SIEMENS

Table of Contents

EasyCODE

Import and Export of Structure Diagram Files in Standardized File Format

Techn. SW Documentation

EasyCODE Version 6.xE, 06-20-1996

1 ETF File Format

2 ETF File Structure

An ETF file consists of a sequence of *entries*, with each entry written in a separate line. An entry consists of a *keyword*, one of the three separating symbols '=', ':' or ';' and (optional) text, which will in the following be referred to as the contents of the entry.

3 Keywords and Entries

Keywords consist of alphanumerical signs and underlines. Upper- and lowercases are significant.

White spaces[1] before the entry are not significant and will be ignored during file import. The only exception is the keyword *EasyCODE*, which must always be entered right at the beginning of a file. The separating symbols must be entered immediately after the keyword, without separating white spaces. The contents of the entry always begin with the first character following the separating symbol and end at the end of the line.

The keywords

- *Line*
- *Level*
- *EasyCODE*
- *Label*
- and the keywords representing defined options,

are followed by a sign of equality and then text.

All keywords that are assigned to construct elements for which the structure diagram may contain line numbers and, in the current case, actually contain line numbers other than 0, are followed by a colon and the line number.

In all other cases, the keyword is terminated by a semi-colon.

During file import, semi-colons, colons and signs of equality will be accepted after all keywords. The only exception is the *Level* keyword, which usually contains both a LevelID and a line number. Here it is implicitly assumed that the line number will follow immediately after the colon. All three characters must follow immediately after the keyword (or LevelID or line number in the case of the *Level* keyword) (no white spaces in between!).

Examples:

While keyword with line number=0:

```
While;
```

While keyword with line number=5:

```
While:5
```

Level keyword with line number=2, LevelID=3:

```
Level=3:2
```

Level keyword with line number=0, LevelID=3:

[1] Blanks, tabs, line breaks

Level=3

Level keyword with line number=12, no LevelID (in EasyCODE(COL)):
Level:12

Level keyword with line number=0, no LevelID (in EasyCODE(COL)):
Level;

4 ETF File Characteristics

The first entry of an ETF file (keyword: EasyCODE) looks as follows:
EasyCODE={component} {version} {date/time of saving}

Example: EasyCODE=SPX V5.0 1993-11-11 11:11:11

The keyword EasyCODE is necessary for identifying the export format, while the component name and the version will be analyzed during file import.

5 Short Info

Short infos are generated in a separate paragraph.

```
ShortInfo;
(Line={one Line entry per line})*
EndShortInfo;
```

The *ShortInfo;* and *EndShortInfo;* entries are required entries, the number of *Line*= entries is optional.

6 File-Specific Options

Options are generated in the *Options* paragraph.

```
Options;
IfLayout={horizontal/vertical}
LevelNumbers={yes/no}
LineNumbers={yes/no}
ScreenFont={System,,100,1,-16,0,700,0,0,0,0,0,0,1,2,1,34}
PrinterFont={Courier,,120,3,-50,0,400,0,0,0,0,0,0,2,1,0,49}
LastLevelId={number}
EndOptions;
```

The *Options;* and *EndOptions;* entries are required entries. The sequence in which the options are entered is, however, not relevant; incorrect or unclear entries will be skipped during file import (warning in the ERR file). In case of double entries, it is always the last one that will be analyzed.

7 Line Numbers

In version 5.0 and higher versions, line numbers will be stored in the ETF file, if they are other than 0. The line number of a construct element will be recorded wiith the corresponding keyword, after a colon. For details regarding the use of construct numbers in case of export errors see chapter *Export*.

8 Character Set of ETF Files

Depending on the INI entry *EtfFileFormat=ANSI* or *EtfFileFormat=OEM*, the ETF file will be treated as ANSI or OEM coded. For all components, this entry will be stored in the *Settings* section of the private INI file and cannot (yet) be modified with the help of the user interface. Default setting is *OEM*.

In V4.0 and higher versions, the INI entry *AnsiToOemConvert* has been renamed *SourceFileFormat* and may have the same values as *EtfFileFormat*. When the

application is started, the old INI entry will be accepted; it will, however, be deleted and replaced by the new entry when the option settings are saved.

9 INFO Strings for Import/Export

Some entries[2] (e.g. keywords: *SdfCommand, Variant*) have contents which will not be analyzed or taken into account by the import/export interface. Instead, they will be passed on as unchecked[3] strings from the ETF file to the application or vice versa, depending on whether the file is imported or exported. The contents of these entries are referred to as *info strings*.

Please note that info strings may - and in general do - comprise several lines. White spaces within the string are significant and must not be modified by the import/export interface.

If the INI entry *EtfWrapSDF* exists and is *true (*or *yes* or *1*), the line breaks following the info strings for SDF commands or SDF statements in an exported file will correspond to those in the structure diagram of the application. If this is not the case, even the longest SDF command will be generated in one line in the ETF file. Default setting is *yes*. This entry cannot be modified with the help of the user interface, but should nevertheless be documented for the user.

For details concerning the info strings (e.g. generation of info strings during export and analysis of info strings during import) see Appendix A.

10 Constructs and Their ETF Format

11 General Notes

In the following sections, the ETF format for all construct types is specified. At the beginning of each section, you will find a list of the components in which the construct is available and the name of the construct as provided in the respective *Insert* menu, then the construct itself (if several alternatives are possible, one of them has been selected at random. This is possible because all constructs of one construct type have the same structure.) and, to the right of the construct, the keywords of the ETF file. The dots ... indicate a lower recursion level. Entries enclosed in simple brackets are optional.

12 dummy_typ

Available in: all components

Dummy;

13 if_typ:

Available in: EasyCODE(COB)…IF-THEN-ELSE
EasyCODE(C++)….if
EasyCODE(SP)……IF-THEN-ELSE
EasyCODE(SPX)....IF-THEN-ELSE

If;
(Label=)
...
Then;
...
Else;
...

[2] All info strings that may occur are listed in the chapter *Constructs and Their ETF Format*.
[3] When it comes to the import/export interface !

EndIf;

During file import, empty ELSE branches will be added and, if this is the case, a warning will be displayed. During file export, the complete construct will be generated.

14 while_typ

Available in: EasyCODE(COB)…PERFORM TEST BEFORE
EasyCODE(C++)….while
EasyCODE(SP).....WHILE
EasyCODE(SPX)....WHILE

While/PerformTestBefore;
(Label=)
...
WhileBody/PerformTestBeforeBody;
...
EndWhile/EndPerformTestBefore;

15 cycle_typ

Available in: EasyCODE(SP).....LOOP
EasyCODE(SPX)....LOOP

Loop;
(Label=)
...
EndLoop;

16 break_typ

Available in: EasyCODE(SP).....EXIT
EasyCODE(SPX)....EXIT

WhenExit;
...
EndWhenExit;

17 case_typ

Available in: EasyCODE(SP).....SWITCH
EasyCODE(SPX)....SWITCH

OnCondition;
(Label=)
...
OnConditionRest;
...
EndOnCondition;

If the otherwise branch does not exist, an empty otherwise branch will be added during file import, and a warning will be displayed. During file export, the complete construct will be generated.

18 casebranch_typ:

Available in: EasyCODE(COB)…WHEN condition
EasyCODE(SP)……WHEN
EasyCODE(SPX)....WHEN

OnConditionBranch;
(Label=)
...
OnConditionBranchBody;
...
EndOnConditionBranch;

19 and_typ

Available in: EasyCODE(SP)……AND
EasyCODE(SPX)....AND

And;
...
EndAnd;

20 or_typ

Available in: EasyCODE(SP)……OR
EasyCODE(SPX)....OR

Or;
...
EndOr;

21 not_typ

Available in: EasyCODE(SP)……NOT
EasyCODE(SPX)....NOT

Not;
...
EndNot;

22 block_typ

Available in: EasyCODE(C++)….Block
EasyCODE(SP)……Block
EasyCODE(SPX)....Block

Block;
(Label=)
...
BlockBody;
...
EndBlock;

23 level_typ

Available in: all components…….LevelId

Level={dwLevelId}
*(EntryName={one line of the entry name per line will be generated})**
...
LevelBody;
...
EndLevel;

The EntryName entry is used for COL only.

For details regarding the specification of line numbers and LevelIDs for segments see chapter **3**

24 comment_typ

Available in: EasyCODE(COB)…Statement
EasyCODE(C++)….Statement
EasyCODE(DS).....Data element
EasyCODE(SP).....Statement
EasyCODE(SPX)....Statement

Text;
Offset={Offset of the Line entries}
*(Line={one Line entry per line will be generated})**
EndText;

The Offset entry is used for COB only (required entry).

25 switch_typ

Available in: EasyCODE(SP).....CASE
EasyCODE(SPX)....CASE

OnSelector;
(Label=)
...
OnSelectorList;
...
OnSelectorRest;
...
EndOnSelector;

If the otherwise branch does not exist, an empty otherwise branch will be added during file import, and a warning will be displayed. During file export, the complete construct will be generated.

26 switchbranch_typ

Available in: EasyCODE(COB)…WHEN expression
EasyCODE(SP).....OF
EasyCODE(SPX)....OF

OnSelectorBranch;
(Label=)
...
OnSelectorBranchBody;
...
EndOnSelectorBranch;

27 for_typ

Available in: EasyCODE(COB)…PERFORM TEST BEFORE VARYING
EasyCODE(C++)….for
EasyCODE(SP).....FOR
EasyCODE(SPX)....FOR

For/PerformBeforeVarying;
...
ForBody/PerformBeforeVaryingBody;
...
EndFor/EndPerformBeforeVarying;

28 repeat_typ

Available in: EasyCODE(COB)…PERFORM TEST AFTER
EasyCODE(C++)….do while
EasyCODE(SP).....REPEAT
EasyCODE(SPX)....REPEAT

Repeat/PerformTestAfter/DoWhile;
...
Until;
...
EndRepeat/EndPerformTestAfter/EndDoWhile;

In EasyCODE(COB), the first part of the tree (i.e. before the *Until*) contains the condition and the second part contains the body. In all other components, it is the other way round.

29 call_typ

Available in: EasyCODE(COB)…PERFORM Outline
EasyCODE(SP).....Procedure call
EasyCODE(SPX)....Procedure call

Call/PerformOutline/InternalProcedureCall
...
EndCall/EndPerformOutline/EndInternalProcedureCall

30 exit_typ

Available in: EasyCODE(COB)…EXIT
EasyCODE(C++)….break

Exit/Break;

31 detach_typ

Available in: EasyCODE(COB)…GOBACK
EasyCODE(C++)….continue

Goback/Continue;

32 c_switch_typ

Available in: EasyCODE(C++)….switch

CSwitch;
...
CSwitchBody;
...
EndCSwitch;

33 c_case_typ

Available in: EasyCODE(C++)….case

CCaseBranch;
...
CCaseBranchBody;
...
EndCCaseBranch;

34 default_typ

Available in: EasyCODE(C++)….default
EasyCODE(DS).....Alternative

Default/Alternative;
...
EndDefault/EndAlternative;

35 return_typ

Available in: EasyCODE(C++)….return

Return;
...
EndReturn;

36 cob_programm_typ

Available in: EasyCODE(COB)…Cobol program

CobProgram;
IdDiv;
...
EnvDiv;
...
DataDiv;
...
ProcDivParam;
...
ProcDivBody;

...
EndCobProgram;

CobProgram and *IdDiv* must follow immediately one after the other. During file analysis, *ProcDivParam* will be added, if required, and a warning will be displayed. During file export, the complete construct will be generated.

When the contents of *ProcDivParam* are imported from an ETF file created by V4.0, the *USING* keyword will be added. If this results in exceeding the maximum text length, the warning "Parameters of PROCEDURE DIVSION or CALL are too long and were cut off." will appear.

37 cob_section_typ

Available in: EasyCODE(COB)…SECTION

Section;
...
SectionBody;
...
EndSection;

38 cob_paragraph_typ

Available in: EasyCODE(COB)…PARAGRAPH

Paragraph;
...
ParagraphBody;
...
EndParagraph;

39 cob_inline_typ

Available in: EasyCODE(COB)…PERFORM Inline

PerformInline;
...
EndPerformInline;

40 cob_times_typ

Available in: EasyCODE(COB)…PERFORM TIMES

PerformTimes;
...
PerformTimesBody;
...
EndPerformTimes;

41 cob_varyingafter_typ

Available in: EasyCODE(COB)…PERFORM After Varying

PerformAfterVarying;

...
Varying;
...
EndPerformAfterVarying;

42 cob_exitper_typ

Available in: EasyCODE(COB)…EXIT PERFORM

ExitPerform;

43 cob_exittest_typ

Available in: EasyCODE(COB)…EXIT TO TEST

ExitToTest;

44 cob_exitprog_typ

Available in: EasyCODE(COB)…EXIT PROGRAM

ExitProgram;

45 cob_call_typ

Available in: EasyCODE(COB)…CALL

ExternalCall;
...
ExternalCallParam;
...
EndExternalCall;

If *ExternalCallParam* does not exist, the branch will be added during file import, and a warning will be displayed. During file export, the complete construct will be generated.

When the contents of *ProcDivParam* are imported from an ETF file created by V4.0, the *USING* keyword will be added. If this results in exceeding the maximum text length, the warning "Parameters of PROCEDURE DIVSION or CALL are too long and were cut off." will appear.

46 cob_exception_typ

Available in: EasyCODE(COB)…Exception

Exception;
...
OnException;
...
ExceptionBody;
...
NotExceptionBody;
...
EndException;

47 cob_evaluate_typ

Available in: EasyCODE(COB)…EVALUATE

Evaluate;
...
EvaluateBody;
...
EvaluateOther;
...
EndEvaluate;

If *EvaluateOther* does not exist during file import, the branch will be added, and a warning will be displayed. During file export, the complete construct will be generated.

48 cob_search_typ

Available in: EasyCODE(COB)…SEARCH

Search;
...
AtEnd;
...
SearchBody;
...
EndSearch;

49 entry_typ

Available in: EasyCODE(COB)…ENTRY

Entry;
...
EntryUsing;
...
EndEntry;

If the *EntryUsing* entry does not exist, it will be added during file import, and a warning will be displayed. During file export, the complete construct will always be generated.

50 proc_typ

Available in: EasyCODE(C++)….Function
EasyCODE(DS).....Object
EasyCODE(SP).....Procedure
EasyCODE(SPX)....Procedure

Procedure/Object/Function;
...
ProcedureBody/ObjectBody/FunctionBody;
...
EndProcedure/EndObject/EndFunction;

51 auswahl_typ

Available in: EasyCODE(DS).....Selection

Choice;
...
EndChoice;

52 wieder_typ

Available in: EasyCODE(DS).....Iteration, Option

Iteration;
...
IterationBody;
...
EndIterationBody;
...
EndIteration;

53 rahmen_typ

Available in: EasyCODE(SP).....Frame
EasyCODE(SPX)....Frame

Frame;
...
FrameBody;
...
EndFrameBody;
...
EndFrame;

54 class_typ

Available in: EasyCODE(C++)….Class

Class;
...
ClassBody;
...
EndClassBody;
...
EndClass;

55 public_typ

Available in: EasyCODE(C++)….public

Public;

56 private_typ

Available in: EasyCODE(C++)….private

Private;

57 protected_typ

Available in: EasyCODE(C++)….protected

Protected;

58 func_typ

Available in: EasyCODE(SP).....Function
EasyCODE(SPX)....Function

This construct is available only in V5.0 and higher versions.

Function;
...
FunctionBody;
...
EndFunction;

59 Import Function

60 Import on File Level

To import a file, choose *Open...* in the File menu. Before opening a file, EasyCODE checks first whether the file has an internal format (ECB, DS or SP file), then whether it is an ETF file (the export format is identified by the first entry - keyword: *EasyCODE*), and, last, whether it is a source file (C, COB source, DS source and SPX source with available parser).

Network behavior and naming are analogous to the parser (e.g. COB). Before a file is imported, the following message will appear: "*File %s will be converted from ETF into EasyCODE(XXX) format"*. This message will be suppressed when you add the *SuppressETFConvMsg=true* entry to the INI file. In the filter box, the "Export files (*.ETF)" entry will be added before "All files (*.*)".

File import is possible only when the import DLL *EASY-IMP.DLL* is stored in the EasyCODE installation directory, otherwise the following error message will appear*: "Dynamic Link Library %s not found or invalid!"*.

61 Inserting Imported Files

If an imported file is inserted into another file, the identification procedure is analogous to the opening of an imported file. EasyCODE will check whether the file has an internal format, export format (identified by the keyword EasyCODE) or source format (C, COB if you use the new parser, DS, SPX if you use the available parser). Information representing options and short info must, however, not be evaluated. The filters in the "Insert" dialog window are analogous to those in the "Open " dialog window.

62 Import Errors

In case of errors other than general problems such as lack of memory on the hard disk or other disk drives, which may occur anywhere and anytime, an ERR file is created which contains one or several warnings and/or error messages. Every message is structured as follows:

Line <number>: <error/warning>: <message text>

Errors:

- *Import not allowed*. Violation according to chapter: 65 Origin of the ETF File.
- *Keyword %s expected*. Violation according to chapter: 67 Check of Semantics.
- *Keyword incorrect*. The specified line contains some characters or words which cannot be interpreted as a keyword.
- *Entry exceeds maximum length*. The entry exceeds MAX_TEXT_SIZE[4].
- *Keyword %s will not be imported by EasyCODE(%s)*. Violation according to chapter: 66 Keyword Acceptance.
- *Unexpected keyword %s.* Violation according to chapter: 67 Check of Semantics.
- *Unexpected end of file.* Violation according to chapter: 67 Check of Semantics.

Warnings:

[4] This value is predefined by the parser interface (in *parse.h*). Currently, the maximum length is about 30000.

- *Entry concerning keyword %s is not empty.*
- Parameters of PROCEDURE DIVSION or CALL are too long and were cut off.[5]

For more detailled information please refer to the section on info strings in Appendix A

63 Naming Conventions for ETF Files

Since in case of import errors an ERR file will be created, the filename extension ERR is not allowed for ETF files which are imported or exported. This naming conventions are extended to all EasyCODE files, because when a file is opened, EasyCODE does not yet know whether the file will be imported or not.

64 Checking of ETF Files During Import

During file import, the following independent checks will be made.

65 Origin of the ETF File

Before a file is imported, the origin of the ETF file will be checked on the basis of the first entry, the EasyCODE keyword.

- All components can import their own 'exports'.
- SP/SPX can import files exported by all SE components except DS.

If none of these two statements can be applied, file import will not be started, and an error message will be displayed. (For details see chapter 62) If a user wants to avoid these requirements and modifies the first line in order to simulate that the ETF file has been created by a different component, the user him-/herself will be responsible for all undesirable consequences that may occur.

When SP/SPX files are imported into language components, no semantic conversion will occur. For this operation, the existing internal interface for transferring SP/SPX files will be useful.

66 Keyword Acceptance

When ETF files are imported, no matter where they come from, EasyCODE will check the keywords on the basis of control tables to make sure that only those keywords of the ETF file will be accepted which are available in the importing component. All other keywords will be rejected. For details see chapter 62

67 Check of Semantics

Although there is no thorough analysis or check of structure during file import, the constructs are checked for completeness and correctness in order to avoid extreme deviations. Excluded from this rule are those constructs which were made up from several internal structures: In particular, this relates to all derivatives of *SWITCH* and *SWITCH-SELECTOR* constructs. An IF-THEN as a branch of a *SWITCH* will, for example, not be rejected during file import. For details see chapter 62

[5] May occur only if V4.0 ETF files are imported.

68 Export

69 User Interface

To export a file, choose the menu item *FEXPAS:* "Export as..." in the File menu. The export directory and the default extension (**.ETF*)[6] are managed analogously to the *Generate as* path/extensions (INI file, Save settings...). The "*Export as..."* dialog window does not contain the *incl. Save* option or the *Short info* edit control.

The export function is not available in the DEMO version. If you try to export a file, the following error message will be displayed: *"The demo version cannot export"*.

70 Check of Semantics During Export

Since in various EasyCODE components the same constructs are (internally) used with different keywords, it is sometimes not advisable to create a definite identifier for a construct during file generation. The internal case construct, for example, is used as a SWITCH in SPX and SP, but as a CASE in COL and JET, and the internal cycle construct is used as a LOOP in SP and SPX, but as a CYCLE in COL and JET. In order not to confuse the user, while at the same time maintaining the interchangeability between the individual components, component-specific identifiers are generated for such constructs which will, however, lead back to the same construct when the file is imported. The alias identifiers may be exchanged regardless of the component. One of these identifiers must always be the default value, so that components which do not provide this construct at all, but are able to import it (in Russian, Turkish or Greek)or load it somehow, will also be able to export it. The default identifier is underlined in chapter *10 Constructs and Their ETF Format*.

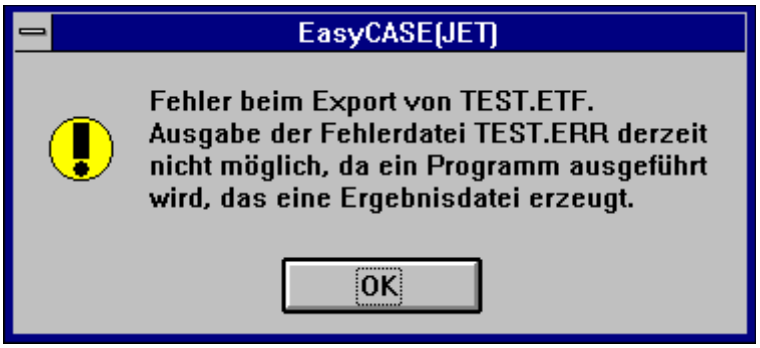
71 Export Errors

In addition to general problems such as lack of memory on the hard disk or other disk drives, which may occur anywhere and anytime, there are also export error messages which may occur in EasyCODE(JET) and EasyCODE(PET) only. These errors may occur because all info strings are checked before they are written into the ETF file. (For details see the chapter on info strings.)

In case of errors, the procedure is the same as during file generation in EasyCODE(JET) and EasyCODE(PET). The following message window

or the message

[6] *ETF*: Abbreviation of *EasyCODE Text Format*.



will appear, if the error file cannot be displayed because it is at the same time used by another program. The only error which may occur in V5.0 is the following: *SDF command/statement cannot be re-imported because of structural errors/syntax errors*.

Every error file has the filename extension *ERR*. Since the file is not automatically saved before it is exported, there may be constructs which have not been assigned construct numbers. When the errors are displayed, the construct numbers will be preceded by the prefix # (as in EasyCODE(JET), EasyCODE(PET)). In the structure diagram, the display of construct numbers is automatically switched on in case of errors.

72 Interface to Other Topics and Subsystems

73 Program Linking

In the *Program settings* dialog window, the *Export before Program Call* check box is available in addition to the *Generate before Program Call*[7] option. If both boxes are checked, the file will first be generated and/or saved and then exported before the program call.

In the *Program settings* dialog window you may, in addition to the existing symbols, also enter *%EXPDIR%*, *%EXPDRIVE%, %EXPEXT%, %EXPFILE%, %EXPNAME%, %EXPPATH%* into the individual editing boxes.

The user will be asked to specify the name of the export file only if the name has not yet been assigned (analogous to file generation).

74 File Transfer

If an export filename exists, the export file will also be included into the default transfer list (analogous to file generation).

75 Parser Interface

Since file import is based on the parser interface, all modifications concerning the parser interface may affect the import interface specified in this document.

76 Info Strings

For effects on the import/export interface see the chapter concerning info strings above.

[7] In COL and C/C++: save.

Appendix A - Info Strings

Special problems concerning the import/export interface may be caused by JET-specific structures which are not displayed in the tree structure of the structure diagram, including

1. BS2000 commands
2. BS2000 statements
3. Variants
4. Procedure attributes
5. conditions
6. Variables

A.1 Info String Layout

The conversion of the internal representation to the info string corresponding to the JET structure concerned must be bijective and must not affect in any way the values of both forms of representation.

A.2 Conventions & General Notes

A.2.1 Conventions Concerning Syntax Description

- Normal characters (without italics or boldface) indicate fixed text.
- Symbolic names, and nothing else, are written in italics and enclosed between < and >.
- Alternatives are enclosed between **{** and **}** and separated by "**|**".
- The meta-symbols { } | are written in bold characters.

A.2.2 Notes on the Info String Syntax

It is absolutely necessary to adhere to the sequence of entries, the line breaks and the syntax (therefore the info strings must be complete !), otherwise they will be rejected when the file is analyzed (during file import).

According to the syntax, the end of an editable section (represented by a symbolic name) must always be at the end of a line and is followed only by the line break (or even by the end of file). Thus, these sections are not subject to contents restrictions, because no special symbol or internal structure is required for identifying their ends. This is of vital importance because although the editable sections may contain any text, EasyCODE must be able to import every exported ETF file without any modifications.

Info string entries should, if possible, contain self-explanatory names which follow more or less the SDF (BS2000) terminology and are therefore quite long, which on the one hand makes the ETF file easier to read and understand, but on the other hand may considerably reduce the performance during file import and export. These two factors are to be weighed against each other.

The ETF interface should be a bit more restrictive than the internal format interface, which on the whole does nothing else but write the internal *tree* into a file when a structure diagram is saved and retrieve it without checking when the file is opened. Since the ETF interface will also allow manual modification, a certain amount of internal consistency should be guaranteed. This is why the structure of the info strings in some ways differs slightly from the structure of the internal EasyCODE files (*.JET, *.ECB,...). Example: POS and LEN data for JV/variable comparisons should only be exported (and

imported) if they are really required. Self-explanatory names (see above) for types and various fields.

A.3 BS2000 Command

node_typ: aktion_typ; akt_typ: bs2_typ; bs2struct_typ: bs2_cmd

-> Diplomarbeit (Roland)

A.4 BS2000 Statement

node_typ: aktion_typ; akt_typ: bs2_typ; bs2struct_typ: bs2_statement

-> Diplomarbeit (Roland)

A.5 Variant

node_typ: aktion_typ; akt_typ: variante_typ

NAME=<*variant name*>
VALUE=<*variant value*>

A.6 Procedure Attributes

node_typ: *jetproc_typ*. Inhalt des Felds: *attribute* of *jetproc_struc*

PROCEDURE-TYPE=**{**BATCH**|**DIALOG**}**
JOBNAME=<*jobname*>
USERID=<*userid*>
JOBACCOUNT=<*jobaccount*>
PASSWORD=<*password*>
JOB-CLASS=<*job-class*>
JOB-PRIO=<*job-prio*>
RUN-PRIO=<*run-prio*>
TIME=<*time*>
PARAMETERS=<*parameters*>
SYSTEM-OUTPUT=**{**PRINT**|**DELETE**|**TAPE-OUTPUT**}**
LOGGING=<*character*>
INTERRUPTION-ALLOWED=**{**YES**|**NO**}**
ESCAPECHAR=**{**&**|**@**|**#**|***|**%**}**
GENSTATUSJV=**{**YES**|**NO**}**
RESTARTLOGIC=**{**GEN**|**NOTGEN**|**NOTALLOWED**}**

A.7 Conditions

node_typ: cond_typ

{
COND-TYPE=EMPTY
|
COND-TYPE=FILE
NAME=<*filename*>
STATE=**{**CAT**|**NOTCAT**|**EMPTY**|**NOTEMPTY**}**
|
COND-TYPE=JV
LINK=**{**YES**|**NO**}**
NAME=<*jv-name or jv-link-name (depending on LINK=)*>
STATE=**{**CAT**|**NOTCAT**|**EMPTY**|**NOTEMPTY**}**
|
COND-TYPE=JVVARCOMP

OPERAND1={JV|JV-LINK|VAR}
<description of first operand (see description of second operand)>
OPERATOR={=|<>|<|>|<=|>=}
OPERAND2={JV|JV-LINK|CSTRING|XSTRING}
<description of second operand:
For OPERAND1 and OPERAND2 the following applies:
If: OPERANDx=JV or *JV-LINK*
LINK={YES|NO}
NAME=*<jv-name or jv-link-name (depending on operand type)>*
POS=*<pos-value>*
LEN=*<len-value>*
Falls OPERANDx=VAR
NAME=*<var-name>*
Falls OPERANDx=CSTRING
STRING=*<c-string>*
Falls OPERANDx=XSTRING
STRING=*<x-string>*
>
|
COND-TYPE=DATAPOOL
CALLID=*<Call ID>*
SELECTION-TYPE={NONE|FILENAME|FILELEN|VSN}
SELECTION=*<filename/filelength/VSN (for SELECTION-TYPE=NONE: empty/without any significance>*
DATA={AVAILABLE|NOTAVAILABLE}
|
COND-TYPE=JOBSWITCH
0={ON|OFF|-}
...
31={ON|OFF|-}
|
COND-TYPE=USERSWITCH
0={ON|OFF|-}
...
31={ON|OFF|-}
USERID=*<User ID>*
}

A.8 Variable

node_typ: *variable_typ*. Field contents: *std* of *variable_struc*

NAME=<variable name>
VALUE=<variable value>
POSVAR={YES|NO}

A.9 Conversion: Internal Structure <--> Info Strings

A.9.1 Import

During file import, the JET structure corresponding to the info string delivered at the import interface is reconstructed and written to the TMP file. In case of errors, the following messages are written to the error file:

1. In case of incorrect SDF commands or SDF statements: *SDF command/statement is incorrect*.
2. In all other cases: *JET structure is incorrect.*

For a more detailled description of import errors and how to proceed, see chapter 3.3 "Import Errors".

A.9.2 Export

All JET structures must be checked for correctness before a file can be exported. If an error is identified, the error messages mentioned in chapter 4.3 will be displayed. Otherwise, the JET structure will be converted into an info string, which will be written into the ETF file via export interface.

77 Index