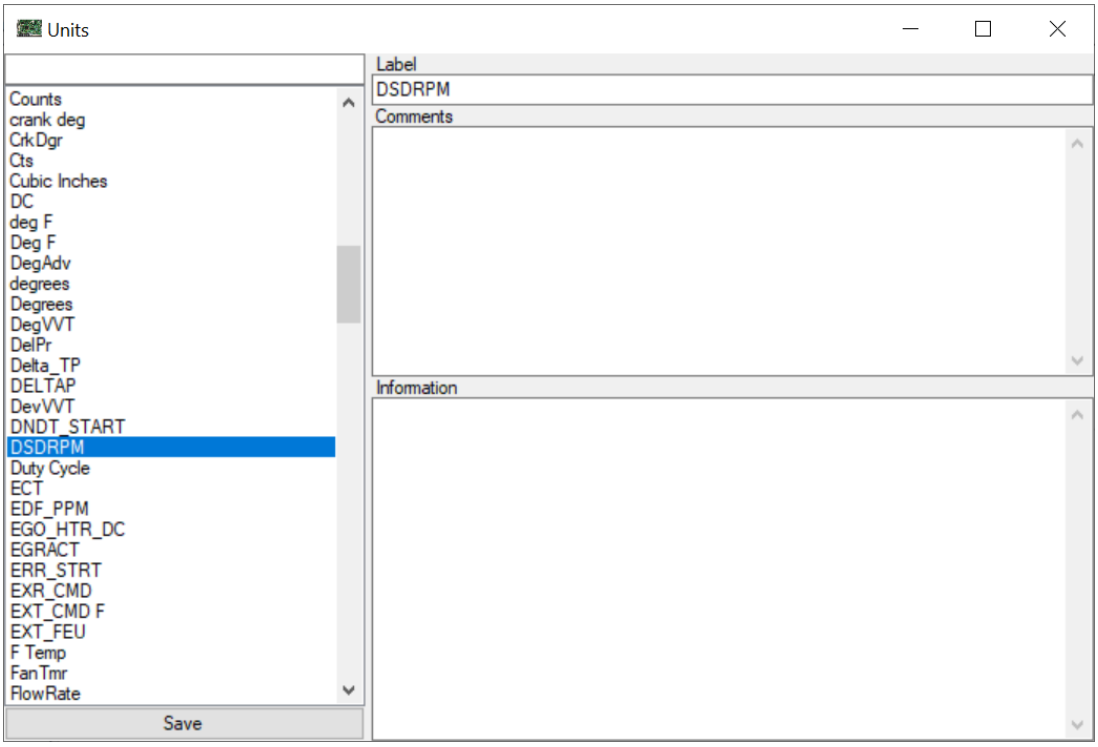


'Elements' repositories (for tables, functions, scalars and structures):



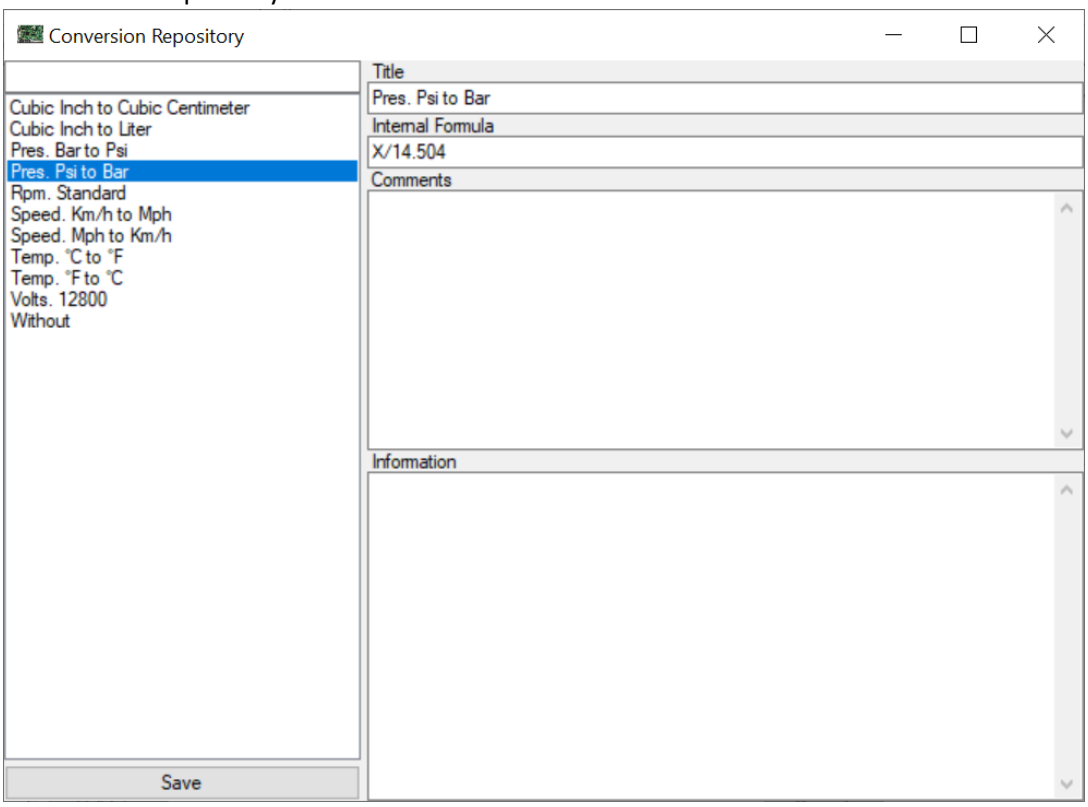
Exactly the same principle here, but with a 'Short Label' in addition, which will be used for filling 'Short Label' field on elements.
Same options are available, including, 'Load from S6x'.

- ‘Units’ repository:



‘Load from S6x’ is still available, ‘Label’ will be used for filling ‘Units’ fields on elements, ‘Comments’ is for repository only.

- ‘Conversion’ repository:



This one is a bit different, no ‘Load from S6x’ is available, for quality reasons.
It possesses a ‘Title’ as information and an ‘Internal Formula’ which will be used for filling

‘Scale’ fields on elements or to directly add an additional conversion level on diplayed data.

SAD 806x command line options:

Most important part of the work, which you will do with SAD 806x, will essentially be on definitions setup, but sometimes it can be useful to do mass disassembly for, for example, finding a strategy name, by having only the EEC catch code or its part number. A mass disassembly is also interesting for me to detect issues on some binaries, when testing a new version of this tool.

So yes, SAD 806x can do some things from command line, even if it stays really limited.

These are the syntaxes to be used:

```
C:\SAD806x>SAD806x.exe "C:\SAD806x\BIN\KBAN7H4.BIN"
```

It opens application with the related binary and its default SAD 806x definition, if it exists in the same folder, here it should be 'C:\SAD806x\BIN\KBAN7H4.s6x'.

```
C:\SAD806x>SAD806x.exe -D "C:\SAD806x\BIN\KBAN7H4.BIN"
```

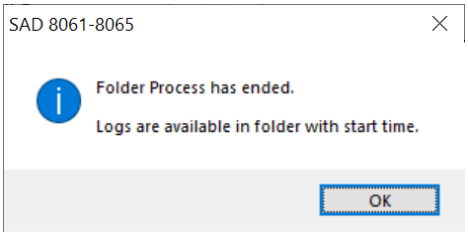
Same thing than previously, but it starts directly the disassembly, application can be used after this.

```
C:\SAD806x>SAD806x.exe -O "C:\SAD806x\BIN\KBAN7H4.BIN"
```

Same thing than previously, but it starts directly the disassembly and it does the text output, with the default text output path, in this case 'C:\SAD806x\BIN\KBAN7H4.txt'. Application can be used after this.

```
C:\SAD806x>SAD806x.exe -F "C:\SAD806x\BIN"
```

This one is the most interesting, because it works at folder level. All binary files (only .bin files) present in this folder (not in sub directories), will be disassembled (with their default definition if it exists) and text output will be generated in the same folder (with its default name).
The process can take some time, based on the number of binaries to be processed.
At the end of the process, this message will appear:



A log file will be available in this folder 'SAD 8061-8065.20XXYYZZ.AABBCC.txt', including details on what was really done.

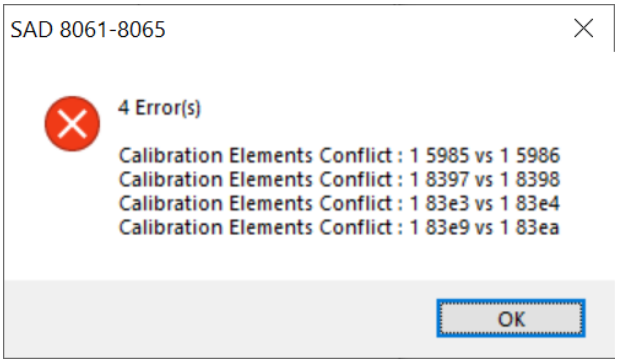
```
SAD 8061-8065 - Folder Process (*.bin files) on folder : C:\SAD806x\BIN
        10:01:56 - Starting.
Processing Binary file : C:\SAD806x\BIN\BADBIN.BIN
        10:01:56 - Starting.
        Binary file is invalid.
Processing Binary file : C:\SAD806x\BIN\KBAN7H4.BIN
        10:01:57 - Starting.
        10:01:57 - Loaded.
                Strategy KBAN7(H4)
                Part Number XS7V-12A650-AJ
        10:02:03 - Disassembled.
        10:02:05 - Output done.
```

Tips:

Disassembly/Output errors management:

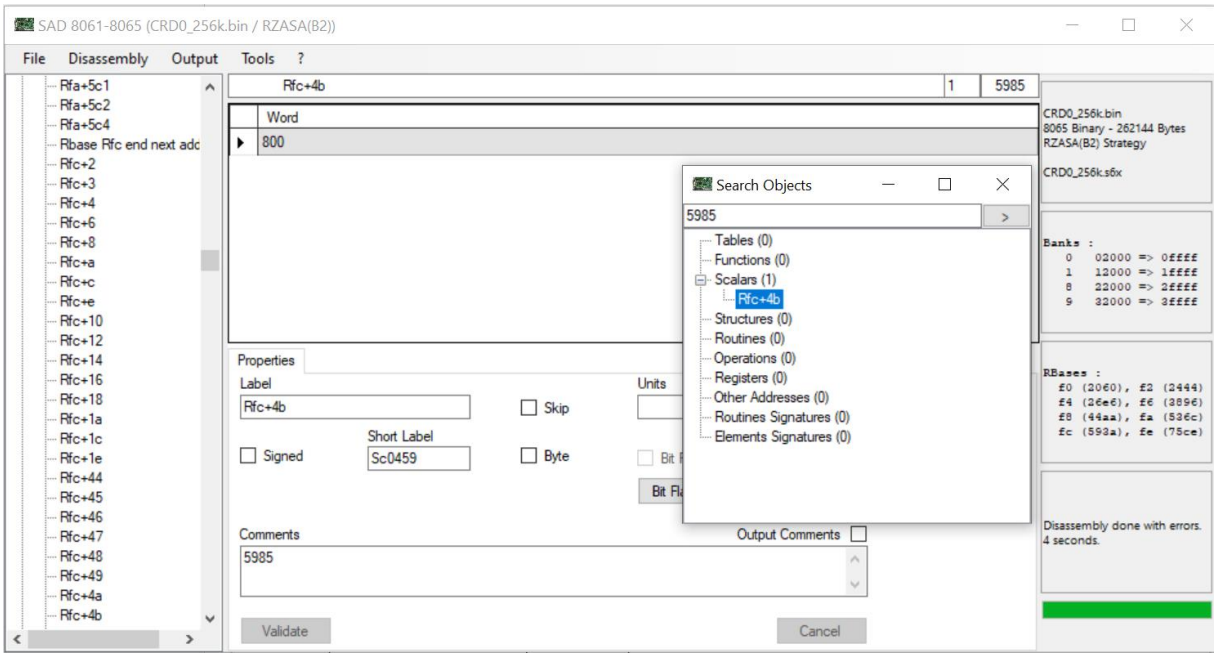
I will give you a good example, based on CRD0 catch code, with strategy RZASA.
For information, RZASA is one of the most complete and clean definition for EECV and is available thanks to Decipha (http://www.efidynotuning.com/).

I have started from scratch, without a definition, I have disassembled binary, seen following errors, done the text output without error, to analyze them:

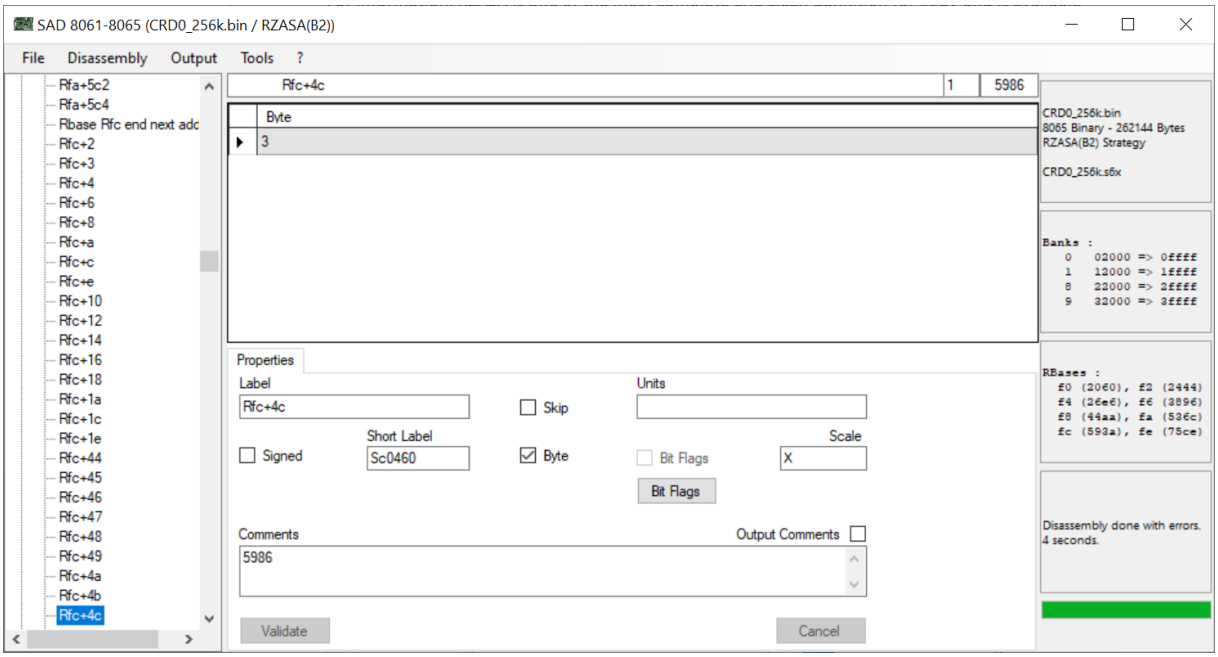


Errors were only at disassembly level, but the output is needed to analyze them, so I open it in parallel and for sure I keep SAD 806x opened.

I search for the first address.



Ok, I can see a word scalar and issue is on its second byte.



Ok, another byte scalar is defined at this place.

```
1 5985: 20,03             Rfc+4b Sc0459       word       320                 800
        Inc    :
        Inc    1 5986: 03      Rfc+4c Sc0460       byte         3                   3
```

Same thing in the output, second byte is managed as an included element (‘Inc’).

```
8 eeda: af,fc,4b,40         ldzbw R40,[Rfc+4b]       R40 = (uns)[Sc0459];
8 eede: 8b,e4,bc,40         cmpw  R40,[Re4+bc]
8 eee2: da,2f               jle   ef13               if ((sig) R40 <= [123c]) goto ef13;

8 f13f: a3,fc,4b,3c         ldw   R3c,[Rfc+4b]       R3c = [Sc0459];
8 f143: 8b,e4,bc,3c         cmpw  R3c,[Re4+bc]
8 f147: da,24               jle   f16d               if ((sig) R3c <= [123c]) goto f16d;

8 ec7d: 15,34               decb  R34                R34--;
8 ec7f: c7,e4,39,34         stb   [Re4+39],R34       [12b9] = R34;
8 ec83: 9b,fc,4c,34         cmpb  R34,[Rfc+4c]
8 ec87: d3,02               jnc   ec8b               if ((uns) R34 < [Sc0460]) goto ec8b;
```

Sc0459 is firstly used as byte, then as word, so yes it is a word scalar and Sc0460 is really used as byte.

As conclusion, Sc0460 can be ignored, but if you skip it, it will do nothing, because SAD 806x will still detect it. The best way to deal with it, is to set Sc0459 as byte scalar, thinking second use is an error in code or a trick to simplify code.

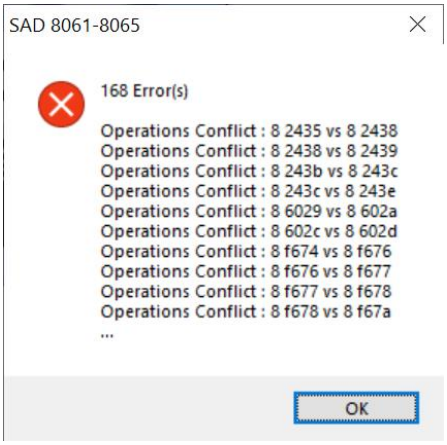
Decipha gives details on Sc0459 and set it as SLPRMPOPN (TCC Ramp Open Exit Slip RPM) which is defined as byte.

Like this issue is corrected.

This is the simplest example, it was not a real error, but yes, sometimes, calibration elements are used in a strange way, both word and byte. Sometimes functions are defined properly, but somewhere, a code part just want to read one of the output values, so it creates this type of message. SAD 806x can not understand that, so it has to be warned and analysed and corrected if necessary.

When errors are on operations, it is more interesting.

Same binary, started from scratch with no definition, a disassembly was done, a SAD directives file was imported, another disassembly and now these errors:

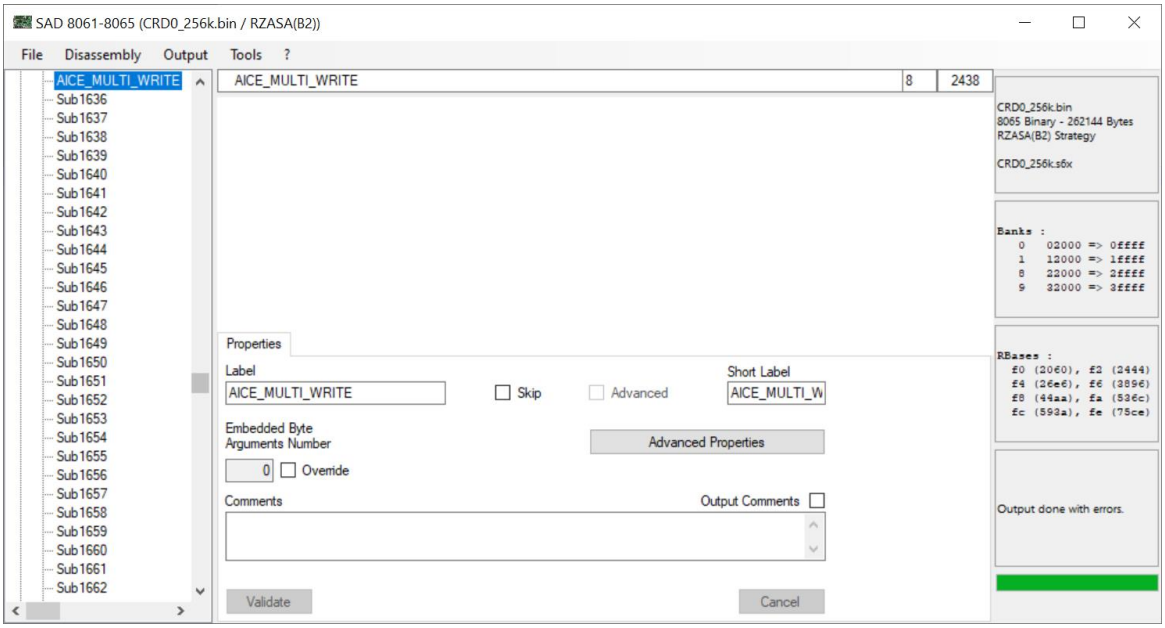


Yes, it is something, much more impressive and as you can see message shows only the first conflicts. No way in this case to obtain an output without errors:

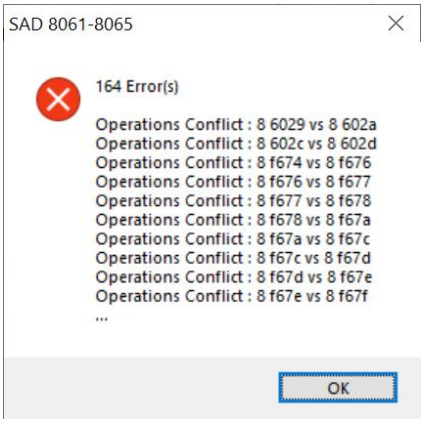
Let's start with first operations in conflict and from text output.

```
8 2413: f3                popp                pop(PSW);
8 2414: 45,c8,02,f0,46    ad3w  R46,Rf0,2c8   [tmp0l] = FN036M;
8 2419: a3,e6,fa,36       ldw   R36,[Re6+fa]  [tmp2l] = [maf_ptr];
8 241d: 88,36,46          cmpw  R46,R36
8 2420: d9,13             jgtu  2435          if ((uns) [tmp0l] > [tmp2l]) goto 2435;
8 2422: 45,78,00,46,34    ad3w  R34,R46,78    [tmp1l] = [tmp0l] + 78;
8 2427: 8b,e6,fa,34       cmpw  R34,[Re6+fa]
8 242b: d3,08             jnc   2435          if ((uns) [tmp1l] < [maf_ptr]) goto 2435;
8 242d: 94,46,36          xrb   R36,R46       [tmp2l] ^= [tmp0l];
8 2430: 71,03,36          an2b  R36,3         [tmp2l] &= 3;
8 2433: df,04             je    2439          if ([tmp2l] == 3) goto 2439;
8 2435: c3,e6,fa,46       stw   [Re6+fa],R46  [maf_ptr] = [tmp0l];
8 243f: c4,15,34          stb   R15,R34       LSSI_A = [tmp1l];
8 2442: c4,1d,38          stb   R1d,R38       LSSI_D = [tmp3l];
8 2445: c4,19,37          stb   R19,R37       LSSI_C = [tmp2h];
8 2448: c4,17,36          stb   R17,R36       LSSI_B = [tmp2l];
8 244b: 08,0d,00          shrw  0,d           0 = 0 / 2000;
8 244e: 94,35,34          xrb   R34,R35       [tmp1l] ^= [tmp1h];
8 2451: c4,15,34          stb   R15,R34       LSSI_A = [tmp1l];
8 2454: 08,09,00          shrw  0,9           0 = 0 / 200;
8 2457: 08,09,00          shrw  0,9           0 = 0 / 200;
8 245a: f3                popp                pop(PSW);
8 245b: f0                ret                 return;
```

I see nothing strange here, but I have not operation 8 2438 (or 2439, 243b, …). I can see 2 operations with a goto to 8 2435, so this one should be good and another one with a goto to 8 2439.
In definition, nothing (no operation, no routine) is defined at address 8 2435, but at 8 2438, yes, coming from directives import.



I skip it to test, no need to save SAD 806x definition, then another disassembly and another output.



```
8 2413: f3                popp                  pop(PSW);
8 2414: 45,c8,02,f0,46    ad3w  R46,Rf0,2c8     [tmp0l] = FN036M;
8 2419: a3,e6,fa,36       ldw   R36,[Re6+fa]    [tmp2l] = [maf_ptr];
8 241d: 88,36,46          cmpw  R46,R36
8 2420: d9,13             jgtu  2435            if ((uns) [tmp0l] > [tmp2l]) goto 2435;
8 2422: 45,78,00,46,34    ad3w  R34,R46,78      [tmp1l] = [tmp0l] + 78;
8 2427: 8b,e6,fa,34       cmpw  R34,[Re6+fa]
8 242b: d3,08             jnc   2435            if ((uns) [tmp1l] < [maf_ptr]) goto 2435;
8 242d: 94,46,36          xrb   R36,R46         [tmp2l] ^= [tmp0l];
8 2430: 71,03,36          an2b  R36,3           [tmp2l] &= 3;
8 2433: df,04             je    2439            if ([tmp2l] == 3) goto 2439;
8 2435: c3,e6,fa,46       stw   [Re6+fa],R46    [maf_ptr] = [tmp0l];
8 2439: f0                ret                   return;

Sub1621:
8 243a: f2                pushp                 push(PSW);
8 243b: 90,35,34          orrb  R34,R35         [tmp1l] |= [tmp1h];
8 243e: fa                di                    disable ints;
8 243f: c4,15,34          stb   R15,R34         LSSI_A = [tmp1l];
8 2442: c4,1d,38          stb   R1d,R38         LSSI_D = [tmp3l];
8 2445: c4,19,37          stb   R19,R37         LSSI_C = [tmp2h];
8 2448: c4,17,36          stb   R17,R36         LSSI_B = [tmp2l];
8 244b: 08,0d,00          shrw  0,d             0 = 0 / 2000;
8 244e: 94,35,34          xrb   R34,R35         [tmp1l] ^= [tmp1h];
8 2451: c4,15,34          stb   R15,R34         LSSI_A = [tmp1l];
8 2454: 08,09,00          shrw  0,9             0 = 0 / 200;
8 2457: 08,09,00          shrw  0,9             0 = 0 / 200;
8 245a: f3                popp                  pop(PSW);
8 245b: f0                ret                   return;
```

Ok, now it is fine for this part, routine set at 8 2438 was wrong, it can be removed.
164 errors remain to be corrected, but it should be the same thing.

You can now understand how to correct this type of issues, when they really are issues. I will not described all conflicts, which are normal and properly working, like operations with ‘fe’, used with or without it, you will have to analyze them yourself, SAD 806x gives just an information.

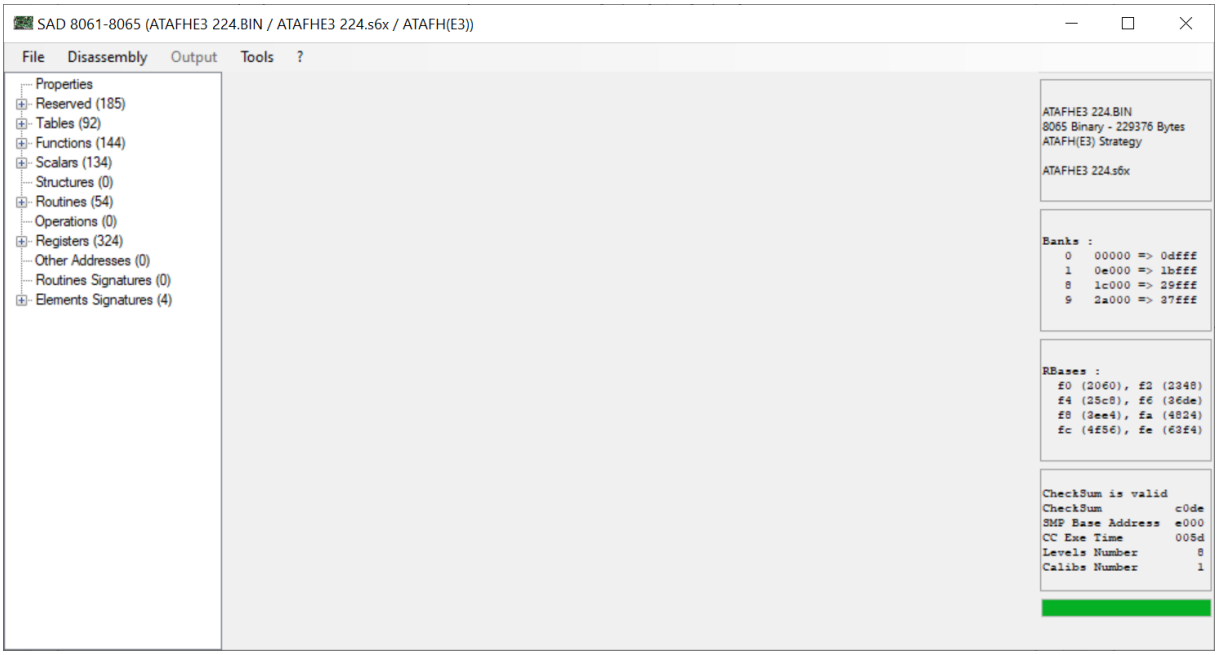
In a real conflict, one of the operation is wrongly defined, the first address given in message or the second one. Sometimes it is easy to find and to correct, sometimes wrong operation comes from an initial goto or call, which is not easy to find. The worst case, is related with a routine call, when arguments were not properly identified or counted, because arguments are now identified as operations and managed like this and because this call is done in many places.

By the way, you will always be able to correct these issues, by modifying SAD 806x definition.

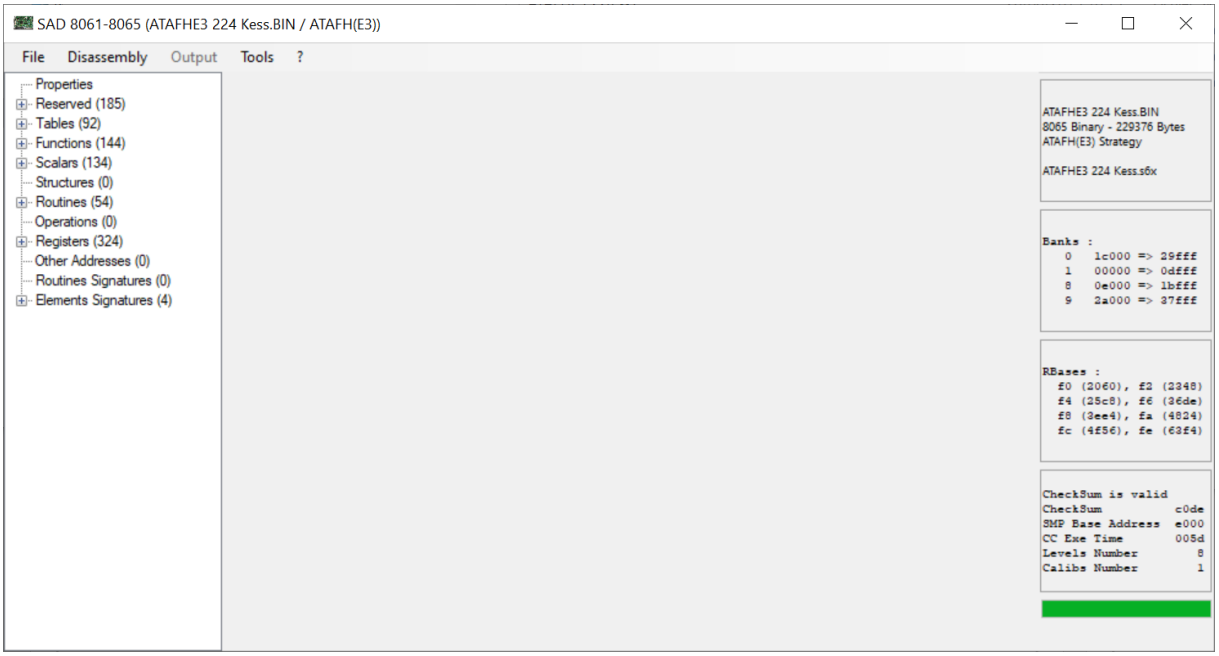
Banks Order and SAD 806x:

I write this chapter to describe, how SAD 806x works with banks and their order, following a ridiculous issue I had with an EEC, just because I had forgotten some details.

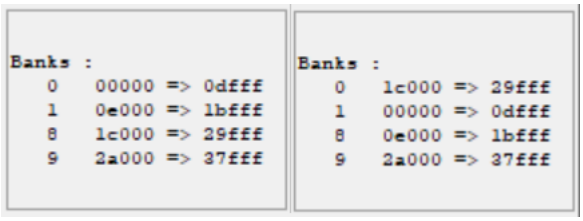
SAD 806x is not dependent from banks order in rom. Let's take the strategy which gives me some troubles, ATAFH. I was working since some time, on a binary coming from Ford IDS.



Before updating it, I have compared it with the one on the car, with SAD 806x and I have seen no difference at all, so I have updated it and I have sent it on the car and nothing was working. I have updated it with PATS, VID block coming from car, I have sent it another and another time, with the same result.



Because of the title of this chapter, you know what was the issue.



```
Banks :
   0    00000 => 0dfff
   1    0e000 => 1bfff
   8    1c000 => 29fff
   9    2a000 => 37fff
```

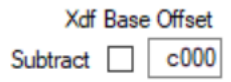
```
Banks :
   0    1c000 => 29fff
   1    00000 => 0dfff
   8    0e000 => 1bfff
   9    2a000 => 37fff
```

With the same definition, except here and in the header of the text output, it is impossible to know, that binaries are different. Yes, banks order is not the same and in SAD 806x, your definition works perfectly on both binaries, each element compared in 2 different SAD 806x sessions is identical.

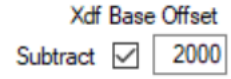
This is really practical, to have a definition, which is banks order free, but you should not forget this information.

TunerPro is not banks order free and if you export this definition for TunerPro, in one case it will not work, nothing will be exported, because of the Xdf base offset, which is defaulted by SAD 806x.

For the first binary you have by default:



For the second one:



If you want to use the same definition, with binaries having a different banks order, it works perfectly inside SAD 806x, but for using TunerPro, you will have to play with the Xdf base offset. Same thing, if you want to change the banks order in your binary, this is the only information to update in definition, to be totally compatible with TunerPro.

And for sure, before updating an EEC, be sure to use the right banks order.

Glossary:

EEC : Ford Electronic Engine Control is the Ford engine control unit. EEC-IV uses 60pin connector, 8061 processor and 1 bank, EEC-V uses 104pin (sometimes 60pin) connector, 8065 processor and 4 banks (not always activated). Both possesses a J3 connector in addition. Additional information will not be described here.

J3 connector : Ford J3 connector is a service connector for EEC-IV and EEC-V. It will not be described here.

Rom/Binary : Each EEC contains a specific Rom/Binary, stored in a Flash memory. It contains both instructions and calibration values.

Strategy : Each Binary is based on a Strategy, which is in fact position of instructions and calibration values in the binary. It is not the EEC Catch Code.

Strategy version : For the same strategy, where version is different, just calibration values are modified.

EEC Catch Code : The main information visible on the EEC. One Strategy version gives one Catch Code.

EEC Hardware Code : The hardware code for the EEC. One hardware code permits to use different strategies, but one strategy requires a specific hardware code.

Bank : Memory bank, in binary. 56ko maximum.

Instructions : Instructions provided to processor to process parameters.

Operations : Set of instructions and parameters.

Routines : Set of operations, virtually created to be able to better understand disassembled code.

Registers : A register is an EEC memory address, not related with rom, used to share data or information inside program. Globally on EECV addresses start at 0x0000 to go to 0x1FFF and another part can be used from 0xF000 to 0xFFFF.

Calibration Values : Calibration values can be split in 4 categories:

- Structures : a variable set of bytes, words based on conditions.
- Tables : a table of bytes or words with fixed size. 3 input values, columns number, column scaled value and row scaled value. 1 output value.
- Functions : a table with 2 columns. 1 input value. 1 output value.
- Scalars : a byte or a word value.

RBases : Most important part of EEC-IV and EEC-V use RBase shortcuts for defining the calibration element addresses. RBases are in fact dedicated registers, containing a base address inside the calibration rom part. By adding a value to it, it gives an element address, still in the calibration rom part. The

first calibration element at the address pointed by a RBase is the end address of its own part. RBases are mainly 8 word registers, which follow themselves.

RConst : Late EEC-IV and EEC-V use, what I call, RConst. It is working like RBases, but essentially for register addresses. They are still dedicated registers, containing a base value and by adding another value to them, it gives a register address.

Disassembly : It is the human understandable version of the instructions and their parameters, separated from the data, which are the calibration values.

Checksum : Rom contains a value, stored at a defined address, which permits to control validity of the whole rom, to prevent copy errors. When updating something in rom (except in certain parts), it is required to modify the checksum, to be sure related routine, will not generate an error code.

Files:

Rom/Binary files : 1 bank (8) for EEC-IV, 2 banks (8 and 1) minimum for EEC-V. (.bin, .hex …).

S6x files : SAD806x definition file (.s6x). Basically an xml file. Use one by strategy.

SAD 806x repository files : SAD806x repository files. Basically an xml file. 'registers.xml', 'structures.xml', 'tables.xml', 'functions.xml', 'scalars.xml', 'units.xml' or 'conversion.xml'.

SAD files : SAD disassembler directives files (.dir) and comment files (.cmt).

TunerPro files : TunerPro definition file (.xdf).