# ETAC Compilation and Run-time Error Codes

28 April 2018



This document presents a description of all the error and warning symbolic codes that may occur while compiling or running an ETAC program. The **RED** headings indicate errors; the **BLUE** headings indicate warnings.

The following symbolic conventions are used in this document.

| Symbol | Meaning |
|---|---|
| ‹*x*› | separates *x* as a unit of information from the surrounding text. |
| … | represents omitted text (as usual). |
| ʷₛ | represents a whitespace character. |
| <EOF> | represents the end of the file data. |

## Error and Warning Code Reference

### ADDAP_A_BAD

Invalid destination token of addition reassignment statement, ‹… +=›.

**Details**

The destination of an addition reassignment statement ‹… +=› can only be a variable.

### ANDAP_A_BAD

Invalid destination token of ‘and’ reassignment statement, ‹… and=›.

**Details**

The destination of an ‘and’ reassignment statement ‹… and=› can only be a variable.

### BA_COCOMB

Invalid destination argument of compound object append statement, ‹… ++:=›.

**Details**

The destination argument of a compound object append statement ‹… ++:=› must evaluate to a compound object.

### BA_DO

Invalid argument for ‘do’ statement, ‹do …›.

### BA_EXITDOIF_DONEXTIF

Invalid argument for procedure exit statement, ‹exitdo_if› or ‹donext_if›.

### BA_SCACCESS

Invalid argument for string character access expression, ‹…#…›.

### BA_SCASN

Invalid argument for string character assignment statement, ‹…#… :=›.

### BA_SCLACCESS

Invalid argument for string (last) character access expression, ‹…#*›.

### BA_SCLASN

Invalid argument of string (last) character assignment statement, ‹…#* :=›.

**Details**

The argument of a string (last) character assignment statement ‹…#* :=› can only be a variable or a data member

name containing a string.

### BA_SEACCESS

Invalid argument for sequence element access expression, ‹…%…›.

### BA_SEQAP

Invalid destination argument of sequence append statement, ‹… +:=›.

**Details**

The destination argument of a sequence append statement ‹… +:=› must evaluate to a sequence or procedure.

### BAD_ARG

Invalid destination argument of an assignment or reassignment statement, or of an increment or decrement expression.

**Details**

An assignment statement ‹…<%… :=›, ‹…<%%… :=›, ‹… @=›, ‹…%… :=›, or reassignment statement ‹… +=›, ‹… -=›, ‹… *=›, ‹… /=›, ‹… and=›, ‹… or=›, or an increment ‹…++› or decrement ‹…--› statement has an invalid argument.

### BAD_CMD

A script command has an invalid syntax.

### BAD_COMMA_POS

Illegal position of comma token ‹,› within sequence or procedure.

**Details**

A comma token may not exist after the opening bracket, or before the closing bracket of a sequence or procedure. The following examples are syntactically invalid ‹[,…]›, ‹[…, ]›, ‹{ ,…}›, ‹{…,}›, ‹[,…,]› because a (highlighted) comma exists after the opening bracket, or before the closing bracket of a sequence or procedure (white-spaces are not relevant). However, the following examples are syntactically valid within a procedure: ‹`[,…,`]›, ‹`{,…,`}› because they are not actual sequences or procedures. They are label commands to create a sequence and a procedure when the procedure in which they exist is executed.

### BAD_COMMA_SCOPE

Illegal scope of comma token ‹,› outside sequence or procedure.

**Details**

Comma tokens may exist only inside sequences or procedures. They must not exist outside all sequences and procedures.

### BAD_DEC

A decimal number is badly formed.

### BAD_END_OPR_BRAK

Invalid delimiter following end operator bracket ‹)›.

**Details**

The character following an end operator bracket must be one of the following delimiter characters: $^{W}_{S}$, ‹)›, ‹]›, ‹}›, ‹,›, ‹;›. The end operator bracket (highlighted) in the following example is not followed by one of the valid delimiter character:
‹(A + B)add2 3 5› (invalid – perhaps there ought to be a space or semicolon after the end operator bracket).

### BAD_END_PROC_BRAK

Invalid delimiter following end procedure bracket ‹}›.

**Details**

The character following an end procedure bracket must be one of the following delimiter characters: $^{W}_{S}$, ‹)›, ‹]›, ‹}›, ‹,›, ‹;›. The end procedure bracket (highlighted) in the following example is not followed by one of the valid

delimiter character:
‹{…}add2 3 5› (invalid – perhaps there ought to be a space or semicolon after the end procedure bracket).

**BAD_END_SEQ_BRAK**

The character following an end sequence bracket ‹]› is not a valid delimiter.

**Details**

The character following an end sequence bracket must be one of the following delimiter characters: ʷₛ, ‹)›, ‹]›, ‹}›, ‹,›, ‹;›. The (highlighted) end sequence bracket in the following example is not followed by one of the valid delimter character:
‹[3, 9, "hello", 2.7]add2 3 5› (invalid – perhaps there ought to be a space or semicolon after the end sequence bracket).

**BAD_INCL_FILE**

Invalid inclusion file specified via the preprocessor inclusion directive ‹#include …›.

**Details**

An inclusion file could be invalid either because it is a binary TAC file or not an E\TAC script file.

**BAD_INSTR**

A text instruction has an invalid syntax.

**BAD_INT**

An integer is badly formed.

**BAD_LABEL**

Invalid label command ‹`*a*›.

**Details**

A script label command ‹`*a*› has an invalid syntax *a*.

**BAD_MARK_VAL**

Invalid mark object ‹!*n*›.

**Details**

A mark object ‹!*n*› has an invalid value (*n* must be an integer from 0 to 7).

**BAD_MEM_VAL**

Invalid memory object ‹&*a*› syntax.

**Details**

A memory object script command ‹&*a*› has an invalid syntax *a* (*a* must be a non-negative integer or a hexadecimal string beginning with ‹0h›).

**BAD_OBJ_TYPE**

A *comop* determined that a required stack object was not of the correct type.

**BAD_OBJ_VAL**

A *comop* determined that a required stack object was of the correct type but had an inappropriate value.

**BAD_OPR**

Invalid script operator ‹&*o*›.

**Details**

A script operator ‹&*o*› has an invalid syntax *o*.

## BAD_PP_COMMAND

Undefined preprocessor command.

**Details**

Valid preprocessor commands are: ‹::define›, ‹::include›, ‹::ifdef›, ‹::then›, ‹::elsedef›, ‹::else›, and ‹::endif›. Anything else beginning with two colons ‹::› will cause this error.

## BAD_PP_OPERAND_NUM

Incorrect number of operands in conditional preprocessor directive.

**Details**

The ‹&and› and ‹&or› operators in the condition of a conditional preprocessor directive (‹::ifdef› or ‹::elsedef›) must have two or more operands. The ‹&not› operator must have exactly one operand. For example,

‹&and ( Name1 Name2 Name3 )›

is syntactically valid, while

‹&and ( Name )›

is invalid. Notice the required space around the parentheses.

## BAD_PP_OPR

Undefined operator in conditional preprocessor directive.

**Details**

The only valid operators in the condition of a preprocessor conditional directive (‹::ifdef› or ‹::elsedef›) are: ‹&and›, ‹&or›, and ‹&not›. This error message will occur if an ampersand ‹&› in the condition is not followed by one of the three operator names. The operators in the following examples are invalid: ‹&AND›, ‹&And›, ‹&+›, ‹&go›, ‹& or›, ‹&or( ›.

## BAD_PP_THEN

Invalid delimiter for preprocessor ‹::then› command.

**Details**

At least one white-space must follow a preprocessor ‹::then› command. For example,

‹::thenABC›

is syntactically invalid.

## BAD_PPNAME

A *preprocessor definition name* contains invalid characters or is badly formed.

**Details**

A PDN (*preprocessor definition name*) must begin and end with a colon (:). Any printable characters can be used between the colons except any of the following delimiter characters or colons. A PDN must be right-delimited by one of the following delimiter characters: ‹$_s^w$›, ‹)›, ‹]›, ‹}›, ‹,›, ‹;›. For example:

**(1)** ‹:MyPPName:sub2 …› is an invalid PDN because the (highlighted) character s is not an appropriate delimiter (the PDN is supposed to be ‹:MyPPName:›).

**(2)** ‹:MyPP:Name:› is an invalid PDN because it contains a (highlighted) colon.

**(3)** ‹:My PP Name:› is an invalid PDN because it contains spaces.

**(4)** ‹:MyPPName …› is an invalid PDN because it does not have an ending colon (the PDN is supposed to be ‹:MyPPName:›).

**(5)** ‹:=10; …› is an invalid PDN because it does not have an ending colon before the (highlighted) semicolon (perhaps it was meant to be part of an assignment statement, which requires at least one white-space after the assignment symbol, as in ‹:=$_s^w$10; …›).

## BAD_SEMICOLON_SCOPE

Illegal scope of semicolon token ‹;› (terminator token) within expression or sequence.

**Details**

Terminator tokens ‹;› may not be enclosed within TAC or ETAC operator expressions ‹(…)› or sequences ‹[…]›. Nor may they be enclosed within TAC procedures ‹{…}›. A procedure is a TAC procedure when the script file containing the procedure is not identified as an ETAC script file. Terminator tokens may, and usually do, exist within ETAC procedures.

## BAD_STRING

A quoted string is badly formed (all strings in E\TAC are double-quoted).

**Details**

This error could be caused by:

**(1)** A string beginning but not ending with a quote character. For example, ‹"my string; add2 A B;<EOF>› is invalid because the string was not terminated before the end of the file (perhaps an ending quote character ought to be present before the first semicolon). Note that if the ending quote character of a string is missing, the next found quote character, which could be the beginning quote character of another string, is taken as the ending quote character. This situation could cause spurious error events.

**(2)** An invalid escape sequence within a string. For example, **(a)** ‹"my string\N"›. The ‹\N› within the string is invalid (perhaps it ought to be ‹\n›); **(b)** ‹"my string\"›. An ending quote character must not be preceded by a backslash ‹\›. To indicate a string ending with a backslash, a space must follow the backslash, as in the following example: ‹"my string\ "›. ‹\ › represents a single backslash within a string.

**(3)** An inappropriate delimiter after the ending quote character of a string. For example, ‹"my string"add2› is invalid because the ending quote character (highlighted) is not followed by an appropriate delimiter (perhaps a space or semicolon ought to be present after the ending quote character). The delimiter following an ending quote character of a string must be one of the following characters: $^{WS}$, ‹)›, ‹]›, ‹}›, ‹,›, ‹;›.

**(4)** An appropriately delimited quoted string does not follow a preprocessor inclusion command ‹::include›. For example, **(a)** ‹::include "My File"add2› is invalid because the ending quote character (highlighted) is not followed by an appropriate delimiter; **(b)** ‹::include 'My File"› is invalid because the beginning quote character (highlighted) is not a double-quote character; **(c)** ‹::include My "File"› is invalid because the next non-whitespace character (highlighted) after the inclusion command is not a double-quote character. The argument of an inclusion command must be a double-quoted string delimited by at least one white-space on each side.

## BAD_SYNTAX

Invalid syntax for object or sequence insertion statement, or for a sequence assignment or append statement.

**Details**

An object insertion statement ‹…<%… :=›, or a sequence insertion statement ‹…<%%… :=›, or a sequence element assignment statement ‹…%… :=›, or a sequence element append statement ‹…%… +=› has an invalid syntax, namely, the ‹:=› or ‹+=› token is not present where it is expected.

## BAD_TERM_TOK

Missing last terminator token ‹;› in script.

**Details**

An E\TAC script file must end in a semicolon (terminator token).

## BS_DO

Invalid syntax for ‘do’ statement, ‹do …›.

## BS_EXITDO_DONEXT

Invalid syntax for procedure exit statement, ‹exitdo› or ‹donext›.

**BS_FNT**

Invalid syntax for function definition statement ‹fnt:...›.

**BS_FNTCALL**

Invalid syntax for function call expression ‹... ()›.

**BS_IF**

Invalid syntax for 'if' statement, ‹if...endif›.

**BS_SCASN**

Invalid syntax for string character assignment statement, ‹...#... :=›.

**BS_SCLASN**

Expected assignment token ‹:=› in string (last) character assignment statement, ‹...#* :=›.

**Details**

A string (last) character assignment statement requires an assignment token ‹:=› (as in: ‹...#* :=›).

**BS_SELN**

Invalid syntax for data member selection expression ‹... . ...› chain.

**BS_WHEN**

Invalid syntax for 'when' statement, ‹when...endwhen›

**COCOMB_A_BAD**

Invalid destination argument of compound object append statement, ‹... ++:=›.

**Details**

The destination argument of a compound object append statement ‹... ++:=› must evaluate to a compound object.

**COPYASN_A_BAD**

Invalid destination token of copy assignment, ‹... @=›.

**Details**

The destination of a copy assignment ‹... @=› can only be a variable.

**DATA**

Expected procedure for data definition statement, ‹data:...›.

**Details**

A data definition statement must have an explicit procedure following it (as in: ‹data:{...}›).

**DATA_A_NONE**

Expected procedure for data definition statement, ‹data:...›.

**Details**

A data definition statement must have an explicit procedure following it (as in: ‹data:{...}›).

**DEC_A_BAD**

Invalid argument of decrement statement, ‹...--›.

**Details**

The argument of a decrement statement ‹...--› can only be a variable or data member containing an integer.

## DIVAP_A_BAD

Invalid destination token of division reassignment statement, ‹… /=›.

**Details**

The destination of a division reassignment statement ‹… /=› can only be a variable.

## DO_BODY_NONE

Expected procedure of ‘do’ statement, ‹do …›.

**Details**

A ‘do’ statement must have an explicit procedure following the “do” options (as in: ‹do [repeat […]] {…}› or ‹do […] [from …] [to …] [step …]] [with … of …] [while …] {…}›, where [*X*] indicates that *X* is optional).

## DO_REPA_BAD

Invalid ‹repeat› argument of a ‘do’ statement, ‹do …›.

**Details**

The optional ‹repeat› argument of a ‘do’ statement must evaluate to an integer.

## DOFOR_FROMA_BAD

Invalid ‹from› argument of ‘do’ statement, ‹do …›.

**Details**

The ‹from› argument of a ‘do’ statement must evaluate to an integer.

## DOFOR_STEPA_BAD

Invalid ‹step› argument of ‘do’ statement, ‹do …›.

**Details**

The ‹step› argument of a ‘do’ statement must evaluate to an integer.

## DOFOR_TOA_BAD

Invalid ‹to› argument of ‘do’ statement, ‹do …›.

**Details**

The ‹to› argument of a ‘do’ statement must evaluate to an integer.

## DOFOR_VAR_BAD

Expected variable in ‘do’ statement, ‹do …›.

**Details**

The “for” clause of a ‘do’ statement must begin with a variable.

## DONEXT_SC_NONE

Expected semicolon ‹;› following ‘donext’ statement.

**Details**

An ‘donext’ statement must have a semicolon following it (as in: ‹donext;›).

## DONEXTIF_A_BAD

Invalid ‹donext_if› argument.

**Details**

A ‹donext_if› argument must evaluate to an integer interpreted as a boolean value.

## DOWHILE_A_BAD

Invalid ‹while› argument of a ‘do’ statement, ‹do …›.

**Details**

The ‹while› argument of a ‘do’ statement must evaluate to an integer interpreted as a boolean value.

## DOWITH_OF_NONE

Expected ‹of› token after ‹with› variable of ‘do’ statement, ‹do …›.

**Details**

A ‘do’ statement must have an ‹of› token after the ‹with› variable (as in: ‹do … with … of …›).

## DOWITH_OFA_BAD

Invalid ‹of› argument of “with” clause in ‘do’ statement, ‹do …›.

**Details**

The ‹of› argument of the “with” clause in a ‘do’ statement must evaluate to a sequence or procedure.

## DOWITH_VAR_BAD

Expected variable after ‹with› token of ‘do’ statement, ‹do …›.

**Details**

A variable is required after the ‹with› token of a ‘do’ statement, ‹do …›.

## DUPL_COMMA

Illegal duplication of comma token ‹,›.

**Details**

A comma token may not exist after another one. It is invalid to have one comma following another with only white-space or no space in-between. The following examples are syntactically invalid ‹[…, ,…]›, ‹{…,,…}› because they contain a (highlighted) comma following another with only white-space or no space in-between. Note that semicolons ‹;› within an ETAC procedure (the usual case) are internally converted into commas, so this error message may occur if a semicolon follows another in an ETAC procedure. A procedure is an ETAC procedure when the script file containing the procedure is identified as an ETAC script file.

## EMPTY_STACK

An attempt was made to access an object from an empty stack.

## EXITDO_SC_NONE

Expected semicolon ‹;› following ‘exitdo’ statement.

**Details**

An ‘exitdo’ statement must have a semicolon following it (as in: ‹exitdo;›).

## EXITDOIF_A_BAD

Invalid ‹exitdo_if› argument.

**Details**

An ‹exitdo_if› argument must evaluate to an integer interpreted as a boolean value.

## EXPECTED_CLOSE_BRAK

Expected closing parenthesis ‹)› in conditional preprocessor directive.

**Details**

An ‹&and› or ‹&or› operator in a conditional preprocessor directive (‹::ifdef› or ‹::elsedef›) must be followed by at least one white-space then the operands of the operator enclosed within white-space delimited parentheses. For example,

```
‹&and ( Name1 Name2 Name3 )›
```

is syntactically valid. Notice the required space around the parentheses. Note that a missing ending parenthesis does not necessarily result in this error message — a different error message can occur instead.

## EXPECTED_OPEN_BRAK

Expected opening parenthesis ‹(› in conditional preprocessor directive.

**Details**

An ‹&and› or ‹&or› operator in a conditional preprocessor directive (‹::ifdef› or ‹::elsedef›) must be followed by at least one white-space then the operands of the operator enclosed within white-space delimited parentheses. For example,

```
‹&and ( Name1 Name2 Name3 )›
```

is syntactically valid. Notice the required space around the parentheses.

```
‹&and (Name1 Name2 Name3 )›
```

is invalid because the opening parenthesis is not followed by a white-space.

## EXPECTED_THEN

Excess conditions in conditional preprocessor directive.

**Details**

There should be exactly one condition each for a preprocessor ‹::ifdef› or ‹::elsedef› command. A ‹::then› command should be where the second condition is (the second and subsequent conditions are syntactically invalid). For example,

```
‹::ifdef Name1 Name2 ::then ... ::endif›
```

is syntactically invalid because the ‹::then› command should exist in place of the (highlighted) condition ‹Name2›. The example is made syntactically valid as follows

```
‹::ifdef Name1 ::then ... ::endif›.
```

This error condition typically occurs when a space is mistakenly inserted into a condition consisting of a single name, as in the following (syntactically invalid) example.

```
‹::ifdef FILE NAME ::then ... ::endif›
```

The programmer intended to write

```
‹::ifdef FILENAME ::then ... ::endif›.
```

## FEW_SEQ_ELMS

An attempt was made to access a non-existent sequence element that should have existed at a specified index.

## FEW_STACK_ARGS

A *comop* requires more arguments than were found on the relevant stack.

## FNT_A1_NONE

Expected opening parenthesis ‹(› for function definition statement, ‹fnt:...›.

**Details**

A function definition statement must have an opening parenthesis following the ‹fnt:› token (as in: ‹fnt: (...) ...›).

## FNT_A2_NONE

Expected procedure for function definition statement, ‹fnt:...›.

**Details**

A function definition statement must have an explicit procedure following the closing parenthesis (as in: ‹fnt: (...) {...}›).

## FNT_AP_BAD

Invalid parameters for function definition statement, ‹fnt:...›.

**Details**

The parameters of a function definition statement must be explicit variable names (as in: ‹fnt: (*par* ···) ...›, where *par* represents a parameter).

**FNTCALL_A1_BAD**

Invalid variable for function call expression, ‹... ()›.

**Details**

A function call expression must have an explicit variable name preceding the parenthesis.

**FNTCALL_A2_NONE**

Expected opening parenthesis ‹(› for function call expression, ‹... ()›.

**Details**

A function call expression must have an opening parenthesis immediately following the variable name.

**GENERAL_ERR**

A non-specific error event occurred.

**IF_A_NONE**

Missing condition for ‘if’ statement, ‹if...endif›.

**Details**

An ‘if’ statement must have at least one condition (followed by the ‹then› token) after the ‹if› token.

**IF_AC_BAD**

Invalid condition of ‘if’ statement, ‹if...endif›.

**Details**

Each condition of an ‘if’ statement must evaluate to an integer interpreted as a boolean value.

**IF_B_NONE**

Unmatched ‹endif› of ‘if’ statement, ‹if...endif›.

**Details**

An ‘if’ statement must begin with the ‹if› token.

**IF_BAD**

Invalid ‘if’ statement structure, ‹if...endif›.

**Details**

An ‘if’ statement must have the following structure: ‹if *([not] ... then {...}) [else {...}] endif›, where *(*X*) indicates one or more *X*, and [*X*] indicates that *X* is optional.

**IF_E_NONE**

Unmatched ‹if› of ‘if’ statement, ‹if...endif›.

**Details**

An ‘if’ statement ‹if...endif› must end with the ‹endif› token.

**INC_A_BAD**

Invalid argument of increment statement, ‹...++›.

**Details**

The argument of an increment statement ‹...++› can only be a variable or data member containing an integer.

**INVALID_LABEL**

Invalid destination for an allocation ‹... :-› or assignment ‹... :=› statement.

**Details**

In a TAC script file, the destination token of an allocation or assignment statement must be in the form of a command name or label command name. The destination is not an actual command or label command that is executed. The destination is internally converted to a string as appropriate for an allocation or assignment command. However if the destination is not in the form of a command name or label command name, for example it might be a string, then this error message will occur. For example, the following are syntactically invalid:

‹"Var" :- 10;›, ‹&Var := "string";›, ‹[3, 2] := 3.3;› because the (highlighted) destination of the allocation or assignment statement is not in the form of a command name or label command name. The following example ‹`Var := "string";› is syntactically valid because the destination of the assignment statement is in the form of a label command. However, the destination is typically in the form of a command, not a label command.

In an ETAC script file, the destination token of an allocation or assignment statement may be other than in the form of a command name or label command name.

## LIBERR

Unhandled programmer-initiated external TAC library error.

**Details**

A function in an external TAC library was called (typically via `@ImportLib`), and returned an error which was not handled by the caller. Functions in external TAC libraries are custom functions implemented with the C++ programming language, and created by external manufacturers, but can be called from ETAC code. Such a function can return custom errors created by the external manufacturer. If those custom errors are not handled by the caller of the function, this error event will occur (this error event can still occur even if the custom errors are handled by the caller). The caller can define `@OnLibErr()` in the data object returned by `@ImportLib` to handle this error event.

The following code illustrates the essentials of how to handle a custom error returned from a function called in an external TAC library.

```
MyLibDataObj :- @ImportLib "ExternalTACLib.dll";
MyLibDataObj.@OnLibErr := fnt:(pFntOrd[*int*] pErrCode[*int*] pDataSeq[*seq*])
{
   ... [* Your code to handle the error goes here. *]
   exit_err :#ETP_RTN_LIBERR:; [* Optional if you want to produce this error
event and end the ETAC session. *]
};
MyLibDataObj.LibFnt();
```

The italic sections are replaced with your own appropriate text. `@OnLibErr()` is automatically executed if a function in *MyLibDataObj* (eg: `LibFnt()`) returns `ccETP_RTN_LIBERR` when called.

## MISMATCHED_BRAKS

Unmatched sequence, procedure, or operator bracket.

**Details**

At least one sequence, procedure, or operator bracket is not matched with its corresponding bracket. Sequence brackets are ‹[› and ‹]›, procedure brackets are ‹{› and ‹}›, and operator brackets are ‹(› and ‹)›.

## MISSING_ARG

Missing size expression ‹|...|› argument.

**Details**

This error message occurs when an ETAC size expression has no argument. For example, ‹| |› and ‹||› are syntactically invalid because each has no argument.

## MISSING_ENDIF

Unmatched ‹::ifdef› in conditional preprocessor directive.

**Details**

Each ‹::ifdef› command in a conditional preprocessor directive must correspond with an ‹::endif› command. This implies that there must be the same number of ‹::ifdef› commands as there are ‹::endif› commands. This error message occurs if there is an ‹::ifdef› command that does not correspond with an ‹::endif› command.

**MULTAP_A_BAD**

Invalid destination token of multiplication reassignment statement, ‹… *=›.

**Details**

The destination of a multiplication reassignment statement ‹… *=› can only be a variable.

**NEG**

Invalid negation expression ‹~…› argument.

**Details**

A negation expression argument must evaluate to an integer or decimal number.

**NEG_A_BAD**

Invalid negation expression ‹~…› argument.

**Details**

A negation expression argument must evaluate to an integer or decimal number.

**NO_APP_FNT**

No application call-back function was specified.

**Details**

An attempt was made to call a function defined in a user-designed application program via the ‘`run_app_fnt`’ TAC command in an ETAC or TAC code file, but that function was not specified to the ETAC for Applications program (`AppETAC.dll`) within the user-designed application program. This error event could occur if the user-designed application program was improperly designed. See the provider of the user-designed application program for an update.

This error event will also occur if `run_app_fnt` is executed in an ETAC or TAC code file via the Run ETAC Scripts program (`RunETAC.exe`). The Run ETAC Scripts program is not designed to execute functions in a user-designed application program.

**NO_DICT_ITEM**

A dictionary item name was searched for on the dictionary stack but was not found.

**NO_INCL_FILE**

Inclusion file specified via preprocessor inclusion directive ‹`#include` …› not found.

**NO_LIB_ACCESS**

Function not accessible in external TAC library.

**Details**

An attempt was made to call a function in an external TAC library, but that function was not accessible even though the library was successfully loaded. This error event could occur if the library was improperly designed. See the provider of the library for an update.

This error event can also occur if an attempt was made to call a function belonging to an external TAC library that was unloaded, and subsequently the same or a different library was loaded before the call. When a library is unloaded (typically via `@Release()`, or sometimes `unload_lib`), any references to the library’s functions become permanently invalid, even if the same library is loaded again.

**NO_LOADED_LIB**

External TAC library not loaded.

**Details**

An attempt was made to call a function in an external TAC library, but that library was not loaded or was previously loaded then unloaded. An external TAC library needs to remain loaded (typically via `@ImportLib`, or sometimes `load_lib`) before any of its functions can be run.

## NO_PP_THEN

Expected ‹::then› in conditional preprocessor directive.

**Details**

The condition of a conditional preprocessor directive (‹::ifdef› or ‹::elsedef›) must be followed by the ‹::then› command. For example,

‹::ifdef Name ... ::endif›

is syntactically invalid because the ‹::then› command does not follow the condition ‹Name›. The example is made syntactically valid as follows

‹::ifdef Name ::then ... ::endif›.

## NOT_A_CMD

Intrinsic operator specified as a command.

**Details**

An intrinsic operator must be specified as an operator not as a command. The following two examples, ‹add› and ‹CMD:add›, are syntactically invalid because they are operators not commands, and should be expressed as ‹&add› and ‹OPR:add›, respectively.

## NOT_A_CMD_LBL

Intrinsic label operator specified as a label command.

**Details**

An intrinsic label operator must be specified as a label operator not as a label command. The following two examples, ‹`add› and ‹LBC:add›, are syntactically invalid because they are label operators not label commands, and should be expressed as ‹`&add› and ‹LBO:add›, respectively.

## NOT_AN_OPR

Intrinsic command specified as an operator.

**Details**

An intrinsic command must be specified as a command not as an operator. The following two examples, ‹&copy› and ‹OPR:copy›, are syntactically invalid because they are commands not operators, and should be expressed as ‹copy› and ‹CMD:copy›, respectively.

## NOT_AN_OPR_LBL

Intrinsic label command specified as a label operator.

**Details**

An intrinsic label command must be specified as a label command not as a label operator. The following two examples, ‹`&copy› and ‹LBO:copy›, are syntactically invalid because they are label commands not label operators, and should be expressed as ‹`copy› and ‹LBC:copy›, respectively.

## NOT_LIB

Library is not recognised as an external TAC library.

**Details**

An attempt was made to load an external TAC library (typically via `@ImportLib`, or sometimes `load_lib`), but that library (DLL) was not recognised as an external TAC library. This error event occurs when the DLL was in fact loaded but was not found to be an external TAC library. Specifying a non external TAC library as the argument for `@ImportLib` or `load_lib` will cause this error event.

## OBJINS_A1_BAD

Invalid left argument of object insertion statement, ‹...<%... :=›.

**Details**

The left argument of an object insertion statement must evaluate to a sequence or procedure.

**OBJINS_A2_BAD**

Invalid right argument of object insertion statement, ‹…<%… :=›.

**Details**

The right argument of an object insertion statement must evaluate to a flat integer sequence.

**OBJINS_ASN_BAD**

Expected assignment token ‹:=› in object insertion statement, ‹…<%… :=›.

**Details**

An object insertion statement requires an assignment token ‹:=› (as in: ‹…<%… :=›).

**ORAP_A_BAD**

Invalid destination token of ‘or’ reassignment statement, ‹… or=›.

**Details**

The destination of an ‘or’ reassignment statement ‹… or=› can only be a variable.

**RES_WRD**

Invalid use of reserved word.

**SCACCESS_A1_BAD**

Invalid left argument of a string character access expression, ‹…#…›.

**Details**

The left argument of a string character access expression must evaluate to a string.

**SCACCESS_A2_BAD**

Invalid right argument of a string character access expression, ‹…#…›.

**Details**

The right argument of a string character access expression ‹…#…› must evaluate to an integer.

**SCASN_A1_BAD**

Invalid destination argument of string character assignment statement, ‹…#… :=›.

**Details**

The left argument of a string character assignment statement ‹…#… :=› can only be a variable or a data member name containing a string.

**SCASN_A2_BAD**

Invalid right argument of string character assignment statement, ‹…#… :=›.

**Details**

The right argument of a string character assignment statement ‹…#… :=› must evaluate to an integer, but cannot be a sequence access ‹…%…› or member selection ‹… . …› expression.

**SCLACCESS_A_BAD**

Invalid argument of a string (last) character access expression, ‹…#*›.

**Details**

The left argument of a string (last) character access expression must evaluate to a string.

**SCLASN_A_BAD**

Invalid argument of string (last) character assignment statement, ‹…#* :=›.

**Details**

The argument of a string (last) character assignment statement ‹…#* :=› can only be a variable or a data member name containing a string.

## SCLASN_ASN_BAD

Expected assignment token ‹:=› in string (last) character assignment statement, ‹…#*  :=›.

**Details**

A string (last) character assignment statement requires an assignment token ‹:=› (as in: ‹…#*  :=›).

## SEACCESS_A1_BAD

Invalid left argument for sequence element access expression, ‹…%…›.

**Details**

The left argument of a sequence element access expression must evaluate to a sequence or procedure.

## SEACCESS_A2_BAD

Invalid right argument for sequence element access expression, ‹…%…›.

**Details**

The right argument of a sequence element access expression ‹…%…› must evaluate to a flat integer sequence.

## SEAP_A1_BAD

Invalid left argument of a sequence element append statement, ‹…%…  +=›.

**Details**

The left argument of a sequence element append statement ‹…%…  +=› must evaluate to a sequence or procedure.

## SEAP_A2_BAD

Invalid right argument of a sequence element append statement, ‹…%…  +=›.

**Details**

The right argument of a sequence element append statement ‹…%…  +=› must evaluate to a flat integer sequence.

## SEAP_ASN_BAD

Expected append token ‹+=› in sequence element append statement, ‹…%…  +=›.

**Details**

A sequence element append statement requires an append token ‹+=› (as in: ‹…%…  +=›).

## SEASN_A1_BAD

Invalid left argument of a sequence element assignment statement, ‹…%…  :=›.

**Details**

The left argument of a sequence element assignment statement ‹…%…  :=› must evaluate to a sequence or procedure.

## SEASN_A2_BAD

Invalid right argument of a sequence element assignment statement, ‹…%…  :=›.

**Details**

The right argument of a sequence element assignment statement ‹…%…  :=› must evaluate to a flat integer sequence.

## SEASN_ASN_BAD

Expected assignment token ‹:=› in sequence element assignment statement, ‹…%…  :=›.

**Details**

A sequence element assignment statement requires an assignment ‹:=› token (as in: ‹…%…  :=›).

## SELN_A1_BAD

Invalid first argument of a data member selection expression, ‹….…› chain.

**Details**

The first argument of a data member selection expression ‹….…› chain must evaluate to a data object.

**SELN_AML_BAD**

Invalid middle or last argument of data member selection expression ‹... . ...› chain.

**Details**

Each middle argument of a data member selection expression ‹... . ...› chain must be a variable, procedure, function call expression ‹... ( ) ›, or sequence element access expression ‹...%...›, and the last argument must be a variable, procedure, function call expression, sequence element access expression, or variable followed by increment\decrement tokens ‹++|--›.

**SEQAP_A_BAD**

Invalid destination argument of sequence append statement, ‹... +:=›.

**Details**

The destination argument of a sequence append statement ‹... +:=› must evaluate to a sequence or procedure.

**SEQINS_A1_BAD**

Invalid left argument of sequence insertion statement, ‹...<%%... :=›.

**Details**

The left argument of a sequence insertion statement ‹...<%%... :=› must evaluate to a sequence or procedure.

**SEQINS_A2_BAD**

Invalid right argument of sequence insertion statement, ‹...<%%... :=›.

**Details**

The right argument of a sequence insertion statement ‹...<%%... :=› must evaluate to a flat integer sequence.

**SEQINS_ASN_BAD**

Expected assignment token ‹:=› in sequence insertion statement, ‹...<%%... :=›.

**Details**

A sequence insertion statement requires an assignment token ‹:=› (as in: ‹...<%%... :=›).

**SIZE**

Invalid size ‹| ... |› expression argument.

**Details**

A size expression argument must not be an integer or decimal number and cannot contain commas or semicolons at the top level.

**SIZE_A_BAD**

Invalid size ‹| ... |› expression argument.

**Details**

A size expression argument must not be an integer or decimal number and cannot contain commas or semicolons at the top level.

**SIZE_A_NONE**

Missing size expression ‹| ... |› argument.

**Details**

This error message occurs when an ETAC size expression has no argument. For example, ‹| |› and ‹| |› are syntactically invalid because each has no argument.

**SUBAP_A_BAD**

Invalid destination token of subtraction reassignment statement, ‹... -=›.

**Details**

The destination of a subtraction reassignment statement ‹... -=› can only be a variable.

## TOO_MANY_ENDIF

Unmatched ‹::endif› in conditional preprocessor directive.

**Details**

Each ‹::ifdef› command in a conditional preprocessor directive must correspond with an ‹::endif› command. This implies that there must be the same number of ‹::ifdef› commands as there are ‹::endif› commands. This error message occurs if there is an ‹::endif› command that does not correspond with an ‹::ifdef› command.

## UNDEF_PPNAME

Undefined preprocessor definition name.

**Details**

A *preprocessor definition name* was not defined (or was inaccessible) via the ‹::define› preprocessor directive or the command-line keyword, ‹PPDEFS=*pp-defs*›.

## UNDEFINED_CUST

Undefined *custom comop* numeric identifier.

**Details**

A *custom comop* numeric identifier is not defined in any loaded comop DLL.

## UNEXPECTED_ELSE

Unexpected ‹::else› in conditional preprocessor directive.

**Details**

The ‹::else› command of a conditional preprocessor directive must be after the ‹::then› command of the last ‹::ifdef› or ‹::elsedef› command of that directive. For example,

‹::ifdef Name ::then … ::else … ::else … ::endif›

is syntactically invalid because the (highlighted) ‹::else› command is not after the last ‹::ifdef› or ‹::elsedef› command. Instead, it is after another ‹::else› command. There must be no more than one ‹::else› command in a conditional preprocessor directive, so the example is made syntactically valid as follows

‹::ifdef Name ::then … ::else … ::endif›.

This error message can also occur if an ‹::else› command is outside of all conditional preprocessor directives.

## UNEXPECTED_ELSEDEF

Unexpected ‹::elsedef› in conditional preprocessor directive.

**Details**

An ‹::elsedef› command of a conditional preprocessor directive must be after the ‹::then› command of the last ‹::ifdef› or ‹::elsedef› command of that directive. For example,

‹::ifdef Name1 ::then … ::else … ::elsedef Name2 ::then … ::endif›

is syntactically invalid because the (highlighted) ‹::elsedef› command is not after the ‹::then› command of the last ‹::ifdef› or ‹::elsedef› command. Instead, it is after the ‹::else› command. The example is made syntactically valid as follows

‹::ifdef Name1 ::then … ::elsedef Name2 ::then … ::else … ::endif›.

This error message can also occur if an ‹::elsedef› command is outside of all conditional preprocessor directives.

## UNEXPECTED_THEN

Unexpected ‹::then› in conditional preprocessor directive.

**Details**

A ‹::then› command of a conditional preprocessor directive must be after the ‹::ifdef› command or each ‹::elsedef› command of that directive. For example,

‹::ifdef Name ::then … ::else … ::then … ::endif›

is syntactically invalid because the (highlighted) ‹::then› command is not after the last ‹::ifdef› or

‹::elsedef› command. Instead, it is after an ‹::else› command. The example is made syntactically valid as follows

‹::ifdef Name ::then ... ::else ... ::endif›.

The following example is also syntactically invalid for the same reason as in the previous example

‹::ifdef Name ::then ... ::then ... ::then ... ::endif›.

The (highlighted) ‹::then› command must not follow another ‹::then› command.

This error message can also occur if a ‹::then› command is outside of all conditional preprocessor directives.

## UNEXPECTED_TOK

Invalid token in conditional preprocessor directive.

**Details**

The condition of a preprocessor ‹::ifdef› or ‹::elsedef› directive contains an invalid token.

## VOID_BAD

Invalid use of the void statement.

**Details**

An attempt was made to redefine the void statement (void). The void statement must not be altered.

## PAREN_CMS_BAD

There must be no comma tokens ‹,› within the top level of parentheses.

**Details**

Comma tokens may exist only inside sequences or procedures. They must not exist within the top level of parentheses.

## W_OPEXP_WNA

Wrong number of arguments in operator expression, ‹(...)›.

**Details**

An operator expression should contain at least one argument. An operator expression need not actually contain arguments if the arguments are accounted for in some other way. For example, a custom operator can produce its own arguments, but this is unnecessary and should be avoided.

Note that this warning message may also occur if an intended function call does not have the opening parenthesis adjacent to the function variable. For example, ‹...$_{WS}$()› will cause this warning message if the parentheses are meant to be empty function call arguments. If a function call is intended, then the (highlighted) white-space must be absent.

## W_OPEXP_WNO

Wrong number of operators ‹&...|*|/|+|-|^|=|!=|<|<=|>|>=|++› in operator expression, ‹(...)›.

**Details**

An operator expression should contain exactly one operator, but need not actually contain an operator if the operator is accounted for in some other way. For example, the operator can be placed after the ending parenthesis of the operator expression as in ‹(3 5 9) +›, but this is considered quirky coding and should be avoided (it would cause less problems if such an expression were written as ‹(+ 3 5 9)› with the (highlighted) operator inside the parentheses).

Note that this warning message may also occur if an intended function call does not have the opening parenthesis adjacent to the function variable. For example, ‹...$_{WS}$(...)› will cause this warning message if the parentheses are meant to enclose function call arguments. If a function call is intended, then the (highlighted) white-space must be absent.

## W_PROC_CMS

Procedure with commas ‹,›.

**Details**

A procedure contains commas at the top level. In ETAC, a procedure (including the procedure of a function

definition ‹fnt:(…){…}›) usually contains semicolons ‹;› separating the token statements. Commas can be used, but the relevant tokens will then be activated as TAC code not as ETAC code.

## W_PROC_NLSC

No last semicolon ‹;› in procedure.

**Details**

The last token statement in a procedure does not end with a semicolon. In ETAC, a procedure (includes the procedure of a function definition ‹fnt:(…){…}›) usually contains a semicolon ‹;› following the last token statement. The semicolon can be omitted, but the last token statement will then be activated as TAC code not as ETAC code.

## W_SEACC_REQ

No sequence element access token ‹%›.

**Details**

A sequence element access token is required for a sequence element assignment ‹…%… :=›, sequence append ‹… +:=›, and compound object append ‹… ++:=› statements where the destination object is a sequence element ‹…%…›. If the destination object is not intended to be a sequence element, then this warning message may be ignored. This warning message applies if the programmer had mistakenly omitted the sequence element access token from the said statements. A statement of the form ‹$V^{N}_{S}$[…] (:=|++:=|+:=) …;› (where $V$ is a variable) is equivalent to ‹[…] (:=|++:=|+:=) …; V›, which may not be what is intended. If the intention is to assign or append to a sequence element, then the form of the statement should be ‹V%[…] (:=|++:=|+:=) …;› with the (highlighted) sequence element token present.

## W_SEQ_DUP

Empty sequence ‹[]› not duplicated before allocation ‹… :-›.

**Details**

An empty sequence is not duplicated before being allocated to a variable. When an empty sequence is allocated to a variable within a procedure (including the procedure of a function definition ‹fnt:(…){…}›), it is usually intended that the sequence remain empty at the start of each call to that procedure. This is achieved by duplicating the empty sequence using the object replication expression ‹@› or `dupl` command before being allocated so that the code within the procedure assigns elements to the duplicated sequence rather than to the original empty sequence. If it is intended that the initial empty sequence retain its elements for each procedure call then the object replication expression must be absent.

In the following example, the programmer wants to have the variable `Seq` initialised with an empty sequence each time the function `MyFnt` is called.

```
MyFnt :- fnt:()
{
Seq :- @ []; [* A new empty sequence is allocated to Seq each time this
                function is called *]
   Seq +:= 10; [* Seq now contains only the element 10. *]
};
```

Without the (highlighted) object replication expression, `Seq` would retain all the `10`'s assigned on each previous call to `MyFnt`.

## W_STMT_NESC

No semicolon ‹;› at end of statement.

**Details**

A statement does not end with a semicolon. Statements usually end with a semicolon. However, if it is intended that the token following the statement be activated before the statement, then the semicolon must be absent. For example, token `A` is activated before statement `S` in ‹S A;› because there is no semicolon following `S` (giving rise to this warning message). A semicolon placed after `S` would cause it to be activated before token `A`.

The statements affected by this warning message are: ‹data:{...}›, ‹fnt:(...){...}›, ‹if...endif›, ‹when...endwhen›, ‹do {...}›, ‹...++›, ‹...--›, ‹...--›, and ‹....{...}›.

## WHEN_B_NONE

Unmatched ‹endwhen› of ‘when’ statement, ‹when...endwhen›.

**Details**

A ‘when’ statement must begin with the ‹when› token.

## WHEN_BAD

Invalid ‘when’ statement structure, ‹when...endwhen›.

**Details**

A ‘when’ statement must have the following structure:
‹when ... ^(is = != < > <= >=) *(... then {...}) [else {...}] endwhen›,
where ^(*X*) indicates that only one of the space-separated options in *X* must be present, *(*X*) indicates one or more *X*, and [*X*] indicates that *X* is optional.

## WHEN_E_NONE

Unmatched ‹when› of ‘when’ statement, ‹when...endwhen›.

**Details**

A ‘when’ statement ‹when...endwhen› must end with the ‹endwhen› token.