

All rights reserved. All parts of the documentation are covered by copyright. Without explicit consent in writing from the copyright owner, it is not allowed to copy the documentation or parts thereof in any form, by photocopy or any other process or means, or to distribute them.

Copyright © 1991-2015 Klaus Schultz Reverse- & und Software-Engineering,  
D 87437 Kempten, Germany

# Contents

<b>1. PREFACE</b>	<b>6</b>
<b>2. FIRST STEPS WITH REASM</b>	<b>7</b>
<b>2.1 The demo version</b>	<b>7</b>
<b>2.2 Installation and running</b>	<b>8</b>
2.2.1 Installing the demo version	8
2.2.2 Installing the production version	9
2.2.3 Uninstalling	9
2.2.4 Running the program	9
2.2.5 reASMgen	9
<b>2.3 Displaying the assembly source</b>	<b>9</b>
<b>2.4 Displaying control flow</b>	<b>10</b>
<b>3. OPERATING PRINCIPLES OF THE REASM USER INTERFACE</b>	<b>12</b>
<b>3.1 Menu</b>	<b>12</b>
<b>3.2 Function keys</b>	<b>12</b>
<b>3.3 Shortcut keys</b>	<b>12</b>
<b>4. THE CONTROL FLOW MENU</b>	<b>13</b>
<b>4.1 Displaying pseudocode</b>	<b>13</b>
<b>4.2 Cursor and select</b>	<b>13</b>
<b>4.3 Flow</b>	<b>14</b>
<b>4.4 Call hierarchy</b>	<b>14</b>
<b>5. THE DATA MENU</b>	<b>15</b>
<b>5.1 Display</b>	<b>15</b>
<b>5.2 Zoom</b>	<b>15</b>
<b>5.3 Select</b>	<b>15</b>
<b>5.4 Data / Input, Output, Usage</b>	<b>15</b>
5.4.1 Output	16
5.4.2 Input	16
5.4.3 Usage	16
5.4.4 Method of operation	16
<b>5.5 Data / Extended Output Backward</b>	<b>18</b>
<b>5.6 Extended Usage Forward</b>	<b>19</b>
<b>5.7 Parameter input with Extended Usage</b>	<b>21</b>

5.8 Data / Reg free	21
5.9 Data / Assignment Xref of...	22
5.10 Data / Condition Xref of...	22
5.11 Prompt window: Select Data by Name	23
5.12 Displaying the number of statements found	23
<b>6. THE OPTIONS MENU</b>	<b>25</b>
6.1 Options / Autoscroll	25
6.2 Options / Font	25
6.3 Options / forward & backward	25
6.4 Options / one-choice	25
6.5 Options / Text PLH- & ASM-like	27
<b>7. THE FILE MENU</b>	<b>28</b>
7.1 Processing an assembly program	28
7.2 File / Source View	29
7.3 File / Open and Save Work	29
7.4 File / Print	29
7.5 File / MergePrint	29
7.6 File / Stop Task	29
<b>8. THE NAVIGATE MENU</b>	<b>30</b>
8.1 Navigate / GOTO Label...	30
8.2 Navigate / GOTO Line...	30
8.3 Navigate / End-Results & User Defined Results	30
8.4 Copy / Merge User Defined Results	32
8.5 Navigate / Clear Selections	32
8.6 Navigate / Keep List	32
8.7 Navigate / Choice points	33
8.8 Navigate / Exclude lines	33
<b>9. CONCEPTS</b>	<b>35</b>
9.1 Computed branch targets	35

9.2 Procedures	36
9.3 Explicitly-addressed data	37
9.4 Redefined and variable-length data	37
9.5 Dynamic code modifications	38
9.6 SECTIONs	38
9.7 USING	38
9.8 EQU	38
<b>10. DESCRIPTION OF SUPPLIED AND GENERATED FILES</b>	<b>39</b>
10.1 REASM.ENV: ENVIRONMENT file	39
10.2 REASM.INI	39
10.3 XYZ.IDB and XYZ.IDC: Internal database	39
10.4 XYZ.PRO: Company and program profiles	39
10.4.1 Options for reASM	40
10.4.2 Options for reASMgen	40
10.5 STMACRO.ARI and MACRO.ARI: Macro definitions	41
10.6 XYZ.ASM, XYZ.LST: Assembler source, assembler listing	42
10.7 XYZ.REF: Reference file	42
10.8 XYZ.LOG: Log file	43
10.9 XYZ.DBR: Import database record	44
10.10 REPORT.ARI	45
10.11 REASM.HLP	45
10.12 XYZ.LRE: List file	45
10.13 XYZ.RLH: Restruct file	45
10.14 XYZ.TMP: Temporary file	45
10.15 PROLOG files notation (*.ARI , *.PRO and *.REF)	46
<b>11. APPENDIX</b>	<b>47</b>
11.1 Storage requirements and performance	47
11.2 Assembly-language subset	47
11.3 Error messages	51
<b>12. OPTIONAL RESTRUCTURING MODULE</b>	<b>53</b>

<b>13. USER INTERFACE EXTENSIONS FOR REASMGEN</b>	<b>56</b>
13.1 Data declaration hierarchy	56
13.2 Generate statement(s)	56
13.3 Restructuring	57
<b>14. INDEX</b>	<b>58</b>

# 1. Preface

*For what programmers has this tool been designed?*

It is **not** meant for those programmers who know “their” programs like the back of their hand, or those who believe they see *at a glance* what to do. It is meant for those programmers who want to work systematically, who take some time to analyse the assembler code in order to see what to do.

Also, it is a help for those programmers who have little experience with assembly language but want to extract information from programs.

*For what programs has the tool been designed?*

It has not been made for programs that use the special features of assembly language, such as those that operate at near-machine level, or SAP programs. It covers an assembly language subset that is usual in commercial programming – much like what COBOL uses (for a more accurate description refer to the Appendix).

It has been designed to support **common tasks**. Special problems are still covered best by human experts. With **reASM** they will regain time to handle these special problems.

The **reASM** product offers maintenance support: program flow and data flow analysis.

In addition to this, there is a program called **reASMgen**, which analyses the assembly program to such a depth that a semi-automatic conversion to another language (PL/I or COBOL) will be possible.

The demo version, which is available on floppy disk, contains the complete program, but without the possibility to load assembly programs. Instead of this, the program’s database file can be restored, so that you can try out all features.

## 2. First steps with reASM

### 2.1 The demo version

Put the floppy disk into the drive and run from there “INSTALL C:” (or “INSTALL D:” if you want to install on the D partition). All files will then be copied to directory C:\REASM\, and data files to directory C:\REASM\DATA.

Verify that the DLL files that are supplied are also installed. E.g. under OS/2 be sure that the current directory is included in your LIBPATH and you start the program from directory REASM.

Switch to directory REASM. Before the program itself is launched, we'll have a look at file UMSATZ.LOG, which resides in subdirectory DATA. UMSATZ.LOG is the **processing log** for the program UMSATZ.ASM. The branch nesting level is 4, with 10 branch labels. At one point, a NOP instruction will be changed at runtime. Two places in the program contain “dead code”.

These numbers are also contained in database record UMSATZ.DBR, with which a history database of the assembly program quality can be built up.

Now switch back to directory REASM and launch the program by entering “REASM”.

#### Loading the example:

With “Open” the file selection window is opened. In directory DATA, select file UMSATZ.IDB. This is the database file of the sample program UMSATZ ; it is supplied with [reASM](#).

The assembly source code which appears on the screen, can only be read, not edited. The *cursor* is a reversed-video bar, which can be positioned with a mouse click. Operation is object-oriented: when the cursor is in the left part of the line, e.g. on the “MVC” of an MVC instruction, a command will apply to the entire instruction; when the cursor is more to the right, on an operand name, the command will apply to this operand.

**Display:** With a mouse click, position the cursor at the “BE” in line 15 and type CTRL + D (*display*) on the keyboard: you will see the pseudocode of the statement. However, if you position the cursor on an operand, e.g. on “MASKE” in line 35, and then type CTRL + D, the local and global definition of the operand will be displayed.

**Data flow:** Place the cursor on a data item, e.g. on field ZWISUMK in line 36, and type CTRL + S (*select*). This selects a data item (green background) with which you can start a data flow search: where has ZWISUMK been set? To answer this question, type CTRL + O (*output*). After the search, you will find in the top-right corner in “end” that two points have been found. You can find these points in two ways: by browsing (the points found are displayed in red), or via the menu under “Navigate / End results”. In this case the points found are:

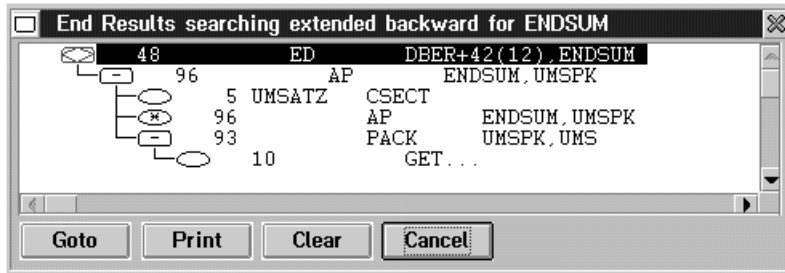
1. line 8        ZAP        ZWISUMK , =P ' 0 '
2. line 95      ZAP        ZWISUMK , UMSPK

The path searched to the results is displayed on a yellow background. When you find this result manually, you will see that [reASM](#) has gone back to label EOF1, from there to the GET instruction, from there back to label SCHLEIFE. From this point, the path branches: one branch to the start of the program, the other one in the direction to the start of the branch in line 32, and from there back into procedure RUMPF.

In the menu item “Options” you can set the direction in which the data flow is to be analysed. Default is backward. That is the usual direction when searching for output: where has a data item most recently been set? When searching for Input (CTRL + I = *input*), the forward direction is usually chosen: where has the field been used as input to a statement?

An example of input: Switch to forward search by clicking on menu item “backward”. The menu item text now changes to “forward”. Position the cursor at line 6, e.g. by means of the menu item “Navigate / GOTO Line...”, and select register 3 with CTRL + S in line 6 (this removes previous selections). Type CTRL + I. **reASM** searches forward, enters into procedures where necessary and finds exactly two statements: 154 and 115. So you can trace how a variable’s value propagates.

Backward data flow analysis is also available in a multi-level form. In line 48, variable ENDSUM is placed into a print string. But how did ENDSUM get its value? Select ENDSUM and select menu item “Data / Extended Output backward”. You will be asked how many levels deep you want to search. Enter e.g. “3”, The result is:



The first level finds line 96 as the source for ENDSUM. As this is an addition, the second level will not only search for ENDSUM, but also for UMSPK. The second level results in three sources: again line 96 (the \* indicates that it is a in loop), line 5 (the start of the program and consequently variable initialisation with DC) and line 93. The third level will search from line 93 backward for variable UMS, which was read in line 10 with a GET command.

Data flow analysis is evidently more powerful than the well-known cross-reference lists. A cross-reference is a survey of the use in the entire program. In menu item “Data” you find a classified cross-reference: the **Assignment Xref** shows all places where a variable is assigned a value, and the **Condition Xref** shows all places where a variable is used in a condition.

**Procedure hierarchy:** under menu item “Control flow / Call hierarchy” you will get the procedure call hierarchy displayed.

You can see the code of these procedures parallel in the source view window when you switch on the “Autoscroll” for the source view window in menu item Options. This will automatically position the source view window accordingly whenever you position the cursor in the call hierarchy window.

## 2.2 Installation and running

### 2.2.1 Installing the demo version

In case a version of **reASM** has already been installed, please save the files that have been modified in a client-specific way (REASM.PRO, MACRO.ARI).

Put the **reASM** floppy disk into the drive and run the installation command file INSTALL, which is on the floppy disk, with parameter <drive letter><colon>: “INSTALL C:” (or “INSTALL D:” in case you want to install on partition D). All files will then be uncompressed and copied into directory C:\REASM. A subdirectory C:\REASM\DATA is created into which the generated files are written.

OS/2: Copy the DLL files (e.g., ARITY32.DLL) into a directory which is specified in the LIBPATH. When the current directory is included in the actual LIBPATH, you may store the DLL files in the REASM directory.



WIN95, WIN NT: You may copy the DLL files into a directory specified in the PATH, or you can run the programs from the REASM directory.

### 2.2.2 Installing the production version

The production version is protected with a hardware key (dongle). While running, the programs REASM.EXE and REASMB.EXE check whether the correct dongle is connected at the parallel printer port. The dongle is transparent to other dongles and to the printer.

Installing the production version requires the same steps as installing the demo version.

OS/2: Additionally, you must copy the dongle driver HASPOS2.SYS in an appropriate directory and insert the following line in file CONFIG.SYS:

```
DEVICE=<pathname>\HASPOS2.SYS
```

WIN95: Additionally, you must install the dongle driver by entering:

```
HINSTALL -i
```

Entering “HINSTAL -r” uninstalls the driver.

WIN NT: Additionally, you must install the dongle driver by logging in as administrator and entering:

```
HINSTALL -i
```

Entering “HINSTAL -r” uninstalls the driver.

### 2.2.3 Uninstalling

Neither under OS/2 nor under WIN95 / WIN NT are registry entries etc. made. To uninstall, just delete the REASM directory, delete the INI file in the operating system directory, and uninstall the dongle driver.

### 2.2.4 Running the program

The command for batch-processing: “REASMB <file name>”

The assembly program <file name> is processed and stored. Then REASMB returns control to the operating system. This way, several programs can be processed over night by means of a command file; they can be inspected interactively afterwards.

The command for interactive operation: “REASM”

You can interactively select for display an assembly program that has previously been processed by REASMB.

Because of a limitation of the class library used, only one instance of REASM can be active at any one moment.

### 2.2.5 reASMgen

When **reASMgen** is used, these explanations are valid, too. Just replace “REASM” with “REASMGEB” and “REASMB” with “REASMGEB”.

## 2.3 Displaying the assembly source

Generally, assembly statements are displayed the same way they appear in source code listings, except for the following differences:

- Operator and operand are placed in one column.

- Line continuation is displayed in the same line (logical line).
- When data or branch targets have not been given names, they are given a name FILLERxyz, which, in order to show the difference with names in the original source, are in lower case.
- If an EQU is used as a data item, the EQU is redeclared as a DS data item, which effectively is a redefinition.
- With STM and LM, both register operands are combined into one variable, or separated in statements assigning the separate registers.
- Certain instructions (BCT, BXH, BXLE, etc., sometimes LM and STM too) are separated in single actions. This is shown on screen by "...". E.g. the instruction

```
LABEL      BCT REG4, SCHLEIFE
```

is separated in:

```
LABEL      BCT . . .
           BCT      REG4, SCHLEIFE
```

When the line RECH MVC A, H is a procedure header, it is separated into:

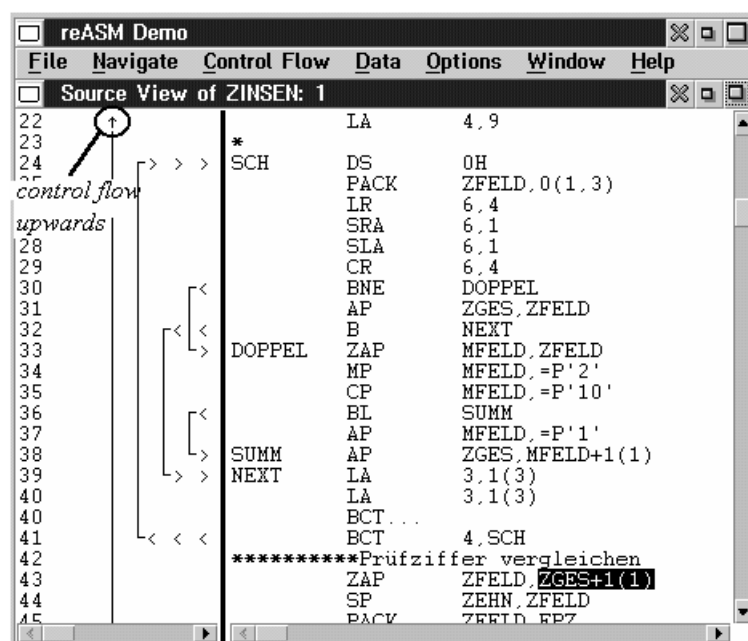
```
RECH      MVC . . .
           MVC . . .
           MVC  A, H
```

With CTRL + D (display) you will see that different interpretations are attached to different lines.

Simulated code modifications are also inserted as additional statements and indicated with "...".

## 2.4 Displaying control flow

At the left side of the code, the program's control flow is displayed by means of semi-graphical arrows. These arrows are nested in such a way that longer-distance branches are shown outside and shorter-distance branches inside. Up to a nesting level of 18, a one-blank distance is kept between the arrows; at deeper nesting levels, the arrows are placed closer together.



When you click on an arrow, you are shown the jump target. By double-clicking on the arrow, both panes are positioned at the jump target.

With the mouse you can freely move the split bar between the control flow pane and the source view pane of a source view window. Hint: when you move the split bar very far to the right and then release the mouse button, the bar will jump back to the position corresponding with the maximum nesting level of the arrows.

## 3. Operating principles of the reASM user interface

### 3.1 Menu

The menu is activated with the F10 key, or with the combination ALT + key (where “key” is the letter marked in the menu).

The options in the menu “Options” are selected or toggled with the ENTER key or with a mouse click. Active options are shown with a tick or by a changed text.

### 3.2 Function keys

F1:	Context-sensitive help, especially in a dialogue window
ALT + F1:	Switching help balloon in the menu on or off
ALT + F4:	Closing the program
Page Down:	One page down
Page Up:	One page up
Arrow Down:	One line down
Arrow Up:	One line up
F10:	Menu

### 3.3 Shortcut keys

The major functions are not only available via the menu, but also directly through key strokes:

CTRL + S: Select	(operates on cursor position)
CTRL + K: Keep	(operates on cursor position)
CTRL + D: Display	(operates on cursor position)
CTRL + F: Flow for-/backward	(operates on selected statement)
CTRL + I: Input for-/backward	(operates on selected statement + data item)
CTRL + O: Output for-/backward	(operates on selected statement + data item)
CTRL + U: Usage for-/backward	(operates on selected statement + data item)

Additionally, with [reASMgen](#):

CTRL + G: Generate statement	(operates on cursor position)
------------------------------	-------------------------------

## 4. The control flow menu

Under this menu item you find everything relating to a single statement and the control flow.

### 4.1 Displaying pseudocode

As far as it can be generated in a meaningful way, pseudocode text is displayed for a statement. This pseudocode mimics PL/I and REXX.

**Displaying a condition:** at a branch instruction, branch target *and* condition are displayed. The condition is composed from the branch instruction and the preceding compare instruction. The data flow analysis takes into account the use of data in a condition in the branch instruction (not in the compare instruction).

Example: the lines

```
CR      REG6,REG4
BNE     DOPPEL
```

are displayed:

```
if REG6 =\= REG4 then goto DOPPEL
```

Displaying an assignment:

```
FELD1 := FELD1 + FELD2
```

In pseudocode, a **variable** is primarily displayed as a name, even when internally more detailed information is available. (For example, DBERR+44(3) will just be displayed as DBERR+44.) A data item explicitly addressed through REG6 is displayed as →(REG6). Register 5 is displayed as REG5. If only parts of a register are accessed, then offset and length in bytes are appended to the name. Example: In the instruction

```
STCM    5,B'0111',ADR
```

only the three right-most bytes are filled, the offset is 1. The register is displayed as REG5\_1\_3.

Instructions that cannot be interpreted (such as unknown macros) are displayed as “unknown”.

Instructions that do “nothing” are displayed as “none”.

For instructions of the type **USING**, instead of pseudocode the actual USING constellation is shown, i.e., taking account of previous PUSHs, POPs, DROPs, etc. (For further explanations refer to page 38).

### 4.2 Cursor and select

**reASM** does not have a cursor like a word processor. A data item or an instruction is clicked with the mouse. That positions the cursor at the corresponding syntactical element (displayed in reversed-video).

The Display and Keep functions only operate on the cursor position, just like the Select function, which selects the statement or data item (green representation). A statement or a data item is selected in order to use it as a starting point for a control flow or data flow analysis.

The list of *end results* is cleared at each select command. If you want to save the results, you have to copy them to a “user-defined result” *before* the next Select command.

## 4.3 Flow

The “Flow” function displays the control flow. Starting from the selected statement, **reASM** searches forward or backward (depending on the option Forward / Backward) until the control flow branches. With the one-choice option switched off, the window is positioned on the branch statement and the first statements of the branches are marked red, i.e., they are copied into the list of “end results”.

Example: In the UMSATZ program, select line 41 and do a backward search. **reASM** traces control flow backward to the label EOF1 – however, that is not a branch, but only one path goes on to the GET statement, and from there to the SCHLEIFE label (line 10). At that point, the control flow branches. So the window is positioned at line 10, the “end results” are lines 9 and 32. In the “End Results” window, which can be reached via menu item “Navigate”, you will see this result, too.

Another example: In the UMSATZ program, select line 153: ST 14, KUNDE-4 and do a backward search. The control flow branches at the procedure header, as there are several (more exactly, three) points from where procedure KUNDE is called. The calls to procedure KUNDE are coloured red as a result.

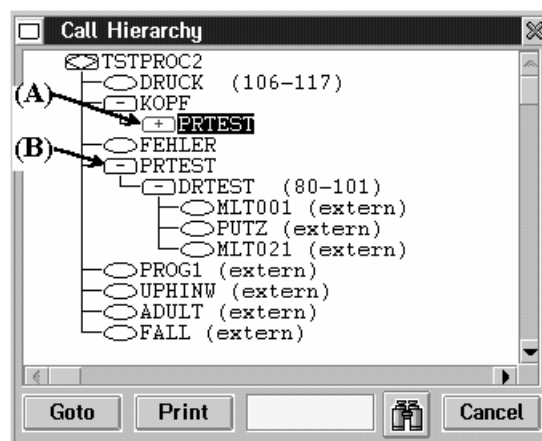
With the one-choice option switched on, you are prompted to select one of the program branches. The method using the one-choice option is described in the chapter on page 25.

## 4.4 Call hierarchy

The procedure call hierarchy is displayed as a tree.

After every procedure name, the line numbers of the start and the end of the procedure are given in parentheses.

When a procedure is called from several points, it will be resolved in-depth at only one point in the tree.



*If you want to see the resolution of PRTEST at (A) with a second mouse click, the cursor jumps to (B), where PRTEST is also called and where the call is resolved in-depth.*

Corresponding **macro declarations** (cf. page 41) ensure that such macros, that in fact represent calls, are included in the tree. The above sample program TSTPROC2 contains e.g. line

```
DFHPC TYPE=LINK, PROGRAM=ADULT
```

Without the declaration of the DFHPC macro in file **STMACRO.ARI**, the call to ADULT would not have been recorded in the call hierarchy.

The option **Autoscroll** allows the source view window to be synchronised with the cursor movements in the call hierarchy.

## 5. The data menu

Under this menu item you will find:

- the display of data declarations extracted from the instruction;
- several data flow retrievals – these start from a selected statement and search the next use;
- special cross-references – these take a data name and produce program-wide references.

### 5.1 Display

When the cursor is placed on a statement's operand, the command CTRL + D will display the data attributes. Displayed are:

- the local name and the local length;
- the variable's global declaration, if necessary with name of the DSECT in which the variable is declared;
- comment in the declaration line, if any.

An external variable (a variable declared with EXTRN in assembler, or an unknown variable that reASM has inserted as EXTRN afterwards) is given the fictitious length of 999 bytes. The purpose of this fictitious length is that all retrievals based on the knowledge of a length can still be processed.

### 5.2 Zoom

When the cursor is placed on the operand of a statement, the key combination CTRL + '+' (on the numeric key pad) opens another window which is positioned on the data declaration.

### 5.3 Select

The variable on which the cursor is placed, is selected. It is displayed in green.

A variable is selected in order that e.g. a data flow starting from it be displayed.

### 5.4 Data / Input, Output, Usage

These functions allow the data flow to be analysed. When e.g. in statement 93 of program UMSATZ the field UMS is used and you wish to know where UMS has previously been assigned a value, you position the cursor on UMS with a mouse click, select UMS by typing CTRL + S (it will be displayed in green) and then type CTRL + O. The data flow analysis goes back along all *control flow paths* that lead to statement 93. This way, it will find all potential data sources for UMS. Normally, the result will consist of several instructions; in this example it is only the GET statement in line 10.

The direction of the data flow analysis is set under menu item “Options Forward / Backward”.

The meaning of Input / Output / Usage has been chosen in such a way as to be as close to the intuitive meaning as possible.

### 5.4.1 Output

The memory location (or field, or register) searched for is set (normally direction “backward” is selected).

Example: You select the ZWISUMK field in a statement and type CTRL + O. The search would stop at a statement like

```
ZAP    ZWISUMK , =P ' 0 '
```

or

```
AP     ZWISUMK , =P ' 1 '
```

because the statement sets ZWISUMK, its “output” is ZWISUMK.

When the search reaches the start of the program and the field has not been initialised (either by a DC statement or by a profile entry), a message “... not initialised” is produced. If the field is initialised, the *start of the program* is recorded in the “end results” list (not the DC statement). E.g., in the UMSATZ program, the program start is the UMSATZ label.

### 5.4.2 Input

The memory location (or field, or register) searched for is used in a calculation or in a condition (normally direction “forward” is selected).

Example: You select the ZWISUMK field in a statement and type CTRL + I. The search would stop at a statement like

```
ED     DBER , ZWISUMK
```

or

```
AP     ZWISUMK , =P ' 1 '
```

The branch instruction takes account of the use of data in a condition. Example: a backward input search for REG4 stops first at the BNE statement:

```
CR     REG6 , REG4  
BNE    DOPPEL
```

### 5.4.3 Usage

The memory location (or field, or register) searched for is used as input or output in the statement or is used in it in another way, e.g. as a pointer. Using FIELD in a length expression like L ' FIELD is not considered a usage.

### 5.4.4 Method of operation

When you request a data flow analysis with respect to a certain statement for a variable which is not used in that statement (so it cannot be selected with the cursor), then just select the statement and choose a data flow function (e.g. by means of CTRL + O). Then a window “Select data by name” appears (cf. page 23), in which you can choose a register or enter a field name.

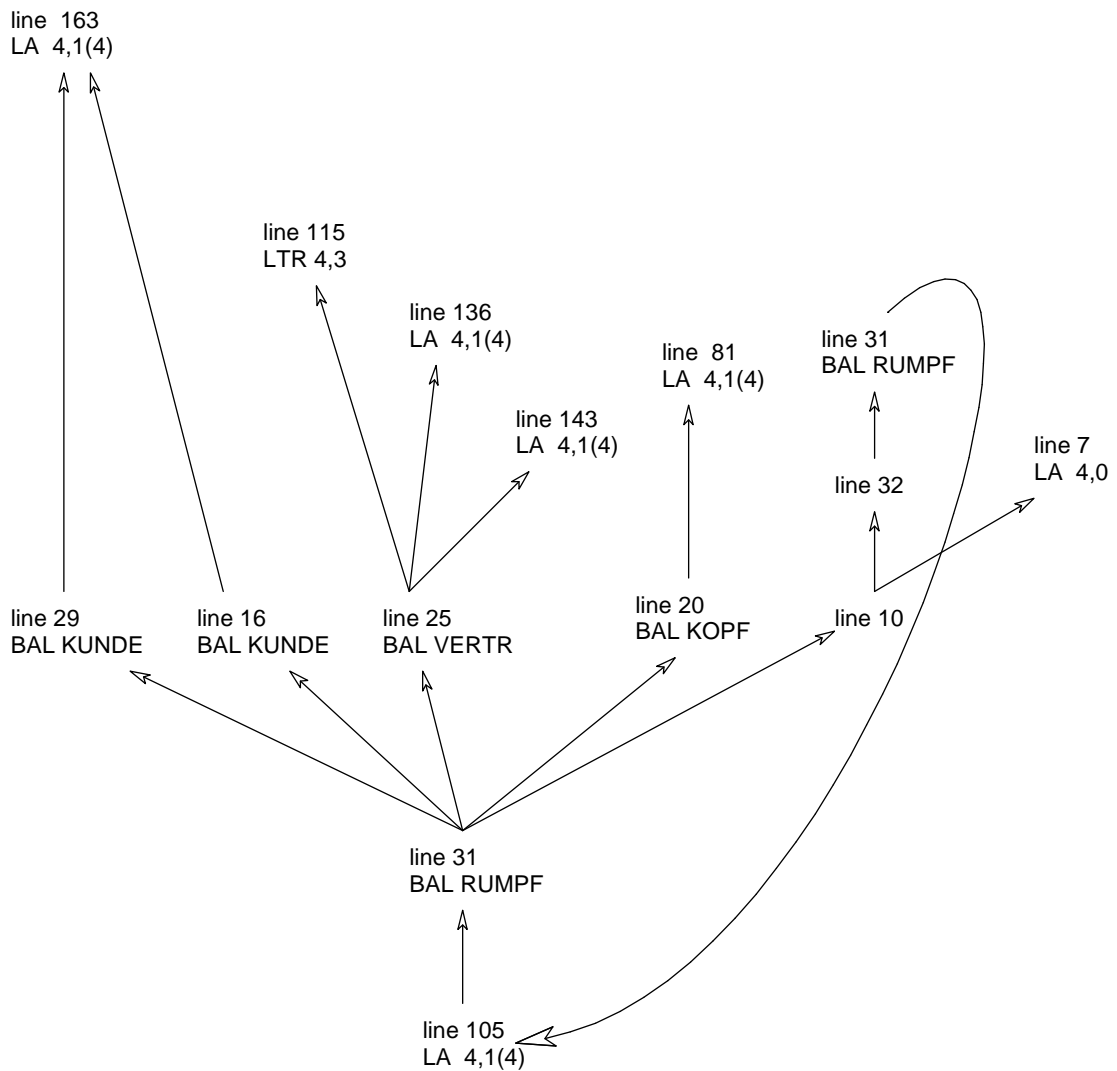
The data flow analysis takes into account redefinitions by not using the variable’s *name*, but its *memory location* and *length*.

When you select the variable to be searched, the search will be executed corresponding to the local length of the variable, e.g. with length 18 in

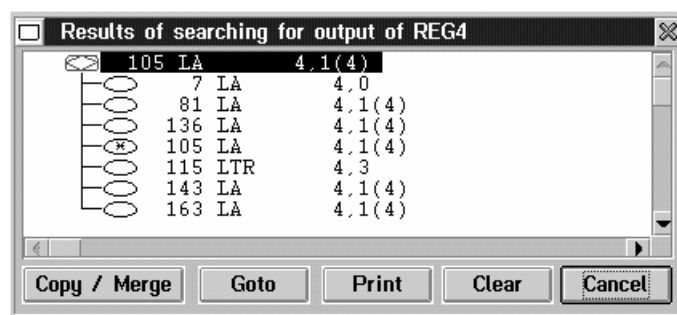
```
MVC    DBER+43 ( 18 ) , DBER+42
```



If, however, you select the variable using “Select data by name”, the length of the DS or DC data item is used in the search.



*Result of a manual single-level data flow search for the sources of REG4 in line 105*



*Result of a single-level data flow search for the sources of REG4 in line 105*

The data flow analysis is executed in the background. During the search you may switch to another program via the OS/2 task list, or browse within [reASM](#) and call up the display function.

When the data flow analysis takes too much time, you may cancel the background thread via menu item “File / Stop task”. No results will then be available.

Explicitly-addressed data are treated in a special way (see page 37).

Error sources:

- \* The data flow analysis follows the control flow. When the control is not correctly determined (e.g. at computed branch targets), the data flow analysis results will also be incorrect.
- \* Explicitly-addressed data (q.v.)
- \* Unknown macros: in case they read/write/modify data, this will not be recognised as long as the macro has not been defined accordingly in **STMACRO.ARI**.
- \* Wrong read/write macro parameter lengths: e.g. in a GET macro or a DL1 macro a field X, which is declared with a length of 2, can be specified as an input field, but VSAM or DL1 write a greater length, e.g. 1000 bytes. Data flow analysis for a variable with offset 4 with respect to X will not stop at this macro but will proceed – possibly though the entire program.

## 5.5 Data / Extended Output Backward

Backward data flow analysis is also available in an extended function. It goes back stepwise to the different sources that contribute to the value of the specified variable.

The first step runs like the usual output analysis. The second step takes the end points of the first step and further traces the corresponding variables. Assume the original variable is REG4 and the end statement of the first step assigns to REG4 the sum of REG5 and REG6:

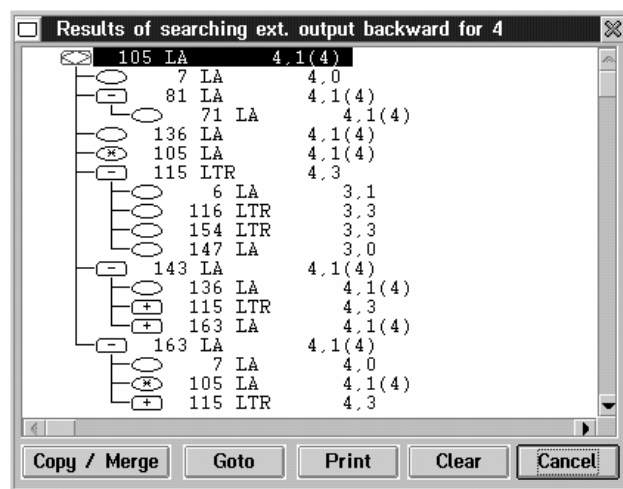
```
LA      REG4 , 0 ( REG5 , REG6 )
```

Then the second step traces REG5 and REG6 backwards. REG5 may be filled from a variable:

```
L       REG5 , FIELD1
```

Then the third step will trace FIELD1 backwards.

When you call up the function “Extended Output backward”, you will be prompted to specify how many steps have to be analysed.



Result of the two-level data flow search for REG4.

From line 115, **reASM** no longer searches backwards for REG4, but for REG3.

Line 136 does not show deeper results, because line 136 is dead code.

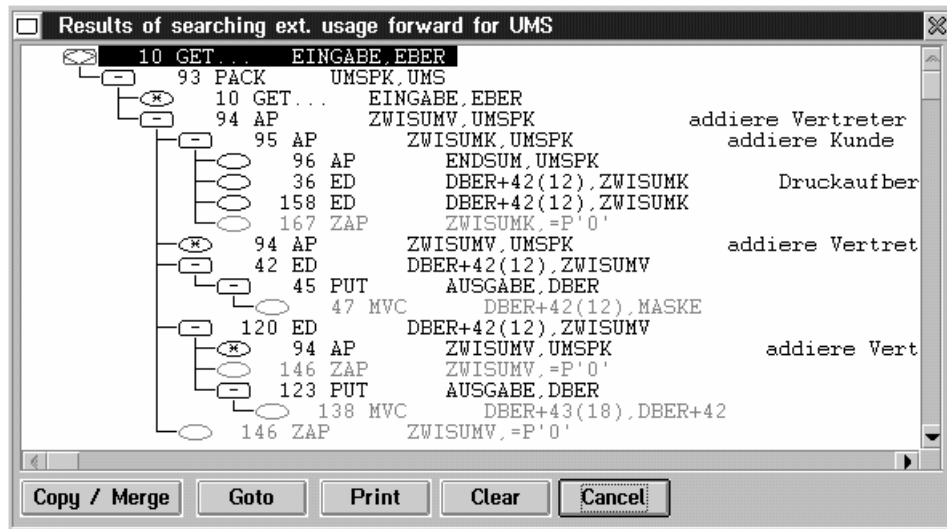
Together, the end points found make up the set of **end results**. Under menu item “Navigate / End results” you will see them displayed as a tree. When a variable references itself in a loop, the statement will be marked with an asterisk (“\*”). If prior to the start of the data flow analysis the set of **end results** is not empty, it will be cleared before the analysis starts.

Internally, the “Extended Output backward” function is composed of several calls to the single step data flow analysis “Output backward” function. The paths followed are not displayed.

## 5.6 Extended Usage Forward

You may use this function to stepwise trace the data flow forward. You will see the usage of a field and how its value propagates into other variables.

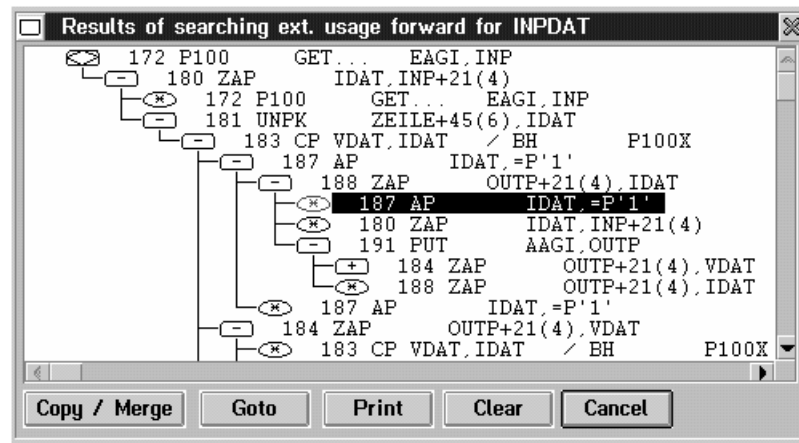
Suppose you know that in the UMSATZ program (which is supplied with the [reASM](#) demo version) the field UMS is read, and you want to check the company's turnover calculation rules. You select the GET command as a starting point, choose function *Extended usage forward* and specify UMS as the field to be traced. The result of this analysis will be a set of statements, and the stepwise character of the analysis is displayed as a tree,



The first step only finds one result: line 93. Based on the assignment, UMSPK will be included in the next search set, as is ZWISUMV from line 94 and ZWISUMK from line 95 – the search set already contains 4 variables. The analysis from line 95 leads to both line 96 (tracing UMSPK) and lines 36, 158 and 167 when tracing ZWISUMK.

The assignment in line 167 ends the lifetime section of the old ZWISUMK and starts a new incarnation. Line 167 is the “fringe” of the validity area of the analysed ZWISUMK and is therefore displayed grey; the tracing for ZWISUMK stops here. Similarly for ZWISUMV in lines 94 => 146.

An example from another program in the context of converting to Year-2000 compliance. Suppose you know that this program reads record INP, which contains a 6-digit date field named INPDAT. You want to know how this field INPDAT propagates through the program, e.g. in order to identify other 6-digit date fields. You select the GET command as a starting point, choose the *Extended usage forward* function and specify INPDAT as the field to be traced. The result of this analysis is a tree of statements found:



The INP data structure

INP	DS	0CL51
	DS	CL2
INP2	DS	CL14
	DS	CL5
INPDAT	DS	PL4

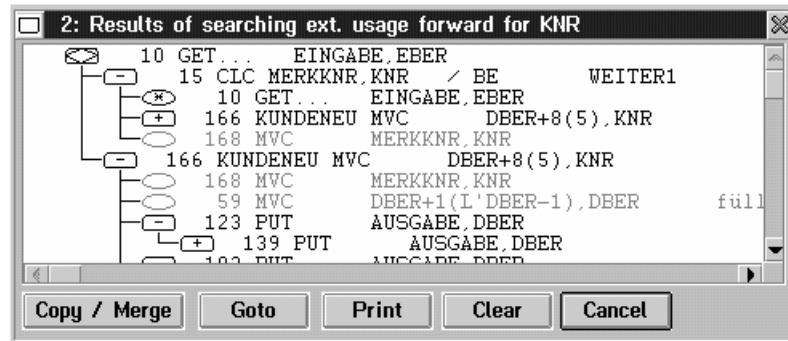
has been inserted into the program afterwards, line 180 still references the field by offset. **reASM** recognises such redefinitions and addressings by offset. Because of the ZAP assignment, from line 180 **reASM** will also trace IDAT and will find statement 181, in the next step statement 183, then 187 and then 188. In statement 183 the control flow branches to P100X according to the CP compare of VDAT and IDAT. As the corresponding option “Search mode like date search” (see below) was switched on, **reASM** will from here on also trace VDAT and will find statement 183 (in a loop) and 184. After the assignment in statement 188, OUTP+21(4) will also be recognised as a date, and a further trace of this field component will end up at the PUT in line 191.

The *Extended usage forward* function consists of several calls to the data flow analysis function *Usage*. At the end of every single call, the results are analysed in order to determine the appropriate search set for the next step:

- When XYZ is moved into a field ABC, from here on both XYZ (which is still valid) and ABC will be traced.
- When XYZ is assigned something, the search for XYZ will end at this point – unless this assignment also involves XYZ itself (example above: AP in line 187). When the trace is finished, the statement will be displayed in grey in the result in order to make clear that at this point XYZ’s first incarnation ends and a second one begins.
- When a pointer is placed on the date searched, such as in LA R5,XYZ, the pointer (the R5 register) is also traced. Explicitly-addressed data, such as 0(3,R5), can be searched in a similar way.
- When the variable searched XYZ is compared to ABC, the option “Search mode like date search” allows you to control whether or not ABC will be included in the search set. In the above UMS example this was not very useful, but in the case of the search for the INPDAT date it is. In general, this option is useful in those cases where you actually know a *data type* and want to trace its propagation through the program. Examples of such data types are 6-digit dates, amounts of money (in view of the conversion to Euro), or just a simple client number. The search for the newly introduced date ABC will of course follow the control flow forward, where in this case it might also be interesting to find out the result of a

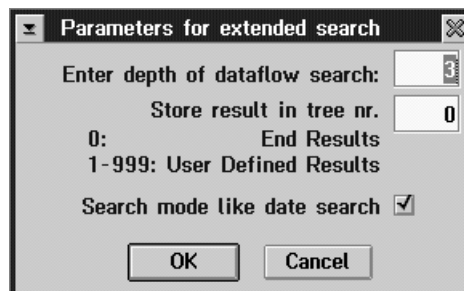
backward search (in the above INPDAT example: where does VDAT come from?). You can achieve this in a second evaluation via VDAT.

Another example for the “Search mode like date search” option: the client number KNR has a length of only 5 characters and has to be made longer. You know that KNR is read into the UMSATZ program as part of EBER, and you let **reASM** search for KNR, starting at the GET statement. The result:



Lines 14 and 15 compare KNR to MERKKNR, from which we may conclude that MERKKNR also has data type “5-character client number”. Line 168 is the end of the incarnation of MERKKNR used in the comparison of line 15, and a new incarnation begins. In line 166, a part of DBER is included in the search set – when the client number is made longer, this one should also be made longer. In line 59, DBER is re-initialised (displayed in grey), or alternatively output in line 123 in AUSGABE.

## 5.7 Parameter input with Extended Usage



In this dialogue window you enter:

- The desired number of steps for the analysis.  
Every step will cause an exponential increase of the number of control flow branches that are analysed, as well as the number of variables traced. Therefore begin with a “prudent” value, such as “3”.
- Where to store the results: in the *end results* or in one of 999 *user-defined results*. The *user-defined result* with the number specified will be created; if it already exists, you are prompted for permission to overwrite it.
- Whether or not the search is to be executed as a date search (variables in a compare statement will also be included in the search set). (Cf. description of *extended usage forward*.)

## 5.8 Data / Reg free

How do you determine whether or not register X is free at a certain point in the program?

- Select with CTRL + S the element concerned (select the statement only, no data).

- In the Data menu, select “Reg free”.
- You are prompted to choose a register.
- When you click “OK”, the search starts.
- When no statements are found as a result, the register is “free”.
- When the register is used as an input in the data flow further on, it is not “free”; the corresponding statements are displayed as a result.

This menu item is a combination of the normal data flow analysis. First it checks whether register X is used as a base register. If not, a forward data search for register X is started. In case register X is used – not as output, but as input – it is *not* free.

If the set of the *end results* is not empty at the beginning of the data flow analysis, it is cleared before the analysis starts.

## 5.9 Data / Assignment Xref of...

Frequently those points in the program are of interest where a variable is assigned a value. You are prompted to enter a variable name or to choose a register; then you will get a list of those statements.

Unlike with the data flow analysis, the memory location is not taken into account here, but the name only. Always the first name of an expression is used for this purpose. If an assembly instruction looks as follows:

```
MVC    EBER+OFF(L'FELD2) , FELD2
```

the statement will be entered in the Xref under the name EBER, not under the names OFF and FELD2.

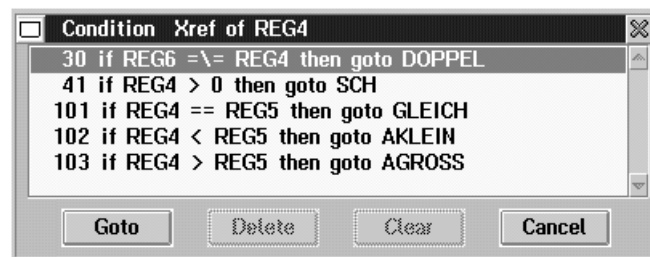
## 5.10 Data / Condition Xref of...

Frequently those points in the program are of interest where a variable is used in a condition. You will be prompted to enter a variable name or to choose a register; then you will get a list of those statements.

The statement entered into the list will always be the one where the condition has its effect. E.g. in the case of

```
CLC    MERKKNR , VNR
BE     WEITER1
```

the BE statement will be entered in the condition. This is done firstly because the CLC alone is not a complete condition (the compare operator is still missing), but only together with e.g. a BE is the condition complete. The second reason is that the BE statement is the one that in the textual representation represents the complete information PLH-like.

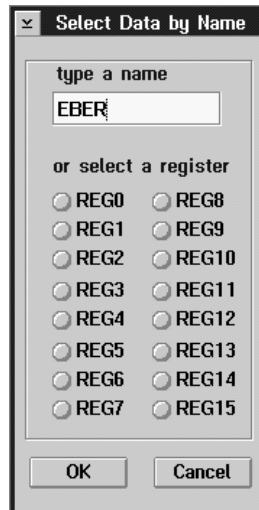


*A Condition Xref of the ZINSEN program.*

*Because under Options the text representation has been set to PLH-like, the window shows pseudocode.*

## 5.11 Prompt window: Select Data by Name

You are prompted to either enter a variable name or choose a register.



If you want to enter a variable name:

- Type the name. The text currently under the cursor is displayed as a default value.
- if only part of this variable should be used for a dataflow search, enter also offset and length: <name>+<offset>(<length>) e.g. DBER+22 ( 5 )
- Click the OK button with the mouse.
- If you have typed a non-existing variable name, a message window is displayed saying “variable not found”. Click on OK and correct the name.

If you want to choose a register:

- With the TAB key, leave the entry box and with the arrow-down key move to the register, or simply click the register with the mouse.
- Click the OK button with the mouse.

With the CANCEL button you leave the window without triggering any activity.

In case of *Extended usage forward* you can enter several fields to be searched, e.g. because they are strongly related semantically. If you want to specify only one field, you proceed as described above. If you want to specify more than one field, you enter the first field as described above and then click on the MORE button. The field is then moved from the left-hand side of the window to the right-hand side, and you can enter the next field in the left-hand side. When you have collected in the right-hand box all fields to be searched, you click the OK button.

## 5.12 Displaying the number of statements found

With control flow and data flow, the top-right corner of the **source view window** shows two numbers (with Extended data flow only if you have specified “end results” as the target):

**path:** The number of statements between the selected statement and the end statements (i.e., the statements searched).

**end:** **End points** of the search: the number of statements found with the function (F, I, O or U).

On the screen, the statements are shown coloured:

Start statement:	green
Start data:	dark green
Path statement:	yellow
End statement:	red

In addition to this, the text under the cursor is shown in reversed-video.

The path number is not always shown. When e.g. a search can use the results of an earlier search, or when an extended data flow search is started, or when the path is too long, then the path is not shown. In that case, a “zero” or a “one” is displayed as the path.

When a statement or a data item is re-selected, the path and end results sets are cleared.

When the selection has been cancelled with “Navigate / Clear selections”, the yellow-red box with the path and end numbers will also disappear.



## 6. The Options menu

### 6.1 Options / Autoscroll

The Autoscroll option refers to only one source view window. In another source view window this option can be set differently.

When the Autoscroll option is activated for a source view window, this window will be permanently positioned at the cursor, when the cursor is moved in another window.

In the following windows, a cursor movement will automatically induce a repositioning of those windows for which the Autoscroll option is activated:

- Cursor movements in another **source view window**.
- Cursor movements in the **call hierarchy window**. In practice, this means that the autoscrolling window always shows the procedure header on which the cursor is placed in the call hierarchy window.
- Cursor movements in the **end results window**. In practice, this means that in the end results window you will see an abstract representation consisting of several statements, and in the autoscrolling window you will always see the context of the current statement of the end results window.

### 6.2 Options / Font

This allows you to change the font of the active source view window. Only monospace fonts are available. The new font is remembered in an INI file and used at a new program start.

The semigraphical control flow display uses special ASCII character that are not available in all fonts. Under OS/2 the VIO fonts are recommended, under Windows the “Terminal” font (a bitmap font), or the font SANSMP.TTF, which is supplied on the **reASM** distribution disk.

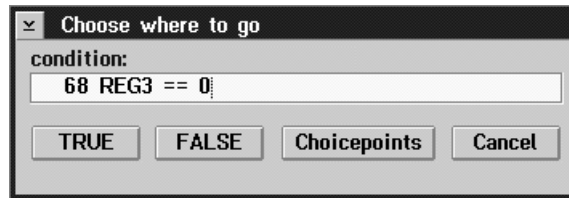
### 6.3 Options / forward & backward

This selects the direction in which the Flow, Input, Output and Usage function operate. The option is toggled between forward and backward with the ENTER key or a mouse click.

### 6.4 Options / one-choice

The One-choice option is designed to follow a program path step by step (e.g. for a “desktop test”). A backward operation is also possible, e.g. when a dump is available and you want to find out by what path the program has reached the point of abnormal termination.

- Choose the “One-choice” option and the “Forward” or “Backward” option (in this scenario we assume that you chose “Forward”).
- With CTRL + S select the statement at which you want to start.
- With menu item “Control flow / Flow” or CTRL + F, let **reASM** follow the program flow. As soon as the control flow branches or executes a procedure call, a prompt window appears asking you which path you want to follow.



REASM08A

When the forward direction has only one branch (e.g. an unconditional branch), **reASM** will nevertheless stop in order to inform you about the branch. In this case, only the TRUE button is available.

While the prompt window is waiting for an answer, you can browse in the main window, e.g. in order to more accurately study the branch statement.

*d.* In the prompt window, you choose with one of the buttons TRUE and FALSE which branch you want to follow. Your decision is stored in the choice points list, the new statement is automatically selected and the cursor is positioned on it (green plus reversed-video looks purple). You can now proceed with *c* (i.e., follow the controlflow with CTRL + F) or with *e*.

*e.* Any action

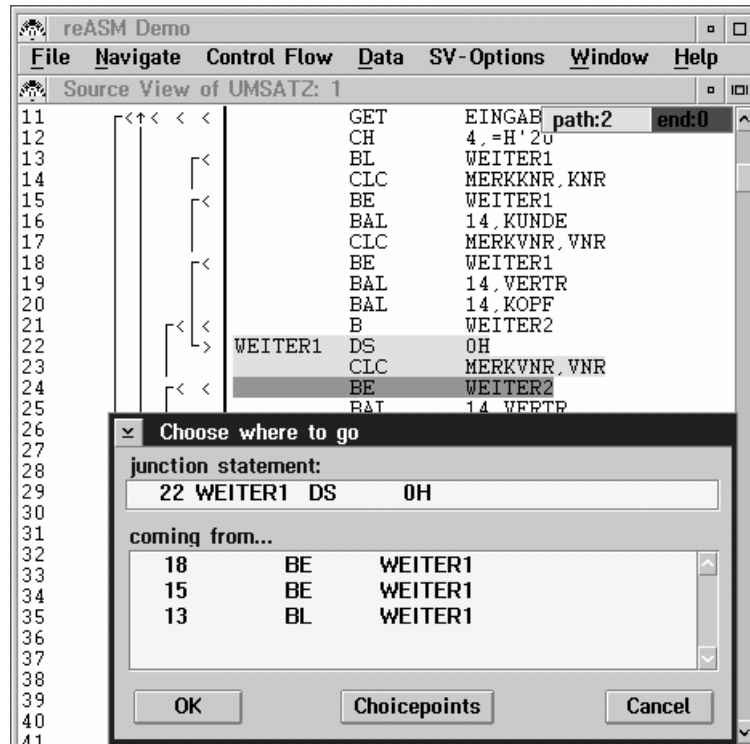
*f.* Proceed with *c*.

At any time you can call up the choice points list via menu item “Navigate / Choice points”, in order to:

- Look up what statements you have followed so far. If you have switched on option “Text PLH-like”, the branch conditions are also visible.
- to “reset” the path, by deleting choice points bottom-up, up to the choice point where you would like to follow another path. Then you select “GOTO” in the box, so that **reASM** will position the display on this statement, and then you select it with CTRL + S. Now you proceed with *c*.

A procedure call is also treated as a branch (although the call has no condition). This allows you not to follow a procedure call when it is currently not of interest.

Calling a data flow function will automatically switch off the one-choice option.



*One-choice backward, from the selected statement in line 24.*

*Line 22 shows the junction: from which one of the three possible points did we come to line 22?*

## 6.5 Options / Text PLH- & ASM-like

In the end result, user-defined result and choice points windows, statements are displayed. With this option, you control whether the statements are shown in ASM-style or in pseudocode (PLH) style in these windows. The option is toggled with the ENTER key or a mouse click. Default is *ASM-like*.

## 7. The File menu

### 7.1 Processing an assembly program

By means of file transfer, the assembly program must have been transferred to the PC as an ASCII file. It can be available either as 80-column source text, or as a compiler listing. The compiler listing is recommended. The file to be read must have a filename extension. The extensions “ASM” or “LST” are recommended.

There are two programs:

- REASM.EXE is a presentation manager program.
- REASMB.EXE is a batch program receiving a parameter: the file name.

Because of the deep semantic analysis, processing takes some time (for a big program possibly an hour). That is why there is no menu item “File / Read ASM-File”. Instead, the batch program REASMB must be used, which displays the progress of the processing on the OS/2 console. Warnings and errors that occur during this time are not displayed in a window that has to be released with a key stroke, but are logged in a file **XYZ.LOG**.

Processing consists of the following steps:

1. Syntactical analysis (“performing read ...”)
2. Program flow analysis (“performing pflow pass”)
3. Analysis of assembler actions (“performing action (\_\_\_\_)”)
3. Preparing screen output (“performing cpl\_list (\_\_\_\_)”)

and under **reASMgen** some more steps that are selected in the profile.

The ASM source to be input is supposed to be syntactically correct. Syntax errors may cause the program to abort. Unknown instructions or macros only produce warnings; their operation (input/output) is ignored. When an unknown macro contains a branch (like the GET macro branching to an EOF label), this is of course not taken account of in the control flow, and consequently not in the data flow either. This situation may sometimes cause a label like EOF1 to be reported as “dead code”.

When a listing file is processed, possible macro expansion lines (i.e., lines preceded by a “+”) will generally be masked. The reason is that the semantic level of macro expansion is the system programming level, not the commercial programming level. In our opinion, macro expansion lines will disturb reading and may impede analysis by **reASM** more than contributing to it.

In the MACRO.ARI file you may specify for every single macro whether or not its expansion must be taken into account.

COPY statements are executed, i.e., the file to be included has to reside in the “asm” directory or in the “copy” directory. (This only applies when source files are used for input, not listing files.)

However, it is often not useful to read big complete data declarations (e.g. from CICS definitions). That would unnecessarily increase program size (on the PC) and would lead to even longer processing times. You should rather confine this to what is actually used in the application program; most often, this is just a few fields.

The necessary EQU’s should be present in a COPY file. The usual register declarations, R0,..., R15 and REG0,..., REG15 don’t need to be defined by EQU’s, as they are already built into the syntax analysis.

reASM must recognise an unambiguous control flow. Problems may occur with computed **branch addresses** (cf. page 35)

## 7.2 File / Source View

With this menu item you may open another source view window on the ASM source. This window can be positioned independently from the other windows, you may assign it other exclusions, etc.

## 7.3 File / Open and Save Work

It is possible to save the current database file of the program and reload it at a later stage with “Open”. When you prepare an ASM source with REASMB, REASMB will generate such a database file.

These files are called XYZ.IDB and XYZ.IDC. The “Open” function opens a file selection window in which you choose the file to be loaded. Only \*.IDB files are shown.

## 7.4 File / Print

After you have specified *from* what line *until* what line the file is to be printed, the lines will be printed in a background thread to a file (filename extension LRE, in the DATA directory; any old LRE file will be overwritten). When the background process is finished, a beep will sound.

Control flow arrows, assembly code and pseudocode for every line are printed. Consequently, the line length will usually exceed 132 characters.

## 7.5 File / MergePrint

When an ASM listing is used as input, it is often more convenient not to print the reASM information to a separate file, but to merge it into the existing listing file.

The “MergePrint” function reads an ASM listing from disk (it must be the same file as used in the REASMB processing!), inserts control flow arrows and prints it (filename extension LRE, in the DATA directory; any old LRE file will be overwritten). Pseudocode is not printed.

You may control where the arrows are printed. A dialogue window appears in which you enter two columns: from column  $x$  to column  $y$  in the listing will be replaced by the arrows, e.g. from column 0 to column 35.

When the range from  $x$  to  $y$  is smaller than the width needed for the arrows, the resulting listing will become wider. More specifically,  $x$  and  $y$  may be equal; in this case only this particular column in the listing will be overwritten.

It is not (yet) possible to specify *from* what line *to* what line is to be printed.

## 7.6 File / Stop Task

This menu item is only available when a thread is running in the background. When you select this menu item, the background thread is terminated. The executing function is aborted without any result.

## 8. The Navigate menu

### 8.1 Navigate / GOTO Label...

With this dialogue, the source view window can be positioned at a specific label. The label may be a branch label or the name of a data declaration.

The dialogue window that appears, contains the following support for fast navigation:

- As far as can be determined in a useful way, the string on which the cursor is positioned is displayed as a default in the entry field. Therefore, if you have first placed the cursor at a convenient point, the correct string will already be in the entry field.
- With the key combination SHIFT + INS you can copy a text from the clipboard to the entry field.
- You can pull down the combo box which shows the last three entries used.
- With the combo box, you can select the “Program start” as a target. This is not the same as line 1, but the line at which the program’s control flow starts.

### 8.2 Navigate / GOTO Line...

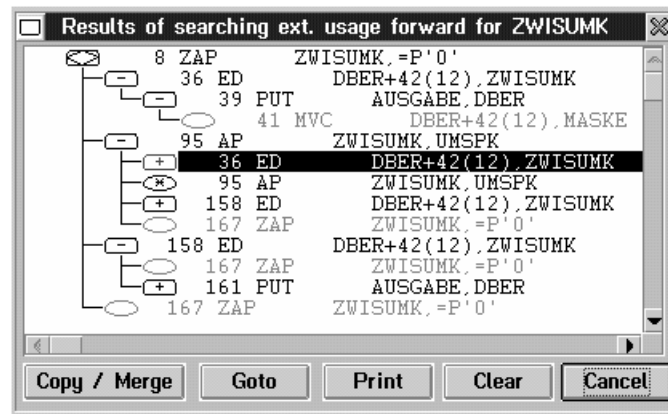
With this, you can position the source view window at a particular line number.

### 8.3 Navigate / End-Results & User Defined Results

The statements found in a data flow analysis are displayed.

When a data flow analysis function is “extended”, you can choose whether the results are to be stored in the *end results* (default) or in one of 999 *user-defined results*. The differences between these two options are:

- The *end results* are automatically cleared upon every Select and are thus made ready to store new results. The set of *user-defined results* is managed by yourself.
- In the source view window, the *end results* are visualised in red, the number of end results is displayed in red in the top-right corner. The *user-defined results* are not displayed in the source view window.
- The *user-defined results* can be saved with “File / Save Work” when you end a [reASM](#) session, so that they are available for the next session.



If you have used “extended data flow”, as in this example, the tree displayed is as deep as the number of steps you specified, and at some points even deeper due to the search method used. In non-extended data flow analyses the tree is one level deep.

As is usual with trees, two mouse clicks on a minus sign in the tree will collapse the tree at this point, while the minus sign becomes a plus sign. In the example shown here, this has been done with the second-last line, the PUT.

When a tree branch has already been expanded at another point, this expansion is not repeated, but the point is marked with a plus sign. In the example shown this is the case with the ED at line 36, on which the cursor is placed. When you now give another mouse click at the cursor position, the cursor jumps to the point where the expansion is available (in the example shown, to the ED in the second line).

When the data flow analysis refers to itself in a loop, the statement is marked with an asterisk (in the example, the AP in statement 95).

In case of an “extended usage forward” data flow analysis there are statements displayed in grey. Here the variable is given a new value; in fact the validity range of this variable traced ends here.

In the options menu, you may select whether the statements are to be represented in an “ASM-like” or a “PLH-like” way. When a particular statement is included in the tree and you do not understand why, just try to switch the representation to “PLH-like”. Most likely you will then see why.

When you enlarge the window by dragging with the mouse, the inner pane is enlarged correspondingly. So you may set the visible section either by means of the scroll bar or by dragging the window.

With the “Goto” button you can position the source view window at the statement that is under the cursor. This way you can very easily show the context of a certain statement.

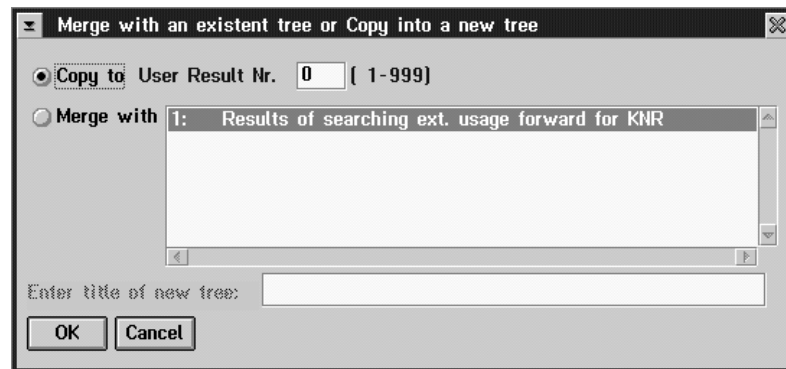
With the “Print” button you can export the tree in ASCII to a file. The stepwise character is shown in ASCII by an indentation with blanks. The plus and asterisk characters have the same meaning as in the graphical display, and the statements displayed in grey will be marked with an “f” (“fringe”).

With the “Clear” button you clear the results. As for the *end results*, this is equivalent to menu item *Clear selections*.

With the “Copy/Merge” button you can copy the tree or merge it with other trees.

## 8.4 Copy / Merge User Defined Results

The *end results* and the *user-defined results* can be copied or can be merged with other trees.



**Copy:** Enter the number of a user-defined results which is still free.

**Merge:** Choose a tree from the list of user-defined results; the tree from where you clicked the **Copy/Merge** button will then be added to it. In the lower entry field you enter a title for the tree that results from this merge. In case you want to retain the original version of the user-defined result, you must first copy it to another user-defined result number.

The merge operation can be useful for several reasons, such as:

- You have traced a variable XYZ with *extended usage forward* with a search depth of 3, and then you have separately selected the branch's end point and searched it in-depth. You may want to combine the results in one tree.
- You search two variables, X1 and X2 which play a major role in a business rule. Starting from statement N you did an *extended output backward* search for X1, and then another search for X2. It may be useful to combine the two results in one tree.

You must decide for yourself whether or not this merging trees makes sense. It is, e.g., senseless to merge a tree from an *extended usage backward* search with one from an *extended usage forward* search.

## 8.5 Navigate / Clear Selections

The *path* and *end results* sets are cleared; no more statement or data item is selected.

## 8.6 Navigate / Keep List

Keep List is used to “bookmark” a statement. Suppose you have positioned the cursor at line 41, and you want to have a short look elsewhere and want to return to line 41 afterwards, you type CTRL + K, the shortcut for the KEEP command. The statement will then be added to the Keep list.

In order for you to easily return to the currently selected statement (green), it is always contained in the Keep list. It cannot be removed with DELETE.

Via menu item “Navigate / Keep List” you can display and manage the Keep list:

- GOTO: positions the **source view window** on the corresponding statement.
- DELETE: the corresponding statement is removed from the Keep list.
- CLEAR: all statements are removed from the Keep list.
- CANCEL: quit the window.

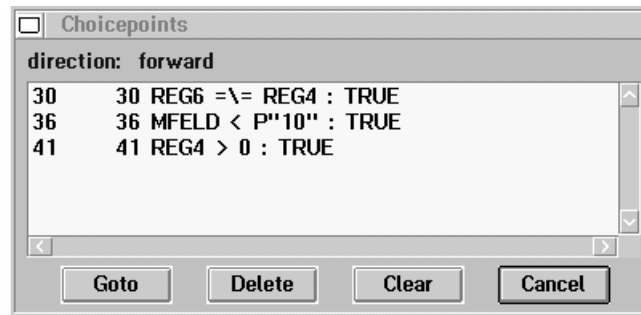


## 8.7 Navigate / Choice points

Choice points are only available when the one-choice option is switched on and you follow the program flow step-by-step with *Flow*. The choice points are the branch points that have been collected so far; they represent the path followed so far.

With the GOTO button you position the source view window on a statement. To do so, you select a line in the list box and click on the GOTO button. The choice point window will be closed.

Positioning with the GOTO button has no effect on where a command like “Flow” will proceed. That is determined by the “Select” command. In the one-choice process, you can position on other points in the program without disturbing the program path you were already tracing.



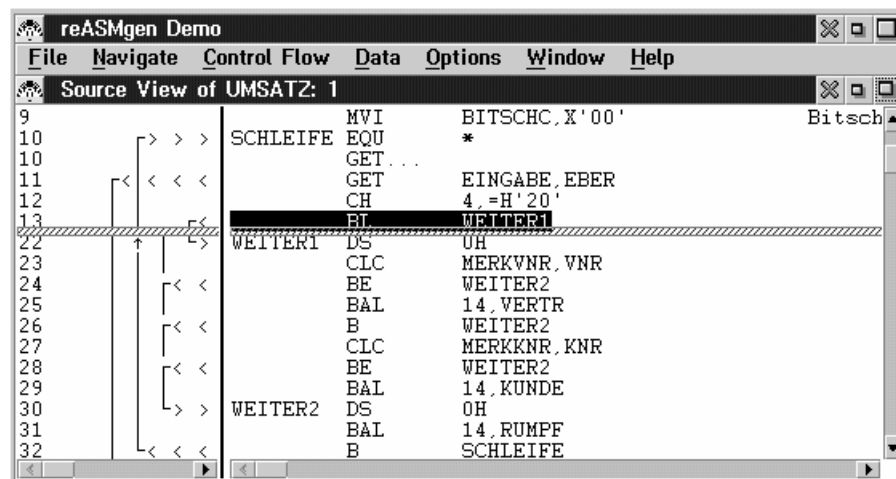
*The list of the choice points passed in a one-choice forward search.  
Each line records the condition and the TRUE or FALSE decision.*

With the DELETE button you may delete the choice point at the bottom of the list. This means resetting a path chosen. When you have “reset” this way, you can choose an alternative path with a GOTO (cursor on the last line in the list box) and a “Select”.

The CLEAR button deletes all choice points.

## 8.8 Navigate / Exclude lines

You can *exclude* lines from the screen representation in order to get a better overview of the assembly program. Where lines are “excluded”, a blue-striped bar is shown.



You can exclude lines in several ways:

1. In the menu, select “Navigate / Exclude from... to...”. A dialogue window appears where you can enter *from* what line *to* what line you want to exclude.

For your convenience, the cursor line number and, if available, the line number of the selected statement are displayed as default values in the dialogue window. This way, you can enter the line number range with one mouse click and a CTRL + S.

2. Position the cursor on a GOTO statement or on the target label of a branch and type key combination CTRL + x. The complete branch range is excluded from being displayed on screen. In the figure above, the cursor was on line 13 before CTRL + x was typed. The same applies to a procedure's return statement or its header.

When you type CTRL + x while the cursor is not on such a statement, an error beep is given.

3. In the menu, select "Navigate / Exclude Level". This level is the indentation depth of the arrows. The outer arrows have level 1, the inner ones 2, 3, 4, etc. until the deepest level (nesting depth). If you enter, e.g., "4", all deeper levels will be removed from the display. If you enter "3", even more will be excluded.

### ***Resetting:***

1. You may reset all exclusions by selecting "Navigate / Clear Exclusions" from the menu.
2. You may reset a single exclusion by placing the cursor on the line immediately above the blue-striped bar and typing CTRL + y.  
Lower-level exclusions may appear. If you have excluded e.g. level 3 and then restore it, then level-4 exclusions may appear.

Exclusions are related to a single **source view window**, i.e., you may use different source view windows and work with different exclusions in each one of them.

## 9. Concepts

### 9.1 Computed branch targets

**reASM** must recognise an unambiguous control flow. Without an unambiguous control flow no useful data flow will result.

In a BR, a BALR and similar instructions, the branch targets are not explicitly given but they are addressed through register contents. The original concept was that **reASM** tries to determine at which branch address the register points (register evaluation). If **reASM** cannot unambiguously determine this, an error message was produced.

This exact algorithmic determination of the register contents faces the following problems:

- It takes relatively much time.
- It is very complicated.
- It may fail at any intermediate result that cannot be interpreted (e.g., when a SAVEAREA is explicitly addressed – which is not unusual).
- Several sources for a register's contents may be found.

For this reason, a heuristic is used which roughly works as follows:

- Every BR is interpreted as a return from a procedure, as far as the register fits to the procedure header.
- If the line in which a register is used as a computed branch target has a comment beginning with a "reference", it is used as described below.
- Otherwise, register evaluation is attempted.

An example of using the reference instruction `use_jumtable`:

Suppose register 11 is calculated in some way and finally points to an entry in a jump table TABLE:

```

                AH      R11,ZHW
                SLL     R11,1
D15            BR      R11          ref 227
                ...
TABLE         DC      A(LABEL1)
                DC      A(LABEL2)
                DC      A(LABEL3)
```

The reference file contains the instruction:

```
227: use_jumtable('TABLE')
```

This way, you inform **reASM** that the possible contents of register 11 can be found in the table TABLE. **reASM** expands the BR instruction to three conditional branches to LABEL1, LABEL2 and LABEL3.

Jump tables may be defined in several ways. In the above example, it only consists of addresses. The question is how many of the consecutive data declarations must be considered to belong to the jump table. In the above example, **reASM** will assume the table to be ended when it sees a statement that is no longer an address declaration.

When the table contains holes, or when table values of another kind are found between the addresses, you will also have to indicate the beginning and the end of the table by a comment. The

line with the table's name must contain a reference instruction `table(begin)`. The end of the table must be marked with `table(end)`, which is to be placed in a separate comment line *after* the last table element. Example:

```
TABEL    EQU    *                                ref 82
          DC     C'A  ',S(ADR1)
          DC     C'AH ',S(ADR2)
          DC     C'W  ',X'0000'
          DC     C'WA ',S(ADR3)
*                                ref 83
```

The reference file contains the instructions

```
82:    table(begin)
83:    table(end)
```

**reASM** takes account of the addresses ADR1, ADR2 and ADR3 and ignores the other data declarations.

## 9.2 Procedures

When the line `RECH MVC A,H` is addressed as a procedure, **reASM** separates this line into three lines:

```
RECH    MVC.....
          MVC...
          MVC    A,H
```

The first line is the procedure header, the second one remembers the register and the third one is the MVC.

A procedure can be called from several points in the program. In the control flow, a procedure call is like a branch to the procedure header, possibly from several points in the program.

With respect to the data flow, several constellations can be distinguished.

Suppose you want to search backwards from a CLC for a field XYZ:

```
          MVC     XYZ,...
          BAL     14,MYPROC
MYPROC    .....
          EQU     *
          CLC     XYZ,..
```

Suppose MYPROC is called from several points in the program; the example shows only one call. When control flow is followed backwards from the procedure header, **all** calls will be found, and from there the search goes on. That is, more than one MVC is likely to be found. When the procedure is called from many points, the search path will branch highly.

The other constellation is as follows:

```
MVC     XYZ,...
BAL     14,MYPROC
CLC     XYZ,..
```

Suppose you search from the CLC backwards for field XYZ, XYZ is *not* referenced in the PROC procedure and MYPROC is called from several points in the program. If the data flow would work the same way as in the first example, then again many MVCs would be found. To avoid this, a special treatment was necessary for procedures. In this example, **reASM** finds exactly one MVC. In

the UMSATZ demo program you can try this: In line 115, select REG4 and run the O(utput) function. **reASM** only finds three points – otherwise it would have been more.

### 9.3 Explicitly-addressed data

With an explicitly-addressed variable, both the base register and the offset are specified explicitly, e.g. `0(1,REG5)`. We also consider a DSECT data item as being explicitly addressed, as it is mostly used to address several different storage locations at runtime.

For our purpose, we consider “normally” declared variables (symbolic addressing) to be fixed-location variables, members of a large working storage. Note: In case their base registers are changed in the program, **reASM** will *not* take this into account.

Problems will now arise when the data flow functions Input / Output / Usage compare data items that are addressed in different ways: is a variable `0(1,REG5)` identical with variable XYZ which was declared normally?

In a static analysis this question cannot be answered exactly, because it is not possible to tell generally at what storage range REG5 points at runtime.

Therefore we have to make assumptions here. **reASM** basically assumes that a fixed-location variable is *not* used as input or output for an explicitly-addressed data item.<sup>1</sup>

But how is input / output of an explicitly-addressed variable handled? Variables are considered the same when the register and the offset to this register are the same. The second stop criterion of the search is when the address register is assigned a value.

In an MVCL the registers are evaluated in a similar way as in a BALR, in order to assess what storage range is operated upon. In case the four registers cannot be evaluated unambiguously (e.g., when a table is being processed), a message is given.

### 9.4 Redefined and variable-length data

The Input / Output / Usage functions check whether or not a variable is used in an instruction. This check does not rely on the variable’s name but instead uses the memory location plus the length of the data item. This ensures that this check is not cheated by a redefinition.

Although with variable-length data the memory location is fixed, the length is not. The length is contained in a variable and can have very different values during runtime.

An example:

```
MOMO    MVC    FELD(0),FELD3
          .....
          EX     REG5,MOMO
          .....
FELD     DS     CL5
```

The Input / Output / Usage functions require that an assumption regarding the length be made. The length from the declaration is used (in this example: 5), and the message 310408 informs the user about this.

---

<sup>1</sup> Rationale: If we assume (as a compiler does) that REG5 may basically point at anything, including a normally-declared variable like DBER, then a corresponding data flow analysis in which this DBER were used as the next target, would be rather useless, because in that case all such “0(1,REG5)” data would be displayed. This is not in line with the intuitive assumption.

## 9.5 Dynamic code modifications

Many assembly programmers have the bad habit of modifying code at runtime, e.g. to change a NOP into a BNE instruction.

The most usual of these modifications are taken into account by [reASM](#) as follows:

- The modification is executed, the resulting statement is re-interpreted (e.g. as a BNE instruction) and inserted after the source statement. This action is recorded in the log file as e.g.: “Changing instruction ... to ...”.
- Both statements are given a condition in which a switch is used, e.g.: “if switchTAB30 == 1”
- The modifying statement, e.g. an OI, will be displayed as “switchTAB30 := 1”. The switch is assumed to be initialised with a 0 value, so the non-modified instruction gets the condition “if switchTAB30 == 0”, and the modified one gets the condition “if switchTAB30 == 1”.

In case the target address of a branch is modified at runtime with a non-constant value, but with a calculated branch address, we have a combination of dynamic code modification and computed branch target. The explanations of page 35 hold in this case.

## 9.6 SECTIONS

The assembler’s sections concept is not completely modelled by [reASM](#), but only insofar as we considered important: DSECTs. Every DSECT is treated as an independent data region, everything else as a big working storage.

For the Input / Output / Usage data flow functions, the question must be answered whether or not a DSECT variable DFELD overlaps a working storage variable WFELD. This question is discussed in more detail in the chapter on explicitly-addressed data ([see page 37](#)).

## 9.7 USING

With menu item Statement / USING you can display the USING constellation for the statement on which the cursor is positioned: which registers are currently used as base registers and which registers are assigned to which DSECTs? This question is primarily important when using POPs, PUSHs and DROPs, which are all taken into account.

For example, the display

```
REG10, REG9, REG8->TABEL
```

shows that REG10 and REG9 are defined as base registers and REG8 is assigned to the DSECT TABEL.

Error sources: USING instructions are often hidden in macros.

## 9.8 EQU

EQUs can be treated in several ways. Firstly, EQUs can be simply substituted literally – that is the way the assembler-compiler works. That way, however, some semantic information would get lost that is expressed by the EQUs. Therefore [reASM](#) tries to avoid as much as possible substituting EQUs. It only substitutes them when necessary, e.g., when something has to be calculated. This way, the possibility remains to re-interpret the EQUs at a later stage, such as an LTAB EQU 33 as a constant with value 33.

## 10. Description of supplied and generated files

For the PC administrator, only the two files REASM.ENV and REASM.PRO are important,

### 10.1 REASM.ENV: ENVIRONMENT file

This file configures system parameters for the Prolog runtime.

The REASM.ENV or REASMB.ENV file must be available in the directory from which REASM or REASMGEB is launched (current directory). When it is missing, default values are used.

All parameters are in upper case, followed (without blanks) by “=” and the parameter value. Most of the parameters present in the default file are only important for MS-DOS. Under OS/2 the following parameters are used:

**LOCAL** and **GLOBAL** set the size (in kbytes) of two different stacks. Both play a role in deep recursions, such as with data flow search. A value of 255 for GLOBAL and LOCAL will suffice, even for big programs.

WIN95: The OVERFLOW parameter gives a filename for the swap area of PROLOG. When the internal data storage grows in excess of MAXPAGES, the data are swapped to disk.

The MAXPAGES parameter sets the size of the storage to be reserved for the internal PROLOG cache.

Example of an ENV file:

```
GLOBAL=255
LOCAL=255
MAXPAGES=2048
OVERFLOW=TMP.IDB
```

Under MS-DOS, this configuration by means of REASM.ENV was very important. Under OS/2 and WIN95 it has lost its importance and is generally limited to defining the OVERFLOW name.

### 10.2 REASM.INI

This file is placed in the operating system directory and remembers the last settings, e.g. the last used font.

### 10.3 XYZ.IDB and XYZ.IDC: Internal database

The database of an assembly program processed is stored in the two files XYZ.IDB and XYZ.IDC and can thus be re-loaded in another session.

REASM.IDB is the initial database and is needed when starting [reASM](#).

Like REASM.EXE, REASM.IDB is searched along the path(s) defined by the PATH environment variable.

### 10.4 XYZ.PRO: Company and program profiles

**REASM.PRO** is the *company-wide* profile. In particular, it must contain the path where the ASM files are to be searched, and (with [reASMgen](#)) the path where the generated COBOL or PL/I programs have to be stored. For REASMGEB.EXE, the profile is also called REASM.PRO.

REASM.PRO is sought in the current directory or in directory “\REASM”.



In the example supplied, the different possibilities are explained in comments. The profile syntax is PROLOG syntax (cf. page 46).

It is possible to *set up a profile for every program*, e.g. for the UMSATZ program the profile would be UMSATZ.PRO. Such a profile is searched in the current working directory.

First of all, remarks regarding the program – in comment form – can be stored in this profile. Example:

```
/* Contains an MVCL with table processing -> reASM prints a warning */
```

Secondly, definitions from REASM.PRO can be overridden here.

Thirdly, special program-specific options can be switched on or off in this file.

#### 10.4.1 Options for reASM

`path( asm, <path> )`: The path in which the \*.ASM or \*.LST files are to be found. Example:  
`path( asm, $C:\ASS\$ )`.

`path( copy, <path> )`: The path in which the COPY files are to be found (only relevant when the assembly program is read in source form).

`start_label( <String> )`: If in your programming environment a label is required where program execution starts, you can make this label known to reASM [here](#). For example, a label “START”:  
`start_label( $START$ )`.

`extern_label( <name>, <action> )`: This specifies labels that are not to be used in the control flow analysis. This may be truly external labels (or labels not included in the source code, such as “ENDE” as a general end of program), or it may be labels that are included in the source but must be ignored.

The first parameter is the label name (a string, and consequently placed between \$...\$). The second parameter indicates the semantics and is one of “abend” (abnormal end of program), “exit” (normal end of program) or “address”. Example:

```
extern_label( $ENDE$ , exit )
```

`bell_level( <level> )`: Level from which a signal tone is produced while reading in batch. The levels range from 0 to 16 and are explained in REPORT.ARI.

`info_counter( <Integer> )`: In batch processing, frequently screen messages appear which show the progress of the reading process. “Info\_counter” indicates how often these messages are to appear. If, e.g., you want the messages to appear after every other line (not counting comment lines), you specify:  
`info_counter(2)`.

#### 10.4.2 Options for reASMgen

In addition to those for reASM, reASMgen has the following options:

`path( gen, <path> )`: The path where the generated code is to be stored. Example with a relative path:  
`path( gen , $COBOL$ )`.



`language( <language> )`: This indicates what language is to be generated. Languages supported so far are COBOL and PL/I. A parameter may be added to indicate the language dialect; the effects of this dialect must be programmed into GEN.ARI. Examples:

```
language( pl1(saa)).  
language( cobol(ibm)).
```

`pass( <name of a pass>, <yes|no> )`: This controls which analyses [reASMgeb](#) must execute. When e.g. analysing the table heuristics takes too much time, it can be switched off using

```
pass( tab_hyp, no ).
```

`editor(<program name>)`: When program parts are generated from [reASMgen](#), they are loaded into an editor. With this option you specify which editor is to be launched. A Windows example:

```
editor( 'NOTEPAD' ).
```

`generate_asm_comment(<parameter>)`: This option allows you to specify in what extent the generated COBOL or PL/I source is to contain references to the original assembly statements. The parameter “linenr” causes the line numbers of the original assembly statements to be included in the generated source. The parameter “asmline(<string>)” causes the complete original assembly statement to be included as comment in the generated source. To ensure that these comments can be simply removed after the conversion is complete, they are marked with the string you specified. Example:

```
generate_asm_comment( asmline( $asm$ ) ).
```

causes line 125 in the UMSATZ program to generate the following COBOL lines:

```
125 *asm      LA      REG5,TABVTR  
125          MOVE 1 TO REG5-TABVTR-ix.
```

In this case, both parameter “linenr” and parameter “asmline” are switched on (they can be combined).

`right_margin(<integer>)`: This option sets the right-hand column limit for the generated code. It works both for COBOL and PL/I generation. For PL/I, the left margin is fixed at 2.

`decimal_point(<parameter>)`: This defines the decimal character in the edit masks. Values allowed are “comma” and “point”.

`substring_limit(<integer>)`: This option controls whether a NAME+OFFSET is to be generated as a substring (in COBOL: reference modification) or as a redefinition. When in the program the same NAME+OFFSET occurs more often than the limit specified in this option, a redefinition will be generated.

`literals_as_constant(<parameter>)`: With this option, literals are removed from the procedure division and will be accessed via a generated name.

The parameter “hex” means that all hexadecimal literals are removed.

The parameter “char” means that all character literals are removed.

## 10.5 STMACRO.ARI and MACRO.ARI: Macro definitions

[reASM](#) does not process assembler macro statements. It is better to directly describe the macro semantics.

A more elaborate explanation is found as comment in the STMACRO.ARI file itself.

The **STMACRO.ARI** file contains macro definitions that are not restricted to one company. They are updated by the [reASM](#) manufacturer. Do not modify **STMACRO.ARI** unless absolutely

necessary (e.g., when a macro contained in it is defined differently in your company). In this case you have to make your changes again upon every update of **STMACRO.ARI**.

The **MACRO.ARI** file is intended for company-specific macros. You create this file as a normal ASCII file. In it, you declare your company-specific macros, according to the examples in **STMACRO.ARI**. The installation does not contain a **MACRO.ARI** file to avoid overwriting your own version.

The **STMACRO.ARI** and **MACRO.ARI** files are found in the current working directory.

When **reASM** finds a macro that is not declared in **STMACRO.ARI** or in **MACRO.ARI**, it assumes the simplest form (no parameters, no semantics) and proceeds. Nevertheless, certain macros should be declared from the beginning. These are, with decreasing priority:

- Macros representing a procedure call, a procedure header or a procedure end.
- Macros that modify program flow, i.e., macros containing a branch. When such a macro is not declared, the branch will not be contained in the control flow. With many macros this is not a problem, e.g., with the exception conditions of the DFHBMS macros. The branch targets are not part of the programme's mainstream, but handle an exception, which may then be seen by **reASM** as "dead code".  
Macro branches that are part of the regular control flow, must be declared.
- Macros that represent data declarations. They are very simple to declare, because **STMACRO** contains a class for this.
- Macros that set registers.  
A similar case are fields that at program start contain the address of some storage region which is then assigned to a register. This fact must be declared in the profile, e.g.:  
`init_value( $CWALOTAB$, address( $TAB_AREA$ ) ).`

## 10.6 XYZ.ASM, XYZ.LST: Assembler source, assembler listing

Contains the S/370 assembly code.

Avoid editing XYZ.ASM files on a PC. If you have to do so, be sure not to use the TAB key, or more accurately, not to insert any ASCII TAB character into the file, because all characters below hex 20 are treated as control characters. Sometimes also a declaration like  
DFHERROR CHAR (1) INIT ('□')  
has such an ASCII character as init value, which hinders processing even in a comment line.

Also be sure that after transferring a listing file to the PC there it does not contain any hex 0 byte (which on the PC may be interpreted as an end-of-file mark).

Basically, any file extension can be used for source and list files that are to be prepared; however, there has to be a file extension. It is recommended to use „ASM“ for source files and „LST“ for list files.

## 10.7 XYZ.REF: Reference file<sup>2</sup>

This file contains further possibilities to influence the semantic interpretation and the generation by **reASMgen**. In the comment part of the ASM or LST files, reference numbers are inserted; in the reference file commands for **reASMgen** can be entered under the corresponding reference number.

### Reference file syntax

---

<sup>2</sup> The most recent explanations on this topic can be found in the REF.REF file.

The filename is <program name>.REF, e.g. W200.REF.

A reference starts with <number><colon> and ends with a period.

The reference numbers have to be ascending, “holes” are allowed.

A reference may contain more than one instruction for the ASM statement; they are separated by a semicolon.

The instruction is a command, e.g. “procedure\_pattern( multipleentry)”, or setting an attribute, like type, array size, width, etc.

“st( )” indicates the complete ASM statement. “op1( )” and “op2( )” indicate the operands.

Only tokens beginning with a lower-case letter are used. Upper-case letters must be quoted, e.g. 'B100', or placed between dollar signs, e.g. \$B100\$. Numbers are also allowed.

“message( <message no.>, <parameter>,...)” is used for classifying and controlling messages. The first parameter is the message number, any number of parameters may follow.

The PROLOG syntax rules apply.

The **semantics** of the **reference file** can be read in the REF.REF file.

## 10.8 XYZ.LOG: Log file

In every analysis of an assembly program, a log file is generated which contains several kinds of messages. The type of the different messages can be configured in REPORT.ARI. The log file contains informative messages, such as the time used for the **processing**, references to problematic points in the code, warnings and errors.

The most important references to problematic code are:

- dead code: i.e. unreachable code.

**reASM** analyses the programme’s control flow. The “dead code” message is generated when a statement is cut off from the control flow.

Sources of inappropriate messages:

\* When computed branch targets are not correctly recognised.

\* When a macro contains a branch to a label, but the macro has not yet been declared.

- Branches that violate the procedure region, e.g. branches from one procedure into another. A procedure starts at its procedure header and ends at its last RETURN.

Sources of inappropriate messages:

\* When a RETURN is not correctly placed with respect to its procedure.

\* When a RETURN cannot be matched with one procedure but is used for several procedures. If need be, you may solve this by an appropriate instruction in the reference file.

- When code is “dangling” outside a procedure: e.g., it jumps to a point after the RETURN and then back again to a point just before the RETURN. Whether or not the message is appropriate in this case, is a matter of opinion. Although according to control flow the code is unambiguously connected to the procedure, this is not easily seen when reading. The programmer may have saved a few minutes by inserting this piece of code after the procedure, but causes unnecessary problems for anyone else who reads the program
- **Dynamic code modifications**, i.e. at runtime statements (e.g., a NOP) might be modified.

All these points represent sources of maintenance error and should be cleaned up in view of preventive maintenance.

## 10.9 XYZ.DBR: Import database record

After the assembly program has been processed, a **database record** (XYZ.DBR) is generated which contains basic information on the program and stores the number of problematic code points. The purpose of this database is that it can be imported into a dBase or SQL database, so that a history database can be built with objective maintenance information.

The basic data have the following format (unless indicated otherwise, all fields have 8 positions; each separated from the next field by a field separator):

- Program name
- Date: 10 positions, format: YYYY-MM-DD

If the input was a list file, the list file date stamp is used. This allows a better synchronisation with other data, e.g. when the program is transferred to the production area.

If the input was a source, then the calendar date is used.

- Number of functional (executable) instructions.  
This is a measure for program size. Comments and data declarations are not counted. This number is often called FLOC, as it is an improvement on the LOC number (LOC = lines of code).
- Number of data instructions (DS, DC, ORG, USING, and the like).
- Number of comment lines.
- Number of labels referenced in branches.
- Branch nesting depth.  
For at least one point in the program the control flow is so complex that branches must be displayed with this nesting depth. This may, however, be a local point and may not indicate an overall program complexity – that is covered by the next number.
- Sum of nesting depths.  
This is a measure for the control flow complexity of the entire program. This corresponds to the maximum intersection number described by Chen (1978) in an article in *IEEE Transactions on Software Engineering*. This article also presents empirical values on how the control flow complexity, when measured this way, influences the *productivity* of the programmers.

Sources of discontinuous numbers:

In the profile, labels are declared that [reASM](#) must consider as being external. They are therefore not included in the control flow nesting – this is intentional. With this, and with the option `return_lookahead`, you strongly influence the value of the nesting depth number. Be sure to first correctly set the options for program XYZ, before you begin to collect these numbers for XYZ. Also be sure that the options are not changed afterwards.

From these basic numbers, further numbers can be derived. Especially the SumNestingDepth per FLOC is a useful metric to compare different programs.

Then the number of problematic code points follows, with 8 positions each. In the REPORT.ARI file you can configure which numbers you want to appear. In the [reASM](#) configuration supplied as default, these are (a more elaborate explanation for each number is found in REPORT.ARI):

- Number of dynamically changed statements (310501)
- Number of (supposed) dead code blocks (310502)

- Number of branches that violate procedure boundaries (310203)
- Number of overlapping procedures

In the case of **reASMgen**, DBR will contain more metric numbers, which are especially useful to assess the smooth convertibility to a 3GL.

## 10.10 **REPORT.ARI**

This file contains the texts corresponding to the log file messages. If you prefer other message texts or want to suppress a message, you can do so in this file. Where necessary, a message is explained in a comment. For better readability, you get an extract of these comments in REPORT.TXT.

In particular, it contains the reference texts to the problematic code points. Which ones are counted and are output to XYZ.DBR, can be configured here.

## 10.11 **REASM.HLP**

This file contains the help texts.

REASM.HLP is the help file for REASM.EXE, while REASMGEN.HLP is the help file for REASMGEN.EXE.

WIN95: REASMGEN.CNT contains the help contents.

## 10.12 **XYZ.LRE: List file**

The processed assembly source, which above all shows the indented control flow arrows, can be printed to a file by means of **FILE / PRINT** or **MERGEPRINT**. The name of the file thus generated will be <program name>.LRE.

## 10.13 **XYZ.RLH: Restruct file**

When the optional module for **restructuring** is called up, the restructured representation is output to this file.

## 10.14 **XYZ.TMP: Temporary file**

Used for outputting temporary texts.

## 10.15 PROLOG files notation (\*.ARI , \*.PRO and \*.REF)

In these files, the PROLOG syntax must be followed:

- Comment is to be enclosed between /\*...\*/, or between “%” and end of line.
- PROLOG clauses look either as follows:

```
abc( ... ) :-  
    defg( ... ) ,  
    hijk( ... )
```

or purely relational:

```
abc( ..... ) .
```

The left parenthesis must immediately follow the identifier, without a blank.

The PROLOG clause is always terminated with a period.

- Everything must be lower-case. Upper case letters are to be enclosed between dollar signs (e.g., \$LIEFERKZ\$). PROLOG is case-sensitive, e.g., \$Namelist\$ and \$NameList\$ are different.

Capitalised names without dollar signs are interpreted by PROLOG as variables. In case of incorrect use, this may lead to unexpected program execution. A single underscore is a blind variable, a variable without a name.

Examples:

```
options( substring_limit, 2 ).  
extern_label( $DUMP$ , abend ).
```

When you launch REASMB.EXE, the files MACRO.ARI etc. are read. Possible PROLOG syntax errors are flagged at this moment. This would look as follows:

```
SYNTAX 9 Right paren not found where expected.  
'RSRBR'( 'Self' , return_def( set , 'REG15' , [ 0 : 'STATUS' ==  
'OK' , 4 : 'STATUS' == 'EOF' ] ) <<. >>
```

Indeed a closing parenthesis was omitted in the RSRBR declaration. The correct declaration looks as follows – correctly indented:

```
'RSRBR'( Self, return_def( set, 'REG15',  
    [ 0: $STATUS$ == 'OK',  
      4: $STATUS$ == 'EOF'  
    ] ) ).
```

## 11. Appendix

### 11.1 Storage requirements and performance

**reASM** runs on an IBM-compatible PC under OS/2 2.11 and OS/2 Warp, as well as WIN95.

Further information can be found in the Environment file chapter.

### 11.2 Assembly-language subset

**reASM**'s purpose is to improve the understanding of commercial assembly-language programs. Commands in relation to systems programming are not included, nor are floating-point instructions. Similarly, bits and half-bytes are in most cases ignored.

Macro instructions are ignored. Instead, macros can be declared in a programmable interface.. With macro parameters, syntax analysis treats two consecutive commas as one, so that position parameters cannot be recognized. As the macro expansion is not processed, but only the macro call, a correct interpretation of the macro parameters is not always possible.

The following list shows the language subset as processed by **reASM**. The abbreviation "370-ASM" indicates the behaviour of the IBM 370-assembler compiler.

Restrictions with storage operands:

when in an explicit addressed operand instead of using REG0 literally the parameter is simply omitted (e.g. ST 9 , ( , 1 ) ), the **reASM** syntax analysis interprets this as an error, whilst the 370-ASM inserts a zero.

Restrictions with literal lists:

Literal lists are processed correctly only with DC and LM, and in fact only in such a way that the DC and the LM are split up into as many single instructions as there are literals. E.g.,

DC        A ( ADR1 , 0 , ADR3 )

becomes

DC        A ( ADR1 )

DC        A ( 0 )

DC        A ( ADR3 )

On the other hand, with

MVC       FELD ( 6 ) , =P ' 00 , 000000 '

only the first one of the two literals is processed.

All instructions:

For every instruction, the condition code, the control flow and input/output from variables is modelled.

Both for data declarations and for executable instructions an internal location counter is maintained. When a listing is used as input, the location counter from the listing is used. An "\*" in an operand is interpreted in relation to this location counter, When a calculated address has no label, a lower-case FILLERxx name is attached.

No difference is made between 24-bits and 31-bits addressing mode. E.g., BAL and BAS are treated identically, and LA is treated as if it would set the entire register.

Setting and reading the program mask is not modelled.

With arithmetic instructions, sign, overflow and carry are not modelled. Shift instructions are treated like arithmetic instructions.

The instruction semantics is modelled in a PL/I-like way, which is called PLH. In PLH too, NOP means "no operation".

AP, AR, A, AH

Semantics: addition

AL, ALR, SL, SLR, CL, CLR

These instructions are treated the same way as the corresponding signed instructions (SL like S, SLR like SR, etc.). Carry is not modelled.

AMODE, RMODE

Ignored.



BAL, BAS

Semantics: “call”.

The label is transformed into a procedure header.

BALR, BASR, BASSM, BSM

Semantics: “call”, when the register is non-zero and can be evaluated.

When the register is zero, the instruction loads the register with the next address.

BCT, BXH, BXLE

The instruction is split into one that decrements the register and one with a conditional branch.

BCTR

When the register is non-zero, an attempt is made to evaluate it.

When the register is zero, the instruction only decrements the register.

B, NOP, BH, BL, BE, BNH, BNL, BNE, BO, BP, BM, BNP, BNM, BNZ, BZ, BNO

Except with B and NOP, an instruction is searched backwards in which the condition code is set. The variables from this instruction are inserted into the condition. Example:

```
CP      OP1,OP2
BNE     LABEL
```

The BNE instruction is assigned the semantics “if X =\= Y then goto LABEL”. X will be replaced by OP1 and Y by OP2. The CP instruction is assigned a NOP.

The condition code itself is not modelled.

Overflow is not modelled.

BR, NOPR, BHR, BLR, BER, BNHR, BNLR, BNER, BOR, BPR, BMR, BNPR, BNMR, BNZR, BZR, BNOR

As far as possible, the register is assigned to a procedure with a return register.

The semantics of, e.g., BNER is then “if X =\= Y then return”.

The other possibility considered is a branch table. When it is not dynamically modified, [reASM](#) may possibly recognise it.

BC

An attempt is made to derive a mnemonic from the mask. It will, however, not be unambiguous,

BCR

Ignored.

CLC, CLI, C, CH, CR, CP

Semantics: no operation (none).

Setting the condition code is not explicitly modelled (e.g. by setting a flag variable). Instead, an explicit condition is formed together with the corresponding branch instruction.

CLCL

Ignored.

CLM

Compare, with the register split up the same way as with STCM and ICM.

CNOP

Ignored.

COPY

Is executed in case no listing file was used as input.

CSECT

The 370-ASM section concept is not emulated. More specifically, [reASM](#) cannot handle in a clever way several CSECTs that have the same label. From a CSECT, a procedure header is derived.

CVB, CVD

Assignment.

D, DP, DR

Division. Only integer division is modelled. Setting the remainder is not modelled.

DSECT

Treated in a similar way as explicitly addressed data ([cf. page 37](#)).

DS

Data declaration.

Syntactical restriction: As a duplication factor, only a number will be accepted, not an expression.



The alignment that 370-ASM performs with types like H, F, etc., is not taken into account.

In the context of executable code, the semantics of a declaration like “NAME DS 0H” is a label with a “no operation”; however, when the length is greater than zero, it is “exit (abend)”.

DC

Data declaration with an initial value. Otherwise treated like DS.

ED, EDMK

Semantics: assignment.

With EDMK, setting REG1 is not taken into account.

END

Ends an assembler source.

ENTRY

The name is entered.

EXTRN

An external name is entered.

EQU

Cf. chapter on EQU.

EX

Setting the condition code, forming the condition and the input/output behaviour are redirected to the statement designated by EX. In contrast to 370-ASM, **reASM** does not OR the register with the byte, but only substitutes the byte value. A useful conclusion is only found when the statement designated by EX has no further degrees of freedom, e.g., when its operands are not explicitly addressed.

Cf. the chapter on variable-length data.

EXEC CICS and EXEC DLI

The syntax of these commands is accepted. However, the semantics must be declared in STMACRO.ARI. When a listing has been read – then the EXEC CICS command is commented out and instead of it a DFHECALL is inserted in the code – the EXEC CICS command is uncommented and the DFHECALL is ignored.

L, LH, LM, LR, LTR, IC, ICM

Semantics: assignment. With IC, ICM, STC and STCM, the register is treated with a granularity of one byte, with the other instructions with 4 bytes.

LA

Semantics: 370-ASM only loads an address. **reASM** differentiates whether a register addition or an address assignment is involved.

LCR

Semantics:  $OP1 := 0 - OP2$

LNR

Semantics:  $OP1 := 0 - \text{abs}(OP2)$

LPR

Semantics:  $OP1 := \text{abs}(OP2)$

LTORG

Ignored. In the listing, the lines with literals that follow, are ignored.

MACRO, MEND

This embodies a macro declaration in 370-ASM. **reASM** ignores the lines between MACRO and MEND. because **reASM** does not interpret macros. Instead, the macro operations must be declared in MACRO.ARI. Whether or not a macro expansion in a listing is to be read, can also be specified in MACRO.ARI.

MP, MR, M, MH

Semantics: multiplication.

MVC, MVI

Semantics: Assignment  $OP1 = OP2$ .

Next, the operand OP1 is analysed to find out whether it represents data or executable code.

MVCL

An attempt is made to evaluate the registers. If successful, i.e., when an unambiguous value can be derived from the register, this value is used as a length or an operand address. Three instructions are inserted:

- fill with a padding byte;

- setting the source register;
  - setting the target register.
- Decrementing the two length registers is not modelled.  
A condition code is not modelled.

MVN, MVZ

A semantics is only assigned when the operands have a length of 1.

OP1 is analysed to find out whether it represents data or executable code. In the case of data, it is tested whether or not setting a sign in packed or unpacked data is involved.

MVO

[reASM](#) does not assign a semantics; [reASMgen](#) partially assigns a division.

NR, NI, N, NC

Semantics: AND-operation “`bitand(OP1,OP2)`”. Is reduced to MVZ or MVN, when the half-bytes are 0 or F respectively.

OR, OI, O, OC

Semantics: OR-operation “`bitor(OP1,OP2)`”. Is reduced to MVZ or MVN, when the half-bytes are 0 or F respectively.

ORG

prepares a data redefinition.

PACK

Assignment.

S, SP, SR, SH

Semantics: subtraction.

SLA, SLDA, SLL, SLDL

Semantics: Multiplication by a power of 2. The sign is not modelled; SLA and SLL are treated identically.

SRA, SRDA, SRL, SRDL

Semantics: Division by a power of 2. The sign is not modelled.

SRP

Division or multiplication

START

Indicates the program starting point.

ST, STH, STM, STC, STCM

Semantics: assignment. With STC and STCM, the register is modelled with a granularity of one byte.

TM

Simultaneously testing several bits. Is represented as a condition with AND and OR operations.  
Only works in co-operation with BO and BZ, not with BM and BNM.

TR

Semantics: `translate(string, Tablein, Tableout)`.

TRT

Semantics: two standard cases are modelled:

- searching characters: “`search_char(,)`”
- testing against a given contents of a field : “`allowed_char(,)`”

[reASM](#) tries to derive these two cases from the table declaration. This process is restricted insofar as it is just one interpretation of the statically declared code, whilst 370-ASM dynamically searches the table during runtime.

UNPK

Assignment.

USING, DROP, PUSH, POP

Cf. the corresponding chapter.

XR, XI, X, XC

Semantics: When both operands are equal, set at low value. Also, the pattern “Exchange of two fields” is recognised and represented while an auxiliary field is generated.

ZAP

Assignment.

TITLE, EJECT, SPACE, PRINT, NOPRINT  
Ignored.

## 11.3 Error messages

There are several sources of error messages.

### 1. Error messages from the ASM syntax analysis

A syntax error messages consists of two parts:

```
E          : syntax error
E          :      OI      TWAS^, TWAFELD
```

The second line shows the incorrect assembly line; the “^” indicates the error position.

In the example above, the error is not caused by faulty code, but by the “\$” character, which originates from a file transfer error. The syntax analysis cannot recognise a valid name.

In a DS statement, the syntax analysis cannot evaluate an expression used as a duplication factor. An example:

```
DS          (X'C8' -*+DFHCSABA)C
```

This line is syntactically correct, but **reASM** cannot evaluate it and produces an error message:

```
error: can't analyze X'C8' -*+DFHCSABA as dimension
```

Instead of the compound expression, the syntax analyser produces a value of 1 as a repeat factor, so that the analysis can proceed.

When a listing is read, the EXEC CICS is commented out and the expanded call is inserted in the code. **reASM** produces the EXEC CICS command as if it were not commented out. In case **no** DFHECALL follows – i.e., the EXEC CICS command was actually commented out in the source code – the EXEC CICS production cannot be undone and the syntax analyser reports:

```
reASM reactivated EXEC CICS/DLI call
```

### 2. Error messages from C subprograms

The control flow arrows are indented in a C subprogram that uses a fixed-size storage for this purpose. When this storage size is insufficient, an error message is produced:

```
pflow_add(.....)
```

In that case, you may order a version with a larger C storage.

The GOTO nesting depth is not limited. Up to a nesting depth of 16, the arrows are kept one blank apart; if the nesting depth exceeds 16, the arrows will be placed at a closer distance.

The error message “not understood”, as in

```
S : not understood sendput DSECTinsert_left(~510F54,~510F50)
```

is an error in the object-oriented processing and is to be reported to the manufacturer.

### 3. Error messages from the PROLOG engine

A PROLOG syntax error message looks as follows (see 11.12):

```
SYNTAX 9 Right paren not found where expected.
```

Processing error messages look as follows:

ERR 103 Database error

The most importing errors, with explanation, are:

ERR 103 Database error

An error occurred during a database access. Often this error will occur if an attempt is made to store a term larger than the 4K page size. Alternatively, check for bad database reference or malfunctioning disk.

ERR 108 Error writing a page to disk - disk probably full

The disk to which writing is attempted is full.

ERR 212 Not enough global stack

In most cases an internal loop of the program.

ERR 214 Error while saving database

Not enough room to complete save. Since save does not remove the old database until a new one is created (hence a “safe save”), there must be twice the size of the database available on diskette.

ERR 221 Error allocating database memory - term too big

A PROLOG term exceeds a size of 4K. In this case you may order another version of the [reASM](#) and include the assembly program that exceeds this limit.

#### **4. Error and informational messages from REPORT.ARI**

These look as follows:

W 311401: taking DFHTCTZE as unknown macro

All message numbers from REPORT.ARI begin with a 3 and have 6 digits. [reASM](#)-specific message numbers begin with a 4. The messages in the REPORT.ARI file are sorted in ascending order; an explanation of every message can be found in that file.

## 12. Optional restructuring module

With this optional module, you can restruct the program's control flow range-by-range. The restructuring eliminates all branches – except for those branching to points outside the selected line range. Branches are transformed into if-then-else constructs or loops and an output in pseudocode form is produced.

When the menu item “Statement /Restruct control flow” is selected, a prompt window appears in which you specify from what line to what line you want to do restructuring. When restructuring is to be began at program start, then enter zero as “from line”.

Depending on control flow complexity, there are limits to the size of the line range to be restructured. Exceeding this limit causes the program to abort.

For the program semantics to be preserved in the restructuring process, it is often necessary in loops to temporarily store a condition in a switch. This is done at the point where there was originally a branch with this condition. Between this point and the point where the condition is used in a structured construct, there are several lines of code, which may change the values of the variables involved in the condition. Therefore, the condition must be temporarily saved in a switch.

Output is sent to a file with extension RLH. At the left margin, the original line numbers are printed, so that the origin of the line can be seen. The switch generated also contains the original line number as a part of its name.

- The starting statement of the line range you selected must be reachable from the program starting point via a control flow path. This does not hold for areas that turned out to be dead code or show other control flow clarity flaws. The message

“is not basic block”

is caused by such flaws. The message

“W 310903: Line xxx no control flow connection to program start”

results from a more strict check than the dead code check. This message shows that the control flow is discontinued and causes the next lines not to be included in the restructuring process.

- Start and end of the selected range may be extended to the next branch statement.

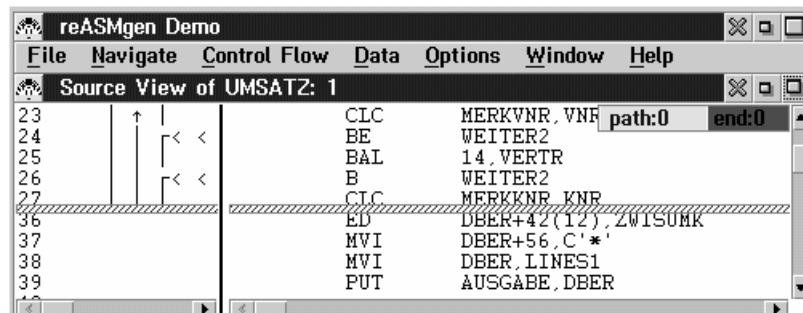
Beginning at the starting statement, those statements are included in the restructuring process that are in the line range selected and that can be reached via a control flow path from the starting statement. More specifically, this means:

**No branch from outside the line range is allowed to jump into that line range.**

When a branch jumps into the line range, then two quasi-parallel control flow paths lead into the selected range. The restructuring process can only take one path into account.

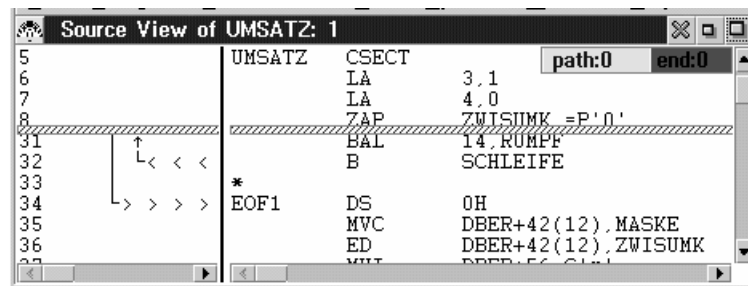
Example: When in the UMSATZ program you select lines 27-36 for restructuring, only lines 27-29 will be restructured.

When lines 27-36 have been excluded from screen display with the **exclude command**, you see that in this case two branches point into this range:



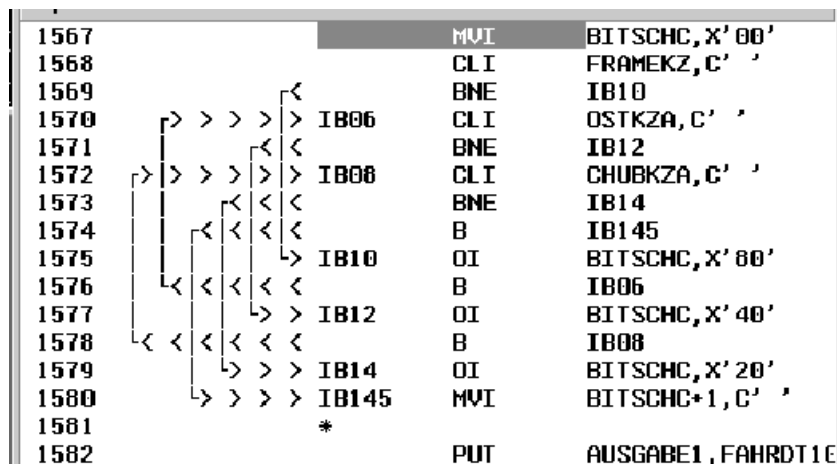
One branch (the central one) leaves the line range – that will not disturb the restructuring process.

Another **exclude** example shows how a branch from below enters into the line range and how a branch leaves the range:



The exclude command therefore shows rather quickly whether or not a line range is appropriate for restructuring. Unfortunately you do not see all branches that leave the excluded range: when at the left end a “goto” arrow passes by that starts far above the excluded range and ends far below it, this screen display does not show whether or not an arrow from the excluded range joins this long-distance branch.

A special feature of the restructuring algorithm is that not all backward branches are transformed to loops, but only the “real ones”. An example with two backward branches from a real program:



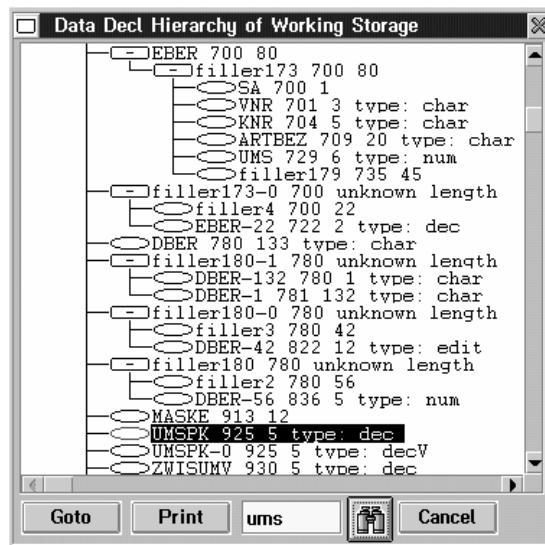
is transformed to:

```
1567     BITSCHC := low_value
1569     if FRAMEKZ = \= C" "
        then
1575     BITSCHC := bitor(BITSCHC, X"80")
1571     if OSTKZA = \= C" "
        then
1577     BITSCHC := bitor(BITSCHC, X"40")
1573     if CHUBKZA = \= C" "
        then
1579     BITSCHC := bitor(BITSCHC, X"20")
1580     BITSCHC+1 := C" "
1582     write(AUSGABE1,FAHRDT10)
```

## 13. User interface extensions for reASMgen

### 13.1 Data declaration hierarchy

Under “Data” you find a [reASMgen](#)-specific menu item “Data decl hierarchy”, which produces a hierarchical representation of the data declarations.



The data item name is displayed, as well as an array size in square brackets, the storage address, the length and the type as assessed by [reASMgen](#).

The binoculars button searches for text within the display. In the edit field, you specify the beginning of a name, e.g., “ums” (the search is case-insensitive). Searching starts at the current cursor position, and when the beginning of the name is found, the cursor is positioned at it. In the above example, “ums” was searched starting at the top. When the binoculars button is clicked again, the search continues from the cursor position, so “UMSPK-0” will be found in the next line.

The “Print” button allows you to print the data in several formats:

- Hierarchically, much like the graphical representation in the window
- As a flat DIFF file: this format allows the assembly program storage format to be compared with that of the COBOL program. The assembly program storage layout is output in the following format:
  1. the field name;
  2. array size, if any, in square brackets;
  3. the storage address of the beginning of the field;
  4. the field length.
- The data declarations are generated in working storage format.
- The data declarations are generated in file section format (not yet implemented). The difference between these two formats in COBOL is in the REDEFINES clause:  
in working storage format, REDEFINES clauses are allowed at level 1, while in file section format the redefinition is simply realised by a new start of level 1.

### 13.2 Generate statement(s)

Under “Control flow”, you find menu item “Generate statement”. Like “Display”, it refers to the cursor position (the current statement).

In order to allow you to simply transfer the text to the clipboard, it is shown in an editable field.



With menu item “Generate statements” you can have [reASMgen](#) generate a series of target language statements. The text generated is written to a file XYZ.TMP (a previous version of XYZ.TMP is overwritten) and this file is loaded into an editor. The name of the editor is specified in REASM.PRO.

These two menu items only generate executable statements. Data declarations can be generated from the “Data decl hierarchy” window.

### **13.3 Restructuring**

Under [reASMgen](#), restructuring does not generate pseudocode but PL/I or COBOL code.

## 14. Index

---

### *A*

AP 47

---

### *C*

Condition 12; 15; 35; 37; 48

Control flow 12; 13; 24; 28; 29; 34; 42; 43; 47; 51; 53; 56

---

### *D*

Dead code 27; 42; 43; 53

DS 49

---

### *E*

EQU 27; 35; 37; 49

EXEC CICS 49; 51

---

### *M*

Macro 17; 27; 41; 42; 46; 49

Memory location 21; 36

Message 310408 36

Message 310903 53

MVCL 50

---

### *N*

Nesting depth 44

---

### *O*

Offset 12

---

### *P*

PLH 47

Procedure 13; 24; 34; 35; 42; 43; 47; 48

---

### *R*

REG5\_1\_3 12

---

### *S*

STM 50

Storage offset 15; 17; 36; 56

---

### *Z*

ZAP 51