ETAC Code Generator

TM

ETAC Code Generator Language Version: **1-1**

Legal Information

 (the ETAC Code Generator logo) is an unregistered trademark (™) of Victor Vella.

ETAC is an unregistered trademark (™) of Victor Vella for *computer software incorporating an implementation of a computer programming language*. There may be other owners of the "ETAC" trademark used for other purposes.

MS-DOS and **Windows** are registered (®) or unregistered (™) trademarks of Microsoft Corporation.

Unicode is a registered trademark (®) of Unicode, Inc. in the United States and other countries.

The author of this document shall not be liable for any direct or indirect consequences arising with respect to the use of all or any part of the information in this document, even if such information is inaccurate or in error. The information in this document is subject to change without notice.

ETAC Code Generator

Victor Vella

Published by Victor Vella
(1 August 2020)

First Published: 1 February 2019
Second Edition: 1 August 2020

Preface

The **ETAC Code Generator** began its life as a simple non-programmable general text code generator written entirely in the C++ programming language in the year 2006 for my own personal use. At that time, I had intended to later create a more sophisticated version of the code generator in C++, but also programmable via the (unreleased) TAC programming language. However, in the year 2015, instead of creating a new C++ version of the code generator programmable via TAC, I decided to redesign the internal structure of the code generator and implement it entirely in the ETAC™ programming language. And thus, the **ETAC Code Generator** was born.

Because the **ETAC Code Generator** is written in ETAC, the **Run ETAC Scripts** package must already be installed to use the **ETAC Code Generator**. However, for those users who do not wish to install the **Run ETAC Scripts** package on their computer, I decided to also release an executable implementation of the **ETAC Code Generator** which internally contains a minimal implementation of **Run ETAC Scripts**. The executable implementation of the **ETAC Code Generator** is therefore also a portable implementation which can be installed on a removable disk and run from any suitable computer.

This is the first production release of the **ETAC Code Generator**, updated to process Unicode® template and data files.

Victor Vella

Perth, Western Australia
1 August 2020

Contents

Preface v
Contents vi
Tables and Diagrams viii
Document Conventions ix
Introduction 1
1 Overview 3
1.1 General Features 3
1.2 Requirements 3
1.3 Overview of the ETAC Code Generator 3
1.4 The Input and Template Arguments 4
1.5 Overview of a Template File 5
2 The Template File 7
2.1 The Header Block 7
2.1.1 Parameters 7
2.1.2 Header Block Example 11
2.2 The Template Line Block 12
2.2.1 Special Symbols 14
2.2.2 Instructions 18
2.2.3 Instruction Summary 18
2.2.4 Instruction Definitions 19
2.2.5 Commands 29
2.2.6 Command Summary 29
2.2.7 Command Definitions 30
3 Processing Stages 52
3.1 Meta-code Processing Stages 53
4 Input Dialog Box 55
4.1 Dialog Box Details 55
5 Operating the ETAC Code Generator 57
5.1 Command Line 57
5.2 Initialisation File 60
5.3 Executing from ETAC Script 61
5.4 Executing from ECGL Commands 62
6 Programming the ETAC Code Generator 63
6.1 Using ETAC Script 63
6.2 Intrinsic Global Functions 63
6.3 Text Array Functions 64
6.4 Debugging ETAC Script 64
7 ETAC Code Generator Examples 65
7.1 Example 1 65
7.2 Example 2 66
7.3 Example 3 68

7.4 Example 4....70
7.5 Example 5....73
7.6 Example 6....75
8 EGCL Function Reference....79
8.1 Global Variables....79
8.2 General Functions....80
8.2.1 Functions by Category....81
8.2.2 Function Summary....82
8.2.3 Function Definitions....83
8.3 Data Object: *text array*....123
8.3.1 Data Members....123
8.3.2 Function Summary....124
8.3.3 Function Members....124
Appendix A: HTML Template Files....129
A.1 Introduction....129
A.2 HTML in a Template File....129
A.3 HTML in an External File....130
Appendix B: Self-contained ETAC Code Generator....132
B.1 Introduction....132
B.2 System Requirements....132
B.3 Self-contained Installation....132
B.3.1 The Initialisation File....133
B.4 ETAC Code Generator Execution....134
B.4.1 Direct Execution....134
B.4.2 Command Line Execution....134
B.4.3 ETAC Script Execution....135
B.5 ETAC Script Debugging....135
B.6 Uninstalling the ETAC Code Generator....136
Bibliography....137
Glossary....138

Tables and Diagrams

Document Conventions....ix
ETAC Code Generator Input Output....4
Special Symbol Syntax Diagram....14
ECGL Instruction Summary....19
DATE Formatting Characters....21
Header Block Parameters....23
ECGL Command Summary....30
Meta-code Stage Numbers....53
Input Dialog Box for the ETAC Code Generator....55
Command Line Input Arguments....57
ECGL Function Summary for Scripts....82
Text Array Function Summary....124
Pattern String Types....126

Document Conventions

The following symbolic conventions are used in this document.

Document Conventions

Symbol	Meaning
‹*x*›	separates *x* as a unit of information from the surrounding text.
x···	means zero, one, or more of the same kind as *x*.
[*x*]	means that *x* optional.
{*x*}	means that *x* is the default value.
(*x*)	groups *x* as a unit.
x|*y*	means that only *x* or *y* applies, but not both (could have more than two options).
...	represents omitted text (as usual).
$^{W}_{S}$	represents a whitespace character (9_{16} to D_{16}, or 20_{16}).
$^{S}_{P}$	represents a space character (20_{16}).
$^{C}_{R}$	represents a carriage return character (D_{16}).
$^{L}_{F}$	represents a linefeed character (A_{16}).
$^{E}_{L}$	represents the character or characters indicating the end of a text line.
n_{16}	represents a number in base 16 (hexadecimal).
U+*x*	represents a Unicode code point where *x* is in hexadecimal notation.
text	maroon coloured italic text is a link to the text's definition.
text	underlined green text is a link into the document.
text	bold green text is a link into the document.
♦	indicates the end of a block of document text.

Introduction

This document is for the **ETAC Code Generator** version **2-0-3-ena** which requires **Run ETAC Scripts** version **3-0-6-ena** or later unless the self-contained implementation of the **ETAC Code Generator** is used. This document defines *ECGL* version **1-1**.

The **ETAC Code Generator** is a computer program, written in the ETAC™ programming language, that uses programmable text *template files* containing special descriptive codes and text to generate and maintain user-specified text files. The **ETAC Code Generator** is ideally suited to generate source code for computer programs, but is completely universal, and can therefore generate any form of text file. Because it is written in ETAC, the **ETAC Code Generator** is released only on the **Windows**® operating system.

A template-based code generator typically consists of a computer program which reads a special text file acting as a model for generating one or more text files for desired purposes. The model file, called a template file, is produced by the user or obtained from an external source. There can be any number of template files, each of which contains fragments of the generated text and other special text that describe how those fragments are to be produced into the generated files. The user supplies keywords and arguments that are used in conjunction with a template file to produce the desired output.

At the time of this writing, most template-based code generators produce HTML output files via any one of a number pre-existing host programming languages adapted for such a purpose. With such code generators, program code of the host language is directly or indirectly incorporated into a template file, and manipulates a copy of the text fragments within the template file to produce the output files. Effectively, therefore, such a template file operates as a computer program containing instructions based on the host language but optimised for generating text files. The template file is processed from top to bottom in sequence during processing.

The **ETAC Code Generator** is a highly advanced programmable template-based universal text and source code generator that uses a unique sophisticated declarative template language with capabilities extended by *ETAC scripts*. An instance of the declarative template language, *ECGL* (**E**TAC **C**ode **G**enerator **L**anguage), is incorporated into a *template file*. *ECGL* not only specifies to create *generated files*, but also specifies to maintain existing text files. Thus, the **ETAC Code Generator** can not only generate any number of new source files of any programming language, but can also update existing source files with new code fragments. Unlike the template files of currently existing code generators, an **ETAC Code Generator** *template file* is processed in a number of passes (“stages”); one or more *ECGL* statement types are processed in each pass.

It should be noted that *ECGL* is a declarative language, not a procedural (algorithmic) programming language. A declarative language is one that uses descriptions of the intended output, rather than using an algorithm to specify how the output is to be constructed. A declarative language, however, is not as versatile and as a procedural language, and so *ECGL* incorporates the ability to use the procedural language ETAC for producing intricate output where required.

For advanced uses of the **ETAC Code Generator**, the user needs to be familiar with the ETAC programming language. The overview of that language is contained in the document ETACOverview.pdf, and the full definition of the language is contained in the document ETACProgLang(Official).pdf. This document assumes that the reader is familiar with the ETAC programming language.

The ETAC programming language has basic support for the full Unicode® codespace (U+0000 to U+10FFFF). However, the support is only up to the Unicode scalar value level; character strings are not normalised. ETAC recognises only strict conformance to the UTF-8, UTF-16, and

UTF-32 encoding schemes; unpaired surrogate code points are not supported. For certain functionalities or parts thereof, only UCS-2 (BMP Unicode scalar value) characters are supported.

Changes from Previous Publication

The following sections indicate the changes made in this publication from the previous publication (1 February 2019). Most of the changes are adaptations to Unicode®.

New Items

&U+ ▪ **@cgECGVrsnID** ▪ **@cgGetStrU** ▪ **@cgPutStrU**

Enhanced Items

&DEL ▪ **&MI** ▪ **@SYMBOL** ▪ **@cgAddCmdSymb** ▪ **@cgGetCmdSymbVals** ▪ **@cgGetNumSymbVals** ▪ **@cgGetSpecSymbVal** ▪ **@cgGetSymbValAtOff** ▪ **@cgGetSymbCount** ▪ **@cgIncrSymbCount** ▪ **@cgSetSymbCount** ▪ **@cgCreateFile** ▪ **@cgCreateNewFile** ▪ **@cgGetWindowsDir** ▪ **@cgIsOnlyDirPath** ▪ **@cgPathExists** ▪ **@cgAddFileData** ▪ **@cgRenameDataFile** ▪ **@cgFormatStr** ▪ **@cgParseString** ▪ **@cgReplSubStr** ▪ **@cgRunETACFile** ▪ **@cgWriteAllToOne** ▪ **@cgWriteFile** ▪ 2.1.1_Parameters (‹**@C=**› and ‹**@P=**› parameters) ▪ 2.2.1_Special Symbols (*symbol-name*)

1

Overview

This chapter is an overview of the **ETAC Code Generator**. Because the **ETAC Code Generator** is written in the ETAC™ programming language, *template files* and *generated files*, as well as other files used by the **ETAC Code Generator**, can contain Unicode® characters (the **ETAC Code Generator** has basic support for the full Unicode codespace).

1.1 General Features

The **ETAC Code Generator** has at least the following capabilities.

1. From a single *template line* in the *template file*, a number of variant text lines can be generated based on a sequence of user-specified arguments.
2. *Special symbols* in *template lines* can be automatically replaced by user-specified arguments.
3. *Template lines* can be conditionally processed.
4. While a *template file* is being processed, the included *instructions* and *commands* can be modified and reprocessed (the internal copy of the *template file* is self-modifying).
5. Text lines can be automatically inserted into the *template lines* from any text file during processing.
6. *Generated lines* can be incorporated into a *template file* from the processed output of other *template files*.
7. *Generated lines* can be merged and aligned with parts of other *generated lines* or external text files.
8. Text lines can be edited via *ETAC script* after they have been generated.
9. *ETAC scripts* can be defined to edit blocks of text lines in a *template file* in any desired way while being generated.
10. *ETAC scripts* can provide user interaction during processing of the *template files*.

1.2 Requirements

The **ETAC Code Generator** is released in two implementations. Officially, the **ETAC Code Generator** is released as a compiled TAC binary instruction file named ETACCodeGen.btac, along with the ETAC source files. The **Run ETAC Scripts** package must therefore already be installed to use the **ETAC Code Generator** in this implementation. The **ETAC Code Generator** is also released as a self-contained executable program named ETACCodeGen.exe, which does not require the **Run ETAC Scripts** package. The self-contained executable is actually a silent installer that temporarily installs and runs a portable version of the **ETAC Code Generator** and **Run ETAC Scripts** (see Appendix B: Self-contained ETAC Code Generator for more details).

1.3 Overview of the ETAC Code Generator

The **ETAC Code Generator** is a computer program used to generate one or more instances of text files belonging to a class of files. Each class of files is defined in its own *template file*, and along with *input arguments*, a user specifies a *template file* to the **ETAC Code Generator** to generate the desired instances of the class of files. Different *template arguments* (which exist within the *input arguments*) that are used with the same *template file* generate different instances of the text files belonging to the same class of files. A *template file* can be designed by a user or obtained from an external source.

The **ETAC Code Generator** can also read and maintain existing or generated text files and *template files* as part of its processing. Thus, not only can the **ETAC Code Generator** generate text files but it can also maintain existing text files, making it suitable for maintaining and updating the source code files of a computer program. The **ETAC Code Generator** updates existing text in a text file by matching pairs of text lines within the existing text and replacing or adding to the text between the matching lines. The text lines that are matched are defined by the designer of the particular *template file* in use.

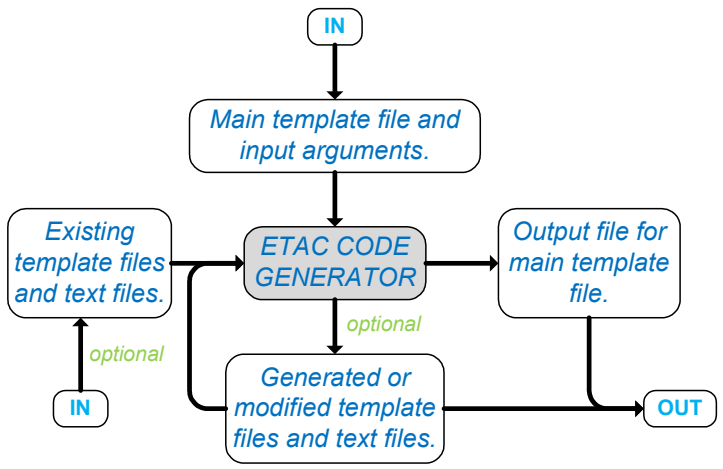
A file that is read by the **ETAC Code Generator** can be a Windows-1252, UTF-8, UTF-16, or UTF-32 file. If an input file is a UTF file, it is highly recommended that the file has a BOM signature (including for UTF-8) to avoid a possible misinterpretation of the file data.

If all the characters of a generated file are a subset of the Windows-1252 character set, then the file data will be written as a Windows-1252 file. Otherwise (if not all a subset), the file data will be written in the same UTF encoding scheme as the original file on disk, or if the file data was not obtained from disk or the original file was a Windows-1252 file, then the file data will be written as a UTF-8 file with a BOM signature.

During processing, an internal copy of the *template file* is modified and finally written to the *output file* before current *ECG session* ends.

The following diagram is an overview of the input and output data required by the **ETAC Code Generator**.

ETAC Code Generator Input Output



Code Generator Workflow

The **ETAC Code Generator** accepts a main *template file* along with *input arguments* as initial input (via a dialog box or the command line). The main *template file* may contain *commands* to accept additional *template files* and text files if required. Copies of the input files are modified, and may be reprocessed any number of times to produce the desired *generated files* if requested.

The **ETAC Code Generator** can run interactively if a *template file* requires it.

1.4 The Input and Template Arguments

The *input arguments* for the **ETAC Code Generator** are in the form of keywords and their arguments. The *input arguments* are directly supplied on the command line when the **ETAC Code Generator** program is run. In addition, the *input arguments* can be optionally supplied or modified by the user in the **ETAC Code Generator** input dialog box.

The *template arguments* within the *input arguments* are used by the **ETAC Code Generator** in conjunction with the *template file* for generating specific *generated files*. The *input arguments* can specify that the *template arguments* exist in a text file rather than within the *input arguments* themselves. Alternatively, *ETAC script* existing within a *template file* can create an appropriate dialog box to request the *template arguments* from the user.

1.5 Overview of a Template File

A *template file* contains *template lines* consisting of *special symbols*, *instructions*, and *commands* enclosed within angle brackets (<...>) and embedded in the text to be generated. The *special symbols*, *instructions*, and *commands* are called *meta-codes*. An internal copy of the *template lines* is modified by the *meta-codes* over a number of stages, then written to a specified *output file*. That *output file* is the main *generated file*. The *template lines* can also be modified by *ETAC scripts* embedded within *instructions* and *commands* in the *template lines*. There are *commands* that allow some of the *template lines* to be generated into other *generated files*, and for text from other files, including files generated from other *template files*, to be included into the *generated files*. A user supplies the name of a *template file*, a default output destination folder for *generated files*, and some *template arguments* (in the form of keyword-arguments) to generate the desired files. Alternatively, if a *template file* is appropriately designed, the contents of some or all of the *generated files* can be output to the single *output file*.

A *template file* requires a *header block*, which includes a *keyword template* used for parsing the *template arguments* into the appropriate values for use with the *special symbols*. To generate files, a user specifies the *template file* to be used and *template arguments* to match the *keyword template* specified in the *template file*. The user can also specify the default *output file* path. The *template arguments* supplied by the user determine the files and their contents to be generated from the *template file*. For example, if the user specifies `CPPSourceFiles.ecgt` as the *template file* and `‹CLASS= abcTextWin, CWnd WNDBASE MFC_BC›` as the *template arguments*, then the **ETAC Code Generator** generates source files `abcTextWin.h` and `abcTextWin.cpp` based on the `CWnd` class. A *template file* can also use *ETAC scripts* to manipulate *generated files* and the copy of the internal *template file* itself. The scripts are typically part of the contents of a *template file*.

A *template line* can contain *special symbols*. The value of a *special symbol* is supplied by the user as part of the *template arguments*. A *special symbol* can modify the supplied value by making it all upper-case or lower-case, and appending it with the result of a simple mathematical expression based on the currently *generated line* number. In the simple case, a *special symbol* is merely replaced by its value in a *template line*. For example, if a *template line* is `‹<CLASS_NAME>::<CLASS_NAME>(CWnd *pParent) :<BASE_CLASS>(<CLASS_NAME>::IDD, pParent)›` (where `‹<CLASS_NAME>›` and `‹<BASE_CLASS>›` are *special symbols*), and the values of the *special symbols* as supplied by the user are `‹abcTextWin›` and `‹Window›`, respectively, then the line `‹abcTextWin::abcTextWin(CWnd *pParent) :Window(abcTextWin::IDD, pParent)›` internally replaces that *template line*, producing a *generated line*. This example is the simplest use of *special symbols*.

A *template line* can contain embedded *instructions* which are specified in the form of keyword-arguments. *Instructions* are mainly used to perform text editing operations on the *template line* before it is converted to a *generated line*. An *instruction* can move the current output character position to a different position, insert and delete text, and align the output text while the line is being generated. An important *instruction* activates an *ETAC script* which returns text replacing the *instruction*. As a simple illustration, the following *instruction* (shown in **bold blue**) embedded in a *template line* will replace itself with the string "red" if the ETAC variable, `Condition`, is `true`, or with "green" if `Condition` is `false`:

```
It was a <&FNT:[=({@IfElse(Condition "red" "green");})]> ball
```

The *generated line* will be either `‹It was a red ball›` or `‹It was a green ball›` depending on the value of `Condition`.

A *template file* can contain *commands* which are specified in the form of keyword-arguments. *Commands* exist alone as text lines in the *template file*. There are *commands* to direct the output of the *generated lines* to various files, delete and insert text into the *generated files* (and other text files) at positions specified by a search *pattern string*, conditionally execute *template lines*, iterate through a sequence of *template lines*, define *ETAC script* functions, insert the output of

another session of the **ETAC Code Generator**, insert the text lines of another *template file*, reprocess sections of the *template lines*, modify the *template lines* including modifying *commands* and *instructions*, and more. Note that when the *template lines* are modified, only an internal copy of those lines are modified, the actual disk content of the *template file* is not modified. That modified internal copy is finally written to the *output file*.

2

The Template File

A *template file* consists of two sections — the *header block* and the *template line block*. The *header block* contains information for the **ETAC Code Generator** for processing the *template lines*. The *header block* of a *template file* must be first. The *template line block* follows the *header block* and contains the actual *template lines* and, if required, *ETAC script* fragments to produce the *generated files*.

2.1 The Header Block

The format of the *header block* is as follows. The very first line of a *template file* is the start of the header section, and must contain the version number the *ECGL* used with the *template file* as in the following example: ‹@ECG V1@›. The line must exist as the first line alone. The second line in a *template file* is the heading which will appear at 'Title' in the **ETAC Code Generator** input dialog box. The format of the heading is ‹@*string*@›, where *string* is the heading which can contain spaces, for example, @Infinite Array Derivation@. The heading must exist on a single line alone. The rest of the lines in the *header block* are the parameters of the *template file* in the form of keyword-arguments as follows: ‹@D=*description*› ‹@C=*comment*› ‹@O=*out-path*› ‹@S=*settings*› ‹@T=*ka-template*› ‹@P=*symbol-names*›. The last line in the *header block* must be ‹@endhead@›, which must exist on a single line alone.

The *header block* (the first part of the *template file*) is therefore of the following format:

@ECG V*version*@
@*heading*@
parameters
@endhead@

The *parameters* are described below.

2.1.1 Parameters

A description of the *header block* parameters follows. *ETAC comments*, which are ignored, can be present among the parameters.

@D=*description* [optional]
description is text indicating a short description of the text generated by the *template file*. This description will appear in the **ETAC Code Generator** input dialog box under 'Description'.

@C=*comment* [optional, repeated]
comment is the text for an arbitrary comment. There can be any number of these keywords. If *comment* of the first keyword (@C=) is delimited by double-quote ('"' U+0022) or single-quote ('' U+0027) characters, then the text contained within those quote characters is displayed to the user when the 'Details' button is clicked in the input dialog box. Otherwise, if no ‹@C=› keyword exists, or the first *comment* is not delimited by double-quote or single-quote characters, then the 'Details' button will be disabled. If *comment* is delimited by double-quote characters, then a double-quote character within *comment* must be expressed by having a backslash (U+005C) immediately before it. A backslash followed by a space (U+0020) within *comment* will be expressed as a backslash. If *comment* is delimited by single-quote characters, then the first single-quote must be preceded by '«' (U+00AB), and the last single-quote must

be followed by '»' (U+00BB), and the text in-between the single quotes is interpreted as given, except that unmatched '«' and '»' characters must be immediately followed by the macron character '¯' (U+00AF).

The second and subsequent `@C=` keywords (if they are present) are not displayed to the user.

For example,

```
@C="This is a template file comment
displayed to the \"user\" in three lines.
»And this is an 'escape-quoted' file path: \ \"file\path\ \"."

@C=This comment will not be displayed to the user.
```

will display the first *comment*, but not the second, as

```
This is a template file comment
displayed to the "user" in three lines.
»And this is an 'escape-quoted' file path: \"file\path\".
```

when the 'Details' button is clicked in the input dialog box.

The first *comment* could have also been written with single-quote character delimiters as

```
@C=«'This is a template file comment
displayed to the "user" in three lines.
»¯And this is an 'escape-quoted' file path: \"file\path\".'»
```

Notice the ‹»¯› at the beginning of the third line above, which is interpreted as '»'. Single-quote character delimiters effectively allow raw text to be entered.

@O=*out-file* [optional]

out-file specifies the file path of the default *output file* into which the *generated lines* are to be output if the *output file* path is not specified by the user in the *input arguments*. If this option is absent, the special directory ‹`Desktop::?`› is assumed for *out-file*. The ‹`Desktop::`› part specifies to output the *generated lines* to the **Windows**® `Desktop`; the ‹`?`› part indicates a unique program-generated file name of the form ‹ECGOutput**...**.txt›, where **...** is an eight digit random number. ‹`Desktop::`› and ‹`?`› can be used separately.

If *out-file* contains backslashes, it should be enclosed within single-quote or double-quote characters, otherwise double backslashes must be used.

Examples of this option follow.

```
[* out-file is a generated file on the Windows Desktop. *]
@O=Desktop::?
@O=Desktop::\\?

[* out-file is a generated file in the folder 'MyFolder' on the Windows Desktop. *]
@O="Desktop::\MyFolder\?"
@O="Desktop::MyFolder\?"
@O=Desktop::\\MyFolder\\?
@O=Desktop::/MyFolder/?

[* out-file is a generated file in the folder 'C:\User\Files'. *]
@O=C:\\User\\Files\\?

[* out-file is the specified file. *]
@O="C:\User\Files\Generated.txt"
@O=C:/User/Files/Generated.txt
```

```
[* out-file is the specified file relative to the current directory. *]
@O=Files\\Generated.txt
```

@S=*settings* [optional]

settings is for the settings, in keyword-arguments format, relating to the *template file*. The syntax for *settings* is:

```
[{PROCESS} | NO_PROCESS] [{NO_CHECK} | CHECK] [BACKUP] [IGNORE_BAD_SYMB]
```

PROCESS

This is the default option to allow the **ETAC Code Generator** to produce its *generated files*.

NO_PROCESS

This option prevents the **ETAC Code Generator** from producing its *generated files*. `NO_PROCESS` is typically used with `CHECK` to check the syntax of the *template file* without generating files.

NO_CHECK

This is the default option that prevents the **ETAC Code Generator** from syntax checking the *instructions* and *commands* within the *template file*. However, some less thorough syntax checking is still performed during the code generating process.

CHECK

This option allows the **ETAC Code Generator** to syntax check the *instructions* and *commands* within the *template file*. If a syntax check fails, then the **ETAC Code Generator** will not perform the code generation process, and an error file (called a "**check-file**") will be created on the **Windows**® `Desktop` (or the current directory if the `Desktop` could not be written to). The *check-file* contains the *output file* as far as it could be processed. The log file will refer to the line number in the *check-file* that caused the error. The format of the *check-file* is: ‹ECGTCheck-*date-time*.txt›, where *date* is in the form `YYYYMMDD`, and *time* is in the form `HHmmss` (for example, ECGTCheck-20150423-223843.txt). The *check-file* will be written as a UTF-8 file (with a BOM signature), unless the file characters are all a subset of the Windows-1252 character set, in which case the file will be written as a Windows-1252 file.

BACKUP

This option creates a backup of the *output file*, if it exists, before being written to by the **ETAC Code Generator**. If *file.ext* is the format of the *output file* name, then the backup file name will be *file*~backup.*ext*. If the backup file already exists then it will be overwritten automatically without warning.

IGNORE_BAD_SYMB

This option ignores *special symbols* that are undefined or invalid. If this option is absent, undefined *special symbols* will have the text ‹`***undefined***`› appended to them, and invalid *special symbols* will have the text ‹`***invalid***`› appended to them. For example, an undefined *special symbol*, `<FRUIT>`, will be replaced with `<FRUIT***undefined***>`. If this option is present, the said text is not appended.

An undefined *special symbol* is one that has the correct format, but does not exist in the list of *special symbol* definitions. An invalid *special symbol* is one that is defined, but is used in an inappropriate context.

@T=*ka-template* [required]

ka-template is a *keyword template* composed of keywords and their arguments as defined in **Appendix A: Keyword-arguments Specifications** of the document "*The Official ETAC Programming Language*", ETACProgLang(Official).pdf.

The *keyword template* is used to allow the user to specify how the *template arguments* are parsed so that they can be substituted in for the *special symbols* within the *template file*. An example of such a *keyword template* is:

```
@T={//CLASS=(#class-name,#class-init)}
{//SUBCLASS=(#subclass-name,#subclass-init,#field-name,#field-type,?)}
```

The example above consists of two 'template blocks', each of which is enclosed within a pair of braces. Although not shown in the example, there can be more than one keyword within a *template block*. The two keywords in the example are ‹CLASS=› and ‹SUBCLASS=›; the lowercase strings represent the arguments of the keywords.

ka-template can contain groups nested *template blocks* to any desired level. The nested *template blocks* syntactically follow the main *template blocks* and exist within a 'keyword block' of the following form:

[$^{W}_{S}$···]*block-label*:[$^{W}_{S}$···][*template-block*···][$^{W}_{S}$···]

An example of a nested *keyword template* is:

```
@T={/Keyword1=(#A,#B)} {/Keyword2=(#KW1:block1)} {/Keyword3:(#block2,?)}
KW1:[{KW1} {/KW1KW=(#C,#D)}] Keyword3:[{Keyword3KW=(#E,#F,?)}]
```

The example above consists of three *template blocks* (comprising the main part of the *keyword template*) on the first line, followed by two *keyword blocks* on the second line. The first *keyword block* contains two *template blocks*, and the second *keyword block* contains only one *template block*.

Given that the full specification of a *keyword template* is intricate, it will not be presented in this document. Suffice it to say that nested *template blocks* are rarely used except in the most complex systems of producing *generated files* from multiple *template files*.

@P=*symbol-names* [required]

symbol-names specifies a mapping between the *special symbol* names existing in the *template file*, and the user-supplied *template arguments* for the *keyword template* indicated at the ‹@T=› keyword.

The syntax for *symbol-names* is as follows:

(*symb-name*$^{W}_{S}$···(*tb-num*$^{W}_{S}$···*arg-num*$^{W}_{S}$···)···;[$^{W}_{S}$···])···

where *symb-name* is a *special symbol* name (alphanumeric-underscore characters beginning with an alphabetic character), *tb-num* is a *template block* number, and *arg-num* is the argument number within that *template block*. *tb-num* and *arg-num*, together, are effectively an index into the *keyword template* (indicated at ‹@T=›). Only UCS-2 (BMP Unicode scalar value) characters are recognised in *symb-name*.

The *template block* number (*tb-num*) is the block number, beginning with 1 (one), of the *template block*, within the *keyword template*, that corresponds to the *special symbol* name.

The argument number (*arg-num*), beginning with 1 (one), indicates the position of the user-specified argument (existing within the *template arguments*), indicated within the *template block*, to replace the *special symbol* name in the *template lines*. An argument number of 0 (zero) indicates the keyword name itself of the *template block*.

There may be more than one pair of *tb-num* and *arg-num* to access nested *template blocks*. Each pair refers to the next level of the nested structure.

An example of *symbol-names* corresponding with the first example, above, shown at ‹@T=› is:

```
@P=CLASS_NAME 1 1; CLASS_INIT 1 2; SUBCLASS_NAME 2 1; SUBCLASS_INIT 2 2;
FIELD_NAME 2 3; FIELD_TYPE 2 4;
```

In the example above, CLASS_INIT is the name of a *special symbol* existing within the *template file*. During processing of the *template file*, that *special symbol* (<CLASS_INIT>) will be replaced by (typically) a user-supplied argument matching the *keyword template* specified at ‹@T=› (see the first example at ‹@T=› above). The *tb-num*, 1, refers to the first *template block* of the *keyword template* (ie: ‹{//CLASS=(#class-name,#class-init)}›), and the *arg-num*, 2, refers to the second argument of that *template block* (indicated by ‹#class-init›). Note that the *special symbol* name could be completely different from the corresponding argument name in the *template block* (in this example, the *special symbol* name, CLASS_INIT, is different from the argument name, class-init). In addition, note that, for example, if *arg-num* were 0 instead of 2, then the *special symbol*, <CLASS_INIT>, would have been replaced by the keyword name itself, ‹CLASS=›. *Special symbol* names are case-sensitive.

As mentioned above, there can be additional pairs of *tb-num* and *arg-num* for a given *symb-name*, indicating nested *template blocks*. Referring to the second example shown at ‹@T=› above, to refer to ‹#A›, we have ‹@P=**...A** 1 1;...› as usual. To refer to ‹#D› we have ‹@P=**...D** 2 1 2 2;...›. To refer to ‹#F› we have ‹@P=**...F** 3 1 1 2;...›. A whole block can be referred to as well, for example, to refer to ‹#block2› we have ‹@P=**...B2** 3 1;...›.

2.1.2 Header Block Example

The following is an example of a *header block*.

```
@ECG V1@
@General CPP source file template@
@D=Template to generate the code necessary for a general C++ source file
and header file for a class structure.
@C="This template implements a C++ class.

   Format: <CLASS= class-name, [base-class]> [WNDBASE\DLGBASE] [MFC_BC]

   Keyword:
   CLASS                Information relating to the class.
   WNDBASE              The base class is derived from CWnd or CCmdTarget.
   DLGBASE              The base class is derived from CDialog.
   MFC_BC               base-class is an actual MFC class.

   Symbol Names:
   CLASS:-
    class-name          Full name of the class.
    base-class          Full name of base class. (optional)

Example:
CLASS= abcTextWin
"
@O=Desktop::? [* Indicates to output the generated file to a uniquely
named file on the desktop if an output file is not specified in the input
arguments *]
@T={//CLASS=(#class-name,$``base-class)}{/WNDBASE/DLGBASE}{/MFC_BC}
@P=CN 1 1; BC 1 2; WD 2 0; MB 3 0;
@endhead@
```

The invariant parts of the *header block* are shown as bold type. *ETAC comments* enclosed within ‹[*› and ‹*]› can be inserted among the *header block* parameters, and are ignored. The *special*

symbols in the *template lines* are expressed as ‹<CN>›, ‹<BC>›, ‹<WD>›, and ‹<MB>› (or variations of those). Those *special symbols* are replaced by the arguments indicated at ‹@P=› when the *template lines* are converted to *generated lines*. For example, if the user supplies the *template arguments*, ‹CLASS=abcTextWin, Window DLGBASE›, matching the *keyword template* at ‹@T=›, then the *special symbols* will have values as follows: ‹<CN>› will be replaced with abcTextWin, ‹<BC>› will be replaced with Window, ‹<WD>› will be replaced with DLGBASE, and ‹<MB>› will be replaced with nothing (an empty string). Because the argument to ‹@C=› is enclosed within double-quotes, that text will be displayed to the user via the 'Details' button of the input dialog box.

For illustration, if the user supplies the *template arguments* ‹CLASS=abcTextWin, Window DLGBASE›, *template lines* such as

```
<@JOIN:[]>
The main class is <CN> with base class "<BC>" and is <WD:L> based. <BC> is
<@IF:[COND=("<MB>" != "MFC_BC")]>
 not
<@END:[IF]>
 based on MFC.
<@END:[JOIN]>
```

generate the single line ‹The main class is abcTextWin with base class "Window" and is dlgbase based. Window is not based on MFC.›. Notice that *special symbols* within quotes and *commands* are also replaced with their value.

The **@IF** *command* in the example above is evaluated as

```
<@IF:[COND=("" != "MFC_BC")]>
 not
<@END:[IF]>
```

before being processed; the condition ‹"" != "MFC_BC"› is true, and so the text between the **@IF** part and the **@END** part (ie: the ‹ not›) is generated. If the condition were false, then the text between the **@IF** part and the **@END** part would not have been generated.

2.2 The Template Line Block

The *template line block*, which consists of text lines and *template lines*, follows immediately after the *header block* (ie: the next line after the ‹@endhead@› line). When the output is generated, the possibly modified *template lines* are written to the specified destinations (which could be other sections of the *template line block*). *Meta-codes* modify *template lines* in various ways, as described in the following sections, before being written.

A text line ending with ‹\$^{C}_{R}$^{L}_{F}›, ‹\$^{C}_{R}›, or ‹\$^{L}_{F}› in a *template line block* is called a *continued line*. The backslash (\) at the end of a *continued line* is called the *line continuation character*. A series of *continued lines*, and the following text line, are treated as a single *template line* by the **ETAC Code Generator**, but the *line continuation characters* are removed. The end-of-line characters remain. So when the resulting *template line* is converted to an *output line*, that *output line* will retain the end-of-line characters. Thus more than one text line is produced but without the *line continuation characters*. This feature is typically used with *multi-lines* to allow sequential text lines in a *template file* to be treated as a single *template line*.

For example, the following text lines in a *template line block*,

```
This is one line\^C_R^L_F
and this is another one\^C_R^L_F
and this is the final one.^C_R^L_F
```

represents the single *template line*

```
This is one line␍␊and this is another one␍␊and this is the final one.
```

The *output line* generated from that *template line* will contain the three text lines. Note that the *template line* does not retain the final ␍␊.

A *template line* is processed as follows ("character" means *u-char* character). Each ordinary character (an ordinary character is a character that is not part of a *meta-code*), in turn, operates as an instruction to insert that character before the *output point* of the *output line*. *Special symbols* and *instructions* perform the operations that they are designed for as they are encountered in the *template line*. An *output point* is an imaginary point that exists between characters or before the first or after the last character in the *output line*. There can be only one *output point* in an *output line*. The first position before any characters in an *output line* is *output point* zero. *Output point* one exists after the first character (or before the second character), and so on for other *output points*. For illustrative purposes only, an *output point* is shown as a vertical bar (|). An *input point* is the same as for an *output point* but applies to the *template line*.

The following illustration shows how a *template line* is processed. ‹<...>› represents a *special symbol* or *instruction*. Consider the following *template line*.

```
|This is a template<...> line.
```

The *input point* begins at position zero (before the first character, T), and then moves to position one (after the first character). The character (T), existing before the new *input point*, is inserted before the *output point* (which is also initially at position zero). The *output line* will therefore have the character T alone, and its *output point* is after that character, as shown below.

```
T|
```

The *input point* then moves to the next position (after h) and the character before the *input point* (the h) is inserted before the *output point*. The *output line* will therefore have the characters Th, with the *output point* after the h.

```
Th|
```

The process is repeated for all characters until the opening angle bracket (<) is encountered. At this point, the *template line* and its corresponding *output line* are shown below.

```
template: This is a template|<...> line.
output:   This is a template|
```

The opening angle bracket and its corresponding closing bracket (>), along with the text in-between, is a *special symbol* or *instruction*. The *input point* moves to the position just after the closing bracket. That *special symbol* or *instruction* is then activated as specified. If it is a *special symbol* then the value of that symbol is inserted before the *output point*; if it is an *instruction* then the *output line* may be altered and the *output point* may be in a different position than the next position. For example, if the *instruction* is to put the *output point* nine positions back, then the *template line* and *output line* will be as shown below.

```
template: This is a template<...>| line.
output:   This is a| template
```

The *input point* then moves to the position before the character l in line and then before subsequent characters, outputting the rest of the *template line* to subsequent positions before the current *output point*. The result is shown below.

```
template: This is a template<...> line.|
output:   This is a line.| template
```

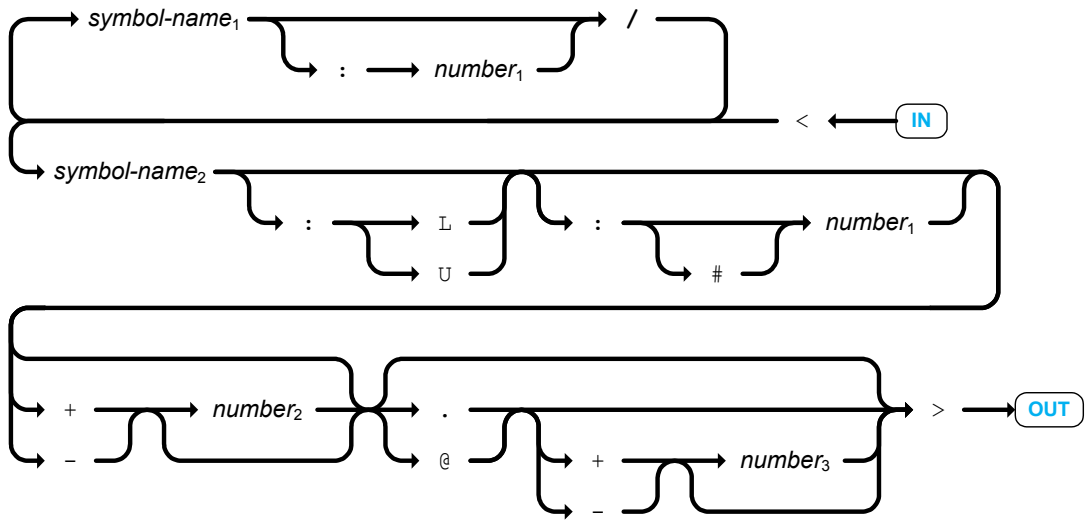
In summary, ordinary characters in a *template line* are output as given, but *special symbols* output their values, and *instructions* modify the *output line* and the position of the *output point*.

2.2.1 Special Symbols

Special symbols are always enclosed within angle brackets, < (U+003C) and > (U+003E). They cannot contain embedded white-space. If the text enclosed within angle brackets is not of the proper format for a *special symbol* then the angle brackets and the enclosed text is generated in the *output line* as given. During the processing of a *template line*, *special symbols* are replaced by specified arguments within the *template arguments*, or for *command symbols*, by the values of those *command symbols* existing on the internal list of *command symbols*.

The syntax diagram for a *special symbol* is as follows.

Special Symbol Syntax Diagram



where

symbol-name$_1$

is a *special symbol* name as defined for the ‹@P=› keyword. The *special symbol* is used with nested *template blocks* and indicates the location of a *template block*, not the location of an argument. For example, if ‹@P=ABC 2 1; DEF 0 3 1 2; GHI 4 9;› then the *special symbol* ‹<ABC/DEF/GHI/...>› indicates the *template block* at location 2 1 0 3 1 2 4 9 .... The effect of this example is the same as using the *special symbol* ‹<XYZ/...>› with ‹@P=XYZ 2 1 0 3 1 2 4 9;›.

A *special symbol* name must begin with an alphabetic character, followed by zero or more alphanumeric characters including underscores. Only UCS-2 (BMP Unicode scalar value) characters are recognised.

number$_1$

has two meanings:

(1) if used without a preceding hash character, '#', it is an integer (positive, negative, or zero) indicating by how much to add to the last index of *symbol-name*$_1$ or *symbol-name*$_2$ in the ‹@P=› keyword. For example, if ‹@P=ABC 2 1; DEF 0 3 1 2; GHI 4 9;› then ‹<ABC:3/DEF/GHI:-2/...>› indicates the *template block* at location 2 4 0 3 1 2 4 7 .... Notice that the last index of ‹ABC 2 1› has been increased by 3 (resulting in the index being 4), and the last index of ‹GHI 4 9› has been decreased by 2 (resulting in the index being 7). Note that the actual indexes of the symbols in ‹@P=› are not modified.

(2) if used with a preceding hash character, '#', the *template line* in which the *special symbol* exists is called a *multi-line*, meaning that it can generate more than one *output line*. *number*$_1$ is a non-negative integer indicating that a line is to be generated for every increase of *number*$_1$ of the last index of the corresponding symbol specified at ‹@P=›, beginning with the

value of the last index. For example, if ‹@P=ABC 0 3 1 **2**;› then the *special symbol* ‹<ABC:#**3**>› indicates that a line is to be generated using the argument located at 0 3 1 **2** of the relevant *template block*, and another line is to be generated using the argument located at 0 3 1 **5** of the same *template block*, and another line generated using the argument located 0 3 1 **8** of the same *template block*, and so on until there are no more arguments for the *special symbol*. Notice that the last index, shown in blue bold, has been incremented by **3** for each *generated line* after the first one.

If all the *number*$_1$ that are preceded by '#' of all *special symbols* in the same *template line* are zero then only one line is generated. For example, the *template line* ‹This <ABC:#**0**> with <DEF:#**0**> and <GHI:-4> generates one line› generates only one line because there is a 0 (zero) following each '#' character.

If more than one *special symbol* contains '#' in the same *template line*, then the indexes corresponding to all such *special symbol* increase simultaneously until the first index increment fails to correspond to an argument. For example, assuming that there are eleven or more arguments corresponding to the *special symbol* ABC, and six arguments corresponding to DEF, then the *template line* ‹This <ABC:#**5**> with <DEF:#**3**> and <GHI:2> generates two lines› generates only two *output lines* because the last index of DEF can only be incremented once (resulting in an index of 4); any further increments of that index would put it beyond the six available arguments. Also note that the last index of ABC is also incremented once.

number$_1$ can also be used with a *special symbol* defined by the **@SYMBOL** *command*, or by the **@cgAddCmdSymb** function.

symbol-name$_2$

is a *special symbol* name as defined in the ‹@P=› keyword, by the **@SYMBOL** *command*, or by the **@cgAddCmdSymb** function. The *special symbol* may be used with nested *template blocks*. It indicates the location of an argument corresponding to the relevant part of the *template block*. For example, if ‹@P=ABC 2 1; DEF 0 3 1 2; GHI 4 9;› then ‹<ABC>› indicates the argument corresponding to the *keyword template* located at index 2 1, and ‹<DEF>› indicates the argument corresponding to the *keyword template* located at index 0 3 1 2. The *special symbol* of *symbol-name*$_2$ is replaced by the possibly modified argument corresponding to the specified index location. Note that this is the most typical use of a *special symbol*.

A *special symbol* name must begin with an alphabetic character, followed by zero or more alphanumeric characters including underscores. Only UCS-2 (BMP Unicode scalar value) characters are recognised.

L

will put the argument in lower case before the *special symbol* is replaced.

U

will put the argument in upper case before the *special symbol* is replaced.

. (dot)

means the current line number (beginning with line number zero) of the line generated from a *multi-line*. The current line number is multiplied by the (non-negative) *number*$_2$ that precedes the dot, if *number*$_2$ is present. If *number*$_1$ and the preceding '#' are absent, then the dot (.) will represent the number zero. If '.' and '@' are both absent, then the number '1' is used instead of the '.' and '@'.

The resulting number modifies the substitution argument corresponding to the *special symbol*, as explained in subsequent paragraphs.

`@`

indicates an internal global counter corresponding to *symbol-name*$_2$. The counter is initialised with `0` (zero). It increases by one each time after a substitution involving the counter is made. The counter is used in the same way as the dot (`.`). In a *multi-line*, the counter is incremented for each *generated line* of the *multi-line*. If *number*$_1$ and the preceding '`#`' are not present, then the counter will be incremented only once for all the *generated lines* of a *multi-line*. If '`.`' and '`@`' are both absent, then the number '`1`' is used instead of the '`.`' and '`@`'.

The resulting number modifies the substitution argument corresponding to the *special symbol*, as explained in subsequent paragraphs.

number$_2$

The plus and minus sign ('+', '−') before *number*$_2$ means to add (or subtract) the multiplied (by *number*$_2$) current line number (indicated by '`.`') or symbol counter (indicated by '`@`') to the argument substituted in for *symbol-name*$_2$ if the argument is all digit characters. If the plus and minus signs and *number*$_2$ do not exist, then the argument is multiplied by the current line number or symbol counter if the argument is all digit characters. If the argument contains non-digit characters, then the (possibly) multiplied current line number or symbol counter is converted to a string and appended to the argument before substitution. For example, if `NUM` represents the three arguments `1 5 6` then `‹XXX<NUM:#1+2.>XXX›` produces `‹XXX1XXX›`, `‹XXX7XXX›`, `‹XXX10XXX›` because `2` × `0` is added to the first argument, `1`, resulting in `1`, `2` × `1` is added to the second argument, `5`, resulting in `7`, and `2` × `2` is added to the final argument, `6`, resulting in `10`. If `VAL` corresponds to the three arguments `A B C` then `‹XXX<VAL:#1.>XXX›` produces `‹XXXA0XXX›`, `‹XXXB1XXX›`, `‹XXXC2XXX›`. If `SYM` corresponds to the argument `A` then `‹X<SYM-1>X›` produces `‹XA-1X›`; with the argument `3` it would produce `‹X2X›`. The default for *number*$_2$ is `1` (one).

number$_3$

is added or subtracted from the current line number (indicated by '`.`') or symbol counter (indicated by '`@`') before the result is multiplied by *number*$_2$. The default for *number*$_3$ is `1` (one) if the preceding '+' or '−' is present. For example, if `NUM` corresponds to the three arguments `1 5 6` then `‹XXX<NUM:#1+.+2>XXX›` produces `‹XXX3XXX›`, `‹XXX8XXX›`, `‹XXX10XXX›` because `0` + `2` is added to the first argument, `1`, resulting in `3`, `1` + `2` is added to the second argument, `5`, resulting in `8`, and `2` + `2` is added to the third argument, `6`, resulting in `10`. If `VAL` corresponds to the three arguments `A B C` then `‹XXX<VAL:#1+.+>XXX›` produces `‹XXXA1XXX›`, `‹XXXB2XXX›`, `‹XXXC3XXX›` because `<VAL:#1+.+>` is equivalent to `<VAL:#1+.+1>`.

The various parts of a *special symbol* can be understood as follows.

case: [`L` | `U`]
symbol-offset: [{`0`} | *number*$_1$]
factor: [{× `1`} | ((+ | −) [{`1`} | *number*$_2$])]
count-type: [{`1`} | `.` | `@`]
count-bias: [{`0`} | ((+ | −) [{`1`} | *number*$_3$])]
number-part: *factor* × (*count-type* + *count-bias*)

A copy of the argument corresponding to *symbol-name*$_2$ is converted to upper-case before it replaces the *special symbol* if *case* is `U`, or converted to lower-case if *case* is `L`. If the argument is an integer, then a copy of it is modified by *number-part*, otherwise, if the argument is not an integer, *number-part* is appended to the argument copy before it replaces the *special symbol*.

The following examples illustrate how this system operates. Note that the term "argument" in the following examples means "argument copy"; the actual arguments are not modified. The *special symbol* names are shown in **bold purple**, and the *number-parts* are shown in **bold blue**.

<**NUM+**>

number-part is: ‹+1 × (1 + 0)›. This is the same as +1, so, for the argument corresponding to NUM, 1 is added to that argument if it is an integer, or appended to the argument if it is non-integral. Note that the default values for *count-type* and *count-bias* are used in this example.

<**ABC**:5**+2**>

number-part is: ‹+2 × (1 + 0)›. This is the same as +2, so, for the fifth argument from the one specified by ABC, 2 is added to that argument if it is an integer, or appended to the argument if it is non-integral.

<**K-@**>

number-part is: ‹−1 × (@ + 0)›. This is the same as ‹−@›, so, for the argument corresponding to K, the negation of the current global counter for K is added to that argument if it is an integer, or appended to the argument if it is non-integral. The global counter for K is automatically increased by one after it is used.

<**SYMB+.-**>

number-part is: ‹+1 × (. + −1)›. This is the same as ‹+ (. − 1)›, so, for the argument corresponding to SYMB, the value of (. − 1) is added to that argument if it is an integer, or appended to the argument if it is non-integral. Note that this *special symbol*, if alone on a *template line*, is not a *multi-line*, so the value of . would be 0. The effective value of (. − 1) would, in that case, be −1.

<**num**:3**@**>

number-part is: ‹× 1 × (@ + 0)›. This is the same as ‹× @›, so, for the third argument from the one specified by num, the current value of @ multiplies an integral argument, or is appended to a non-integral argument, where '@' is the current global counter for num. The value of @ is automatically increased by one after it is used.

<**NAME**:L**@-3**>

number-part is: ‹× 1 × (@ − 3)›. This is the same as ‹× (@ − 3)›, so, the argument corresponding to NAME is multiplied by the difference of the current global counter for NAME and 3 if that argument is an integer, or the resulting difference is appended to a lower-case copy of the argument if it is non-integral. The global counter for NAME is automatically increased by one after it is used.

<**VAR**:#1**+2**>

number-part is: ‹+2 × (1 + 0)›. This is the same as +2, so, for each argument, 2 is added to an integral argument, or appended to a non-integral argument.

<**LOG_NUM**:#2**-2.**>

number-part is: ‹−2 × (. + 0)›. This is the same as ‹−2 × .›, so, for each second argument beginning with the first, the value of ‹−2 × .› is added to an integral argument, or appended to a non-integral argument, where '.' is the current line number beginning with line 0. The values of ‹−2 × .› are: 0, −2, −4, etc. Note that for integral arguments, those arguments are decreased (or remain the same), and for non-integral arguments, the values as shown (including the negation sign) are appended to those arguments.

<**Var7**:#3**.+2**>

number-part is: ‹× 1 × (. + 2)›. This is the same as ‹× (. + 2)›, so, for each third argument of Var7 beginning with the first, the value of (. + 2) multiplies an integral argument, or is

appended to a non-integral argument, where . is the current line number beginning with line 0. The values of (. + 2) are: 2, 3, 4, etc.

<NUM:#3+.+2>

number-part is: ‹+1 × (. + 2)›. This is the same as ‹+ (. + 2)›, so, for each third of NUM argument beginning with the first, the value of (. + 2) is added to an integral argument, or appended to a non-integral argument, where '.' is the current line number beginning with line number 0. The values of (. + 2) are: 2, 3, 4, etc.

<field_id:#4-5@+2>

number-part is: ‹-5 × (@ + 2)›. So, for each fourth argument of field_id beginning with the first, the value of ‹-5 × (@ + 2)› is added to an integral argument, or appended to a non-integral argument, where '@' is the current global counter for field_id. The global counter for field_id is automatically increased by one after each time it is used, for example, if the current value of @ is 3, then the values of ‹-5 × (@ + 2)› would be: -25, -30, -35, etc.

2.2.2 Instructions

Instructions exist within a *template line* and are of the following form.

<&*name*:[ʷˢ···][[ʷˢ···]*arguments*[ʷˢ···]]>

Instructions can be placed anywhere within a *template line* unless otherwise stated, but they cannot span more than one line. The '&' (U+0026) is a symbol within the angle brackets indicating that the angle brackets and the text within it is an *instruction* rather than a *command*. *name* is the name of the *instruction*, and *arguments* is the keyword-arguments format of the *instruction's* arguments. *Special symbols* and **&FNT** (function) *instructions* can be present in *arguments* unless specified otherwise; the *special symbols* in *arguments* get evaluated once first, then the **&FNT** *instructions* get evaluated once. *arguments* can contain *ETAC comments* (outside of double quotes), which are ignored.

The following is an example of a *template line* containing an *instruction* whose argument involves a *special symbol*.

```
Ordinary line for output <&DEL:[CHARS=<NUM>]> with some characters
deleted.
```

In the example above, <NUM> is a *special symbol*. When that *special symbol* gets replaced with its value (which must be a positive integer) then the *instruction* (shown in bold blue) performs the specified number of back-space (ie: character deletion) operations as the *output line* is being generated. If, for example, the value of <NUM> is 8 then the *template line* will be equivalent to

```
Ordinary line for output <&DEL:[CHARS=8]> with some characters deleted.
```

and the *output line* generated from that *template line* would be

```
Ordinary line for with some characters deleted.
```

Notice that the *instruction* is not produced in the *output line*, and that 8 characters before the *instruction* were deleted, as specified by the *instruction*. *Instructions* are effectively removed before they are activated, so they are not part of the *template line* text when activated. Other *instructions* operate in a similar way.

2.2.3 Instruction Summary

The table below contains an alphabetical list of the *instructions*.

ECGL Instruction Summary

Instruction	Description
&	Deletes itself when activated.
&C	Encloses an arbitrary comment in a *template line block*.
&DATE	Replaces itself with the current date and time.
&DEL	Backspaces the current *output point* with deletion.
&FNT	Activates *ETAC script* whose return value replaces the *instruction*.
&HPAR	Replaces itself with the specified *header block* parameter arguments.
&MI	Moves the *output point* of the current *output line* a specified number of character positions.
&OMIT	Replaces itself with nothing on specified *output lines* of a *multi-line*, otherwise it replaces itself with a specified string.
&sq	Replaces itself with a single-quote character.
&dq	Replaces itself with a double-quote character.
&bs	Replaces itself with a backslash character.
&n	Replaces itself with a new-line (line feed) character.
&t	Replaces itself with a horizontal tabulation character.
&v	Replaces itself with a vertical tabulation character.
&b	Replaces itself with a back-space character.
&r	Replaces itself with a carriage return character.
&f	Replaces itself with a form feed character.
&a	Replaces itself with an alert character.
&U+	Replaces itself with a Unicode scalar value character.
&x	Replaces itself with the specified hexadecimal character.
&eol	Replaces itself with the end-of-line characters of the current file.
&lp	Replaces itself with a left parenthesis character.
&rp	Replaces itself with a right parenthesis character.
&lb	Replaces itself with a left brace character.
&rb	Replaces itself with a right brace character.
&ls	Replaces itself with a left square bracket character.
&rs	Replaces itself with a right square bracket character.
&lt	Replaces itself with a less-than character.
&gt	Replaces itself with a greater-than character.
&<>	Replaces itself with the enclosed *meta-code*.

2.2.4 Instruction Definitions

In the text that follows, the "current *output point*" (represented by a red bar: |) means the position of the *output point* just before the specified *instruction* is activated.

The *instructions* are defined as follows. The examples are for illustrative purposes only.

```
<&>
```

The void *instruction* simply deletes itself when activated, it is not produced in a *generated file*. The *instruction* can exist anywhere in the *template line block*, and can be used after a backslash at the end of a line so that the backslash is not interpreted as a *line continuation*

character. After this *instruction* is activated, the end of the line would be just a backslash and not treated as part of the next line during further processing.

For example,

```
this is a line \<&>
```

converts to

```
this is a line \
```

The backslash is not a *line continuation character* since it was not originally at the end of the *template line*. Note, however, that if the void *instruction* was a comment *instruction* instead in the example above, then the backslash would be a *line continuation character* because comment *instructions* are effectively removed before any processing begins.

A void *instruction* can be placed at the end of a *template line* containing trailing spaces to prevent those spaces from being deleted by a text editor. For example, ‹this line will retain the three trailing spaces$^{S_P S_P S_P}$<&>›. A "trim trailing spaces" command used in a text editor will retain the three trailing spaces, as will the **ETAC Code Generator** (after removing the void *instruction*). ♦

<&C:[(*text*)]>

The **&C** (**c**omment) *instruction* encloses an arbitrary comment in a *template line block*, and is not produced in a *generated file*; comment *instructions* are effectively removed from the *template file* before any processing begins. However, comment *instructions* within a *protection instruction* are not removed. *text* comprises the actual comment which can be any text except the string ‹]>›.

Comment *instructions* can exist anywhere outside other *commands* and *instructions*, and can be nested. If a *template line* is empty (does not contain characters including spaces) after the comment *instructions* on it have been removed, the line is deleted when the comment *instruction* is processed.

Examples

```
BEFORE: ‹This is a line <&C:[(with a comment)]> in it›
AFTER:  ‹This is a line  in it›

BEFORE: ‹This is a line <&C:[(with a comment)]>]> in it›
AFTER:  ‹This is a line ]> in it›

BEFORE: ‹This is a line <&C:[(with a <&C:[(comment)]>> in it›
AFTER:  ‹This is a line  in it›

BEFORE: ‹This is a line <&<&C:[(with a protected comment)]>> in it›
AFTER:  ‹This is a line <&C:[(with a protected comment)]> in it›
```

Notice that in the last example, the comment *instruction* was not removed because it was within a *protection instruction*. ♦

<&DATE:[(*date-format*)][UTC]>

The **&DATE** *instruction* replaces itself with the current date and time in the specified form. ‹[dd]/[MM]/[yyyy]› is the default *date-format* if no format is specified (ie: <&DATE:[]>). UTC indicates that the date is in coordinated universal time, otherwise it is in local time. Note that *date-format* must be enclosed within parentheses.

date-format can contain *special symbols* and **&FNT** *instructions*, which are activated prior to this *instruction* being activated.

The following table shows the date symbols and their meaning within *date-format*. Other symbols (eg: '/') are presented as given. Where a single digit is specified, only leading zero digits are suppressed; other non-zero digits are presented. For example, if the seconds is 20, then ‹[s]› will display 20; if the seconds is 3, then ‹[s]› will display 3, but ‹[ss]› will display 03. Alphabetic text should preferably exist outside this *instruction*.

DATE Formatting Characters

Desired Date and Time	Format Symbol
Year (four digits, last two digits)	[yyyy], [yy]
Month (long name, short name, two digits, one digit)	[MMMM], [MMM], [MM], [M]
Day (long name, short name, two digits, one digit)	[dddd], [ddd], [dd], [d]
12 hour (two digits, one digit)	[hh], [h]
24 hour (two digits, one digit)	[HH], [H]
Minute (two digits, one digit)	[mm], [m]
Second (two digits, one digit)	[ss], [s]
Fraction of seconds (3 digits)	[f]
AM/PM (A/P, AM/PM, a/p, am/pm)	[T], [TT], [t], [tt]

Examples

BEFORE: ‹Today is <&DATE:[]> <&DATE:[([HH]:[mm]:[ss])]>.›
AFTER: ‹Today is 20/05/2014 19:06:23.›

BEFORE: ‹Today is <&DATE:[([ddd] [dd]-[M]-[yy] [h]:[mm]:[s].[f] [tt])]>.›
AFTER: ‹Today is Tue 20-5-14 7:06:23.592 pm.›

BEFORE: ‹It is <&DATE:[([dddd])]>, day <&DATE:[([d])]>, in the month of <&DATE:[([MMMM])]>, in the year <&DATE:[([yyyy])]> AD.›
BEFORE: ‹It is <&DATE:[([dddd], day [d], in the month of [MMMM], in the year [yyyy] AD.)]>›
AFTER: ‹It is Tuesday, day 20, in the month of May, in the year 2014 AD.›

BEFORE: ‹Today is <&FNT:[=({@IfElse(("<CC>" = "USA") "<&DATE:[([MM]/[dd]/[yy])]>" "<&DATE:[([dd]/[MM]/[yy])]>");})]>.›
BEFORE: ‹Today is <&DATE:[(<&FNT:[=({@IfElse(("<CC>" = "USA") "[MM]/[dd]/[yy]" "[dd]/[MM]/[yy]");})]>)]>.›
AFTER: ‹Today is 05/20/14.› (if "<CC>" resolves to "USA")
AFTER: ‹Today is 20/05/14.› (if "<CC>" does not resolve to "USA")

Other Information

@cgDateTimeFormatted ♦

<**&DEL**:[CHARS=*num-chars*]>

The **&DEL** (**del**ete) *instruction* backspaces the current *output point* a number (*num-chars*) of times deleting characters as it backspaces. *num-chars* specifies how many *u-char* characters to delete. *num-chars* is a non-negative integer.

Example

BEFORE: ‹This is a |<&DEL:[CHARS=6]> line›
AFTER: ‹This line|› ♦

<&FNT:[=(*ETAC-script*)]>

The **&FNT** (function) *instruction* activates *ETAC script* (*ETAC-script*) whose return value is converted to a string replacing the *instruction*. Conditional statements and variables can be used in the script. The script runs in its own temporary local dictionary, and can access the global **cg** data object. In addition, the inclusion file, TACGlobal.PTAC, and pre-processor definitions for the *text array* data object will have automatically been included. Note that *ETAC-script* must be enclosed within parentheses.

After the *ETAC script* is activated, the top TAC stack object is removed and converted to a string which replaces the whole *instruction* (ie: replaces <&FNT:[...]>); if the stack is empty, or the top stack object is null (?), the whole *instruction* is deleted (ie: replaced with nothing) or the whole line is deleted if the *instruction* exists alone on a line. If an error occurs (for example, the top stack object cannot be converted to a string), the whole *instruction* remains as is.

TAC custom commands and operators can be used within *ETAC-script*. *Special symbols* within *ETAC-script* are replaced by their value before the script is executed, but **&FNT** *instructions* within the script are pointless (they are not processed).

ETAC-script may be an ETAC procedure or operator expression. If it is enclosed within braces or parentheses, then it is activated; otherwise it is automatically enclosed within parentheses and activated. For example: (assume <MY_ARG> has a value of MyVar)

BEFORE: ‹This is a line with <&FNT:[=({**if ("MyVar" = "<MY_ARG>") then {ins_str 2 "_" "<MY_ARG>[1]";} else {"<MY_ARG>"} endif;**})]> in it›
INTERMEDIATE: ‹This is a line with <&FNT:[=({if ("MyVar" = "MyVar") then {ins_str 2 "_" "MyVar[1]";} else {"MyVar"} endif;})]> in it›
AFTER: ‹This is a line with My_Var[1] in it›.

After the *special symbols* in the script have been replaced, the script is activated leaving the string My_Var[1] on the stack, which replaces the **&FNT** *instruction*.

Typically, the script merely consists of a function call. Functions are defined using the **@SCRIPT** *command*. For example (assuming that MyETACFnt() is already defined with the same script as in the previous example),

BEFORE: ‹This is a line with <&FNT:[=({**MyETACFnt("<MY_ARG>");**})]> in it›
AFTER: ‹This is a line with My_Var[1] in it›.

The following example activates a script expression.

BEFORE: ‹This is a line with <&FNT:[=(**("<MY_ARG>" + "[1]")**)]> in it›
BEFORE: ‹This is a line with <&FNT:[=(**"<MY_ARG>" + "[1]"**)]> in it›
AFTER: ‹This is a line with MyVar[1] in it›.

Note that for an expression, only its outer parentheses may be omitted as shown in the example above.

Other Information

@SCRIPT ♦

<&HPAR:[VRSN | HEAD | FORM | ARGS | DESC | OUTF | OUTP | OUTN | GEND | SRCD | SETT | TMPL | SIDX]>

The **&HPAR** (**h**eader **par**ameters) *instruction* replaces itself with the specified *header block* parameter arguments. The following table shows the meaning of the keywords in **&HPAR**. The *instruction* replaces itself with the value of the red ellipsis in the following table.

Header Block Parameters

Keyword	Header Block Parameter Value
VRSN	@ECG V...@ (the version of the *ECGL* used with the *template file*)
HEAD	@...@ (the heading of the *template file*)
FORM	The generated format based on the ‹@T=› argument.
ARGS	The *template arguments*.
DESC	@D=... (a short description of the *template line block*)
OUTF	@O=... (the full path of the *output file*)[a]
OUTP	@O=... (the full path of the *output file* excluding the file name and extension)[a]
OUTN	@O=... (the file name and extension only of the *output file*)[a]
GEND	The full directory path of the specified location of the *generated files*.
SRCD	The full directory path of the location of the *template files*.
SETT	@S=... (settings relating to the *template file*)
TMPL	@T=... (*keyword template*)
SIDX	@P=... (list of *special symbol* name indexes)

a If the current *template file* was evoked via the **@GEN** *command* with the INSERT option, then the value will be an empty string. The *output file* path supplied by the user, if any, will override the one at the ‹@O=› option.

Example

If the *header block* contains ‹@D=This file generates C code›, then:

BEFORE: ‹<&HPAR:[DESC]> which finds an element in an array.›
AFTER: ‹This file generates C code which finds an element in an array.›

Other Information

@cgGetHeaderPar ♦

<&MI:[(POSA=*a-pos* | POSR=*r-pos*) [FILL=(*fill-string*)] [SPACES=*num-spaces*]]>

The **&MI** (**m**ove **i**nsert) *instruction* moves the *output point* a specified number (*a-pos*) of *u-char* character positions from the beginning of the current *output line*, or a specified number (*r-pos*) of *u-char* character positions relative to the current *output point* of the *output line*. It fills any missing characters between the last character of the *output line* and the resulting position of the *output point* with the first character of a string (*fill-string*). The first character of *fill-string* must be a UCS-2 (BMP Unicode scalar value) character. If no character positions were filled, it puts the *output point* after the last printable character and appends a number (*num-spaces*) of spaces.

a-pos is a non-negative integer; the point before the first character in the line is position 0 (zero). *r-pos* is an integer. *num-spaces* is a non-negative integer. If ‹FILL=› is absent, the default for *fill-string* is a space character (U+0020).

The *output point* cannot be moved before position zero. If ‹SPACES=› is omitted then the rest of the line after the specified position of the *output point* is deleted.

A typical usage of the **&MI** *instruction* is to align variable names when generating a programming language source file. For example, suppose that <TYPE1> has the value of ‹unsigned int›, <TYPE2> has the value of long, and <TYPE3> has the value of MyVeryLongClassNameJustBecause. The following *template lines* would align the variables at position 20, except for the long class name, which will have a space after it before the variable name.

```
<TYPE1><&MI:[POSA=20 SPACES=1]>CountNum;
<TYPE2><&MI:[POSA=20 SPACES=1]>Len;
<TYPE3><&MI:[POSA=20 SPACES=1]>MyClass;
```

The *output lines* generated from the *template lines* above would be

```
unsigned int         CountNum;
long                 Len;
MyVeryLongClassNameJustBecause MyClass;
```

The variable names of the top two lines are aligned at position 20 (specified by ‹POSA=20›), but the last variable name, MyClass, is appended to the end of its variable type separated by a space (specified by ‹SPACES=1›).

Examples

In the examples that follow, the first line (at "BEFORE") represents the *template line*, the lines following represent the *output line*.

```
BEFORE:       ‹This is a line|<&MI:[POSA=10 SPACES=2]>›
INTERMEDIATE: ‹This is a |line› (output point at position 10)
AFTER:        ‹This is a line  |› (2 spaces appended to the end)

BEFORE:       ‹to be continued|<&MI:[POSR=3 FILL=(.)]> later›
INTERMEDIATE: ‹to be continued...|› (output point at relative position 3 with fill)
AFTER:        ‹to be continued... later|›

BEFORE:       ‹unsigned int|<&MI:[POSA=20 FILL=(*)]>Var;›
INTERMEDIATE: ‹unsigned int********|› (output point at position 20 with fill)
AFTER:        ‹unsigned int********Var;|›

BEFORE:       ‹unsigned int|<&MI:[POSR=0 SPACES=3]>Var;›
INTERMEDIATE: ‹unsigned int|› (output point unchanged)
INTERMEDIATE: ‹unsigned int   |› (3 spaces appended)
AFTER:        ‹unsigned int   Var;|›

BEFORE:       ‹unsigned int|<&MI:[POSR=-3]>Var;›
INTERMEDIATE: ‹unsigned |int› (output point 3 positions back)
INTERMEDIATE: ‹unsigned |› ("int" deleted)
AFTER:       ‹unsigned Var;|›
```

In the last example, int was deleted because ‹SPACES=› was absent in the **&MI** *instruction*.

♦

<&OMIT:[STR=(*string*) [FIRST | {LAST} | ENDS | NFIRST | NLAST | NENDS]]>

The **&OMIT** *instruction* replaces itself with nothing on the first (FIRST), last (LAST), first and last (ENDS), all but the first (NFIRST), all but the last (NLAST), or all but the first and last (NENDS) *output line* generated from a *multi-line*, otherwise it replaces itself with a string (*string*). LAST is the default if the FIRST, LAST, ENDS, NFIRST, NLAST, and NENDS options are absent.

Any *special symbols* or *instructions* in *string* are activated at the appropriate stage. Note that **&FNT** *instructions* are processed at a later stage than the processing of the **&OMIT** *instructions*. This *instruction* replaces itself with nothing if present on a non *multi-line* or a single *output line* of a *multi-line*.

Example

Suppose that <FRUITS> has values apples, oranges, bananas. Then the following two *template lines*,

```
I have some
<FRUITS:#1><&OMIT:[STR=(,)]>
```

generate

```
I have some
apples,
oranges,
bananas
```

Used with the **@JOIN** *command*, as follows,

```
<@JOIN:[]>
I have some
<&OMIT:[STR=( and) NLAST]> <FRUITS:#1><&OMIT:[STR=(,)]>
.
<@END:[JOIN]>
```

the **ETAC Code Generator** generates

```
I have some apples, oranges, and bananas.
```

Notice the full stop (.) before <@END:[JOIN]>. ♦

<&sq>

The &sq (**s**ingle **q**uote: ‹'›) *instruction* replaces itself with a single-quote character (U+0027). ♦

<&dq>

The &dq (**d**ouble **q**uote: ‹"›) *instruction* replaces itself with a double-quote character (U+0022). ♦

<&bs>

The &bs (**b**ack**s**lash: ‹\›) *instruction* replaces itself with a backslash character (U+005C). ♦

<&n>

The &n (**n**ew-line) *instruction* replaces itself with a new-line (line feed) character (U+000A). ♦

<&t>

The &t (horizontal **t**abulation) *instruction* replaces itself with a horizontal tabulation character (U+0009). ♦

<&v>

The &v (**v**ertical tabulation) *instruction* replaces itself with a vertical tabulation character (U+000B). ♦

<&b>

The &b (**b**ack-space) *instruction* replaces itself with a back-space character (U+0008). This *instruction* is not to be confused with the **&DEL** *instruction* which deletes the previous character. ♦

<&r>

The &r (carriage **r**eturn) *instruction* replaces itself with a carriage return character (U+000D). ♦

<&f>

The &f (**f**orm feed) *instruction* replaces itself with a form feed character (U+000C). ♦

<&a>

The &a (**a**lert) *instruction* replaces itself with an alert character (U+0007). ♦

<&U+*h*⋯*h*>

The &U+ (**U**nicode scalar value) *instruction* replaces itself with a Unicode scalar value character ‹U+*h*⋯*h*›. *h*⋯*h* is from one to eight characters in the range '0' to '9' or 'A' to 'F' or 'a' to 'f', representing a Unicode character (scalar value) in hexadecimal notation (U+*h*⋯*h*). For example, <&U+1F34F> replaces itself with the Unicode character U+1F34F (the "Green Apple" character). *h*⋯*h* cannot be 0 (the NULL character). Note that *h*⋯*h* cannot represent a surrogate code point or a value above U+10FFFF. For the single-byte version of the **ETAC Code Generator**, a value of *h*⋯*h* greater than FF_{16} will convert the *instruction* to the question mark character '?'. ♦

<&x*hh*>

The &x (he**x**) *instruction* replaces itself with the character code *hh*. *hh* is two characters in the range '0' to '9' or 'A' to 'F' or 'a' to 'f', representing a Unicode character in hexadecimal notation (U+00*hh*). For example, <&x0C> replaces itself with the Unicode character U+000C (the form feed character). *hh* cannot be 00 (the NULL character). This *instruction* is compatible with the single-byte version of the **ETAC Code Generator**. Note that <&x*hh*> is the same as <&U+*hh*>. ♦

<&eol>

The &eol (**e**nd **o**f **l**ine) *instruction* replaces itself with the end-of-line (EOL) characters of the current file. The EOL characters are typically ‹$^{C}_{R}{}^{L}_{F}$›. ♦

<&lp>

The `&lp` (**l**eft **p**arenthesis: ‹(›) *instruction* replaces itself with a left parenthesis character (U+0028). ♦

<&rp>

The `&rp` (**r**ight **p**arenthesis: ‹)›) *instruction* replaces itself with a right parenthesis character (U+0029). ♦

<&lb>

The `&lb` (**l**eft **b**race: ‹{›) *instruction* replaces itself with a left brace (“left curly bracket”) character (U+007B). ♦

<&rb>

The `&rb` (**r**ight **b**race: ‹}›) *instruction* replaces itself with a right brace (“right curly bracket”) character (U+007D). ♦

<&ls>

The `&ls` (**l**eft **s**quare bracket: ‹[›) *instruction* replaces itself with a left square bracket character (U+005B). ♦

<&rs>

The `&rs` (**r**ight **s**quare bracket: ‹]›) *instruction* replaces itself with a right square bracket character (U+005D). ♦

<&lt>

The `&lt` (**l**ess **t**han: ‹<›) *instruction* replaces itself with a less-than character (U+003C). ♦

<&gt>

The `&gt` (**g**reater **t**han: ‹>›) *instruction* replaces itself with a greater-than character (U+003E). ♦

<&<*meta-code*>>

This *protection instruction* replaces itself with the enclosed *meta-code* (*meta-code*) including the surrounding angle brackets of that *meta-code*. The enclosed *meta-code* is not processed as such (it is ‘protected’). This *instruction* can be placed anywhere within a *template line*.

The *instruction* can be used to generate *template files* (ie: files containing *meta-codes*). For example, the *template line*

```
This is <&<&DEL:[CHARS=2]>> a line
```

generates the *output line*

```
This is <&DEL:[CHARS=2]> a line
```

Effectively, the *protection instruction* has been removed, leaving the enclosed *meta-code*.

Special symbols and *instructions* are processed as usual if they exist within the *command* or *instruction* that is directly enclosed by the *protection instruction* (such *special symbols* and *instructions* are not *protected*). For example, the *instruction* (shown in **orange** text) in the first line below is within the *protection instruction*, but it is not *protected*. The bold blue text is *protected*.

```
BEFORE:       ‹This is <&<&DEL:[CHARS=<&FNT:[=(1 + 2)]>]>> a line›
INTERMEDIATE: ‹This is <&<&DEL:[CHARS=3]>> a line›
AFTER:        ‹This is <&DEL:[CHARS=3]> a line›
```

The intermediate line above is generated internally after the **&FNT** *instruction* is activated. The last line shows the *output line* after the *protection instruction* itself is activated. The *output line* contains the actual *instruction* that was originally protected. Thus, the **ETAC Code Generator** can use the *protection instruction* to generate *template files* and also activate them.

A *protection instruction* can be nested. There are two kinds of nesting:

(1) A *protection instruction* is 'tightly nested' within another *protection instruction* if the first said *protection instruction* is the only text enclosed. The *protection instruction* shown in bold, in the following example, is *tightly nested* within another *protection instruction*: <&<&<...>>>.

(2) A *protection instruction* is 'loosely nested' within another *protection instruction* if the said *protection instruction* is enclosed within that other *protection instruction* but not *tightly nested*. The *protection instruction* shown in bold, in the following example, is *loosely nested* within another *protection instruction*: <&<&DEL:[CHARS=<&<...>>]>.

Protection instructions can be nested to any degree. When a *protection instruction* is activated (the outer ‹<&› and ‹>› are effectively removed), all *loosely nested protection instructions* (shown in bold in the following examples) are also activated.

Examples

```
BEFORE: ‹<&<&DEL:[CHARS=<&<&FNT:[=(...)]>>]>>›
AFTER:  ‹<&DEL:[CHARS=<&FNT:[=(...)]>]>›

BEFORE: ‹<&<&<&DEL:[CHARS=<&<&FNT:[=(...)]>>]>>>›
AFTER:  ‹<&<&DEL:[CHARS=<&FNT:[=(...)]>]>>›

BEFORE: ‹<&<@DELETE:[...]>>›
AFTER:  ‹<@DELETE:[...]>›
```

The following example *protects* a *command* block.

```
<&<@OUTPUT:[PATH="MyFile.txt"]>>
I am un<&<&DEL:[CHARS=2]>>touchable.
I am un<&DEL:[CHARS=2]>touchable.
<&<@END:[OUTPUT]>>
```

The *output lines*, after the *protection instructions* have been activated, are as follows

```
<@OUTPUT:[PATH="MyFile.txt"]>
I am un<&DEL:[CHARS=2]>touchable.
I am touchable.
<@END:[OUTPUT]>
```

Notice that the second last line was not *protected*, and so the **&DEL** *instruction* took effect. ♦

2.2.5 Commands

Commands exist within the *template line block* and are of the following two forms.

1
```
<@name:[Ws…]
[[Ws…]arguments[Ws…]]>
template-lines
<@END:[name]>
```

2
```
<@name:[Ws…][[Ws…]arguments[Ws…]]>
```

Commands can be placed anywhere within the *template line block*, but cannot exist on the same logical line with text other than comment *instructions* or white space, unless stated otherwise. If such text exists then the *command* is not recognised and is regarded as ordinary text. Parts of a *command* can exist on subsequent lines; those lines are automatically concatenated as a single logical line. Some *commands* can be nested within other *commands* of the same or different type, as specified in the *command* definitions.

The '@' (U+0040) is a symbol within the angle brackets indicates that the angle brackets and the text within it is a *command* rather than an *instruction*. *name* is the name of the *command*, and *arguments* is the keyword-arguments format of the *command's* arguments. *arguments* can contain *ETAC comments* (outside of double quotes), which are ignored. *template-lines* indicates one or more *template lines*. *Commands* are removed after being activated (unless otherwise stated), so they are not themselves part of the *output lines*. *Special symbols* and **&FNT** (function) *instructions* can be present in *arguments* unless specified otherwise; the *special symbols* get evaluated once first, then the **&FNT** *instructions* get evaluated once.

The following is an example of part of a *template line block* containing a *command* whose argument contains a *special symbol*.

```
<@OUTPUT:[PATH="C:\Files\<FILE>.txt" MARK=(//M1//)]>
This line will be output,
and so will this line.
<@END:[OUTPUT]>
```

In the example above, ‹<FILE>› is a *special symbol*, and when that *special symbol* gets replaced with its value (which must be a proper file name) then the *command* (shown in blue) inserts the two lines (shown in black) to the specified *output file* beneath the text line containing ‹//M1//› in that file. If, for example, the value of <FILE> is MyFile then the *command* lines will be equivalent to:

```
<@OUTPUT:[PATH="C:\Files\MyFile.txt" MARK=(//M1//)]>
This line will be output,
and so will this line.
<@END:[OUTPUT]>
```

And MyFile.txt will contain the two black lines beneath the text ‹//M1//› as shown below (the ‹//M1//› is already in the file MyFile.txt).

```
…
//M1//
This line will be output,
and so will this line.
…
```

2.2.6 Command Summary

The table below contains an alphabetical list of the *commands*.

ECGL Command Summary

Command	Description
@CMT	Encloses comments on any number of lines.
@DELETE	Deletes all, the initial, or final text lines in a file matching a *pattern string*.
@DO_FOR	Reproduces a block of *template lines* a specified number of times.
@DO_WITH	Reproduces *template lines* for a subset of the values of a *special symbol*.
@EDIT	Allows the programmer to edit *template lines* via *ETAC script*.
@END	Indicates the end of a *command* block for certain *commands*.
@EVAL	Does a number of additional levels of *special symbol* processing of specified *template lines*.
@GEN	Runs a new internal instance of the **ETAC Code Generator**.
@IF	Selects *template lines* depending on boolean conditions.
@INSERT	Inserts text lines immediately below itself.
@JOIN	Concatenates consecutive *template lines* into a single *template line*.
@OUTPUT	Inserts processed *template lines* into a text file relative to a marked place.
@POSTGEN	Used to post-generate to a file from a given *template file*.
@REPROCESS	Reprocesses specified *template lines* a number of specified times.
@SCRIPT	Activates a block of *ETAC script*.
@SECTION	Used to separate the *template line block* into separate sections for processing.
@SYMBOL	Adds a new *special symbol* and its values to the list of *command symbols*.

2.2.7 Command Definitions

The *commands* are defined as follows. The examples are for illustrative purposes only.

```
<@CMT:[]>
comments
<@END:[CMT]>
```

The **@CMT** *command* encloses comments (*comments*) on any number of lines. A comment *command* is deleted when activated, and not produced in the *output file*.

The following *template lines* contain a comment *command*.

```
AAAA
<@CMT:[]>
I am a comment.
And so am I!
<@END:[CMT]>
BBBB
```

The resulting *output lines* will be as follows.

```
AAAA
BBBB ♦
```

```
<@DELETE:[[PATH="file-path"] LINES=[P | S | E | A](string) [(cust-pat)···] [FIRST]
    [LAST]]>
```

The **@DELETE** *command* deletes all, the initial, or final text lines in a file (*file-path*) matching a *pattern string* (*string*). *file-path* can be a full or relative file path but must be

enclosed within double quotes. If ‹PATH=› is absent, the *output file* path specified in the *header block* (@O=*out-path*) or by the user is assumed. If the file specified by *file-path* does not exist, or no line matches *string*, then no action occurs and this *command* is removed. *string* and *cust-pat* are *pattern strings* and have the same format as *marker-string* and *cust-pat* (respectively) for the **@OUTPUT** *command*.

If FIRST is present, the first lot of contiguous lines matching the *pattern string* are deleted.

If LAST is present, the last lot of contiguous lines matching the *pattern string* are deleted.

If both FIRST and LAST are absent, all lines matching the *pattern string* are deleted.

This *command* operates on an internal copy of the text in the specified file; that text is written to the actual disk file just before the current *ECG session* ends.

Additional Information

@OUTPUT ♦

```
<@DO:[FOR:[IDXNAME=index-name [START=start-val] [STEP=incr] [REPEAT=num-reps]]]>
template-lines
<@END:[DO]>
```

The DO[FOR] *command* reproduces a block of *template lines* (*template-lines*) a specified number of times (*num-reps*). Each reproduced block will have a specified index name (*index-name*) replaced by an integer beginning with an initial value (*start-val*), and incremented by a specified amount (*incr*).

start-val is an integer (default value is 0), *incr* is a positive integer (default value is 1), and *num-reps* is a non-negative integer (default value is 1). Any of the values can be a non *multi-line special symbol* or a **&FNT** *instruction* evaluating to the appropriate type of integer.

index-name is an alphanumeric name (beginning with an alphabetic character) which will contain the next value specified by *incr* for each iteration. *index-name* will have an initial value of *start-val* and is incremented with a value of *incr* after each iteration.

template-lines is reproduced *num-reps* number of times. The reproduced *template-lines* are then reprocessed using simple symbol substitution (as in stage 6). Within *template-lines*, *index-name* is used as ‹<%*index-name*>›. ‹<%*index-name*>› is replaced with its current value, and can be used anywhere an integral value is required (including in a *special symbol*) but is ignored if it exists directly within a *protection instruction* (for example, **...**<&<%Idx>>**...**, remains as is).

The header of a **@DO**[FOR] *command* cannot be a *multi-line*. Non *multi-line special symbols* and **&FNT** *instructions* for *start-val*, *incr*, and *num-reps* are activated prior to this *command* being activated. **@DO**[FOR] *commands* can be nested, but the nested *commands* must not have the same *index-name*.

In the following example, suppose that the *special symbol* <S1> has values Tom, Mick, Mary, Jane.

```
<@DO:[FOR:[IDXNAME=Idx REPEAT=4]]>
Hello <S1:<%Idx>>,
<@IF:[COND=(<%Idx> = 0)]>
You have no apples to eat.
<@ELSE:[COND=(<%Idx> = 1)]>
You have <%Idx> apple to eat.
<@ELSE:[]>
You have <%Idx> apples to eat.
<@END:[IF]>
<@END:[DO]>
```

When the **@DO**[FOR] *command* is activated, the eight *template lines* will be internally duplicated and modified so that there will be four lots of those eight lines (because of REPEAT=**4**). For each duplication, the index name <%Idx> will be replaced with its current value. The intermediate stage of the **@DO**[FOR] *command*, above, with the index name replaced by its values is:

```
Hello <S1:0>,
<@IF:[COND=(0 = 0)]>
You have no apples to eat.
<@ELSE:[COND=(0 = 1)]>
You have 0 apple to eat.
<@ELSE:[]>
You have 0 apples to eat.
<@END:[IF]>
Hello <S1:1>,
<@IF:[COND=(1 = 0)]>
You have no apples to eat.
<@ELSE:[COND=(1 = 1)]>
You have 1 apple to eat.
<@ELSE:[]>
You have 1 apples to eat.
<@END:[IF]>
Hello <S1:2>,
<@IF:[COND=(2 = 0)]>
You have no apples to eat.
<@ELSE:[COND=(2 = 1)]>
You have 2 apple to eat.
<@ELSE:[]>
You have 2 apples to eat.
<@END:[IF]>
Hello <S1:3>,
<@IF:[COND=(3 = 0)]>
You have no apples to eat.
<@ELSE:[COND=(3 = 1)]>
You have 3 apple to eat.
<@ELSE:[]>
You have 3 apples to eat.
<@END:[IF]>
```

In the final stage, the *special symbols* are replaced by their values, and the **@IF** *commands* are activated. The resulting *output lines* are shown below.

```
Hello Tom,
You have no apples to eat.
Hello Mick,
You have 1 apple to eat.
Hello Mary,
You have 2 apples to eat.
Hello Jane,
You have 3 apples to eat.
```

The **ETAC Code Generator** does actually internally produce modified duplicates of *template-lines* — *meta-codes* are duplicated along with those lines.

The **@DO**[FOR] *command* can be combined with the **@DO**[WITH] *command* as shown in the following syntax.

```
<@DO:[FOR:[...] WITH:([...]), ···]>
template-lines
<@END:[DO]>
```

template-lines is reproduced the minimum of the number of times they would be reproduced with the FOR or WITH options alone. For example, if *template-lines* would be reproduced six times with the FOR option alone, and four times with the WITH option alone, then they will be reproduced four times because this is the minimum of six and four.

Other Information

@DO_WITH ♦

```
<@DO:[WITH:([IDXNAME=index-name [STEP=incr] SNAME=symbol-name]), ···]>
template-lines
<@END:[DO]>
```

The **@DO**[WITH] *command* reproduces *template lines* (*template-lines*) for a subset of all the values of a *special symbol* (*symbol-name*) using an index name (*index-name*) for the last index of those values. Note that there may be more than one WITH block in this *command*.

incr is a non-negative integer or a non *multi-line special symbol* or a **&FNT** *instruction* evaluating to a non-negative integer (default value is 1).

index-name is an alphanumeric name (beginning with an alphabetic character) which will represent the incremented value specified by *incr* for each iteration. *index-name* will initially represent a value of zero, and that value is incremented by the value of *incr* after each iteration. The values of all *index-names* are incremented simultaneously. *index-name* can exist in any *special symbol* within *template-lines* but only as $number_1$ (without the '#' prefix) in the *special symbol* syntax diagram. If *index-name* exists anywhere else within *template-lines*, the consequence is undefined. For each iteration of the **@DO**[WITH] *command*, *index-name* is replaced by its current represented value. Note that *index-name* is not an ETAC variable.

symbol-name must evaluate to a restricted *special symbol* format as follows,

```
[(name1[:(number1 | idx-name1)]/)···]namen[:(numbern | idx-namen)]
```

where *name* is in the format of a *special symbol* name (alphanumeric string beginning with an alphabetic character); *number* is an integer (usually positive); and, *idx-name* is an index name (*index-name*) of any outer **@DO**[WITH] *command* (this cannot be *index-name* at ‹IDXNAME=› of the current *command*). *number* is $number_1$ as defined in the *special symbol*

syntax diagram. Note that *symbol-name* is internally used as an actual *special symbol* to determine the number of iterations, and so *name* must be appropriately pre-defined as usual. The number of iterations is the number of values remaining at and after the specified value position (*number*$_n$ | *idx-name*$_n$) corresponding to the *special symbol* (*name*$_n$) in the ‹@P=› keyword or the *special symbol* (*name*$_n$) defined via the **@SYMBOL** *command* or the **@cgAddCmdSymb** function. If ‹*number*$_n$ | *idx-name*$_n$› is absent, then *number*$_n$ effectively has the value of zero.

An example of *symbol-name* is ‹FNT_NAME/CMD:Idx/SUBCMD:3/CMD_TYPE›. In this example, Idx is the *index-name* of any outer **@DO**[WITH] *command*.

template-lines is reproduced a number of times which is the minimum of the number of iterations of all the symbol names (*symbol-name*) specified in all the WITH blocks of the same **@DO**[WITH] *command*. The reproduced *template-lines* are then reprocessed using simple symbol substitution (as is done in stage 6).

Within *template-lines*, *index-name* is used as

<[(*name*$_1$[:(*number*$_1$ | *idx-name*$_1$)]/)···]*name*$_n$[:(L|U)]:*index-name*[(+ | −)[*number*$_2$]]>

— OR —

<[(*name*$_1$[:(*number*$_1$ | *idx-name*$_1$)]/)···]*name*$_n$:*index-name*[/*name*[:*number*$_1$]]···
/*name*[:(L|U)][:*number*$_1$][(+ | −)[*number*$_2$]]>

and is processed as a *special symbol*. *name* , *number*, and *idx-name* are as defined above. *number*$_2$ is as defined in the *special symbol* syntax diagram. *index-name* and *number*$_1$ represent the *number*$_1$ as defined in the *special symbol* syntax diagram.

For example, if *index-name* is Index, an illustration of the first format is

```
<FNT_NAME/CMD:Idx/SUBCMD:3/CMD_TYPE:L:Index>
```

and an illustration of the second format is

```
<FNT_NAME/CMD:Idx/SUBCMD:3/CMD_TYPE:Index/ENTRY:5/SECT/INPAR+1>
```

The header of a **@DO** *command* cannot be a *multi-line*. Non *multi-line special symbols* and **&FNT** *instructions* for *index-name*, *incr*, and *symbol-name* are activated prior to this *command* being activated. **@DO**[WITH] *commands* can be nested, but the nested *commands* must not have the same *index-name*.

Note that the ‹WITH:([...]), ···› syntax can be equivalently written as ‹(WITH:[...]) ···›. For example, ‹<@DO:[**WITH**:[IDXNAME=Idx SNAME=S1], [IDXNAME=J STEP=2 SNAME=S2]]>› and ‹<@DO:[**WITH**:[IDXNAME=Idx SNAME=S1] **WITH**:[IDXNAME=J STEP=2 SNAME=S2]]>› are equivalent (the difference is that the ‹, › is replaced with ‹ WITH:›).

In the following example, suppose that the *special symbol* <S1> has the values Tom, Mick, Jane; and <S2> has the values 4, 7, 2, 10, 5, 3, 8.

```
<@DO:[WITH:[IDXNAME=Idx SNAME=S1] WITH:[IDXNAME=J STEP=2 SNAME=S2]]>
Hello <S1:Idx>,
You have <S2:J> apples to eat.
<@END:[DO]>
```

When the **@DO**[WITH] *command* is activated, the two *template lines* will be internally duplicated and modified so that there will be three lots of those two lines. For each duplication, the index names Idx and J will be replaced with their current values. Idx

indicates an index into each value of S1 (because STEP=**1**, by default); J indicates an index into every second value of S2 (because STEP=**2**). The value of Idx cannot be greater than 3 since there are only three values of S1, therefore the two *template lines* can only be reproduced three times. The intermediate step of the **@DO**[WITH] *command*, above, with the two index names replaced by their values is:

```
Hello <S1:0>,
You have <S2:0> apples to eat.
Hello <S1:1>,
You have <S2:2> apples to eat.
Hello <S1:2>,
You have <S2:4> apples to eat.
```

In the final step, the *special symbols* are replaced by their values. The resulting *output lines* are shown below.

```
Hello Tom,
You have 4 apples to eat.
Hello Mick,
You have 2 apples to eat.
Hello Jane,
You have 5 apples to eat.
```

The **@DO**[WITH] *command* is effectively an elaborate version of a *multi-line* applied to more than one *template line*. In a simple case, however, it is more efficient and readable to use a *multi-line* rather than a **@DO**[WITH] *command*, as illustrated in the following example.

```
Hello <S1:#1>,\
You have <S2:#2> apples to eat.
```

You will notice that the two *template lines* above generate the same text as in the preceding example (assuming the same values for <S1> and <S2> as previously defined). It is only in more complex cases that the **@DO**[WITH] *command* needs to be used.

The **ETAC Code Generator** does actually internally produce modified duplicates of *template-lines* — *meta-codes* are duplicated along with those lines.

The **@DO**[WITH] *command* can be combined with the **@DO**[FOR] *command* as shown in the following syntax.

```
<@DO:[FOR:[...] WITH:([...]), ···]>
template-lines
<@END:[DO]>
```

template-lines is reproduced the minimum of the number of times they would be reproduced with the FOR or WITH options alone. For example, if *template-lines* would be reproduced six times with the FOR option alone, and four times with the WITH option alone, then they will be reproduced four times because that is the minimum of six and four.

Additional Information

Special Symbol Syntax Diagram

Other Information

@DO_FOR ♦

```
<@EDIT:[SCRIPT=(ETAC-script)]>
template-lines
<@END:[EDIT]>
```

The **@EDIT** *command* allows the programmer to edit *template lines* (*template-lines*) via *ETAC script* (*ETAC-script*). ETAC conditional statements and variables can be used in the *ETAC script* as well as ETAC commands and operators. The script runs in its own local dictionary, and can access the **cg** data object. In addition, the inclusion file, TACGlobal.PTAC, and pre-processor definitions for the *text array* data object will have automatically been included. Note that *ETAC-script* must be enclosed within parentheses.

Before the *ETAC script* is activated, the top TAC stack object will be a *text array* data object containing a duplicate of *template-lines* as an ETAC string sequence (accessed via the **tsaTextLines** data member of the *text array* data object). The script can then edit the string sequence within the data object as desired, but must leave or replace the *text array* data object (containing the edited string sequence) on the stack. The edited string sequence will then replace *template-lines* (and the **@EDIT** *command*).

ETAC-script may be an ETAC procedure or just text script code. If it is enclosed within braces then it is executed as a procedure; otherwise it is executed as a top-level script. Typically, *ETAC-script* will be one or more ETAC function or procedure variables that have been previously defined in a **@SCRIPT** *command*.

This *command* can be nested. The inner level nested **@EDIT** *commands* are processed before the outer level **@EDIT** *commands*.

The *text array* data object on the TAC stack is accessed by allocating it to a variable as follows: ‹*variable* :-;› where *variable* is a programmer-defined variable. The ETAC string sequence in the *text array* is then modified as desired via appropriate code, then the *text array* data object is pushed onto the stack by presenting its variable. For example,

```
... SCRIPT=(TAData :-; code; TAData;)...
```

code is the part of the *ETAC script* that modifies the *text array* in TAData via the ‹*variable*.tl*› or the ‹@cg*› functions. Other ETAC commands and functions that operate on a string sequence can be used with ‹*variable*.**tsaTextLines**› (**tsaTextLines** is a string sequence). After *ETAC-script* has finished executing, the contents of **tsaTextLines** of the *text array* replaces *template-lines* (and the **@EDIT** *command*).

The following example removes the duplicate *template-lines*, sorts the lines in ascending alphabetical order, then reverses the order of those lines.

```
<@EDIT:[SCRIPT=({@cgDelDuplLines(); @cgSortLines(); @cgRevLines();})]>
bool      RtnVal1;
int       RtnVal3;
bool      Par1;
int       Par2;
int       RtnVal3;
char     *Par3;
bool      Par1;
<@END:[EDIT]>
```

The resulting *output lines* will be:

```
int       RtnVal3;
int       Par2;
char     *Par3;
bool      RtnVal1;
bool      Par1;
```

The three ‹@cg*› functions in the script modify a *text array* data object which is on the TAC stack.

Note that the braces in the ‹SCRIPT=› keyword may be omitted as is done in the following example. Also note that if the braces are omitted, the last semicolon may also be omitted.

This example indents the two *template-lines* with five asterisks.

```
<@EDIT:[SCRIPT=(TArr :-; TArr.tlIndentLines(5 "*"); TArr;)]>
What's up doc?
Nothing but the sky.
<@END:[EDIT]>
```

The resulting *output lines* will be:

```
*****What's up doc?
*****Nothing but the sky.
```

The **@EDIT** *command* can also be used to enable the user to edit *template-lines* interactively. This can be achieved by creating an ETAC function, via the **@SCRIPT** *command*, to display a dialog box showing *template-lines* for the user to edit. The function would return the edited *template lines* on the stack. For example, if the said function is called EditLines, and takes a *text array* data object as argument and return value, then the **@EDIT** *command* would look like

```
<@EDIT:[SCRIPT=(EditLines())]>
Some lines for the user to edit.
The user will edit these lines.
<@END:[EDIT]>
```

The function EditLines will need to extract the string sequence from the *text array* data object for the user to edit, assigning the modified string sequence to the returned *text array* data object as follows.

```
EditLines :- fnt:(pTAData)
{
StrSeq :- pTAData.tsaTextLines;

   [* --insert code here to edit the text lines in StrSeq via a dialog box or text editor-- *]

   pTAData.tsaTextLines := StrSeq;
   pTAData; [*RETURN*]
};
```

Of course, the function above would include the details to create and display the dialog box, or to display a text editor for the user to edit the *template lines*.

Other Information

@SCRIPT ♦

```
<@END:[CMT | DO | EDIT | EVAL | IF | JOIN | OUTPUT | REPROCESS | SCRIPT]>
```

The **@END** *command* indicates the end of a *command* block for certain *commands* as indicated by the keywords for this *command*. This *command* can only be used in conjunction with the said *command* blocks. ♦

```
<@EVAL:[[COUNT=num-times] [PRIOR]]>
template-lines
<@END:[EVAL]>
```

The **@EVAL** *command* does a number (*num-times*) of additional levels of *special symbol* processing (stages 6 and 7) of the specified *template lines* (*template-lines*). *num-times* is a non-negative integer, or a non *multi-line special symbol* or **&FNT** *instruction* that evaluates to a non-negative integer. If ‹COUNT=› is absent, the default value of *num-times* is one. Note that, regardless of the value of *num-times*, *template-lines* is processed as usual before this *command* takes effect; *num-times* specifies the number of addition processing.

This *command* can be nested. The inner level nested **@EVAL** *commands* are processed before the outer level **@EVAL** *commands*, and are re-processed by those outer level *commands*.

The header of an **@EVAL** *command* cannot be a *multi-line*.

The PRIOR option activates the **@EVAL** *command* before the **@DO** *commands* are activated.

This *command* is useful for processing a *special symbol* whose value is another *special symbol*. For example, suppose that the *special symbol* <FOOD> has the value <FRUIT>, and that <FRUIT> has the value banana, then

```
<@EVAL:[]>
I have a <FOOD>
<@END:[EVAL]>
```

produces the *output line*

```
I have a banana
```

because <FOOD> evaluates to <FRUIT> in the normal processing of *template lines*, but ‹COUNT=1› (the default) causes 1 (one) additional processing of <FRUIT>, resulting in banana. Without the **@EVAL** *command*, the *generated line* would be ‹I have a <FRUIT>›.

The **@EVAL** *command* is also useful for processing *special symbols* that themselves contain *special symbols*. For example, suppose that <NUM> has the three values 1, 5, 6. And also suppose that <MYSYMB> has the value NUM. Then

```
<@EVAL:[COUNT=1]>
XXX<<MYSYMB>:#1+2.>XXX
<@END:[EVAL]>
```

generates

```
XXX1XXX
XXX7XXX
XXX10XXX
```

because in the normal processing of *template lines*, <<MYSYMB>:#1+2.> is not a valid *special symbol*, but <MYSYMB> is, so the normal result is <NUM:#1+2.> after <MYSYMB> gets replaced. However, this normal result is a valid *special symbol*, so the **@EVAL** *command* processes this *special symbol*, generating the three lines shown above.

Other Information

@REPROCESS ♦

```
<@GEN:[INPUT="template-file" (OUTPUT="out-path" | INSERT) ARGS=(template-arguments)
    [SRC_HEAD] [{NO_PROMPT} | PROMPT]]>
```

The **@GEN** *command* runs a new internal instance of the **ETAC Code Generator**. *template-file* is a file path of a *template file*. If *template-file* is a file name only, without a path and extension, then it is assumed to exist in the default *template files* directory, and its extension is assumed to be 'ecgt'. If *template-file* is a relative path, then it is relative to the current directory.

out-path specifies the *output file* path for *template-file*. If *out-path* is not an empty string, it overrides the *output file* path specified (at ‹@O=›) in the *header block* of *template-file*. If *out-path* is a relative path, then it is relative to the current directory. If *out-path* is an empty string then the *output file* path specified in the *header block* of *template-file* takes affect.

Note that *template-file* and *out-path* must be enclosed within double quotes.

INSERT produces the default *generated lines* into the current *template lines* below where this *command* is specified, rather than into an *output file*.

template-arguments is a string containing the *template arguments* used with the *template file* specified in *template-file*. *template-arguments* must be enclosed within parentheses and must conform to *ka-template* specified at the ‹@T=*ka-template*› parameter in the *header block* as specified by the option SRC_HEAD.

SRC_HEAD indicates that the *header block* of the current *template file* in which this *command* is specified takes effect, and *template-file* contains only the *template line block* itself. If this keyword is absent, the *header block* of the specified *template file* takes effect.

PROMPT displays the user input dialog box of the **ETAC Code Generator**, containing *template-file*, *out-path*, and *template-arguments*. NO_PROMPT (the default) does not display the dialog box.

Special symbols and **&FNT** *instructions* are activated prior to this *command* being activated. This *command* is removed after being activated.

Other Information

@POSTGEN ▪ @cgGenerate ♦

```
<@IF:[true | false | COND=(boolean-condition1)]>
[template-lines1]
[<@ELSE:[[{true} | false | COND=(boolean-condition2)]]>
[template-lines2]]
...
[<@ELSE:[[{true} | false | COND=(boolean-conditionn)]]>
[template-linesn]]
<@END:[IF]>
```

The **@IF** *command* selects certain *template lines* (*template-lines*) depending on boolean conditions (*boolean-condition*) ignoring other specified *template lines*. *boolean-condition* is a boolean expression written in the ETAC™ programming language, and returns an integral value. Note that *boolean-condition* is actually activated as *ETAC script*. If the returned value is non-zero then it is regarded as `true`, otherwise the value is regarded as `false`. *boolean-condition* cannot be a *multi-line*; non *multi-line special symbols* in *boolean-condition* are pre-activated.

This *command* can be nested. The outer level nested **@IF** *commands* are processed before the inner level **@IF** *commands*.

The syntax for *boolean-condition* is the same as for *ETAC-script* of the **&FNT** *instruction* with the exception that if *boolean-condition* does not return an integral value or the TAC stack is empty then the whole **@IF ... @END** *command* remains as is given (no processing of the *command* occurs). Note that *boolean-condition* must be enclosed within parentheses.

If ‹`COND=`› is absent, then whichever one is specified of '`true`' or '`false`' is interpreted as the return value of the absent *boolean-condition*; '`true`' is the default value. Any number of `@ELSE` options can exist, or they can all be omitted.

template-lines is any number of *template lines*, and can include **@IF** *command* blocks and other *commands*.

This *command* is processed as follows. Each *boolean-condition* is processed in turn beginning with $boolean\text{-}condition_1$ to $boolean\text{-}condition_n$, but for the first $boolean\text{-}condition_k$ that returns `true`, $template\text{-}lines_k$ is retained and no other *boolean-condition* is activated. All other *template-lines* and the rest of the **@IF** *command* block are internally removed. Any existing **@IF** *command* blocks within $template\text{-}lines_k$ are then processed. If each *boolean-condition* returns `false` then the whole **@IF** *command* block is removed and no *template-lines* are retained.

An illustration of how the **@IF** *command* operates is shown below.

```
<@IF:[COND=(<VISIBLE> &and (<USER1> &or <USER2>))]>
code1
<@ELSE:[COND=("<COLR>" = "red")]>
code2
<@ELSE:[COND=("<COLR>" = "green")]>
code3
<@ELSE:[]>
code4
<@END:[IF]>
```

The *special symbols* are replaced as usual before the *command* is processed. For the first one of the first three *boolean-condition* in the example above that returns `true`, the corresponding *code* is retained, and the rest of the **@IF** *command* block, including each other *code*, is removed. Notice that the *boolean-condition* of the last `@ELSE` option is `true` by default. If the first three *boolean-condition* all return `false`, then only $code_4$ is retained.

Also notice that, in this example, the first *boolean-condition* returns an integral value, but each other *boolean-condition* returns a boolean value.

Additional Information

&FNT ♦

<@INSERT:[[PATH="*file-path*", ["*src-path*"]] [START=[P | S | E | A](*start-string*) [(*cust-pat*)…], [{first} | last | *match-num*$_1$]] [OFFSET=*offset*] [END=[P | S | E | A](*end-string*) [(*cust-pat*)…], [{first} | last | *match-num*$_2$]] [ENDOFF=*end-offset*] [SCRIPT=(*ETAC-script*)] [DEFER]]>

The **@INSERT** *command* inserts text lines immediately below itself. The inserted text lines originally exist between specified lines (*start-string*, *end-string*) of either an external text file (*file-path*) if ‹PATH=› is present, or the current *template lines* if it is not. If DEFER is absent, the processing of this *command* occurs before any substitutions are made in the *template lines*, that is, before the first stage. If DEFER is present, processing occurs after **@IF** *command* processing (at stage 11).

Special symbols and **&FNT** *instructions* are activated prior to this *command* being activated. Note, however, that if the DEFER option is absent, this *command* is processed at the first stage, and so no **@SCRIPT** *commands* would have yet been processed.

file-path specifies the file path to a text file and must be enclosed within double quotes. If *file-path* is an empty string and ‹SCRIPT=› is absent, or no text line in the text file matches *start-string* or *end-string* (where specified), then this *command* is removed and no insertion occurs. If the file specified by *file-path* does not exist, a new one is created. If *src-path* is present, it must be enclosed within double quotes, and the created file will be a copy of the file specified by *src-path*, otherwise the created file will be empty. If *src-path* is a relative path, then it is relative to the directory containing the *template files*.

start-string specifies a ‘start line’, and *end-string* specifies an ‘end line’ within the text file specified by *file-path*. The search for the *end line* begins from the line after the *start line*. The text lines between the *start line* and *end line* (not inclusive) are inserted below this *command* (which is removed); the content of the file specified by *file-path* is unaffected. *start-string* and *end-string* with *cust-pat* can be *pattern strings* as described for the **@OUTPUT** *command*. Note that *start-string*, *end-string*, and *cust-pat* must be enclosed within parentheses.

If ‹START=› is present, then first means that the first text string that matches *start-string* in the text file is the *start line*, and last means that the last matching text line is the *start line*. *match-num*$_1$ could be a positive or negative integer. It determines which matched text line is to be the *start line*, searching from the beginning of the file (if *match-num*$_1$ is positive) or searching backwards from the end of the file (if *match-num*$_1$ is negative). A value of zero for *match-num*$_1$ is invalid. For example, if *match-num*$_1$ is 3 then the third line matching *start-string*, searching from the beginning of the text file, will be the *start line*.

If ‹START=› is present, the ‹OFFSET=› option “moves” (redefines) the *start line* (specified by *start-string*) *offset* lines from where it was found, placing it at another line which now becomes the *start line*. *offset* is an integer. A positive value moves the original *start line* down, and a negative value moves the original *start line* up. The **default** value of *offset* is zero (the original *start line* is not moved and so it remains as specified by *start-string*). The *start line* cannot be moved beyond an imaginary line before the first line or past the last line in the text file.

If ‹START=› is absent and ‹OFFSET=› is present, then *offset* indicates a line number of the text file. The text line at that number will be the *start line*. A positive value of *offset*

indicates a line number from the beginning of the text file; the first line is line number 1 (one), the second line is line number 2, and so on. A negative value of *offset* indicates a line number from the end of the text file; the last line is line number -1 and the second last line is line number -2, and so on. If *offset* is zero, the *start line* is an imaginary line before the first line of the text file.

If both ‹START=› and ‹OFFSET=› are absent, then the *start line* is effectively an imaginary line before the first line of the text file.

If ‹END=› is present, then first means that the first text line after the *start line* that matches *end-string* in the text file is the *end line*, and last means that the last matching text line found after the *start line* is the *end line*. *match-num*$_2$ could be a positive or negative integer. It determines which matched text line is to be the *end line*, searching from the line after the *start line* (if *match-num*$_2$ is positive) or searching backwards from the end of the file (if *match-num*$_2$ is negative). A value of zero for *match-num*$_2$ is invalid. For example, if *match-num*$_2$ is 7 then the seventh line matching *end-string*, searching from the line after the *start line*, will be the *end line*.

If ‹END=› is present, the ‹ENDOFF=› option logically "moves" (redefines) the *end line* (specified by *end-string*) *end-offset* lines from where it was found, placing it at another line which now becomes the *end line*. *end-offset* is an integer. A positive value moves the original *end line* down, and a negative value moves the original *end line* up. The **default** value of *end-offset* is zero (the original *end line* is not moved and so it remains as specified by *end-string*). The *end line* cannot be moved beyond an imaginary line before the *start line* or past the last line in the text file.

If ‹END=› is absent and ‹ENDOFF=› is present, then *end-offset* indicates a line number of the text file. The text line at that number will be the *end line*. A positive value of *end-offset* indicates a line number from the beginning of the *start line*; the first line after the *start line* is line number 1 (one), the second line is line number 2, and so on. A negative value of *end-offset* indicates a line number from the end of the text file; the last line is line number -1 and the second last line is line number -2, and so on. If *end-offset* is zero, the *end line* is an imaginary line after the last line of the text file.

If both ‹END=› and ‹ENDOFF=› are absent, then the *end line* is an imaginary line after the last line of the text file, so the text from the *start line* to the rest of the text file is inserted beneath this *command*.

If ‹SCRIPT=› is present, *ETAC-script* is activated just before the text lines are inserted. *ETAC-script* has the same syntax and operates in the same manner as described in the **@EDIT** *command*, except that the *text array* data object will contain the text lines to be inserted by this *command*. If *file-path* is an empty string, then the *text array* data object will initially be empty but may be filled by *ETAC-script*. This allows *ETAC script* to insert text lines into the current *template lines*. Note that *ETAC-script* must be enclosed within parentheses.

If the file specified by *file-path* contains *template lines* with *commands* that are spread over more than one line, then the **@cgCvtTmplData** function will need to be called at ‹SCRIPT=›, as shown in the following illustration.

```
<@INSERT:[PATH="Template.ecgt" ... SCRIPT=(@cgCvtTmplData())]>
```

Note that the **@INSERT** *command* can be used to copy a section of the existing *template lines* to the position below itself, possibly with modification. For example, the following *template lines*

```
I am an ordinary line
I am special
An ordinary line here
Another one here
I too am special
I am not
<@INSERT:[START=(I am special) END=(I too am special) ENDOFF=1
          SCRIPT=(TArr :-; TArr.tlIndentLines(3 "+"); TArr;)]>
Just another line here
```

results in the following *output lines*

```
I am an ordinary line
I am special
An ordinary line here
Another one here
I too am special
I am not
+++An ordinary line here
+++Another one here
+++I too am special
Just another line here
```

The bold text in the *output lines* above has been inserted by the **@INSERT** *command* into (an internal copy of) the original *template lines* because the keyword ‹PATH=› was absent from that *command*. The *end line* is one line below the specified one at ‹END=› due to ‹ENDOFF=1›. The *command* inserted the text lines between, but not including, the *start line* and the *end line*. The *ETAC script* at ‹SCRIPT=› indented the lines with three “plus” (+) characters before the insertion was made.

The *command* ‹@INSERT:[PATH="*file-path*"]› inserts beneath itself the whole content of the file specified by *file-path* before the existing *template lines* are processed.

Additional Information

@OUTPUT ▪ @EDIT

Other Information

@cgCvtTmplData ♦

```
<@JOIN:[]>
template-lines
<@END:[JOIN]>
```

The **@JOIN** *command* concatenates consecutive *template lines* (*template-lines*) into a single *template line* without the end-of-line characters. This *command* can be nested; the inner **@JOIN** *commands* are processed before the outer ones.

Example

Suppose that <FRUITS> has values apples, oranges, bananas. Then the following

```
<@JOIN:[]>
I have some
<&OMIT:[STR=( and) NLAST]> <FRUITS:#1><&OMIT:[STR=(,)]>
. And I like them all.
<@END:[JOIN]>
```

produces the *output line*

```
I have some apples, oranges, and bananas. And I like them all.
```

♦

```
<@OUTPUT:[[PATH="file-path", ["src-path"]] [MARK=[P | S | E | A](marker-string) [(cust-
    pat)…], [{first} | last | match-num]] [OFFSET=offset] [DELETE=num-lines |
    DELETETO=([P | S | E | A](end-string) [(cust-pat)…] | eof)] [ALIGN | ALIGNA | ALIGNB |
    INDENT=num-spaces] [UPDATE] [BACKUP] [FLUSH]]>
template-lines
<@END:[OUTPUT]>
```

The **@OUTPUT** *command* inserts *template lines* (*template-lines*), after those lines have been processed, into a (possibly existing) text file (*file-path*) before or after a line relative to a marked place (*marker-string*) within that file, and possibly deleting a number of lines (*num-lines*) after the marked place in the file first. Two pieces of information are needed for this: the target *file-path*, and a *marker-string* (ie: a text line within the text file) indicating where the processed *template-lines* are to be inserted.

If ‹PATH=› is specified, *file-path* can be a full or relative file path and must be enclosed within double quotes. If *file-path* is a relative path, it is relative to the path specified on the command line (GEN_DIR=*gen-dir*) or input dialog box for the directory into which *generated files* are to be put, or if no such directory is specified, then *file-path* is relative to the current directory. If the file specified by *file-path* does not exist, a new one is created. If *src-path* is present, it must be enclosed within double quotes, and the created file will be a copy of the file specified by *src-path*, otherwise the created file will be empty. If *src-path* is a relative path, then it is relative to the directory containing the *template files*. If *file-path* is an empty string, a unique program-generated file name of the form ‹ECGOutput**...**.txt›, where **...** is an eight digit random number, will be created on the **Windows**® Desktop, and used as *file-path*. *file-path* could specify a file that was already generated by the **ETAC Code Generator** itself (typically via the **@GEN** *command*).

If ‹PATH=› is absent, the *output file* path specified in the *header block* (at ‹@O=*out-path*›) or by the user is assumed as *file-path*. In that case, the **@OUTPUT** *command* and all the *template lines* below it are ignored when this *command* is processed.

marker-string specifies a 'marker' within the text file specified by *file-path*. The search for the *marker* is from the beginning of the text file. *cust-pat* is a custom *pattern string* that may be used with *marker-string*. Note that *marker-string* and *cust-pat* must be enclosed within parentheses.

marker-string and its qualifier have the following format (*M* is the *marker-string*, and *T* is the current text line in the text file specified by *file-path*):

P(*M*)	implies that *M* contains a *pattern string*. The match is for any part of *T*.
S(*M*)	implies that *M* contains a *pattern string*. The match is for the start (initial) part of *T*.
E(*M*)	implies that *M* contains a *pattern string*. The match is for the end part of *T*.
A(*M*)	implies that *M* contains a *pattern string*. The match is for all of *T*.
(*M*)	without the prefixes above implies that it is a plain string. The match is for all of *T*.

If ‹MARK=› is present, then first means that the first text string that matches *marker-string* in the text file is the *marker*, and last means that the last matching text string is the *marker*. *match-num* could be a positive or negative integer. It determines which matched text line is to be the *marker*, searching from the beginning of the text file (if *match-num* is positive), or searching backwards from the end of the file (if *match-num* is negative). A value of zero for *match-num* is invalid. For example, if *match-num* is 3 then the third line matching *marker-string*, searching from the beginning of the text file, will be the *marker*.

If both ‹MARK=› and ‹OFFSET=› are absent, or if a text line in the text file does not match *marker-string*, then *template-lines* is appended to the text file specified by *file-path*.

If ‹MARK=› is present, the ‹OFFSET=› option "moves" (redefines) the *marker* (specified by *marker-string*) *offset* lines from where it was found, placing it at another line which now becomes the *marker*, and *template-lines* is inserted after that *marker*. *offset* is an integer. A positive value moves the *marker* down, and a negative value moves the *marker* up. The **default** value of *offset* is zero (the *marker* is not moved and so it remains as specified by *marker-string*). The *marker* cannot be moved beyond an imaginary line before the first line or past the last line in the text file. To insert code after the text line just above the *marker*, specify −1 for *offset*.

If ‹MARK=› is absent and ‹OFFSET=› is present, then *offset* indicates a line number of the text file. A positive value of *offset* indicates a line number from the beginning of the text file; the first line is line number 1 (one), the second line is line number 2, and so on. A negative value of *offset* indicates a line number from the end of the text file; the last line is line number −1 and the second last line is line number −2, and so on. *template-lines* is inserted before the specified line number, unless *offset* is zero, in which case *template-lines* is appended to the text file.

The ‹DELETE=› option deletes *num-lines* text lines from the text file specified by *file-path* before *template-lines* are inserted. *num-lines* is a positive integer or zero. The first line deleted is at the position where the first insertion occurs. If text insertion occurs after the last line of the file, then no lines are deleted because there would not be any lines after the last line.

The ‹DELETETO=› option deletes the text lines from the first line after the *marker* up to, but not including, the line specified by *end-string* before *template-lines* are inserted. *end-string* specifies an 'end line' within the text file specified by *file-path*, and has the same format as *marker-string*. The search for the *end line* begins from the line after the *marker*. If the *end line* is not found then no lines are deleted. If eof is specified, then the rest of the text lines in the text file are deleted. Note that *end-string* must be enclosed within parentheses.

The ALIGN option aligns *template-lines* with the first non-white space character of the text line indicated by the *marker*. If that line is blank then this option is ignored.

The ALIGNA option aligns *template-lines* with the first non-white space character of the first non-blank line after the line indicated by the *marker*. If no such line exists then this option is ignored.

The ALIGNB option aligns *template-lines* with the first non-white space character of the closest non-blank line before the line indicated by the *marker*. If no such line exists then this option is ignored.

For ALIGN, ALIGNA, and ALIGNB, the fill characters to produce the alignment are spaces.

The ‹INDENT=› option indents each *template line* in *template-lines* with *num-spaces* space characters just before the *template line* is ready to be output. *num-spaces* is a non-negative integer.

The UPDATE option reloads the disk data of the text file specified by *file-path* into the **ETAC Code Generator** before processing.

The BACKUP option creates a backup of the text file specified by *file-path*, if it exists, before being written to by the **ETAC Code Generator**. If *file.ext* is the format of file name specified in *file-path*, then the backup file name will be *file*~backup.*ext*. If the backup file already exists then it will be overwritten automatically without warning. When the BACKUP option is specified, the data to be written to the text file is internally marked as needing a backup of the existing file data. When the backup file is created, the internal mark is removed. If the text file is the *output file*, then this option is ignored.

The FLUSH option causes the accumulated data for the text file specified by *file-path* to be written to the disk immediately after this *command* ends, rather than being written at the end of the current *ECG session*. This is useful if an external program needs to access the output data (typically via the **@SCRIPT** *command*) before the current *ECG session* ends. If the text file is the *output file*, then this option is ignored.

All output data for which the FLUSH option was not specified is written to the appropriate text files after all the *template lines* in the current *ECG session* have been processed.

The **@OUTPUT** *command* can be nested; the inner levels of the *command* are processed before the outer levels. The *command* is deleted after being processed. If ‹PATH=› is absent, all *template lines* above the parent **@OUTPUT** *command*, and all *template lines* below the beginning of the current **@OUTPUT** *command* are ignored as though they did not exist. If no parent **@OUTPUT** *command* exists, then only the *template lines* below the beginning of the current **@OUTPUT** *command* and are ignored. Line number offsets and searching will be relative to the remaining *template lines*.

The header of an **@OUTPUT** *command* cannot be a *multi-line*.

Examples

The following examples illustrate how the **@OUTPUT** *command* can be used.

```
<@OUTPUT:[]>
Nothing happened here.
<@END:[OUTPUT]>
```

The example above is not particularly useful because it merely replaces itself with the *template line* ‹Nothing happened here.›.

The following **@OUTPUT** *command* inserts variable declarations in a C programming language header file, above a comment, ‹//«DECS»//›, used especially for that purpose. It is assumed that <FILE> has the value of the file path containing existing declarations, <TYPE> has the values int, long, short, and <VAR> has the corresponding values MyVar1, StrLen, Count.

```
<@OUTPUT:[PATH="<FILE>.h" MARK=(//«DECS»//) OFFSET=-1]>
<TYPE:#1><&MI:[POSA=20 SPACES=1]><VAR:#1>;
<@END:[OUTPUT]>
```

If the header file originally contained

```
bool                FileExists;
//«DECS»//
```

then the *command* will insert the new declarations into the header file as shown below

```
bool                FileExists;
int                 MyVar1;
long                StrLen;
short               Count;
//«DECS»//
```

Note that the ‹OFFSET=-1› option causes the insertions to go above ‹//«DECS»//›; if that ‹OFFSET=› option were absent, then the insertions would have gone beneath ‹//«DECS»//›.

The following example is based on the example above, and illustrates how the **@OUTPUT** *command* can be used to update and maintain sections of any text file. The example assumes the same values of <FILE>, <TYPE>, and <VAR> as in the preceding example. If the header file contains

```
//«+DECS+»
bool                FileExists;
//«-DECS-»
```

then the following **@OUTPUT** *command*

```
<@OUTPUT:[PATH="<FILE>.h" MARK=(//«+DECS+») DELETETO=(//«-DECS-»)]>
<TYPE:#1><&MI:[POSA=20 SPACES=1]><VAR:#1>;
<@END:[OUTPUT]>
```

will replace the existing declarations in the header file with the new ones, thus

```
//«+DECS+»
int                 MyVar1;
long                StrLen;
short               Count;
//«-DECS-»
```

Notice the absence of the original declaration of `FileExists`.

The following example replaces the content of `MyFile.txt` after making a backup of it.

```
<@OUTPUT:[PATH="MyFile.txt" BACKUP OFFSET=1 DELETETO=eof]>
...
<@END:[OUTPUT]>
```

Additional Information

See **Pattern String Matching** under chapter 3 of the "*The Official ETAC Programming Language*" document, ETACProgLang(Official).pdf. ♦

```
<@POSTGEN:[INPUT="template-file" OUTPUT="out-file" ARGS=(keyword-arguments)
   [SRC_HEAD] [{NO_PROMPT} | PROMPT]]>
```

The **@POSTGEN** *command* is used to post-generate to a file (*out-file*) from a given *template file* (*template-file*). The *command* is executed as for the **@GEN** *command* but is processed after the **@OUTPUT** *commands* have been executed.

out-path specifies the *output file* path for *template-file*. If *out-file* is an empty string, then the *output file* will be the one specified (at ‹@O=›) in the *header block* of *template-file*. If *out-file* is a question mark (?) character, then the default *generated lines* from *template-file* will be appended to the *output file* of the current *template file*. Otherwise the *output file* will be the one specified at *out-file*, overriding the *output file* path specified in the *header block* of *template-file*. If *out-path* is a relative path, then it is relative to the current directory. Note that *template-file* and *out-file* must be enclosed within double quotes (even when *out-file* is a question mark character).

The other keywords and arguments are as described under the **@GEN** *command*.

Typically, the **@POSTGEN** *commands* are placed last in the *template file*. The *command* can be used to generate from a *template file* that was produced by an **@OUTPUT** *command*.

Additional Information

@GEN

Other Information

@cgGenerate ♦

```
<@REPROCESS:[[PASSES=num-levels] [POST]]>
template-lines
<@END:[REPROCESS]>
```

The **@REPROCESS** *command* reprocesses specified *template lines* (*template-lines*) a number (*num-levels*) of times. If POST is absent, the *command* reprocesses *template-lines* beginning with the stage when the **@SYMBOL** *commands* are activated up to the stage before the **@REPROCESS** *commands* are activated (stages 4 to 17). If POST is present, the *command* reprocesses *template-lines* through all stages up to, and including, the stage that the **@OUTPUT** *commands* are activated (stages 1 to 24). This *command* cannot exist within an **@OUTPUT** *command* if POST is specified.

num-levels is zero or a positive integer and specifies the number of times that *template-lines* is reprocessed. If ‹PASSES=› is absent, the default value of *num-levels* is one. *num-levels* is internally decremented by one each time *template-lines* is reprocessed. When *num-levels* is or becomes zero, the *command* is removed and no further reprocessing of *template-lines* occurs. Note that, regardless of the value of *num-levels*, *template-lines* is processed as usual before this *command* takes effect; *num-levels* specifies the number of addition processing.

The **@REPROCESS** *command* can be nested; the inner level nested **@REPROCESS** *commands* are processed before the outer level **@REPROCESS** *commands*, and are re-processed by those outer level *commands*.

The header of a **@REPROCESS** *command* cannot be a *multi-line*.

Other Information

@EVAL ♦

```
<@SCRIPT:[{ETAC} [PRIOR|POST]]>
ETAC-script
<@END:[SCRIPT]>
```

The **@SCRIPT** *command* activates *ETAC script* (*ETAC-script*). The *ETAC script* has a pre-allocated local dictionary, which is deleted when that script ends. Variable allocations are therefore private to the *ETAC script* within each **@SCRIPT** *command*. However, global variables and ETAC function definitions can be allocated within the **cg** data object (for example: ‹cg.{MyGlobVar :- 10; MyGlobFnt :- fnt:(...){...};};›). Global variables are accessible directly from any *ETAC script* in the *template line block* without using the **cg** data object. If a local variable is identical to a global variable, and the global variable is not accessed via **cg** (for example: cg.MyVar), then the local variable will be accessed.

Various intricacies of the *header block*, *template line block*, and *output file* are available via predefined global functions. Such functions begin with the prefix @cg (for example: **@cg**GetSymbCount(...)). The standard ETAC library procedures are also available for use. In addition, the inclusion file, TACGlobal.PTAC, and pre-processor definitions for the *text array* data object will have automatically been included.

The PRIOR option activates the *ETAC script* before the **@SYMBOL** *commands* are activated.

The POST option postpones the activation of the *ETAC script* until after the **@OUTPUT** *commands* have been activated.

ETAC-script can contain *special symbols*, *instructions*, and *commands* that have been processed at an earlier stage than the **@SCRIPT** *commands*; if the PRIOR option is present, the *ETAC-script* can contain *special symbols* even though they are processed at a later stage.

ETAC-script is written in the ETAC™ programming language, which is a dictionary and stack based interpreted script programming language (the same language in which the **ETAC Code Generator** is implemented). The document ETACOverview.pdf contains an overview of the ETAC programming language, while the document ETACProgLang(Official).pdf contains the official definition of the language.

The **@SCRIPT** *command* is typically used to define global ETAC variables, functions, and procedures for use by **&FNT** *instructions*, or the *command* can be used to do additional processing such as user interaction or disk file data manipulation.

Examples

Assume that the *special symbol* <DAY> has the value today. The following *template lines*

```
<@SCRIPT:[]>
cg.
{
   Message :- fnt:() {"Hello world, I'm joining you <DAY>";};
};
<@END:[SCRIPT]>
Important message: <&FNT:[=({Message();})]>!!!
```

generates the *output line*

```
Important message: Hello world, I'm joining you today!!!
```

Note that the Message function is allocated within the **cg** data object for it to be available globally. Also note that the call to the Message function does not require the **cg** data object to be specified.

The following illustration shows how to assign values to a *special symbol* programmatically (via **@cgAddCmdSymb**) if necessary. Simple straight forward assignments can be done via the **@SYMBOL** *command*; more complicated assignments that cannot be done via the **@SYMBOL** *command* can be done programmatically. Note that *special symbol* assignments usually need to be done with the PRIOR option of the **@SCRIPT** *command* for those symbols to be available at later stages.

```
<@SCRIPT:[PRIOR]>
   [* Assign symbols here. *]
   @cgAddCmdSymb("fruit" ["apples", "oranges", "bananas"]);
<@END:[SCRIPT]>
<@SCRIPT:[]>
cg.Proc :- [* Allocate procedure to the 'cg' data object. *]
{
Fruits :- [* Allocate local variable within the procedure. *]
[
   "<fruit:#1>"<&OMIT:[STR=(,)]>
];

   @cgFormatStr("Which do you like best %1%, %2%, or %3%?" Fruits);
};
<@END:[SCRIPT]>
<&FNT:[=({write_con Proc;})]>
```

The *template lines* above will print the following text to the console window.

```
Which do you like best apples, oranges, or bananas?
```

Notice how the ETAC sequence, Fruits, has been created using a *multi-line*. The resulting sequence is ["apples", "oranges", "bananas"].

Additional Information

See the documents "*An Overview of ETAC*" (ETACOverview.pdf) and "*The Official ETAC Programming Language*" (ETACProgLang(Official).pdf).

Related Information

6_Programming the ETAC Code Generator ▪ 8_EGCL Function Reference

Other Information

@FNT ♦

<@SECTION:[]>

The **@SECTION** *command* is used to separate the *template line block* into separate sections, each of which is fully processed in turn. However, the *output file* and, by default, all other *generated files* are written to disk after the last section has been processed. The scope of all *commands* within a section is limited to that section as though it were the only section in the *template line block*. *ETAC script* variable definitions (within the **cg** data object) and *command symbols* are not affected — they are defined for all subsequent sections. **@SECTION** *commands* take effect as they are encountered — they are not subject to stages.

A *template line block* with no **@SECTION** *commands* is effectively a single section. A **@SECTION** *command* may optionally be the first line of the *template line block*.

Example

The following example illustrates how the *template line block* is partitioned into three sections of *template lines*.

```
...
@endhead@
<@SECTION:[]> (optional)
template-lines1 (section 1)
<@SECTION:[]>
template-lines2 (section 2)
<@SECTION:[]>
template-lines3 (section 3)
```

♦

<@SYMBOL:[NAME="*name*" ARGS=(*value*), ··· [EVAL]]>

The **@SYMBOL** *command* creates a new *special symbol* (*name*) and its values (*value*), and adds them to the list of *command symbols*. The *special symbol* can have more than one *value*, and is used like any other *special symbol*.

name must be enclosed within double quotes, and consists of the proper *special symbol* name syntax (alphanumeric-underscore characters beginning with an alphabetic character). Only UCS-2 (BMP Unicode scalar value) characters are recognised.

value is interpreted literally and *meta-codes* within it are not evaluated unless EVAL is present. Note that *value* must be enclosed within parentheses.

If EVAL is present, then only *special symbols* within *value* are evaluated before the *special symbol name* is created.

If more than one **@SYMBOL** *command* with the same *name* is encountered, each *value* of each subsequent **@SYMBOL** *command* having the same *name* is added to the list of values for that *name*. For example, the following three *template lines*

```
<@SYMBOL:[NAME="FRUIT" ARGS=(apples), (oranges), (bananas)]>
<@SYMBOL:[NAME="FRUIT" ARGS=(pears)]>
<@SYMBOL:[NAME="FRUIT" ARGS=(plums), (apricots)]>
```

are equivalent to the following *template line*.

```
<@SYMBOL:[NAME="FRUIT" ARGS=(apples), (oranges), (bananas), (pears),
(plums), (apricots)]>
```

Example

The following example shows how the **@SYMBOL** *command* can be used.

```
<@SYMBOL:[NAME="FRUIT" ARGS=(apples), (oranges), (bananas)]>
<@SYMBOL:[NAME="AMOUNT" ARGS=(5)]>
I have <AMOUNT+2.-1> <FRUIT:#1>.
```

Noting that the last line above is a *multi-line*, the preceding *template lines* produce the following *output lines*.

```
I have 3 apples.
I have 5 oranges.
I have 7 bananas.
```

The *special symbol*, <AMOUNT+2.-1>, converts to the value of $\langle 5 + 2 \times (c - 1) \rangle$, where c is the current *output line* number beginning with zero.

Other Information

@cgAddCmdSymb ♦

3

Processing Stages

The **ETAC Code Generator** uses a unique and sophisticated declarative template language to produce *generated lines*. The *meta-codes* of that language are not processed in order from the top of the *template line block* to the bottom in one pass. Rather, all the *meta-codes* of the same type are processed in their own pass from top to bottom separately. Such a processing pass is called a "stage". There are about thirty stages in processing a *template line block*. Of course, stages for which there are no corresponding *meta-codes* in the *template file* are skipped.

If the *template line block* is partitioned into sections as defined in the **@SECTION** *command*, then each section undergoes all the stages before the next section is processed. The sections are processed separately in the order in which they occur in the *template line block*.

The following example illustrates the concept of activating the *meta-codes* in stages. The **@IF** *command* is activated at stage 11, **@EDIT** at stage 23, **@OUTPUT** at stage 24, **@POSTGEN** at 26, and **@SCRIPT**[POST] at 27. In the example below, the **@IF** *command* is activated first, resulting in one of the two **@OUTPUT** *command* headers. Next, the two **@EDIT** *commands* are activated in the given order, followed by the two **@OUTPUT** *commands*; the top **@OUTPUT** *command* is activated before the remaining second one. The **@POSTGEN** *command* is activated next, followed by the **@SCRIPT**[POST] *command*. The numbers in parentheses indicate the order that the *meta-codes* are activated.

```
<@OUTPUT:[PATH=...]> (4)
...
<@END:[OUTPUT]>

<@SCRIPT:[POST]> (7)
...
<@END:[SCRIPT]>

<@EDIT:[SCRIPT=(...)]> (2)
...
<@END:[EDIT]>

<@IF:[...]> (1)
<@OUTPUT:[PATH="MyFile1.txt"]> (5)
<@ELSE:[]>
<@OUTPUT:[PATH="MyFile2.txt"]> (5)
<@END:[IF]>
...
<@END:[OUTPUT]>

<@EDIT:[SCRIPT=(...)]> (3)
<@POSTGEN:[...]> (6)
...
<@END:[EDIT]>
```

There are a few points to be aware of in the example above. The *commands* are not activated from top to bottom as written. The **@IF** *command* only outputs one of the two headers for the **@OUTPUT** *command* which is activated later. The script of the second **@EDIT** *command* can alter

the **@POSTGEN** *command* within, before that *command* is activated later. The **@SCRIPT**[POST] *command* is activated last, even though it is near the top of the *template line block*.

The following section defines the stage at which each type of *meta-code* is activated.

3.1 Meta-code Processing Stages

There are a number of processing stages for the *meta-codes* of a *template line block*. The instances of the various types of *meta-codes* are activated at different stages. A **&FNT** *instruction* within a *meta-code* (where permitted) that is activated prior to stage 16, is activated when the *meta-code* itself is activated unless the **&FNT** *instruction* is immediately inside a *protection instruction*. Protected *meta-codes* are not activated while they are protected. A *meta-code* that is not immediately inside a protection *instruction* is not protected.

The following table shows the stage numbers (**SN**) of the various *meta-codes*. A lower-numbered stage is performed before a higher-numbered stage. At each stage, all the *meta-code* instances for that stage in all the *template lines* (in the same section as defined by the **@SECTION** *command*) are activated before the next stage commences. The table also indicates whether a *meta-code* can be nested (**Ns**).

Meta-code Stage Numbers

Meta-code	SN	Ns	Comments
@INSERT (no DEFER)	1		Pre-activates *special symbols* and &FNT *instructions*.
&C	2	Y	
@CMT	2	Y	
@SCRIPT[PRIOR]	3	N	Pre-activates *special symbols* in the *command's* body.
@SYMBOL	4		Pre-activates *special symbols* if EVAL is specified.
continued lines (...\)	5		
special symbols (non *multi-line*)	6		
special symbols (*multi-line*)	7		
&OMIT	7	N	
@SCRIPT (no POST or PRIOR)	8	N	
@EVAL[PRIOR]	9	Y	Pre-activates &FNT *instructions*[a].
@DO	10	Y	Pre-activates &FNT *instructions*[a].
@IF...@ELSE...@END[IF]	11	Y	
@INSERT[DEFER]	12		Pre-activates &FNT *instructions*.
&DATE	13	N	Pre-activates &FNT *instructions*.
&HPAR	13		
@EVAL (no PRIOR)	14	Y	Pre-activates &FNT *instructions*[a].
@GEN	15		Pre-activates &FNT *instructions*.
&FNT	16	N	
<&>	17		
@REPROCESS (no POST)	18	Y	Reprocesses *template lines* from stage 4 to stage 17.
&sq &dq &bs &n &t &v &b &r &f &a &U+ &x &eol &lp &rp &lb &rb &ls &rs &lt &gt	19		
&MI	20	N	
@JOIN	21	Y	

&DEL	22	N	
@EDIT	23	Y	
<&<...>>	24	Y	Activated with @OUTPUT *command*.
@OUTPUT	24	Y	Text lines containing EOLs are treated as separate lines.
@REPROCESS[POST]	25	Y	Reprocesses *template lines* from stage 1 to stage 24.
@POSTGEN	26		
@SCRIPT[POST]	27	N	
@DELETE	28		
<&<...>>	29	Y	

a Applies only within the *meta-code* itself, not to the *meta-code's* body.

"EOL" stands for "end-of-line character sequence".

4

Input Dialog Box

The input dialog box allows the user to enter information for the **ETAC Code Generator** to create *generated files*. The dialog box is displayed only if the `PROMPT` keyword is specified in the *input arguments* of ETACCodeGen.btac, or in the **`@GEN`** or **`@POSTGEN`** *commands*. The initial fields displayed in the dialog box are the ones specified in the command line or in the said *commands*. The input dialog box can also be presented via the **`@cgShowNewDialog`** function from within a *template file*.

The following section describes the components of the input dialog box.

4.1 Dialog Box Details

A user can enter or modify the **ETAC Code Generator** parameters (*input arguments*) through an input dialog box for a particular code generating session (*ECG session*). The dialog box is displayed by specifying `PROMPT` on the command line or when setting up a **Windows**® shortcut to RunETAC.exe, which activates the **ETAC Code Generator** (ETACCodeGen.btac). The dialog box can also be displayed by specifying `PROMPT` for the **`@GEN`** and **`@POSTGEN`** *commands*.

For example, to display the input dialog box from a shortcut to RunETAC.exe, use

```
...\RunETAC.exe NO_EXIT_MSG SCRIPTS="ETACCodeGen.btac" '... PROMPT ...'
```

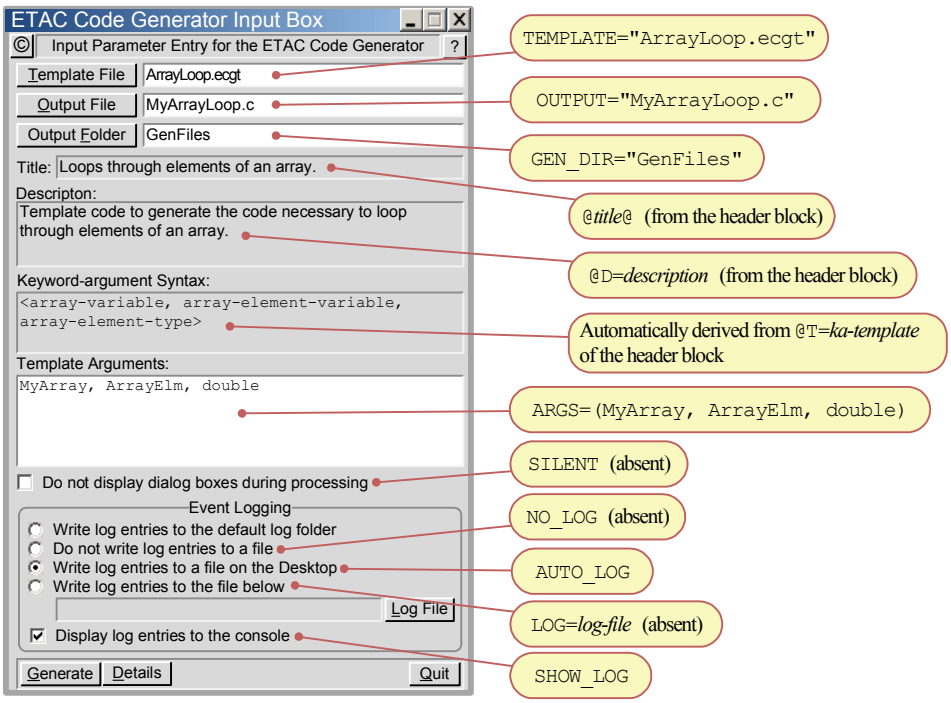
To display the input dialog box from a shortcut to ETACCodeGen.btac, use

```
...\ETACCodeGen.btac '... PROMPT ...' NO_EXIT_MSG
```

Note: specify `NO_EXIT_MSG` if the `RunETAC` exit message box is not required to be displayed.

The input dialog box appears as shown below (the field values are only for illustration).

Input Dialog Box for the ETAC Code Generator



The diagram above is an example of the input dialog box that is displayed when the following command line is entered as the `Target` of a **Windows**® shortcut file (note that, in this example, RunETAC.exe is assumed to exist in the same directory as the shortcut).

```
RunETAC.exe SCRIPTS=ETACCodeGen.etac 'TEMPLATE="ArrayLoop.ecgt"
OUTPUT="MyArrayLoop.c" GEN_DIR="GenFiles" ARGS=(MyArray, ArrayElm, double)
AUTOLOG SHOW_LOG PROMPT'
```

The 'Generate' button generates the requested files, the 'Details' button shows the first comment at ‹@C=› within the specified *template file* if the comment is enclosed within double quotes, and the 'Quit' button exits the **ETAC Code Generator**. The '?' button displays the command syntax for the *input arguments*, and the '©' button displays the copyright information.

The default log folder is the one specified at the ‹LogDir=*log-dir*› parameter of the initialisation file (ETACCodeGen.ini).

5

Operating the ETAC Code Generator

This chapter shows the various ways to execute the official implementation of the **ETAC Code Generator** (which requires the pre-installed **Run ETAC Scripts** package). To execute the self-contained executable implementation, see Appendix B: Self-contained ETAC Code Generator.

5.1 Command Line

The **ETAC Code Generator** can be run from either the **MS-DOS**® or **Windows**® environment. In either case, a command line needs to be specified. From within **Windows**, the command line is typically entered in a shortcut file to the **ETAC Code Generator** (ETACCodeGen.btac).

The current directory must be the directory containing ETACCodeGen.btac or ETACCodeGen.etac.

In the ‹Start In› entry (the current directory) of the shortcut properties, enter the following:

```
ecg-path
```

In a ‹*.cmd› or ‹*.bat› file, enter the following to change the current directory before calling the **ETAC Code Generator**.

```
cd /D "ecg-path"
```

In the Target entry of the shortcut properties, or in a ‹*.cmd› or ‹*.bat› file, enter the following:

```
"path\RunETAC.exe" NO_EXIT_MSG SCRIPTS="ETACCodeGen.btac" 'args'
```

Alternatively, if the ETAC™ programming language has been installed, the following can be entered:

```
"ecg-path\ETACCodeGen.btac" 'args' NO_EXIT_MSG
```

where *path* is the directory path of the directory containing RunETAC.exe, *ecg-path* is the directory path of the directory containing the **ETAC Code Generator** program (ETACCodeGen.btac or ETACCodeGen.etac), and *args* represents the *input arguments* for the **ETAC Code Generator**. *args* may be delimited by single or double quotes, which are ignored. Typically, however, *args* is delimited by single quotes, and file paths within *args* are delimited by double quotes. This results in the least interpretation problems of *args*. NO_EXIT_MSG can be specified to prevent the RunETAC exit message box from being displayed.

The **ETAC Code Generator** command line *input arguments* specification (*args*) is defined as follows.

Command Line Input Arguments

```
[ [INI_DIR=ini-dir] TEMPLATE=tmpl-file (ARGS=(kw-args) | ARG_FILE=arg-file)
[OUTPUT=out-file] [GEN_DIR=gen-dir] [NO_LOG | AUTOLOG | LOG=log-file] [PROMPT]
[SHOW_LOG] [SILENT] ]
```

If no *input arguments* are present, the copyright information and a description of the **ETAC Code Generator** along with a summary of the *input arguments* format is presented in a dialog box. The following keywords are case-sensitive.

INI_DIR=*ini-dir*

ini-dir is the full or relative directory path for the initialisation file (ETACCodeGen.ini) used by the **ETAC Code Generator**. A relative path is relative to the current directory. *ini-dir* may be delimited by single or double quotes, which are ignored. If this option is absent or the initialisation file does not exist in the specified directory, *ini-dir*, then the **ETAC Code Generator** will first search the current directory for the initialisation file, and if the file is not found then the **ETAC Code Generator** will search the `Windows` directory. If the initialisation file is not found, then the **ETAC Code Generator** will use default values for the initialisation file parameters. (See 5.2_Initialisation File)

TEMPLATE=*tmpl-file*

tmpl-file is the full or relative file path of the *template file* used by the **ETAC Code Generator** as the source file for generating the *output file* and other files. A relative path is relative to the directory specified in the ‹`ECGTSourceDir`=*tmpl-dir*› parameter of the initialisation file (ETACCodeGen.ini). If the initialisation file is not found, then the **ETAC Code Generator** will use default values for the initialisation file parameters. *tmpl-file* may be delimited by single or double quotes, which are ignored. A full stop (**.**) at the beginning of *tmpl-file* represents the full path of the current directory. The file name in *tmpl-file* typically has an extension of ‘ecgt’. If the `PROMPT` keyword is specified, *tmpl-file* can be an empty string. (See 5.2_Initialisation File)

ARGS=(*kw-args*)

kw-args is a string specifying the *template arguments* used by the **ETAC Code Generator** to generate the *output file* and other files. *kw-args* must be enclosed within parentheses and must conform to *ka-template* specified at the ‹`@T`=*ka-template*› parameter in the *header block*.

ARG_FILE=*arg-file*

arg-file is a single-quoted or double-quoted file path to a file containing the *template arguments* used by the **ETAC Code Generator** to generate the *output file* and other files. A relative path is relative to the current directory. The content of the file must conform to *ka-template* specified at the ‹`@T`=*ka-template*› parameter in the *header block*.

OUTPUT=*out-file*

out-file is the full or relative file path for the *output file* generated by the **ETAC Code Generator**. A relative path is relative to the current directory. *out-file* overrides the file specified at the ‹`@O`=*out-file*› parameter in the *header block*. *out-file* may be delimited by single or double quotes, which are ignored. If *out-file* is a single or double quoted empty string, then the *output file* will be generated on the **Windows**® `Desktop` as a unique program-generated file name of the form ‹ECGOutput**...**.txt›, where **...** is an eight digit random number. If this option is absent, the *out-file* specified at the ‹`@O`=*out-file*› parameter in the *header block* of the *template file* specified at ‹`TEMPLATE`=› is assumed for *out-file*.

GEN_DIR=*gen-dir*

gen-dir is the full or relative directory path to contain the files (excluding the *output file*) generated by the **ETAC Code Generator**. A relative path is relative to the current directory. *gen-dir* applies only to the *generated files* that are specified as relative file paths of the **`@OUTPUT`** *commands* within the *template file*. *gen-dir* may be delimited by single or double quotes, which are ignored. If this option is absent, then the current directory will be assumed for the value of *gen-dir*.

NO_LOG

If this option is present, then no log file will be produced.

AUTOLOG

If this option is present, then the log file produced by the **ETAC Code Generator** will be written to the **Windows**® `Desktop`. The log file has the format ‹ECGLogFile-*date*.log›, where *date* is of the form `YYYYMMDD`, and is the date that the **ETAC Code Generator** is executed (for

example, if the **ETAC Code Generator** is executed on the 23rd of April, 2015, then the log file will be named ‹ECGLogFile-20150423.log›). Log outputs produced on the same day are appended to the same log file. The log file will be written as a UTF-8 file (with a BOM signature), unless the file characters are all a subset of the Windows-1252 character set, in which case the file will be written as a Windows-1252 file. If the log file does not exist, then a new one is created.

LOG=*log-file*

log-file is the full or relative file path to which log outputs (activity and error messages) are written by the **ETAC Code Generator**. A relative path is relative to the current directory. If *log-file* does not end with a forward-slash (/) or backslash (\), then it specifies the file path of the log file. If the log file already exists, then the log outputs are appended to that file, otherwise a new file is created. If *log-file* ends with a forward-slash (/) or backslash (\), then it specifies a directory into which the log file is written. In this case, the log file has the format ‹ECGLogFile-*date*.log›, where *date* is of the form YYYYMMDD, and is the date that the **ETAC Code Generator** is executed (for example, if the **ETAC Code Generator** is executed on the 23rd of April, 2015, then the log file will be named ‹ECGLogFile-20150423.log›). Log outputs produced on the same day are appended to the same log file. If all the log file keywords (NO_LOG, AUTOLOG, LOG=) are absent from the command line, then the directory specified in the ‹LogDir=*log-dir*› parameter of the initialisation file (ETACCodeGen.ini) will be assumed as the directory path for the value of *log-file*. In this case, the log output is written to the dated log file as explained above. *log-file* may be delimited by single or double quotes, which are ignored. The log file will be written as a UTF-8 file (with a BOM signature), unless the file characters are all a subset of the Windows-1252 character set, in which case the file will be written as a Windows-1252 file. (See 5.2_Initialisation File)

Note that a backslash at the end of a quoted string needs to have a space following it, otherwise the ending quote may not be regarded as a string delimiter.

PROMPT

If this option is present, then the **ETAC Code Generator** will display a dialog box for the user to enter or modify the *input arguments* before generating the files. The initial *input arguments* displayed in the dialog box are the ones set on the command line.

See chapter 4_Input Dialog Box for details.

SHOW_LOG

If this option is present, then log outputs (activity and error messages) produced by the **ETAC Code Generator** are also displayed to the console window.

SILENT

If this option is present, then the **ETAC Code Generator** does not display dialog boxes other than the input dialog box if the PROMPT option is present.

Examples

The **orange** text in the following examples is the *input arguments* for the **ETAC Code Generator**. The current directory must be the directory containing ETACCodeGen.btac.

The following example is entered in a command prompt, the Target of a **Windows**® shortcut, or a ‹*.cmd› or ‹*.bat› file. An input dialog box is presented for the user to type additional arguments to the **ETAC Code Generator**.

```
"C:\...\RunETAC.exe" SCRIPTS="ETACCodeGen.btac"
   'GEN_DIR="C:\Development\MyProject" TEMPLATE="" ARGS=() AUTOLOG PROMPT
   SHOW_LOG'
```

The following example is the same as the preceding example but entered in the Target of a **Windows**® shortcut to ETACCodeGen.btac.

```
"C:\...\ETACCodeGen.btac" 'GEN_DIR="C:\Development\MyProject" TEMPLATE=""
   ARGS=() AUTOLOG PROMPT SHOW_LOG'
```

In the following example, the PATH environment variable is assumed to include the directory where RunETAC.exe is installed, and the current directory is assumed to be the directory containing ETACCodeGen.btac. Two independent *ECG sessions* are run via the command processor (cmd.exe); the second one runs after the first one has finished.

The first session generates files into the MyProject directory relative to the current directory. The session uses the *template file* FindElement.ecgt existing in the directory specified in the ‹ECGTSourceDir=*tmpl-dir*› parameter of the initialisation file (ETACCodeGen.ini) to generate those files. FindElement.ecgt uses the *template arguments* ‹FUNCTION=Find, , pSearchVal, char * ARRAY=CustomerRec, CustName› as the parameters to generate the files. The *output file* is ReadMe.txt generated into the current directory. Events are logged to a dated file on the **Windows**® Desktop (AUTOLOG), and also to the console (SHOW_LOG).

After the first *ECG session* has completed, the second session generates files into the MyOtherProject directory relative to the current directory. It uses the *template file* EventProcessing.ecgt existing in the current directory. The *output file* is MyOtherProject.txt generated into the current directory. Before this session begins, however, an input dialog box is displayed with the dialog box's fields initialised as specified in the command line *input arguments* (shown for the second session in blue below). The user can change the value of those fields, and must enter the appropriate *template arguments* (under Template Arguments in the input dialog box) for the EventProcessing.ecgt *template file*. The *template arguments* in the dialog box is initialised with ‹CLASS=MyClass, mc EVENT=MOUSE_DOWN, ...› for the user to modify.

```
"RunETAC.exe" SCRIPTS="ETACodeGen.btac" 'GEN_DIR="MyProject"
   TEMPLATE="FindElement.ecgt" ARGS=(FUNCTION=Find, , pSearchVal, char *
   ARRAY=CustomerRec, CustName) OUTPUT="ReadMe.txt" AUTOLOG SHOW_LOG',
   "ETACodeGen.btac" 'GEN_DIR="MyOtherProject"
   TEMPLATE=".\EventProcessing.ecgt" ARGS=(CLASS=MyClass, mc
   EVENT=MOUSE_DOWN, ...) OUTPUT="MyOtherProject.txt" PROMPT'
```

5.2 Initialisation File

The **ETAC Code Generator** typically requires an INI file (initialisation file) which contains information relevant to all *ECG sessions*. The INI file is named ETACCodeGen.ini. The format of the content of the INI file is as follows.

```
[Settings]
ECGTSourceDir=tmpl-dir
LogDir=log-dir
```

ECGTSourceDir=*tmpl-dir*

tmpl-dir is the full or relative directory path of the directory containing the **ETAC Code Generator** *template files* (*.ecgt). A relative path is relative to the current directory. *tmpl-dir* may be delimited by single or double quotes, which are ignored. If this parameter is absent, then the current directory will be assumed for the value of *tmpl-dir*.

LogDir=*log-dir*

ini-dir is the default full or relative directory path of the directory into which the log files produced by the **ETAC Code Generator** will be written. A relative path is relative to the current directory. Log files are written into this directory only if all the log file keywords (NO_LOG, AUTOLOG, LOG=) are absent from the *input arguments*, or if the log file is specified to be written to the default log folder in the input dialog box. The log files have the format

‹ECGLogFile-*date*.log›, where *date* is of the form YYYYMMDD (for example, if today is the 23rd of April, 2015, then the log file will be named ‹ECGLogFile-20150423.log›). *date* is the date that the **ETAC Code Generator** is executed. *log-dir* may be delimited by single or double quotes, which are ignored. If this parameter is absent, then the current directory will be assumed for the value of *log-dir*.

The INI file can be specified on the **ETAC Code Generator** command line, or exists in the current directory or Windows directory.

If the **ETAC Code Generator** does not find the INI file, then *src-dir* and *log-dir* will both be assumed to have the value of the current directory.

5.3 Executing from ETAC Script

The **ETAC Code Generator** can be executed directly from *ETAC script*. The current directory must be the directory containing ETACCodeGen.btac or ETACCodeGen.etac.

Important Note

Do not use exec_tac to run the **ETAC Code Generator** — the consequence is unpredictable.

The following example illustrates how to execute the **ETAC Code Generator** from *ETAC script* outside a *template file*. The second argument of fRunETACFile() (shown in **violet** colour) is only for illustration.

```
   ...
::define @F_RUN_ETAC_FILE
::include "etacFunctions.etac"
   ...
RtnCode :- ?; Fnts :- @NewData("etacFunctions");
   ...
   RtnCode := Fnts.fRunETACFile("ETACCodeGen.btac"
           'TEMPLATE=".\Tmplt.ecgt" ARGS=(...) OUTPUT="C:\
           ...\GenFiles\MyGenFile.txt"');

   if ( RtnCode = :#TAC_RTN_SUCCESS: ) then {...} endif;
   ...
```

The following example illustrates how to execute the **ETAC Code Generator** from *ETAC script* inside a *template file*. The second argument of **@cgRunETACFile** (shown in **violet** colour) is only for illustration.

```
<@SCRIPT:[]>
RtnCode :- ?;
   ...
   RtnCode := @cgRunETACFile("ETACCodeGen.btac" 'TEMPLATE=".\Tmplt.ecgt"
              ARGS=(...) OUTPUT="C:\...\GenFiles\MyGenFile.txt"');

   if ( RtnCode = :#TAC_RTN_SUCCESS: ) then {...} endif;
   ...
<@END:[SCRIPT]>
```

The second argument (shown in **violet** colour) of the function call in the two examples above is the command line *input arguments* (see Command Line Input Arguments).

An *ECG session* can be executed from within a *template file* to produce the *generated lines* into an ETAC sequence (OutSeq) as illustrated in the following example (the first argument of **@cgGenerate**, shown in **violet** colour, is only for illustration).

```
<@SCRIPT:[]>
OutSeq :- []; Success :- ?;
   ...
   Success := @cgGenerate(".\\Tmplt.ecgt" OutSeq "..." ? 0);

   if Success then {...} endif;
   ...
<@END:[SCRIPT]>
```

Note the double backslash (\\) used in the file paths of the preceding examples. The double backslash represents a single backslash, and is necessary because the backslash is an escape character in regular ETAC strings. Alternatively, if single-quoted strings ("raw" strings) are used, the backslashes must not be doubled.

5.4 Executing from ECGL Commands

The **@GEN** *command* can be used to cause the **ETAC Code Generator** to insert the default *generated lines* into the current *template lines* (rather than into an *output file*), as illustrated below.

```
<@GEN:[INPUT=".\Tmplt.ecgt" INSERT ARGS=(...)]>
```

In the example above, if INSERT is replaced with the ‹OUTPUT=› keyword, the default *generated lines* are written to the *output file* specified in the keyword.

The **@POSTGEN** *command* can be used to execute a new instance of the **ETAC Code Generator** after all the **@OUTPUT** *commands* have been activated, as illustrated below.

```
<@POSTGEN:[INPUT=".\Tmplt.ecgt" OUTPUT="C:\...\GenFiles\MyGenFile.txt"
    ARGS=(...)]>
```

6

Programming the ETAC Code Generator

The **ETAC Code Generator** uses a unique declarative programming language, *ECGL* (ETAC Code Generator Language), to generate and maintain text and source code files. However, a declarative language cannot practically cater for all possible generation scenarios. *ECGL* is therefore complemented by an algorithmic programming language. Since the **ETAC Code Generator** is implemented in the ETAC™ programming language, the most viable design option is to use that programming language as the complementary language to *ECGL*. The code generating process can therefore be refined to any desired degree by *ETAC script*. For the majority of cases, however, *ETAC script* is not required.

The *ECGL* and *ETAC script* used in a *template file* have different syntaxes; *ECGL* uses a declarative syntax, while *ETAC script* uses an algorithmic syntax (so-called "imperative" syntax). *ETAC script* is specified as fixed "data" to *ECGL instructions* and *commands*. *ETAC script* used in a *template file* can utilise the full power of the ETAC programming language.

It is important to remember that *ECGL* operates in stages, so *ETAC script* is activated only when the appropriate stage is current. For example, just because some *ETAC script* exists below some other *ETAC script* in a *template file*, that does not necessarily mean that the lower *ETAC script* will be activated after the other *ETAC script*.

6.1 Using ETAC Script

ETAC script can be specified directly in the following *ECGL instructions* and *commands*: **`&FNT`** *instruction*, **`@INSERT`**, **`@SCRIPT`**, **`@IF`**, and **`@EDIT`** *commands*. *ETAC script* can be specified in other *instructions* and *commands* via the **`&FNT`** *instruction*. (**Refs**: &FNT, @INSERT, @SCRIPT, @IF, @EDIT)

The most suitable place to include *ETAC script* in a *template file* is in one or more **`@SCRIPT`** *commands*. This allows ETAC variables, functions, and procedures to be defined in the global **`cg`** data object. Other *instructions* and *commands* existing anywhere in the *template line block* can simply refer to those definitions to use them. **`@SCRIPT`** *commands* can be placed anywhere in the *template line block* of a *template file*, however, they are best placed at the bottom of the file. (**Ref**: @SCRIPT)

For *template files* with complicated *template arguments*, the *ETAC script* in a **`@SCRIPT`** *command* can create one or more custom dialog boxes for the user to enter those *template arguments*. However, the **ETAC Code Generator** itself does not provide any facilities for such dialog boxes.

6.2 Intrinsic Global Functions

The **ETAC Code Generator** incorporates a number of predefined intrinsic global ETAC functions for various purposes including for manipulating *template lines*, if required, before they are converted to *generated lines*. The global functions are prefixed by the text ‹@cg›, and can be accessed from anywhere within the *template line block* via the appropriate *instructions* and *commands*. (**Ref**: 8.2_General Functions)

6.3 Text Array Functions

In addition to the intrinsic global functions, the *text array* data object contains some function members to search for and alter text in the *text array*. A particularly useful function is the **tlFindMark** function which uses a *pattern string* to find a text line in the *text array*. Other member functions can append, delete, retrieve, indent, and insert text lines.
(**Ref**: 8.3_Data Object: text array)

6.4 Debugging ETAC Script

ETAC script, whether existing in a **&FNT** *instruction*, on in an **@INSERT**, **@SCRIPT**, **@IF**, or **@EDIT** *command*, is debugged as described in chapter 2, **ETAC Debugger**, of the "*The Official ETAC Programming Language*" document, ETACProgLang(Official).pdf.

To debug *ETAC script*, the **ETAC Code Generator** needs to be started in debug mode via the RunETAC.exe program (with the DEBUG keyword option) or the context menu command, 'Debug', after placing breakpoints at suitable positions within the *ETAC script*. The 'Debug' menu command is available only for ETACCodeGen.etac (not for ETACCodeGen.btac).

In the debug window, the 'Silent Continue' button needs to be clicked repeatedly until the debugger pauses at a set break point. Debugging the *ETAC script* can commence from that point as described in the aforementioned chapter 2. Note that breakpoints can be placed within the *ETAC script* of **&FNT** *instructions* and conditions of **@IF** *commands*.

The following two illustrations show how to start the **ETAC Code Generator** in debug mode via the command line in a command file (‹*.cmd› or ‹*.bat›) or shortcut file. The current directory must be the directory containing ETACCodeGen.etac.

```
"...\ETACCodeGen.etac" 'TEMPLATE="..." ARGS=(...)' DEBUG
```

```
"...\RunETAC.exe" DEBUG SCRIPTS="ETACCodeGen.etac"
   'TEMPLATE="..." ARGS=(...)'
```

Although 'ETACCodeGen.btac' can be specified on the command line, 'ETACCodeGen.etac' is preferable to avoid a confusing debugging session.

In a command file, the current directory (indicated by the ellipsis) is specified by the **MS-DOS®** cd command as shown below.

```
cd /D "..."
```

7

ETAC Code Generator Examples

Some examples of how the **ETAC Code Generator** generates text from *template files* are shown in the following sections. The examples are for illustration purposes only, and do not necessarily represent recommended methods or useful outputs. The shaded line numbers (1) are only for reference and are not part of the examples themselves. Note that to follow the examples below, chapter 3_Processing Stages needs to be understood.

7.1 Example 1

This is a simple example of code, generated in the C programming language, to loop through an array. The *generated file* contains a code fragment that the user includes into some larger C program. The *header block* is shown in **bold aqua** colour. The *template line block* follows the *header block*. *Meta-codes* are shown in **bold blue** colour.

The Template File (ArrayLoop1.ecgt)

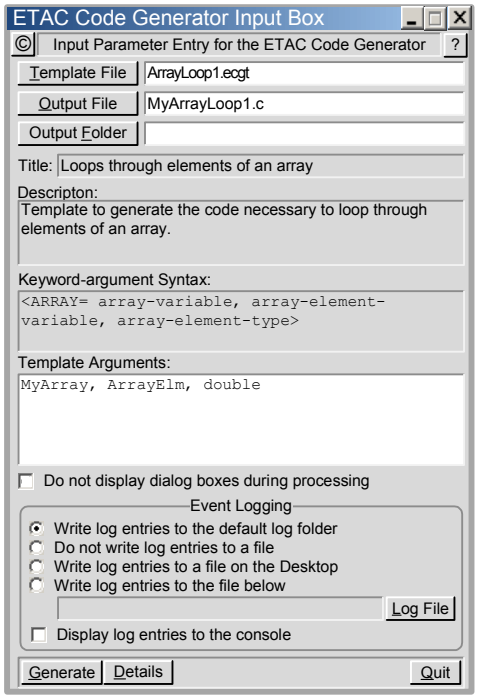
```
@ECG V1@
@Loops through elements of an array@
@D=Template to generate the code necessary to loop through elements of an array.
@C="This code implements a partial C language code segment that loops through the elements of an array.
   The programmer then fills in the rest of the code after pasting it into the source file.

   Arguments:
    array-variable          Array variable name.
    array-element-variable  A variable name for the element.
    array-element-type      The element type.
"
@O=Desktop::?
@T={//ARRAY=(#array-variable,#array-element-variable,#array-element-type)}
@P=ARR_VAR 1 1; ARR_ELM_VAR 1 2; ARR_ELM_TYPE 1 3;
@endhead@
/* This code implements a partial C language code segment that loops through the elements of an array.
   The programmer then fills in the rest of the code after pasting it into the source file. */
unsigned long           ArraySize;
<ARR_ELM_TYPE><&MI:[POSA=23 SPACES=1]>*<ARR_ELM_VAR>;
unsigned long           Idx;

   /* Get the size of the array. */
   ArraySize = GetArraySize(<ARR_VAR>);

   /* Loop through the <ARR_VAR> list. */
   for (Idx = 0; Idx < ArraySize; Idx++)
   {
      /* Get the next element from the <ARR_VAR> array. */
      <ARR_ELM_VAR> = (<ARR_ELM_TYPE> *)<ARR_VAR>[Idx];

      if ( *<ARR_ELM_VAR> == /*TBD: insert value here*/ )
      {
         /*TBD: insert code here*/
      }
      else
      {
         /*TBD: insert code here*/
      }
   }
```

User Request to Generate the File via the Command Line

```
ETACCodeGen.btac 'TEMPLATE="ArrayLoop1.ecgt" OUTPUT="MyArrayLoop1.c" ARGS=(ARRAY=MyArray, ArrayElm, double)'
```

User Request to Generate the File via the Input Dialog Box

```
ETACCodeGen.btac 'TEMPLATE="ArrayLoop1.ecgt" OUTPUT="MyArrayLoop1.c" ARGS=(ARRAY=MyArray, ArrayElm, double) PROMPT'
```



If the user runs the **ETAC Code Generator** with the `PROMPT` keyword, then the dialog box shown above is presented for the user to verify or alter.

The Output File (`MyArrayLoop1.c`)

```
/* This code implements a partial C language code segment that loops through the elements of an array.
   The programmer then fills in the rest of the code after pasting it into the source file. */
unsigned long           ArraySize;
double                 *ArrayElm;
unsigned long           Idx;

   /* Get the size of the array. */
   ArraySize = GetArraySize(MyArray);

   /* Loop through the MyArray list. */
   for (Idx = 0; Idx < ArraySize; Idx++)
   {
      /* Get the next element from the MyArray array. */
      ArrayElm = (double *)MyArray[Idx];

      if ( *ArrayElm == /*TBD: insert value here*/ )
      {
         /*TBD: insert code here*/
      }
      else
      {
         /*TBD: insert code here*/
      }
   }
```

Explanation

This example demonstrates plain *special symbol* substitution. Line `19` of the *template file* positions `‹*ArrayElm;›` so that `ArrayElm` is aligned with `ArraySize`. If `<ARR_ELM_TYPE>` is replaced by a string that is longer than 23 characters, then a space will be inserted before `‹*ArrayElm;›`. Note that the *output file* is actually a modified copy of the *template file*.

7.2 Example 2

In this example, the **ETAC Code Generator** uses the *template file* in **Example 1**, and internally modifies its *output file* to generate a more detailed *output file*. In addition to the arguments supplied in **Example 1**, the user supplies a comparison value, and a function (or other code) that gets inserted in the '`if`' part of the conditional statement, and a function (or other code) that gets inserted in the '`else`' part of the conditional statement. In addition, the user can supply a field name if the array element is a structure or class.

The Template File (ArrayLoop2.ecgt)

```
@ECG V1@
@Loops through elements of an array (V2)@
@D=Template code to generate the C language code necessary to loop through elements of an array and execute some
    code for each element matching a value.
@C="This code implements a code fragment that loops through the elements (array-element-variable) of an
   array (array-variable) and executes some code (if-code, else-code) for each element matching a value
   (array-element-value). The programmer then fills in the rest of the code after pasting it
   into the source file.

   Arguments:
    array-variable          Array variable name.
    array-element-variable  A variable name for the element.
    array-element-type      The element type.
    array-element-value     The value that the array element is to match.
    field-name              The name of the field if the element type is a structure or class (optional).
    if-code                 The code to execute if the array element matches the value (optional).
    else-code               The code to execute if the array element does not match the value (optional).
"
@O=Desktop::?
@T={//ARRAY=(#array-variable,#array-element-variable,#array-element-type,#array-element-value,$``field-name)}
    {/IF_CODE=(#if-code)} {/ELSE_CODE=(#else-code)}
@P=ARR_VAR 1 1; ARR_ELM_VAR 1 2; ARR_ELM_TYPE 1 3; VALUE 1 4; FIELD 1 5; IF_CODE_KW 2 0; IF_CODE 2 1;
    ELSE_CODE_KW 3 0; ELSE_CODE 3 1;
@endhead@
<@SYMBOL:[NAME="DOT" ARGS=(<&FNT:[=({@IfElse(("<FIELD>" != "") "." "");})]>)]>
<@GEN:[INPUT="ArrayLoop1" INSERT ARGS=(ARRAY=<ARR_VAR>, <ARR_ELM_VAR>, <ARR_ELM_TYPE>)]>
<@OUTPUT:[OFFSET=1 DELETE=2]>
/* This code implements a partial code segment that loops through the elements (<ARR_ELM_VAR>) of the
   array <ARR_VAR> and executes some code for each element matching the value <VALUE>.
   The programmer then fills in the rest of the code after pasting it into the source file. */
<@END:[OUTPUT]>
<@OUTPUT:[MARK=(   for (Idx = 0; Idx < ArraySize; Idx++)) OFFSET=4 DELETE=1]>
<@EVAL:[PRIOR]>
      if ( (*<ARR_ELM_VAR>)<DOT><FIELD> == <VALUE> )
<@END:[EVAL]>
<@END:[OUTPUT]>
<@IF:[COND=("<IF_CODE_KW>" != "")]>
<@OUTPUT:[MARK=S(.if.<()>) OFFSET=1 DELETE=1]>
         <IF_CODE>
<@END:[OUTPUT]>
<@END:[IF]>
<@IF:[COND=("<ELSE_CODE_KW>" != "")]>
<@OUTPUT:[MARK=S(.else) OFFSET=1 DELETE=1]>
         <ELSE_CODE>
<@END:[OUTPUT]>
<@ELSE:[]>
<@OUTPUT:[MARK=S(.else) OFFSET=-1 DELETE=4]>
<@END:[OUTPUT]>
<@END:[IF]>
```

User Request to Generate the File via the Command Line

```
ETACCodeGen.btac 'TEMPLATE="ArrayLoop2.ecgt" OUTPUT="MyArrayLoop2.c" ARGS=(ARRAY=TheArray, Element, Records, 20,
Age IF_CODE=Print(Element); break;)'
```

The Output File (MyArrayLoop2.c)

```
/* This code implements a partial code fragment that loops through the elements (Element) of the
   array TheArray and executes some code for each element matching the value 20.
   The programmer then fills in the rest of the code after pasting it into the source file. */
unsigned long           ArraySize;
Records                *Element;
unsigned long           Idx;

   /* Get the size of the array. */
   ArraySize = GetArraySize(TheArray);

   /* Loop through the TheArray list. */
   for (Idx = 0; Idx < ArraySize; Idx++)
   {
      /* Get the next element from the TheArray array. */
      Element = (Records *)TheArray[Idx];

      if ( (*Element).Age == 20 )
      {
         Print(Element); break;
      }
   }
```

Explanation

This example shows how to define *special symbols* within the *template file* itself (line 22). In this case, if ‹"<FIELD>"› resolves to a non-empty string then the symbol DOT will represent a dot character (.), otherwise DOT will represent an empty value. Note that the **&FNT** *instruction* at line 22 is not evaluated at that line. It is evaluated at the function processing stage after <DOT> has been replaced. <FIELD> within that *instruction* is evaluated when the **@EVAL** *command* (lines 30 to 32) is processed. The **@SYMBOL** *command* does not itself evaluate *special symbols* (unless the EVAL option is specified) or embedded *instructions*. The result is that at line 31, if <FIELD> represents a non-empty value then it will be preceded by a dot character, otherwise it will not be preceded by a dot character. So if Age were not specified in the command line, the line generated from line 31 would have been: ‹if ((*Element) == 20)›. Note that if the EVAL option of the SYMBOL *command* at line 22 were specified, then the <FIELD> symbol would have been evaluated at that line, and therefore the **@EVAL** *command* at lines 30 and 32 would not have been required — the **&FNT** *instruction* (at line 22) would have been evaluated normally at line 31. The example uses the **@EVAL** *command* and the absence of the EVAL option for illustrative purposes only.

This example also shows how to internally modify the *generated lines* from another *template file*. Line 23 starts a new independent *ECG session*, and inserts the content of the *output file* of that *ECG session* into the internal copy of the current *template lines* just after line 23 (line 23 itself is automatically deleted). Line 24 deletes the first two lines of the inserted lines (those two lines are comment lines) and inserts lines 25 to 27, thus replacing those first two lines. Line 29 deletes the line ‹if (*ArrayElm == /*TBD: insert value here*/)› of the inserted lines and replaces it with line 31 (after the *special symbols* have been replaced). Likewise, lines 35 to 37 replace ‹/*TBD: insert code here*/› of the inserted lines with ‹Print(Element); break;› because ‹IF_CODE=› has been specified on the command line. If ‹IF_CODE=› were not specified on the command line, then the original inserted line would have remained unchanged. In a similar manner, lines 40 to 42 replace the ‘else’ body with the argument of ‹ELSE_CODE=› if specified on the command line. In this example, ‹ELSE_CODE=› was not specified, so line 44 deletes the whole ‘else’ part (four lines) of the inserted lines.

This example demonstrates an important feature of the **ETAC Code Generator** — that is, to generate more specialised and sophisticated code based on the *output files* of existing *template files*. For example, the *generated file* of this example could have been created by the user by manually modifying the *output file* generated from **Example 1**. This example shows that the **ETAC Code Generator** could do those modifications by having the user merely specify a few extra command line arguments rather than the user having to manually do those modifications himself. Any number of levels of *template files* can be used, each one modifying the *output file* generated from the previous *template file*. And, of course, a single *template file* can modify the *output file* of more than one *template file*. **Example 3** illustrates a *template file* that modifies the *output file* generated in this example.

7.3 Example 3

In this example, the **ETAC Code Generator** uses the *output file* generated from the *template file* in **Example 2** and internally modifies an internal copy of that *output file* to generate a C function that returns the index of a found item in an array (or returns −1 if the item is not found). The user supplies the array element type, and field name if the array element is a structure or class. The *template file* generates three files: A “read me” file (the *output file*) containing some information, a C a header file, and a C function definition file.

The Template File (FindElement.ecgt)

```
1  @ECG V1@
2  @Finds an element of an array@
3  @D=Template code to generate the C language code necessary to find a given element of an array.
4  @C=This code implements a partial code segment that loops through the elements of an array searching
5     for an element matching a value. The element index is returned if found. -1 is returned if not found.
```

```
   The programmer can then modify the code (if necessary).

   Format: <FUNCTION= function-name, [{pArray}\array-par], [{pValue}\value-par], value-type>
           <ARRAY= array-element-type, [field-name]>

   Arguments:
    function-name            The name of the generated function.
    array-par                The name of the array parameter of the function (default: pArray).
    value-par                The name of the value parameter of the function (default: pValue).
    value-type               The value parameter type.
    array-element-type       The element type.
    field-name               The name of the field if the element type is a structure or class (optional).

@O=Desktop::?
@T= {//FUNCTION=(#function-name,$pArray``array-par,$pValue``value-par,#value-type)}
    {//ARRAY=(#array-element-type,$``field-name)}
@P=FNT_NAME 1 1; ARR_PAR 1 2; VAL_TYPE 1 4; ARR_ELM_TYPE 2 1; FIELD 2 2;
@endhead@
<@SYMBOL:[NAME="DOT" ARGS=(<&FNT:[=({@IfElse(("<FIELD>" != "") "." "");})]>) EVAL]>
The generated code implements a function (<FNT_NAME>) that loops through the elements of an
array (<ARR_PAR>) and returns the index of the first element matching the value in <ARR_VAL>.

The files generated are: <&HPAR:[OUTN]> (this file), <FNT_NAME>.h, and <FNT_NAME>.c.
<&C:[]>
<&C:[******** Generate the source file ********)]>
<@OUTPUT:[PATH="<FNT_NAME>.c" OFFSET=1 DELETETO=eof]>
<@GEN:[INPUT="ArrayLoop2" INSERT ARGS=(IF_CODE=return Idx; ARRAY=<ARR_PAR>, ArrElm, <ARR_ELM_TYPE>, <ARR_VAL>,
       <FIELD>)]>
<@OUTPUT:[OFFSET=1 DELETE=3]>
/* This is the source file for <FNT_NAME>(). */

int <FNT_NAME>(<ARR_ELM_TYPE> **<ARR_PAR>, <VAL_TYPE> <ARR_VAL>)
{
<@END:[OUTPUT]>
<@OUTPUT:[MARK=(   for (Idx = 0; Idx < ArraySize; Idx++)) OFFSET=4 DELETE=1]>
      if ( Compare((*ArrElm)<DOT><FIELD>, <ARR_VAL>) )
<@END:[OUTPUT]>
<@OUTPUT:[]>

  return -1;
}
<@END:[OUTPUT]>
<@END:[OUTPUT]>
<&C:[******** Generate the header file ********)]>
<@OUTPUT:[PATH="<FNT_NAME>.h"]>
/* This is the header file for <FNT_NAME>(). */

int <FNT_NAME>(<ARR_ELM_TYPE> **<ARR_PAR>, <VAL_TYPE> <ARR_VAL>);
<@END:[OUTPUT]>
```

User Request to Generate the Files via the Command Line

```
ETACCodeGen.btac 'TEMPLATE="FindElement.ecgt" OUTPUT="ReadMe.txt" ARGS=(FUNCTION=Find, , pSearchVal, char *
ARRAY=CustomerRec, CustName)'
```

The Output File (`Readme.txt`)

```
The generated code implements a function (Find) that loops through the elements of an
array (pArray) and returns the index of the first element matching the value in pSearchVal.
The programmer then fills in the rest of the code after pasting it into the source file.

The files generated are: ReadMe.txt (this file), Find.h, and Find.c.
```

Generated File (`Find.c`)

```
/* This is the source file for Find(). */

int Find(CustomerRec **pArray, char * pSearchVal)
{
unsigned long           ArraySize;
CustomerRec            *ArrElm;
unsigned long           Idx;

   /* Get the size of the array. */
   ArraySize = GetArraySize(pArray);

   /* Loop through the pArray list. */
   for (Idx = 0; Idx < ArraySize; Idx++)
   {
      /* Get the next element from the pArray array. */
      ArrElm = (CustomerRec *)pArray[Idx];

      if ( Compare((*ArrElm).CustName, pSearchVal) )
      {
         return Idx;
```

```
      }
   }

   return -1;
}
```

Generated File (`Find.h`)

```
/* This is the header file for Find(). */

int Find(CustomerRec **pArray, char * pSearchVal);
```

Explanation

Line 32 inserts the default *generated lines* (the content of the *output file*) of the *template file* `ArrayLoop2.ecgt`, which itself uses `ArrayLoop1.ecgt` (note that the *output file* itself is not created on disk). Those *generated lines* are then modified by this *template file* (lines 33 to 46) and output to the file `Find.c` via line 31. Another file, `Find.h`, is generated via lines 49 to 53. Note that line 40 replaces the original '`if`' statement of the *output file* generated from `ArrayLoop1.ecgt`.

In practice, parts of a *generated file* that need to be modified by another *template file* contain specially marked lines to be searched for by the other *template file*. For example, if the line ‹//MOD1//› were inserted after line 30 of `ArrayLoop1.ecgt`, and ‹//MOD2//› inserted after line 32, and ‹//MOD3//› after line 36, then those marked lines could be searched for in the **`@OUTPUT`** *commands* as markers for replacing the desired lines. However, the markers would remain in the *generated file*. There are two options here:

(1) if the markers are not desired in the *generated file*, then they can be removed using the **`@DELETE`** *command* (eg: `<@DELETE:[FILE="<FNT_NAME>.c" LINES=A(.//MOD%%`d//).]>`);

(2) the markers could remain so that a *template file* could modify a previously *generated file* relative to the markers.

A *template file* could read in such a marked file using the **`@INSERT`** *command*, and those markers could be searched for using the **`@OUTPUT`** *command*, which modifies the lines relative to the markers. By using a suitable convention of markers, source files can be constructed so that they contain "protected lines" which are not intended to be modified by users. Those protected lines are modified or replaced by various *template files* which implement various features into the marked source files.

7.4 Example 4

In this example, the **ETAC Code Generator** generates a general *template file*. The user then fills in specified parts of the *generated file* to make it a specific *template file*. All *template files* require a file name, a heading, a *keyword template*, keyword position specification, and a default *output file* path.

The Template File (`ECGTSourceFile.ecgt`)

```
1  @ECG V1@
2  @ETAC Code Generator Template Source File@
3  @D=Template for creating a skeleton template file for use by the ETAC Code Generator.
4  @C=The resulting file can be used with the ETAC Code Generator after appropriate modifications.
5
6     Format:  <ARGS= [{ECGT File}\HEADING], TEMPLATE, SYMPOS, [{Desktop::?}\OUTPUT]>
7
8     Keyword:
9     ARGS                 Information to set up a skeleton ECGT file to be filled in by the user.
10
11    Symbol Names:
12    ARGS:-
13     HEADING             The heading of the ECGT file that exists between two @ symbols. (optiopnal) Default: ECGT
                           File
14     TEMPLATE            The keyword template (inserted after @T=). The full keyword template is specified here.
15     SYMPOS              The symbol position specifications (inserted after @P=). Each specification is of the
                           form: 'symbol idx1 idx2 ...;'
16     OUTPUT              The output path for the ECGT file (inserted after @O=). (optional) Default: Desktop::?
17
```

```
@O=Desktop::?
@T={//ARGS=($ECGT File``HEADING,#TEMPLATE,#SYMPOS,$Desktop::?``OUTPUT)}
@P=HEADING 1 1; TEMPLATE 1 2; SYMPOS 1 3; OUTPUT 1 4;
@endhead@
************************************************************
*  This file contains the skeleton for the ETAC Code Generator template file <&HPAR:[OUTN]>.
*  Replace sections between « and » (inclusively) with appropriate text.
*
*  ARGS=<HEADING>, <TEMPLATE>, <SYMPOS>, <OUTPUT>
*
************************************************************

///// Start of file <&HPAR:[OUTN]> /////
@ECG V1@
@<HEADING>@
@D=Template for creating «...».
@C="«Insert detailed description here»

   Format: <&FNT:[=({@cgTrimStrEOL(@cgGetKWSyntax(["<TEMPLATE>"] ?));})]>

   Keyword:
   «Insert keyword here»                «Insert keyword description here»

   Symbol Names:
   «Insert keyword here»:-
    «Insert parameter here»               «Insert parameter description here» (eg: ...).

Examples:
   «Insert examples here»
"
@O=<OUTPUT>
@T=<TEMPLATE>
@P=<SYMPOS>
@endhead@
/************************************************************
*  This code «describe the purpose of this code here».
*
*  Generated Files:
*  «Insert names of generated files here»
*
*  ECGT Command Line Arguments:
*  «Insert command line arguments here»
*
************************************************************/
«Insert template body here»
///// End of file <&HPAR:[OUTN]> /////

// Some useful template fragments follow. //

<&<&C:[...]>>

<&<@CMT:[...]>>
<&<@END:[CMT]>>

<&<&FNT:[=(...)]>>

<&<@IF:[COND=("<...>" = "...")]>>
<&<@ELSE:[]>>
<&<@END:[IF]>>

<&<@OUTPUT:[PATH="..." MARK=(...) OFFSET=... DELETE=...]>>
<&<@END:[OUTPUT]>>

<&<@GEN:[INPUT="..." OUTPUT="..." ARGS=(...)]>>

<&<@SYMBOL:[NAME="..." ARGS=(...)]>>
```

User Request to Generate the File via the Command Line

```
ETACCodeGen.btac 'TEMPLATE="ECGTSourceFile.ecgt" OUTPUT="MyCGTFile.ecgt" ARGS=(ARGS=Finds an element of an array.,
{//FUNCTION=(#function-name,$pArray``array-par,$pValue``value-par,#value-type)} {//ARRAY=(#array-element-type,
$``field-name)}, FNT_NAME 1 1; ARR_PAR 1 2; ARR_VAL 1 3; VAL_TYPE 1 4; ARR_ELM_TYPE 2 1; FIELD 2 2;)'
```

The Output File (MyCGTFile.ecgt)

```
************************************************************
*  This file contains the skeleton for the ETAC Code Generator template file MyCGTFile.ecgt.
*  Replace sections between « and » (inclusively) with appropriate text.
*
*  ARGS=Finds an element of an array., {//FUNCTION=(#function-name,$pArray``array-par,$pValue``value-par,#value-
type)} {//ARRAY=(#array-element-type,$``field-name)}, FNT_NAME 1 1; ARR_PAR 1 2; ARR_VAL 1 3; VAL_TYPE 1 4;
ARR_ELM_TYPE 2 1; FIELD 2 2;, Desktop::?
*
************************************************************
```

```
///// Start of file MyCGTFile.ecgt /////
@ECG V1@
@Finds an element of an array.@
@D=Template for creating «...».
@C="«Insert detailed description here»

   Format: <FUNCTION= function-name, [{pArray}\array-par], [{pValue}\value-par], value-type> <ARRAY= array-element-
type, [field-name]>

   Keyword:
   «Insert keyword here»                «Insert keyword description here»

   Symbol Names:
   «Insert keyword here»:-
    «Insert parameter here»               «Insert parameter description here» (eg: ...).

Examples:
   «Insert examples here»
"
@O=Desktop::?
@T={//FUNCTION=(#function-name,$pArray``array-par,$pValue``value-par,#value-type)} {//ARRAY=(#array-element-type,
$``field-name)}
@P=FNT_NAME 1 1; ARR_PAR 1 2; ARR_VAL 1 3; VAL_TYPE 1 4; ARR_ELM_TYPE 2 1; FIELD 2 2;
@endhead@
/*********************************************************
*  This code «describe the purpose of this code here».
*
*  Generated Files:
*  «Insert names of generated files here»
*
*  ECGT Command Line Arguments:
*  «Insert command line arguments here»
*
*********************************************************/
«Insert template body here»
///// End of file MyCGTFile.ecgt /////

// Some useful template fragments follow. //

<&C:[...]>

<@CMT:[...]>
<@END:[CMT]>

<&FNT:[=(...)]>

<@IF:[COND=("<...>" = "...")]>
<@ELSE:[]>
<@END:[IF]>

<@OUTPUT:[PATH="..." MARK=(...) OFFSET=... DELETE=...]>
<@END:[OUTPUT]>

<@GEN:[INPUT="..." OUTPUT="..." ARGS=(...)]>

<@SYMBOL:[NAME="..." ARGS=(...)]>
```

Explanation

After filling in the specified sections of `MyCGTFile.ecgt`, the user then deletes the lines up to and including ‹`///// Start of file MyCGTFile.ecgt /////`› and the lines from and including ‹`///// End of file MyCGTFile.ecgt /////`›. The resulting file is the desired *template file*. Note how line `36` automatically creates the format of the arguments for the desired *template file*. The **`&FNT`** *command* activates a small *ETAC script* which produces the desired format from the *keyword template* of the generated *template file*. As for any *generated file*, the output of this *template file* can be the source for another *template file* if specialised *template files* need to be produced. Most systematic changes that a user can make to a *generated file* can be made by a *template file* via suitable parameters as explained in **Example 2**.

Note especially the use of *protection instructions* in this example, from line `67` to line `83`, and how they generate the corresponding text lines in the *output file*.

7.5 Example 5

In this example, the **ETAC Code Generator** generates a skeleton C++ source code to process named events. The user then fills in specified parts of the *generated files* to carry out the actual processing of the events. This example shows how *multi-lines* operate.

The Template File (EventProcessing.ecgt)

```
@ECG V1@
@Class Event Processing@
@D=Template code to generate the C++ code necessary for class event processing.
@C=This template implements code fragments for class event processing.

   Format: <CLASS= CLASS_NAME, CLASS_INIT> <EVENT= EVENT_NAME, ...>

   Keyword:
   CLASS                Information relating to the class.
   EVENT                Information relating to the events to be processed.

   Symbol Names:
   CLASS:-
    CLASS_NAME          Full name of the class. eg: TextWin.
    CLASS_INIT          First initials of the class name specified at CLASS_NAME. eg: tw.

   EVENT:-
    EVENT_NAME          Name of the event to process. eg: READ_FILE. (repeated)

Example:
CLASS=TextWin, tw
EVENT=READ_FILE, WRITE_FILE, UPDATE

@O=Desktop::?
@T={//CLASS=(#CLASS_NAME,#CLASS_INIT)} {//EVENT=(#EVENT_NAME,?)}
@P=CLASS_NAME 1 1; CLASS_INIT 1 2; EVENT_NAME 2 1;
@endhead@
/*************************************************************
*  This code implements code fragments for class event processing.
*
*  Generated Files:
*  <CLASS_NAME>.h, <CLASS_NAME>.cpp
*
*  ECGT Command Line Arguments:
<@JOIN:[]>
*  CLASS=<CLASS_NAME>, <CLASS_INIT> EVENT=
<&OMIT:[STR=( ) FIRST]><EVENT_NAME:#1><&OMIT:[STR=(,)]>
<@END:[JOIN]>
*************************************************************/
<@SYMBOL:[NAME="EVNTNUM" ARGS=(1)]>

   «To be put in appropriate event functions»
   <CLASS_INIT:L>Process(e<CLASS_INIT:U>_<EVENT_NAME:U:#1>);
<&C:[(======== C Header File Creation ========)]>
<@OUTPUT:[PATH="<CLASS_NAME>.h" BACKUP OFFSET=1 DELETETO=eof]>
/* Events for <CLASS_INIT:L>Process(). */
#define e<CLASS_INIT:U>_<EVENT_NAME:U:#1><&MI:[POSA=30 SPACES=1]><EVNTNUM+.>

class <CLASS_NAME>
{
protected:
   void     <CLASS_INIT:L>Process(unsigned long pEvent); /* Event processing procedure for this class. */
};
<@END:[OUTPUT]>
<&C:[(======== C Source File Creation ========)]>
<@OUTPUT:[PATH="<CLASS_NAME>.cpp" BACKUP OFFSET=1 DELETETO=eof]>
#include "<CLASS_NAME>.h"

/* Event processing procedure for this class. */
void <CLASS_NAME>::<CLASS_INIT:L>Process(unsigned long pEvent)
{
   if ( UpdateData(TRUE) )
   {
      switch (pEvent)
      {
      case e<CLASS_INIT:U>_<EVENT_NAME:U:#1> :\
         «TBD: Enter event handling code for e<CLASS_INIT:U>_<EVENT_NAME:U:#1> here»\
         break;\

      default:
         ASSERT(false);
      }

      VERIFY(UpdateData(FALSE));
   }
```

```
76
77    return;
78 }
79 <@END:[OUTPUT]>
```

User Request to Generate the File via the Command Line

```
ETACCodeGen.btac 'TEMPLATE="EventProcessing.ecgt" OUTPUT="ReadMe.txt" ARGS=(CLASS=TextWin, tw EVENT=READ_FILE,
WRITE_FILE, UPDATE)'
```

The Output File (ReadMe.txt)

```
/*********************************************************
*  This code implements code fragments for class event processing.
*
*  Generated Files:
*  TextWin.h, TextWin.cpp
*
*  ECGT Command Line Arguments:
*  CLASS=TextWin, tw EVENT=READ_FILE, WRITE_FILE, UPDATE
*********************************************************/

   «To be put in appropriate event functions»
   twProcess(eTW_READ_FILE);
   twProcess(eTW_WRITE_FILE);
   twProcess(eTW_UPDATE);
```

Generated File (TextWin.h)

```
/* Events for twProcess(). */
#define eTW_READ_FILE         1
#define eTW_WRITE_FILE        2
#define eTW_UPDATE            3

class TextWin
{
protected:
   void     twProcess(unsigned long pEvent); /* Event processing procedure for this class. */
};
```

Generated File (TextWin.cpp)

```
#include "TextWin.h"

/* Event processing procedure for this class. */
void TextWin::twProcess(unsigned long pEvent)
{
   if ( UpdateData(TRUE) )
   {
      switch (pEvent)
      {
      case eTW_READ_FILE :
         «TBD: Enter event handling code for eTW_READ_FILE here»
         break;

      case eTW_WRITE_FILE :
         «TBD: Enter event handling code for eTW_WRITE_FILE here»
         break;

      case eTW_UPDATE :
         «TBD: Enter event handling code for eTW_UPDATE here»
         break;

      default:
         ASSERT(false);
      }

      VERIFY(UpdateData(FALSE));
   }

   return;
}
```

Explanation

Lines 37, 47, and 66 to 69 are *multi-lines*. This means that a single *template line* can produce more than one *generated line*. Line 37 generates three lines, one for each event name READ_FILE, WRITE_FILE, and UPDATE. The first of the three lines will not have a space before it but the others will, and the last line will not have a comma after it but the others will. The three lines are concatenated by the **@JOIN** *command*, resulting in a single *generated line*

‹CLASS=TextWin, tw, EVENT=READ_FILE, WRITE_FILE, UPDATE›. Line 47 generates three lines, one for each event name, with a sequential number for each line beginning with the number one. The sequential numbers are produced by the *special symbol* ‹<EVNTNUM+.>›, which adds the current line number (beginning with line number zero) to the value of the symbol EVNTNUM. The value of EVNTNUM is defined in the *template file* itself at line 40. This results in the three 'define' statements at the top of the file TextWin.h. The lines 66 to 69 are *continued lines* and are treated as a single *template line* with end-of-line characters replacing the backslashes. That single *template line* is a *multi-line* and generates three lines, one for each event name, with the end-of-line characters embedded in each line. The result is three lots of four lines; each lot being a 'case' block of the 'switch' statement. *Continued lines* had to be used because multiple lines can be generated only from a single *multi-line* (unless the **@DO** *command* is used), and the four lines had to be concatenated first to form that single *multi-line*.

This example is a typical case where markers can be used to insert additional event code into the *generated files* using a suitably designed *template file*. If a marker were placed before lines 48 and 70 (the markers would be different from each other), then this or another *template file* can be designed to insert the event code before those markers on an existing file previously generated from this *template file*. The suitably designed *template file* does not generate any files, but modifies existing files (specified via a command line argument). In this way, any number of new events can be added to the previously *generated files* at any time. **Example 6** illustrates how this can be done.

7.6 Example 6

In this example, the **ETAC Code Generator** does not generate an output (other than an information *output file*) but reads two existing files (modified versions of TextWin.h and TextWin.cpp of **Example 5**) and adds additional event code to those files. The user then fills in specified parts of the modified files to carry out the actual processing of the events. The modified parts of those two files are shown in **BOLD**.

Existing File (TextWin.h)

```
/* Events for twProcess(). */
#define eTW_READ_FILE         1
#define eTW_WRITE_FILE        2
#define eTW_UPDATE            3
//«EVNTDEF»//

class TextWin
{
protected:
   void     twProcess(unsigned long pEvent); /* Event processing procedure for this class. */
};
```

Existing File (TextWin.cpp)

```
#include "TextWin.h"

/* Event processing procedure for this class. */
void TextWin::twProcess(unsigned long pEvent)
{
   if ( UpdateData(TRUE) )
   {
      switch (pEvent)
      {
      case eTW_READ_FILE :
         some code
         break;

      case eTW_WRITE_FILE :
         some code
         break;

      case eTW_UPDATE :
         some code
         break;

      //«EVNTBODY»//
      default:
         ASSERT(false);
```

```
      }

      VERIFY(UpdateData(FALSE));
   }

   return;
}
```

The Template File (AddEvent.ecgt)

```
@ECG V1@
@Add Event Processing@
@D=Template code to add the C++ code necessary to existing files containing event processing.
@C=This template implements code fragments for adding events to class event processing files.

   Format: <CLASS= CLASS_NAME, CLASS_INIT> <EVENT= EVENT_NUM, EVENT_NAME, ...>

   Keyword:
   CLASS                Information relating to the class.
   EVENT                Information relating to the events to be processed.

   Symbol Names:
   CLASS:-
    CLASS_NAME          Full name of the class. eg: TextWin.
    CLASS_INIT          First initials of the class name specified at CLASS_NAME. eg: tw.

   EVENT:-
    EVENT_NUM           Event number of the first event.
    EVENT_NAME          Name of the event to process. eg: READ_FILE. (repeated)

Example:
CLASS=TextWin, tw
EVENT=4, DELETE, MODIFY

@O=Desktop::?
@T={//CLASS=(#CLASS_NAME,#CLASS_INIT)} {//EVENT=(#EVENT_NUM,#EVENT_NAME,?)}
@P=CLASS_NAME 1 1; CLASS_INIT 1 2; EVENT_NUM 2 1; EVENT_NAME 2 2;
@endhead@
/*********************************************************
*  This code implements the code fragments added to files containing class event processing.
*
*  Modified Files:
*  <CLASS_NAME>.h, <CLASS_NAME>.cpp
*
*  ECGT Command Line Arguments:
<@JOIN:[]>
*  CLASS=<CLASS_NAME>, <CLASS_INIT> EVENT=<EVENT_NUM>
, <EVENT_NAME:#1>
<@END:[JOIN]>
*********************************************************/
<&C:[(======== C Header File Modification ========)]>
<@OUTPUT:[PATH="<CLASS_NAME>.h" MARK=A(.//«EVNTDEF»//.) OFFSET=-1 BACKUP]>
#define e<CLASS_INIT:U>_<EVENT_NAME:U:#1><&MI:[POSA=30 SPACES=1]><EVENT_NUM:#0+.>
<@END:[OUTPUT]>
<&C:[(======== C Source File Modification ========)]>
<@OUTPUT:[PATH="<CLASS_NAME>.cpp" MARK=A(.//«EVNTBODY»//.) OFFSET=-1 ALIGNA BACKUP]>
case e<CLASS_INIT:U>_<EVENT_NAME:U:#1> :\
   «TBD: Enter event handling code for e<CLASS_INIT:U>_<EVENT_NAME:U:#1> here»\
   break;\

<@END:[OUTPUT]>
```

User Request to Generate the File via the Command Line

```
ETACCodeGen.btac 'TEMPLATE="AddEvent.ecgt" OUTPUT="ReadMe.txt" ARGS=(CLASS=TextWin, tw EVENT=4, DELETE, MODIFY)'
```

The Output File (ReadMe.txt)

```
/*********************************************************
*  This code implements the code fragments added to files containing class event processing.
*
*  Modified Files:
*  TextWin.h, TextWin.cpp
*
*  ECGT Command Line Arguments:
*  CLASS=TextWin, tw EVENT=4, DELETE, MODIFY
*********************************************************/
```

Modified File (TextWin.h)

```
/* Events for twProcess(). */
#define eTW_READ_FILE         1
#define eTW_WRITE_FILE        2
#define eTW_UPDATE            3
```

```
#define eTW_DELETE            4
#define eTW_MODIFY            5
//«EVNTDEF»//

class TextWin
{
protected:
   void     twProcess(unsigned long pEvent); /* Event processing procedure for this class. */
};
```

Modified File (TextWin.cpp)

```
#include "TextWin.h"

/* Event processing procedure for this class. */
void TextWin::twProcess(unsigned long pEvent)
{
   if ( UpdateData(TRUE) )
   {
      switch (pEvent)
      {
      case eTW_READ_FILE :
         some code
         break;

      case eTW_WRITE_FILE :
         some code
         break;

      case eTW_UPDATE :
         some code
         break;

      case eTW_DELETE :
         «TBD: Enter event handling code for eTW_DELETE here»
         break;

      case eTW_MODIFY :
         «TBD: Enter event handling code for eTW_MODIFY here»
         break;

      //«EVNTBODY»//
      default:
         ASSERT(false);
      }

      VERIFY(UpdateData(FALSE));
   }

   return;
}
```

Explanation

‹//«EVNTDEF»//› in TextWin.h, and ‹//«EVNTBODY»//› in TextWin.cpp are markers for the **@OUTPUT** *command* to insert generated event code lines. Those markers remain with their respective files for future addition of event code. Line 42 searches for the marker ‹//«EVNTDEF»//› in TextWin.h, and inserts the *output line* generated from line 43 before the marker. Line 46 searches for the marker ‹//«EVNTBODY»//› in TextWin.cpp, and inserts the *output lines* generated from lines 47 to 50 before the marker. Those inserted lines are aligned to the beginning of the marker (ALIGNA). The markers remain in the modified files. If the ‹OFFSET=› options in lines 42 and 46 were omitted, then the code would have been inserted after the markers.

Markers can be used with any text file that needs to have modifications in particular places within that file. A system of beginning and ending markers can be used inclosing text that may be replaced or modified by the **ETAC Code Generator**. This system can be used to add or modify features of a computer program at any time during its life-cycle. A new feature typically needs programming code to be added at various places within various source files; the **ETAC Code Generator** can accomplish that task via markers in the source files. Markers can also exist in *template files*; the markers can be deleted from the internal copy of *template files* using the **@DELETE** *command* after modifications have been made relative to the markers. The *generated files* will then contain no markers.

In the example above, the event number, obtained from the first argument of the command line keyword ‹EVENT=›, is specified by the user. However, it is possible to include some *ETAC script* in the *template file* to parse the line before the marker (//«EVNTDEF»//) and obtain the event number of the last event in TextWin.h adding one to that event number. That calculated event number can then be used as the default event number if the user does not specify one.

Notice that the six examples above do not use the **@SCRIPT** *command*, that is to say, they do not use explicit ETAC programming. Most *template files* need not involve ETAC programming, and can be constructed by merely declaring the desired structure of the *generated files*.

8

ECGL Function Reference

ETAC script within any *template file* can communicate with the **ETAC Code Generator** via specially designed ETAC functions (*ECGL functions*). There are general purpose functions, and also functions to access and modify various properties of the components of a *template file*, such as accessing the values of a *special symbol*. There are also global variables that can be accessed.

Unicode File Specification

Important Note

A file specification string in ETAC cannot contain unpaired Unicode surrogate code points. If a file specification containing such code points needs to be specified, the **MS-DOS**® short (8dot3) format of the file specification should be used. The short format for files and directories can be displayed from an **MS-DOS**® command prompt window by typing the command `dir` with the option `/X`. The operating system may need to be configured to store the short format of file specifications.

8.1 Global Variables

Global variables can be accessed from any *ETAC script* within the *template line block*. The names of such variables begin with `@cg`, and are directly accessible. The variables can also be accessed via the '**cg**' data object. For example, the two variables `@cgMainTemplate` and `cg.@cgMainTemplate` are the same.

The following boxes contain a description of all the global variables. **R** means that the variable can be read from, and **W** means that the variable can be written to.

cg

cg	
value	A data object. (**RW**)

Details

The **cg** variable is an ETAC data object containing the definitions of all the *ECGL functions* and global variables. Template designers can also define their own global functions, procedures, and variables in the data object. The **cg** variable itself should not be reassigned.

Each *ECG session* contains its own **cg** variable which exists only within that session. ♦

@cgECGVrsnID

@cgECGVrsnID	
value	A string stack object. (**R**)

Details

Contains the **ETAC Code Generator** version identification string. The **@cgECGVrsnID** variable should not be reassigned. ♦

@cgMainTemplate

@cgMainTemplate	
value	A *text array* data object, or a null stack object (?). (**RW**)

Details

Contains all the *template lines* in the *template line block* during the processing of the **@SCRIPT**[POST] *command*. The ETAC sequence containing the *template lines* exists in the **tsaTextLines** data member of **@cgMainTemplate**. If the contents of **@cgMainTemplate** are modified, the actual *template line block* will be affected. The contents of **@cgMainTemplate** should be modified only by the last **@SCRIPT**[POST] *command*.

For all stages other than the **@SCRIPT**[POST] stage, *value* will be a null stack object.

Each *ECG session* contains its own **@cgMainTemplate** variable which exists only within that session. ♦

@cgSectTemplData

@cgSectTemplData	
value	A *text array* data object, or a null stack object (?). (**RW**)

Details

Contains all the *template lines* in the current section of the *template line block* during the processing of the **@SCRIPT**[POST] *command* for that section. The ETAC sequence containing the *template lines* exists in the **tsaTextLines** data member of **@cgSectTemplData**. If the contents of **@cgSectTemplData** are modified, the actual *template lines* of the current section will be affected. The contents of **@cgSectTemplData** should be modified only by the last **@SCRIPT**[POST] *command*.

value will be a null stack object for all stages other than the **@SCRIPT**[POST] stage, or if the *template line block* does not contain any **@SECTION** *commands*.

Each section of *template lines* contains its own **@cgSectTemplData** variable which exists only within that section.

Related Information

@SECTION ♦

8.2 General Functions

This section describes the general *ECGL functions* that can be used within any *template file*. The names of all such functions begin with @cg, and are directly accessible. The functions can also be accessed via the '**cg**' data object. For example, the two function calls `@cgSortLines(X)` and `cg.@cgSortLines(X)` are the same. The *ECGL functions* themselves should not be reassigned.

8.2.1 Functions by Category

The *ECGL functions* are listed below by category.

Date Time

@cgDateTimeFormatted

Disk File

@cgCreateFile ♦ @cgPathExists ♦ @cgWriteAllToOne ♦ @cgWriteFile

File Data

@cgAddFileData ♦ @cgCreateNewFile ♦ @cgGetFileData ♦ @cgGetFileFlags ♦ @cgRemoveFileData ♦ @cgRenameDataFile ♦ @cgReplFileFlags ♦ @cgSetFileFlags ♦ @cgWriteAllToOne ♦ @cgWriteFile

File Path

@cgCvtRelativePath ♦ @cgGetDefaultOutPath ♦ @cgGetWindowsDir ♦ @cgIsOnlyDirPath ♦ @cgIsOnlyFileName ♦ @cgIsRelativePath

Log File

@cgAddLogEntry

Special Symbol

@cgAddCmdSymb ♦ @cgGetCmdSymbVals ♦ @cgGetNumSymbVals ♦ @cgGetSpecSymbVal ♦ @cgGetSymbCount ♦ @cgGetSymbValAtOff ♦ @cgGetTArgsTree ♦ @cgIncrSymbCount ♦ @cgSetSymbCount

String

@cgExtractInnerStr ♦ @cgFindString ♦ @cgFormatStr ♦ @cgGetStrU ♦ @cgIsStrDblQuoted ♦ @cgIsStrInParen ♦ @cgIsStrInt ♦ @cgIsStrNegInt ♦ @cgIsStrPosInt ♦ @cgIsStrZeroInt ♦ @cgParseString ♦ @cgPutStrU ♦ @cgRemQuotes ♦ @cgReplSubStr ♦ @cgSeqToStrLines ♦ @cgStrLinesToSeq ♦ @cgTrimStrEOL ♦ @cgTrimStrSpaces

String Sequence

@cgDelDuplLines ♦ @cgIndentLines ♦ @cgRevLines ♦ @cgSeqToStrLines ♦ @cgSortLines ♦ @cgStrLinesToSeq

Text Array

@cgCvtTmplData ♦ @cgCvtToAngBraks ♦ @cgGetFileData ♦ @cgNewTextArray ♦ @cgRevLines ♦ @cgSortLines

Unicode

@cgGetStrU ♦ @cgPutStrU

Other

@cgExitECG ♦ @cgGenerate ♦ @cgGetCmdLineArgs ♦ @cgGetHeaderPar ♦ @cgGetInputBoxArgs ♦ @cgGetKWArgs ♦ @cgGetKWSyntax ♦ @cgRunETACFile ♦ @cgShowNewDialog ♦ @cgWriteCon

8.2.2 Function Summary

The table below contains an alphabetical list of the *ECGL functions*.

ECGL Function Summary for Scripts

Function	Description
`@cgAddCmdSymb`	Adds a new *special symbol* and its values to the list of *command symbols*.
`@cgAddFileData`	Adds a file path and its *text array* data object to the internal file list.
`@cgAddLogEntry`	Appends an entry to the list of log file entries.
`@cgCreateFile`	Creates a new empty file if it does not exist.
`@cgCreateNewFile`	Creates and loads a new file if it does not exist on disk.
`@cgCvtRelativePath`	Returns the full path of a file path, which may be relative to a specified directory path.
`@cgCvtTmplData`	Converts a *text array* containing *ECGL* template text to *template lines*.
`@cgCvtToAngBraks`	Converts all substrings from ‹`[~`› to ‹<› and ‹`~]`› to ‹>› in a *text array* data object.
`@cgDateTimeFormatted`	Returns a formatted date and time string of the current date and time.
`@cgDelDuplLines`	Deletes duplicate elements of a string sequence.
`@cgExitECG`	Exits the **ETAC Code Generator**.
`@cgExtractInnerStr`	Extracts the substring from a string enclosed within brackets.
`@cgFindString`	Returns the index of a substring existing in a string sequence.
`@cgFormatStr`	Replaces certain substrings within a format string.
`@cgGenerate`	Runs an internal instance of the **ETAC Code Generator**.
`@cgGetCmdLineArgs`	Gets the *input arguments* from the command line.
`@cgGetCmdSymbVals`	Gets the values of a *command symbol*.
`@cgGetDefaultOutPath`	Returns the effective full file path specification of the *output file*.
`@cgGetFileData`	Gets the *text array* data object associated with a file.
`@cgGetFileFlags`	Gets the file data flags affecting all files of the current *ECG session*.
`@cgGetHeaderPar`	Gets the specified header parameter.
`@cgGetInputBoxArgs`	Gets the *input arguments* from the main input dialog box.
`@cgGetKWArgs`	Processes keywords and their arguments.
`@cgGetKWSyntax`	Gets the keyword-arguments syntax of a *keyword template*.
`@cgGetNumSymbVals`	Gets the number of values of a *special symbol* name.
`@cgGetSpecSymbVal`	Gets the value of a *special symbol*.
`@cgGetStrU`	Returns the middle part (specified as *u-chars*) of a string.
`@cgGetSymbCount`	Gets the symbol counter value of a *special symbol*.
`@cgGetSymbValAtOff`	Gets the value of a *special symbol* at a specified offset.
`@cgGetTArgsTree`	Gets the raw keyword-arguments sequence tree of the *keyword template* of the current *template file*.
`@cgGetWindowsDir`	Get the full path of the system `Windows` directory.
`@cgIncrSymbCount`	Increments the symbol counter value of a *special symbol* by one.
`@cgIndentLines`	Indents all text lines in a string sequence.
`@cgIsOnlyDirPath`	Determines if a path specification is a directory path only.
`@cgIsOnlyFileName`	Determines if a file path specification contains only a file name (and extension).

@cgIsRelativePath	Determines if a file path specification is a relative path.
@cgIsStrDblQuoted	Determines whether a string is delimited by double quote characters.
@cgIsStrInParen	Determines whether a string is delimited by matching parentheses.
@cgIsStrInt	Determines whether a string is in the form of an integer.
@cgIsStrNegInt	Determines whether a string is in the form of a negative integer.
@cgIsStrPosInt	Determines whether a string is in the form of a positive integer.
@cgIsStrZeroInt	Determines whether a string is in the form of a zero integer.
@cgNewTextArray	Creates a new empty *text array* data object.
@cgParseString	Parses a string based on a pattern and\or sub-patterns.
@cgPathExists	Determines whether a specified type of disk entity exists for a path specification.
@cgPutStrU	Replaces a substring (specified as *u-chars*) in a string.
@cgRemoveFileData	Removes a file path and its *text array* data object from the internal file list.
@cgRemQuotes	Trims a string by removing leading and trailing single or double quotes and then spaces.
@cgRenameDataFile	Renames an internal data file path to a new path.
@cgReplFileFlags	Replaces the internal flags of an individual file.
@cgReplSubStr	Replaces all substrings of a string that match a *pattern string* with a string or strings.
@cgRevLines	Reverses the sequential order of the text lines of a string sequence or of a *text array* data object.
@cgRunETACFile	Runs an ETAC (or TAC) file as it would be run from RunETAC.exe.
@cgSeqToStrLines	Converts from a string sequence to a string with EOL characters.
@cgSetFileFlags	Sets the file data flags affecting all files of the current *ECG session*.
@cgSetSymbCount	Sets the symbol counter value of a *special symbol*.
@cgShowNewDialog	Shows a new uninitialised input dialog box to the user.
@cgSortLines	Sorts the text lines of a string sequence or of a *text array* data object.
@cgStrLinesToSeq	Converts from a string containing EOL-separated text lines to a sequence.
@cgTrimStrEOL	Trims a string by removing trailing EOL characters.
@cgTrimStrSpaces	Trims a string by removing leading and trailing spaces.
@cgWriteAllToOne	Writes the data of all files on the internal file list to a single disk file.
@cgWriteCon	Displays a message to the console window.
@cgWriteFile	Writes the specified internal data file to disk.

"EOL" stands for "end-of-line".

8.2.3 Function Definitions

The following boxes contain a description of all the *ECGL functions*.

@cgAddCmdSymb

@cgAddCmdSymb *s-name val-seq*	
s-name	A string stack object.
val-seq	A string sequence.

Details

Adds a new *special symbol* (*s-name*) and its values (*val-seq*) to the list of *command symbols*. The *special symbol* can have more than one *value*, and is used like any other *special symbol*.

s-name is the name of a new *special symbol*, and is in the format of the proper *special symbol* name syntax (alphanumeric-underscore characters beginning with an alphabetic character). *s-name* must not be the same as a symbol name specified at the ‹@P=› keyword of the *header block*. Note that *s-name* is only the name of a *special symbol*; it cannot contain other text (eg: ‹INPUT:3› is invalid for *s-name*, but ‹INPUT› is valid). Only UCS-2 (BMP Unicode scalar value) characters are recognised in *s-name*.

val-seq is a string sequence containing one or more *special symbol* values. If this function is called more than once with the same *s-name*, the values in *val-seq* are concatenated to the existing values for that *s-name*.

This function operates in the same manner as the **@SYMBOL** *command*.

Other Information

@SYMBOL ♦

@cgAddFileData

@cgAddFileData *file-path file-data*	
file-path	A string stack object.
file-data	A *text array* data object.

Details

Adds a file path (*file-path*) and its *text array* data object (*file-data*) to the internal file list.

file-path is internally expanded to its full file path specification before being used by this function.

file-data is an ETAC data object representing the internal file data to be associated with *file-path*. *file-data* is typically obtained from a call to the **@cgGetFileData** or **@cgNewTextArray** functions.

The **ETAC Code Generator** maintains data from files in an internal file list, identifying the data by the full file path specification of those files. This function adds *file-path* and *file-data* to that file list. If *file-path* already exists on the file list, the associated *text array* data object is replaced by *file-data*. The data is written to the disk file specified by its associated file specification before the end of the current *ECG session*, or at other specified times.

Additional Information

Unicode File Specification

Other Information

@cgGetFileData ▪ @cgNewTextArray ▪ @cgRemoveFileData ♦

@cgAddLogEntry

@cgAddLogEntry *entry error write*	
entry	A string stack object.
error	An integer stack object containing a logical boolean value.
write	An integer stack object containing a logical boolean value.

Details

Appends an entry (*entry*) to the list of log file entries.

entry is the actual log message desired to be appended to the list of log file entries.

error is `true` if the entry is considered to be an error message, otherwise it is `false`. Logged error messages cause a dialog box to be presented to the user asking whether to view the log file just before the **ETAC Code Generator** terminates.

write is `true` if the log file is to be written immediately, otherwise the log file is written before the **ETAC Code Generator** terminates. ♦

@cgCreateFile

@cgCreateFile *file-path*	
file-path	A string stack object.

Details

Creates a new empty disk file as specified (*file-path*) if it does not exist on disk. If the specified file already exists on disk, this function has no effect.

The created file is not registered by the **ETAC Code Generator**. An *error event* will occur if the file cannot be created.

Additional Information

Unicode File Specification

Other Information

@cgCreateNewFile ♦

@cgCreateNewFile

@cgCreateNewFile *file-path src-path* → *bool*	
file-path	A string stack object.
src-path	A string stack object, or a null stack object (`?`).
bool	An integer stack object containing a logical boolean value.

Details

Creates and loads a new file as specified (*file-path*) if it does not exist on disk. The function returns `true` (*bool*) if the file did not already exist and was actually created and loaded successfully, otherwise the function returns `false` (*bool*).

If the specified file already exists on disk, no action occurs and the function returns `false`.

If *src-path* is a relative path, it will be relative to the directory containing the *template files*. Note that the current directory symbol, ‹.› (dot), is regarded as an absolute path. For example, ‹.\MyFile.txt› is regarded as an absolute path.

The **ETAC Code Generator** maintains data from files in an internal file list, identifying the data by the full file path specification of those files. If *src-path* is a null stack object, and the specified file (*file-path*) does not exist on disk, this function adds *file-path* and an empty *text array* data object to that file list. If *file-path* already exists on the file list, the empty *text array* data object will replace the existing one. If *src-path* is not a null stack object, and the specified file (*file-path*) does not exist on disk, this function copies the file specified by *src-path* as the new file and loads the *text array* data object of that new file. The data is written to the disk file specified by its associated file path specification before the end of the current *ECG session*, or at other specified times.

Additional Information

Unicode File Specification

Other Information

@cgCreateFile ▪ @cgGetFileData ♦

@cgCvtRelativePath

@cgCvtRelativePath *dir-path file-path* → *path-str*	
dir-path	A string stack object.
file-path	A string stack object.
path-str	A string stack object.

Details

Returns the full file path specification (*path-str*) of a specified file path (*file-path*), which may be relative to a specified directory path (*dir-path*). If *file-path* is a relative path then the returned path specification is relative to *dir-path*, otherwise the returned path specification is the full file path of *file-path*.

Note that the current directory symbol, ‹.› (dot), is regarded as an absolute path. For example, ‹.\MyFile.txt› is regarded as an absolute path.

Examples

The following illustrations show how the **@cgCvtRelativePath** function can be used.

```
(1) @cgCvtRelativePath('C:\MyFolder\Other' 'Programs\TextFile.txt');
(2) @cgCvtRelativePath('C:\MyFolder\Other' 'C:\Files\TextFile.txt');
```

Example **(1)** returns the string ‹C:\MyFolder\Other\Programs\TextFile.txt› because the second argument is a relative path to the first argument.

Example **(2)** returns the string ‹C:\Files\TextFile.txt› because the second argument is an absolute path; the first argument is ignored. ♦

@cgCvtTmplData

@cgCvtTmplData *ta-data* → *ta-data*

ta-data	A *text array* data object.

Details

Converts a *text array* (*ta-data*) containing *ECGL* template text to *template lines*.

Specifically, this function concatenates *commands* in the *text array* of *ta-data* spread over more than one line into a single line. For example, if the *text array* in *ta-data* contains

```
...
<@DO:[FOR:[...]
       WITH:[...]
       WITH:[...]]>
...
```

on separate lines as shown, then this function converts the text above to a single line as shown below.

```
...
<@DO:[FOR:[...]       WITH:[...]       WITH:[...]]>
...
```

The reason for the conversion is that *commands* must be on a single text line to be processed correctly. If all *commands* in *ta-data* are already on a single text line, then this function need not be called.

Examples

The following illustrations show how the **@cgCvtTmplData** function can be used.

```
(1) <@INSERT:[PATH="..." SCRIPT=(@cgCvtTmplData())]>;
(2) void @cgCvtTmplData(@cgGetFileData(...));
```

In example **(1)**, a file (indicated by the ellipsis) containing *template lines* is to be inserted into the current *template lines* to be processed as part of the *template file*. The *commands* in the inserted file need to be on a single line, which is accomplished by the **@cgCvtTmplData** function.

In example **(2)**, an existing file (indicated by the ellipsis) containing *template lines* is to be converted to be processed as a *template file* at some other point.

Other Information

@cgGetFileData ♦

@cgCvtToAngBraks

@cgCvtToAngBraks *ta-data* → $ta\text{-}data_1$

ta-data	A *text array* data object.
$ta\text{-}data_1$	A *text array* data object.

Details

Converts all substrings ‹[~› to ‹<› and ‹~]› to ‹>› in the *text array* of a *text array* data object (*ta-data*). For example, a string element, ‹...[~FRUIT~]...›, of the *text array* is converted to ‹...<FRUIT>...›. $ta\text{-}data_1$ is *ta-data* with a possibly modified *text array*.

This function is typically used with *template files* whose *template lines* are based on angle brackets (‹<› and ‹>›), such as HTML files. Such files may not render correctly in their native display program if they contain *meta-codes*. The solution is to use ‹[~› and ‹~]› instead of ‹<› and ‹>› for *meta-codes* within those files. This function can then be used with the **@INSERT** *command* to convert ‹[~› and ‹~]› back to ‹<› and ‹>› so that *meta-codes* operate correctly within the *template file*. See Appendix A: HTML Template Files for more details.

Example

The following illustration shows how the **@cgCvtToAngBraks** function can be used.

(1) `<@INSERT:[PATH="MyHTML.html" SCRIPT=(@cgCvtToAngBraks())]>`

In example **(1)**, the file `MyHtml.html` is assumed to have *meta-codes* enclosed within ‹[~› and ‹~]›, rather than within ‹<› and ‹>›. The **@cgCvtToAngBraks** function converts the ‹[~› and ‹~]› to ‹<› and ‹>›, respectively, during the insertion. The content of the file `MyHtml.html` is unaffected.

Additional Information

Appendix A: HTML Template Files ♦

@cgDateTimeFormatted

@cgDateTimeFormatted *fmt-dt utc* → *dt-str*	
fmt-dt	A string stack object.
utc	An integer stack object containing a logical boolean value.
dt-str	A string stack object.

Details

Returns a formatted date and time string (*dt-str*) of the current date and time based on a specified format (*fmt-dt*). If *utc* is `true`, the returned string represents the UTC (“Universal Time Coordinated”) date and time (previously referred to as “Greenwich Mean Time” or GMT), otherwise it represents the local date and time.

The following table shows the date\time symbols and their meaning within the format string *fmt-dt*. Other symbols (eg: ‘/’) are presented as given. Where a single digit is specified, only leading zero digits are suppressed; other non-zero digits are presented. For example, if the seconds is `20`, then ‹[s]› will display `20`; if the seconds is `3`, then ‹[s]› will display `3`, but ‹[ss]› will display `03`.

Desired Date and Time	Format Symbol
Year (four digits, last two digits)	`[yyyy]`, `[yy]`
Month (long name, short name, two digits, one digit)	`[MMMM]`, `[MMM]`, `[MM]`, `[M]`
Day (long name, short name, two digits, one digit)	`[dddd]`, `[ddd]`, `[dd]`, `[d]`
12 hour (two digits, one digit)	`[hh]`, `[h]`
24 hour (two digits, one digit)	`[HH]`, `[H]`
Minute (two digits, one digit)	`[mm]`, `[m]`
Second (two digits, one digit)	`[ss]`, `[s]`
Fraction of seconds (3 digits)	`[f]`
AM\PM (`A\P`, `AM\PM`, `a\p`, `am\pm`)	`[T]`, `[TT]`, `[t]`, `[tt]`

Examples

The following illustrations show how the **@cgDateTimeFormatted** function can be used.

```
(1) @cgDateTimeFormatted("Today is [dd]/[MM]/[yyyy] [HH]:[mm]:[ss]",
    false);
(2) @cgDateTimeFormatted("Today is [ddd] [dd]-[M]-[yy] [h]:[mm]:[s].[f]
    [tt]", true);
(2) @cgDateTimeFormatted("It is [dddd], day [d], in the month of [MMMM],
    in the year [yyyy] AD.", true);
```

Example **(1)** returns the current local date and time in a form such as ‹Today is 20/05/2014 19:06:23›.

Example **(2)** returns the current UTC date and time in a form such as ‹Today is Tue 20-5-14 7:06:23.592 pm›.

Example **(3)** returns the current UTC date and time in a form such as ‹It is Tuesday, day 20, in the month of May, in the year 2014 AD.›.

Other Information

&DATE ♦

@cgDelDuplLines

@cgDelDuplLines *in-seq* | *in-data* → *out-seq* | *in-data*	
in-seq	A string sequence.
in-data	A *text array* data object.
out-seq	A string sequence.

Details

Deletes duplicate elements of a string sequence (*in-seq*) returning a new string sequence (*out-seq*). Note that the original string sequence, *in-seq*, is not modified. Alternatively, the function deletes duplicate elements of the *text array* in a *text array* data object (*in-data*) returning the same data object without duplicate elements. All strings in the returned string sequence or *text array* will be unique.

The function leaves the first one of the duplicate strings and removes the other duplicates.

Examples

The following illustrations show how the **@cgDelDuplLines** function can be used.

```
(1) Seq := ["hello", "goodbye", "hello"]; RtnSeq := @cgDelDuplLines(Seq);
(2) void @cgDelDuplLines(@cgGetFileData("MyTextFile.txt"));
```

Example **(1)** returns the new sequence ‹["hello", "goodbye"]› in RtnSeq, leaving the original sequence in Seq unmodified.

Example **(2)** assumes that the specified file exists, and removes duplicate text lines from an internal copy of that file. The file is written to disk with no duplicate lines before the current *ECG session* ends. Note that, in this example, the **@cgGetFileData** function returns a *text array* data object of the specified file. ♦

@cgExitECG

@cgExitECG

Details

Exits the **ETAC Code Generator**. This is the same as clicking the 'Quit' button on the input dialog box. ♦

@cgExtractInnerStr

@cgExtractInnerStr *in-str* → *out-str*

in-str	A string stack object.
out-str	A string stack object.

Details

Extracts the substring (*out-str*) from a string (*in-str*) enclosed within brackets. *in-str* must begin with one of the bracket characters, ‹(›, ‹[›, ‹{›, and end with the corresponding bracket character, ‹)›, ‹]›, ‹}›, otherwise *out-str* will be the same as *in-str*. If *in-str* is delimited as said, *out-str* will contain the text in-between, but excluding, the bracket characters.

Examples

The following illustrations show how the **@cgExtractInnerStr** function can be used.

```
(1) @cgExtractInnerStr("[the string]");
(2) @cgExtractInnerStr("not delimited by brackets");
(3) @cgExtractInnerStr("[mismatched brackets)");
```

Example **(1)** returns ‹the string›.

Example **(2)** returns ‹not delimited by brackets› because *in-str* was not delimited by the appropriate brackets.

Example **(3)** returns ‹[mismatched brackets)› because the outer brackets to not correspond. ♦

@cgFindString

@cgFindString *str-seq substr* → *idx*

str-seq	A string sequence.
substr	A string stack object.
idx	An integer stack object.

Details

Returns the index of the first occurrence of a substring (*substr*) existing in a string sequence (*str-seq*). The search is case-sensitive.

idx is the index of the first string in the sequence *str-seq* that contains the substring *substr*. *idx* is 0 if no substring is found.

Example

The following illustration shows how the **@cgFindString** function can be used.

```
(1) @cgFindString(["good morning", "good afternoon", "good evening"]
    "after");
```

Example **(1)** returns the value 2 on the object stack. ♦

@cgFormatStr

@cgFormatStr *fmt-str repl-seq* → *str*	
fmt-str	A string stack object.
repl-seq	A numeric or string sequence, or a null stack object (?).
str	A string stack object.

Details

Replaces symbols within a format string (*fmt-str*), returning a formatted string (*str*). This function creates a new temporary local dictionary when processing *fmt-str*.

The symbols within *fmt-str* are of the form:

1. ‹%*n*%› where *n* is a positive integer. There can be more than one %*n*% for a particular *n*, but no *n* can exceed the number of elements in *repl-seq*. Each *n* is an index into *repl-seq*, which must be a numeric or string sequence. The string or number at that index (*n*) replaces all the %*n*% in *fmt-str*. Whitespaces must not exist between the percent characters and *n*.
2. ‹%*›* is replaced with %. Note that % is special everywhere else, so an actual literal % must be represented as ‹%*›.
3. ‹%(*expression*)%› where (*expression*) is an ETAC expression, which must return a number or string when activated. The symbol is replaced by the string form of the returned value.
4. ‹%{*procedure*}%› where {*procedure*} is an ETAC procedure, which must return a number or string when activated. The symbol is replaced by the string form of the returned value.
5. ‹%*variable*%› where *variable* is an ETAC variable, which must return a number or string when activated. The symbol is replaced by the string form of the returned value. Whitespaces must not exist between the percent characters and *variable*. Note that the value of the dictionary item represented by *variable* can be a procedure, which is executed, returning a number or string.

Any number of the above symbols can be used in the same *fmt-str*. The command cpy can be used within *expression* (3) and *procedure* (4) above. cpy is identical to the ETAC command **copy_top**. If symbol (1) is not used, then the second argument (*fmt-str*) to the function is ignored (it must be the null stack object).

If a symbol is correct but the resulting value could not be converted to a string, then the symbol is replaced by ‹?*text*?›, where *text* is the text between the percent characters of the symbol. If a symbol could not be processed then it remains as is.

The function returns the modified string *str*.

Examples

The following illustrations show how the **@cgFormatStr** function can be used. The symbols mentioned above are highlighted (pink) in the examples.

```
(1) @cgFormatStr("The %1% chased his %2%." ["dog", "tail"]);
(2) @cgFormatStr("Some people %1% %1% %1% themselves 10%* of the time
    %1%edly" ["repeat"]);
(3) Var := "programmers"; @cgFormatStr("Hello %Var%" ?);
(4) Var := {("pro" + "grammers");}; @cgFormatStr("Hello %Var%" ?);
(5) @cgFormatStr('Hello %("pro" + "grammers")%' ?);
(6) @cgFormatStr('Hello %{add2 "pro" "grammers";}%' ?);
(7) @cgFormatStr("The value of %1% times %2% is %(&@ V :- cpy (2 * 3))%.
    That value %V% is %4%." ["two", 3, "incorrect", "correct"]);
```

Example **(1)** returns ‹`The dog chased his tail.`›.

Example **(2)** returns ‹`Some people repeat repeat repeat themselves 10% of the time repeatedly`›.

Examples **(3)** to **(6)** return ‹`Hello programmers`›.

Example **(7)** returns ‹`The value of two times 3 is 6. That value 6 is correct.`›. The variable `V` is allocated temporarily within **`@cgFormatStr`**. Note the use of the special `cpy` command, which is the same as the **`copy_top`** ETAC command. ♦

@cgGenerate

`@cgGenerate` *tmpl-path out-path arg-str gen-path flags* → *bool*	
tmpl-path	A string stack object.
out-path	A string stack object, a sequence, or a null stack object (`?`).
arg-str	A string stack object.
gen-path	A string stack object, or a null stack object (`?`).
flags	An integer stack object containing a binary boolean value.
bool	An integer stack object containing a logical boolean value.

Details

Runs an internal instance of the **ETAC Code Generator** on a *template file* (*tmpl-path*) in a new *ECG session*.

tmpl-path is a file path of a *template file*. A relative path specified for *tmpl-file* is relative to the current directory. *tmpl-path* must not be an empty string.

out-path specifies the file path of the *output file*, or an ETAC sequence, into which the main *generated lines* are to be put. **(a)** If *out-path* is a non-empty string, it overrides the default output path specified (at ‹`@O=`›) in the *header block* of *tmpl-file*. A relative path specified for *out-path* is relative to the current directory. **(b)** If *out-path* is an empty string, a unique program-generated file name of the form ‹ECGOutput**....**txt›, where **...** is an eight digit random number, will be created in the current directory and used as *out-path*. **(c)** If *out-path* is a null stack object (`?`) then the default output path specified (at ‹`@O=`›) in the *header block* of *tmpl-file* is used. **(d)** If *out-path* is a sequence, then the *generated lines* will be appended to the sequence.

The special directory ‹`Desktop::`› at the beginning of *out-path* specifies to output the *generated lines* to the **Windows**® `Desktop`; the character `?` for the file name indicates the aforementioned unique program-generated file name (see at (b) above).

arg-str is the *template argument* string matching the *keyword template* of the *template file* specified by *tmpl-file*.

gen-path is a directory path indicating the directory into which the *generated files* specified by relative output paths of **@OUTPUT** *commands* are to be written. A relative path specified for *gen-path* is relative to the current directory. A null stack object (`?`) for *gen-path* indicates the current directory. *gen-path* must not be an empty string.

flags is reserved and must be `0x00000000`.

bool will be `true` if the function succeeds, otherwise it will be `false`.

Examples

The following illustrations show how the **@cgGenerate** function can be used. For illustration purposes, the return value of the **@cgGenerate** function in the following examples is discarded.

```
(1) void @cgGenerate("C:\\TmplFile.ecgt" "TextFile.txt" "FUNCTION=Find, ,
    pSearchVal, char * ARRAY=CustRec, CustName" "Gen Files" 0x00000000);
(2) void @cgGenerate("MyFiles/MakeEXE.ecgt" ? "FILE=..." ? 0x00000000);
(3) void @cgGenerate("MyFiles/MakeEXE.ecgt" "Desktop::\\Gen\\?"
    "FILE=..." ? 0x00000000);
(4) OSeq :- [];
    void @cgGenerate("MyFiles/MakeEXE.ecgt" OSeq "FILE=..." ? 0x00000000);
(5) Pars := Success := @cgShowNewDialog();
    if Success then {void @cgGenerate(Pars);} endif;
```

Example **(1)** processes the *template file* ‹`C:\TmplFile.ecgt`› producing an *output file* `TextFile.txt` relative to the current directory. The *template arguments* for the *template file* are ‹`FUNCTION=Find, , pSearchVal, char * ARRAY=CustRec, CustName`›. Any **@OUTPUT** *commands* within the *template file* that have a relative path will create *generated files* in the folder ‹`Gen Files`› relative to the current directory.

Example **(2)** processes the *template file* ‹`MyFiles/MakeEXE.ecgt`› relative to the current directory producing an *output file* as specified in the *header block* of the *template file*. The *template arguments* for the *template file* are as specified. Any **@OUTPUT** *commands* within the *template file* that have a relative path will create *generated files* in the current directory.

Example **(3)** is the same as example (2) except that the *output file* will be a unique program-generated file (eg: ECGOutput31014640.txt) created in the `Gen` folder on the **Windows**® `Desktop`.

Example **(4)** is the same as example (2) except that the main *generated lines* will be appended to the sequence `OSeq` rather than being written to an *output file*.

Example **(5)** obtains *input arguments* from the user via a dialog box, which is the same as the main input dialog box. If the user clicks the 'Generate' button on the dialog box, the **@cgShowNewDialog** function returns `true` in `Success`, and also returns a procedure (in `Pars`) containing the user-entered arguments. That procedure is then used with the **@cgGenerate** function to generate the desired files. Note that if `Pars` is activated, then the top stack objects will be the raw arguments for the **@cgGenerate** function; those arguments may be modified if desired before being used with the **@cgGenerate** function, although that is not typically done.

Other Information

@cgShowNewDialog ▪ @GEN ▪ @POSTGEN ♦

@cgGetCmdLineArgs

@cgGetCmdLineArgs → *args-data*

args-data A data object.

Details

Gets the *input arguments* (*args-data*) from the command line. The *input arguments* are returned in an ETAC data object which contains the following data members.

Member	Description
claIniDirPath	A string containing the value of ‹INI_DIR=› without quote characters, or null (?) if ‹INI_DIR=› was not specified.
claTemplateFilePath	A string containing the value of ‹TEMPLATE=› without quote characters.
claOutputFilePath	A string containing the value of ‹OUTPUT=› without quote characters, or null (?) if ‹OUTPUT=› was not specified.
claGenDirPath	A string containing the value of ‹GEN_DIR=› without quote characters, or null (?) if ‹GEN_DIR=› was not specified.
claTmplArguments	A string containing the value of ‹ARGS=› or ‹ARG_FILE=›. If ‹ARGS=› was specified, the string will include the parentheses; if ‹ARG_FILE=› was specified, the string will be without quote characters.
claLogOpt	A string containing the log option specified on the command line. The string will contain one of the following values: ‹NL› for ‹NO_LOG› option, ‹AL› for ‹AUTO_LOG› option, ‹LF› for ‹LOG=› option (see claLogFilePath), or an empty string if a log option was not specified.
claLogFilePath	A string containing the value of ‹LOG=› without quote characters, or null (?) if ‹LOG=› was not specified.
claPrompt	Contains true if ‹PROMPT› was specified, otherwise contains false.
claShowLog	Contains true if ‹SHOW_LOG› was specified, otherwise contains false.
claSilent	Contains true if ‹SILENT› was specified, otherwise contains false.
claArgTree	An ETAC sequence containing a duplicate of the raw keyword-arguments sequence tree obtained directly from the command line as would be obtained from the **@cgGetKWArgs** function.
claECGTSrcDir	Contains the directory path of the directory containing the **ETAC Code Generator** template files (*.ecgt).
claLogDir	A string containing the full directory path (without quote characters) of the log file if claLogOpt is an empty string.

Examples

The following illustrations show how the **@cgGetCmdLineArgs** function can be used.

```
(1) DirPath := @cgGetCmdLineArgs().claGenDirPath;
(2) Args := @cgGetCmdLineArgs();
    Args.
    {
       DirPath := claGenDirPath;
       OutFile := claOutputFilePath;
    };
(3) @cgGetCmdLineArgs().
    {
       DirPath := claGenDirPath;
       OutFile := claOutputFilePath;
    };
```

Notice the full stop (“period”) in the examples above. ♦

@cgGetCmdSymbVals

@cgGetCmdSymbVals *s-name* → *val-seq* | ?	
s-name	A string stack object.
val-seq	A string sequence.

Details

Gets the values (*val-seq*) of the specified *special symbol* (*s-name*) existing on the list of *command symbols*.

s-name is the name of a *special symbol* as defined by the **@SYMBOL** *command* or by the **@cgAddCmdSymb** function. Note that *s-name* is only the name of a *special symbol*; it cannot contain other text (eg: ‹INPUT:3› is invalid for *s-name*, but ‹INPUT› is valid).

val-seq is a string sequence containing the values of the specified *special symbol*. If that *special symbol* (*s-name*) does not exist on the list of *command symbols*, this function returns a null stack object (?).

Related Information

@cgAddCmdSymb

Other Information

@SYMBOL ♦

@cgGetDefaultOutPath

@cgGetDefaultOutPath → *file-path* | ?	
file-path	A string stack object.

Details

Returns the effective full file path specification of the *output file* of the current *ECG session*.

If the current *template file* was evoked via the **@GEN** *command* with the INSERT option, then a null stack object (?) is returned by this function.

Other Information

&HPAR ♦

@cgGetFileData

@cgGetFileData *file-path* → *file-data* | ?	
file-path	A string stack object.
file-data	A *text array* data object for a file.

Details

Gets the *text array* data object (*file-data*) associated with the specified file (*file-path*).

file-path is internally expanded to its full file path specification before being used by this function.

file-data is an ETAC data object representing the internal file data of the specified file.

The **ETAC Code Generator** maintains data from files in an internal file list, identifying the data by the full file path specification of those files. If *file-path* already exists on the file list, the associated *text array* data object is returned in *file-data*. If *file-path* does not exist on the file list, this function will load and register the disk file (if it exists) onto that list, returning the associated *text array* data object in *file-data*. If the file does not exist on the file list and on the disk, this function will return a null stack object (?). The data is written to the disk file specified by its associated file path specification before the end of the current *ECG session*, or at other specified times.

Note that the *text array* of the returned data object from a loaded disk file will have text lines that were delimited only by $^{C_R}{}_{L_F}$, ^{C_R}, or $_{L_F}$ within the file data; text lines delimited by other EOL characters within the file data are not recognised.

Other Information

@cgAddFileData ▪ @cgRemoveFileData ♦

@cgGetFileFlags

@cgGetFileFlags → *flags*	
flags	An integer stack object containing a binary boolean value.

Details

Gets the file data flags (*flags*) affecting all files of the current *ECG session*. A returned value of `:!CGF_NO_WRITES:` indicates that no internal file data of the current *ECG session* is to be written to disk.

Note that the following ETAC pre-processor definition needs to be made in the script that calls this function.

```
[* No data is written to disk. *]
::define !CGF_NO_WRITES 0x00000001
```

Example

The following illustration shows how the **@cgGetFileFlags** function can be used.

(1)
```
::define !CGF_NO_WRITES 0x00000001
if ( @cgGetFileFlags() &and :!CGF_NO_WRITES: ) then {...} endif;
```

Example **(1)** checks if any files of the current *ECG session* are to be written to disk.

Related Information
@cgSetFileFlags

Other Information
@cgRepFileFlags ♦

@cgGetHeaderPar

@cgGetHeaderPar *par-str* → *par-val*	
par-str	A string stack object.
par-val	A string stack object.

Details
Gets (*par-val*) the specified header parameter (*par-str*) of the *header block*.

par-str is a string as defined for the keywords of the **&HPAR** *instruction*.

par-val is a string as is produced by the **&HPAR** *instruction*.

Additional Information
&HPAR

Example
The following illustration shows how the **@cgGetHeaderPar** function can be used.

```
(1) @cgGetHeaderPar("DESC");
```

Assuming that the *header block* contains ‹@D=This file generates C code›, example **(1)** returns the string ‹This file generates C code›. ♦

@cgGetInputBoxArgs

@cgGetInputBoxArgs → *args-data* | ?	
args-data	A data object.

Details
Gets the *input arguments* (*args-data*) from the main input dialog box. The main input dialog box is the one displayed as a result of specifying the PROMPT keyword on the command line. If PROMPT was not specified, this function returns a null stack object (?).

Note that this function does not obtain information from the input dialog box presented via the **@GEN** and **@POSTGEN** *commands*.

The *input arguments* are returned in an ETAC data object which contains the following data members.

Member	Description
ibaTemplateFilePath	A string containing the value of the *template file* (at Template File).
ibaOutputFilePath	A string containing the value of the *output file* (at Output File).
ibaGenDirPath	A string containing the value of the folder path to contain the *generated files* (at Output Folder).
ibaTmplArguments	A string containing the value of the *template arguments* (at Template Arguments:).
ibaLogOpt	A string containing the log option specified in the dialog box. The string will contain one of the following values: • ‹NL› for ‹NO_LOG› option (at Do not write log entries to a file), • ‹AL› for ‹AUTO_LOG› option (at Write log entries to a file on the Desktop), • ‹LF› for ‹LOG=› option (at Write log entries to the file below) (see ibaLogFilePath), • or an empty string if a log option was not specified (at Write log entries to the default log folder).
ibaLogFilePath	A string containing the value of the log file path (ibaLogOpt is ‹LF›), or an empty string if the path was not specified.
ibaShowLog	Contains true if the log entries were specified to be displayed to the console (at Display log entries to the console), otherwise contains false.
ibaSilent	Contains true if silent mode was specified (at Do not display dialog boxes during processing), otherwise contains false.

Examples

The following illustrations show how the **@cgGetInputBoxArgs** function can be used. The examples assume that the main input dialog box was displayed.

```
(1) DirPath := @cgGetInputBoxArgs().ibaGenDirPath;
(2) Args := @cgGetInputBoxArgs();
    Args.
    {
       DirPath := ibaGenDirPath;
       OutFile := ibaOutputFilePath;
    };
(3) @cgGetInputBoxArgs().
    {
       DirPath := ibaGenDirPath;
       OutFile := ibaOutputFilePath;
    };
```

Notice the full stop (“period”) in the examples above. ♦

@cgGetKWArgs

@cgGetKWArgs *tmpl arg-str sep* → *bool str-seq*	
tmpl	A string stack object, or a string sequence.
arg-str	A string stack object.
sep	A string stack object, or a null stack object (`?`).
bool	An integer stack object containing a logical boolean value.
str-seq	A string sequence.

Details

Processes keywords and their arguments (*arg-str*), based on a *keyword template* (*tmpl*), into a string sequence tree (*str-seq*). Note that this function operates in the same way as the ETAC command **kw_args**.

sep is a string containing three UCS-2 (BMP Unicode scalar value) characters. The left-most character determines the character for separating the parameters in the *keyword template* specified in *tmpl*. The middle character determines the source argument separators in the argument string (*arg-str*). That character cannot be a whitespace. The right-most character must be a zero character (`0`). For example, the string `‹,#0›` indicates that the parameters specified in *tmpl* are separated by commas (the default), and the arguments in *arg-str* are separated by a hash character. The default is effectively `‹,,0›`, and is indicated by a null stack object (`?`) for *sep*.

tmpl is either a string indicating a single *keyword template*, or a sequence of strings each of which is part of a nested *keyword template*. The first sequence element specifies the main group of *template blocks*; each other element specifies a *keyword block*. A string value for *tmpl* is effectively a sequence containing that string value.

Important Note

If any element of *tmpl* does not have a valid syntax, the consequence is unpredictable. **@cgGetKWArgs** does not cater for elements with an invalid syntax.

arg-str is the source string to be parsed, consisting of keywords and their arguments. *ETAC comments* within *arg-str* are logically replaced with one space, unless the comments are within a pair of double quotes. Backslashes (`‹\›`) in *arg-str* that are outside of *string blocks* are ignored and the character following a backslash is accepted literally. Escaped *ETAC comments* outside of *string blocks* are retained, as in this example, `‹KW=argument \[\*comment retained\*\]›`. The backslashes are automatically removed, leaving `‹KW=argument [*comment retained*]›` as the effective *arg-str*. Without the backslashes in the example, the comment is replaced with a single space. *arg-str* can include double-angle quoted substrings (the `:!KA_ANGLE_QUOTES:` flag is automatically applied), so the example above can be presented as `‹KW=argument «[*comment retained*]»›` to retain the comment.

bool indicates whether *arg-str* has matched the *keyword template* in *tmpl*. If *bool* is `true`, the match was successful, and the parsed *arg-str* will be contained in the returned output tree (*str-seq*). If *bool* is `false`, the match failed, and *str-seq* will contain a sequence of strings describing the reasons for the failure.

str-seq is the output tree containing nested sequences for each matched block corresponding to the *keyword template* in *tmpl*. Each matched and parsed block consists of a sequence containing one or more subsequences. Each subsequence contains string elements. The first element in the subsequence is the matched keyword, and the subsequent elements are the matched arguments or a matched and parsed block.

For full information on the keyword-argument system see **Appendix A: Keyword-arguments Specification** in the document "*The Official ETAC Programming Language*" (ETACProgLang(Official).pdf).

Example

The following illustration shows how the **@cgGetKWArgs** function can be used.

```
(1) @cgGetKWArgs("{//A(#a1)/k/C(x)}{/D($a2)/-e(#a3,?)}" "-ea,b C -e c" ?);
```

Example **(1)** returns `true` followed by the sequence ‹`[["C", "x"], ["-e", "a", "b", "c"]]`› as the second top stack object.

Additional Information

See **Appendix A: Keyword-arguments Specification** in the document "*The Official ETAC Programming Language*" (ETACProgLang(Official).pdf). ♦

@cgGetKWSyntax

@cgGetKWSyntax *tmpl sep* → *sntx-str*	
tmpl	A string stack object, or a string sequence.
sep	A string stack object, or a null stack object (`?`).
sntx-str	A string stack object.

Details

Gets the keyword-arguments syntax (*sntx-str*) of a *keyword template* (*tmpl*). *sntx-str* will contain a string showing the user-friendly syntax based on *tmpl*.

tmpl and *sep* are as defined for the function **@cgGetKWArgs**, except that for *sep* (if it is a string) the middle character must be a zero character (`0`), and the third character determines the character for separating the options in the returned syntax (*sntx-str*). The default is effectively ‹`,0\`›, and indicated by a null stack object (`?`) for *sep*. (Note that in ETAC, a backslash character in a string is represented as two backslashes or a backslash followed by a space.)

Example

The following illustration shows how the **@cgGetKWSyntax** function can be used.

```
(1) @cgGetKWSyntax("{//A(#a1)/k/C(x)}{/D($a2)/-e(#a3,?)}" "00|");
```

Example **(1)** returns the syntax string ‹`<A a1|k|C> [D {a2}|-e a3, ...]`›.

Additional Information

@cgGetKWArgs ♦

@cgGetNumSymbVals

@cgGetNumSymbVals *s-name* → *num*	
s-name	A string stack object.
num	An integer stack object.

Details

Gets the number of values (*num*) of a *special symbol* name (*s-name*).

s-name must conform to a restricted *special symbol* format as follows,

[(*name*$_1$[:*number*$_1$]/)···]*name*$_n$[:*number*$_n$]

where *name* is in the format of a *special symbol* name as defined in the ‹@P=› keyword, by the **@SYMBOL** *command*, or by the **@cgAddCmdSymb** function. *number* is an integer (usually positive). *number* is *number*$_1$ as defined in the *special symbol* syntax diagram.

An example of an *s-name* is ‹FNT_NAME/CMD:-2/SUBCMD:3/CMD_TYPE›.

num will be a positive number, or zero if the specified *special symbol* does not exist. The number of *special symbol* values is the number of values remaining at and after the argument position corresponding to the *special symbol* in the ‹@P=› keyword (modified by the syntax of *s-name*), or is the number of *special symbol* values for *special symbols* defined via the **@SYMBOL** *command* or the **@cgAddCmdSymb** function.

Examples

The following illustrations show how the **@cgGetNumSymbVals** function can be used.

```
(1) $P=... FRUIT 1 2; ...
    NumSymbs := @cgGetNumSymbVals("FRUIT");
(2) <@SYMBOL:[NAME="FRUIT" ARGS=(apple), (orange), (banana)]>
    NumSymbs := @cgGetNumSymbVals("FRUIT");
(3) @cgAddCmdSymb("FRUIT" ["apple", "orange", "banana"]);
    NumSymbs := @cgGetNumSymbVals("FRUIT");
```

In example **(1)**, assume that the *special symbol* FRUIT has values apple, orange, banana, pear, peach. Then NumSymbs will have the value 4, because the *special symbol* FRUIT at ‹$P=› indicates the value orange (the second index is 2), and there are four values at and after orange.

In examples **(2)** and **(3)**, NumSymbs will have the value 3, being the total number of values of the specified *special symbol*.

In the examples above, if the argument to **@cgGetNumSymbVals** were "FRUIT:2", then the value of NumSymbs would have been two less than specified in the examples because "FRUIT:**2**" indicates the second (**2**) value after the one specified by FRUIT.

Additional Information

Special Symbol Syntax Diagram

Other Information

@SYMBOL ▪ @cgAddCmdSymb ▪ 2.1.1_Parameters (‹**@P=**› parameter) ♦

@cgGetSpecSymbVal

@cgGetSpecSymbVal *s-name* → *s-val* | ?	
s-name	A string stack object.
s-val	A string stack object.

Details

Gets the value (*s-val*) of a *special symbol* (*s-name*).

s-name is the name of a *special symbol* as defined in the ‹@P=› keyword, by the **@SYMBOL** *command*, or by the **@cgAddCmdSymb** function. Note that *s-name* is only the name of a *special symbol*; it cannot contain other text (eg: ‹INPUT:3› is invalid for *s-name*, but ‹INPUT› is valid).

s-val will be the value of the symbol specified by *s-name*. If that symbol contains more than one value, only the nominal value will be returned.

If the specified *special symbol* is undefined or does not have a string value, a null stack object (?) will be returned.

Other Information

@cgGetCmdSymbVals ▪ @cgGetSymbValAtOff ▪ @SYMBOL ▪ @cgAddCmdSymb ▪ 2.1.1_Parameters (‹**@P=**› parameter) ♦

@cgGetStrU

@cgGetStrU *str offset len* → *out-str* | ?	
str	A string stack object.
offset	A non-negative integer stack object.
len	A non-negative integer stack object.
out-str	A string stack object.

Details

Returns (*out-str*) the middle substring (*offset*, *len*) of a string (*str*). *offset* and *len* are in *u-char* character units.

offset is a zero-based *u-char* character offset into *str*. If *offset* indicates a character beyond the last *u-char* character of *str*, then a null stack object will be returned by the function.

len is the maximum number of *u-char* characters to be obtained from *str* beginning at *offset*. If *len* exceeds the remaining characters of *str*, then only the remaining characters are obtained.

out-str is a substring of *str* beginning at *u-char* character *offset* with *u-char* character length up to *len*.

Examples

The following illustrations show how the **@cgGetStrU** function can be used.

```
(1) @cgGetStrU("hello-ha" 5 1);
(2) @cgGetStrU("hello-ha" 6 4);
(3) @cgGetStrU("hello-ha" 8 4);
(4) @cgGetStrU("thumbs\#1F44D#up" 6 2);
(5) @cgGetStrU("thumbs\#1F44D#up" 7 2);
```

Example **(1)** returns the string ‹-›.

Example **(2)** returns the string ‹ha›.

Example **(3)** returns a null stack object because *offset* is beyond the last character of *str*.

Example **(4)** returns the string equivalent of ‹\#1F44D#u›. Note that the Unicode supplementary plane code point U+1F44D (Thumbs Up Sign) is internally represented as a surrogate pair, but is only one *u-char* character wide. Because *len* (2) is the *u-char* character length, both *w-chars* (surrogate pairs) of the character at *offset* 6 and the following character (u), are obtained.

Example **(5)** returns the string ‹up› because it begins at *u-char* character offset 7 of *str*. ♦

@cgGetSymbCount

@cgGetSymbCount *s-name* → *cnt-val*	
s-name	A string stack object.
cnt-val	A non-negative integer stack object.

Details

Gets the symbol counter value (*cnt-num*) of a *special symbol* (*s-name*).

s-name is the name of a *special symbol* as defined in the ‹@P=› keyword, by the **@SYMBOL** *command*, or by the **@cgAddCmdSymb** function. Note that *s-name* is only the name of a *special symbol*; it cannot contain other text (eg: ‹INPUT:3› is invalid for *s-name*, but ‹INPUT› is valid). The *special symbol* (*s-name*) need not be currently defined.

cnt-val is the current counter value of the specified *special symbol* (*s-name*). The initial counter value of a *special symbol* is zero.

Related Information

@cgSetSymbCount ▪ @cgIncrSymbCount

Other Information

@SYMBOL ▪ @cgAddCmdSymb ▪ 2.1.1_Parameters (‹**@P=**› parameter) ♦

@cgGetSymbValAtOff

@cgGetSymbValAtOff *s-name offset* → *s-val* | ?	
s-name	A string stack object.
offset	An integer stack object.
s-val	A string stack object.

Details

Gets the value (*s-val*) of a *special symbol* (*s-name*) at a specified offset (*offset*).

s-name is the name of a *special symbol* as defined in the ‹@P=› keyword, by the **@SYMBOL** *command*, or by the **@cgAddCmdSymb** function. Note that *s-name* is only the name of a *special symbol*; it cannot contain other text (eg: ‹INPUT:3› is invalid for *s-name*, but ‹INPUT› is valid).

offset is a positive or negative integer (or zero), *n*, which indicates the n^{th} value of *s-name* from its nominal value.

s-val is the requested value of the specified *special symbol* as indicated by *offset*; if *offset* is such that it indicates a non-existent or non-string value, then a null stack object (?) is returned.

Examples

The following illustrations show how the **@cgGetSymbValAtOff** function can be used.

```
(1) $P=... FRUIT 1 2; ...
    Value := @cgGetSymbValAtOff("FRUIT" -1);
(2) $P=... FRUIT 1 2; ...
    Value := @cgGetSymbValAtOff("FRUIT" 3);
(3) <@SYMBOL:[NAME="FRUIT" ARGS=(apple), (orange), (banana)]>
    Value := @cgGetSymbValAtOff("FRUIT" 2);
(4) @cgAddCmdSymb("FRUIT" ["apple", "orange", "banana"]);
    NumSymbs := @cgGetSymbValAtOff("FRUIT" 2);
```

In examples **(1)** and **(2)**, assume that the *special symbol* `FRUIT` has values `apple`, **`orange`**, `banana`, `pear`, `peach`. Then `Value` will contain `apple` in example (1), because the *special symbol* `FRUIT` at ‹`$P=`› indicates the value `orange` (the second index is `2`), but the value at position `-1` relative to that value is `apple`. For example (2), `3` positions from `orange` is the value `peach`, which is returned in `Value`.

In examples **(3)** and **(4)**, `Value` will contain `banana`, being `2` values from the nominal value.

Other Information

@cgGetCmdSymbVals ▪ @cgGetSpecSymbVal ▪ @SYMBOL ▪ @cgAddCmdSymb ▪ 2.1.1_Parameters (‹**`@P=`**› parameter) ♦

@cgGetTArgsTree

`@cgGetTArgsTree` → *kw-args*

kw-args A string sequence.

Details

Gets the raw keyword-arguments sequence tree (*kw-args*) of the *keyword template* of the current *template file* as would be obtained from the **`@cgGetKWArgs`** function.

kw-args is a duplicate of the internal keyword-arguments sequence tree, which contains all the user-supplied or command-supplied *template file* keyword-arguments specified at the ‹`ARGS=(...)`› or ‹`ARG_FILE=`› keywords. Note that the ‹`ARGS=(...)`› keyword can also be specified at the **`@GEN`** and **`@POSTGEN`** *commands*.

The keyword-arguments sequence tree is an ETAC sequence containing nested sequences for each matched block corresponding to the *keyword template*. Each matched and parsed block consists of a sequence containing one or more subsequences. Each subsequence contains string elements. The first element in the subsequence is the matched keyword, and the subsequent elements are the matched arguments or a matched and parsed block.

Example

The following illustration shows how the **`@cgGetTArgsTree`** function can be used.

```
(1) User input: ... ARGS=(CLASS=MyClass DLGBASE) ...
    @T={//CLASS=(#c-name,$Base)} {/WNDBASE/DLGBASE} {/MFC_BC}
    KWArgs := @cgGetTArgsTree();
```

In example **(1)**, `KWArgs` will contain the sequence ‹`[["CLASS=", "MyClass", "Base"], ["DLGBASE"], [""]]`›. The result would be the same if the first line of the example were ‹`@GEN[... ARGS=(CLASS=MyClass DLGBASE) ...]`›.

Additional Information

For details of the structure of *kw-args*, see paragraph **A.5 Output Tree** under **Appendix A: Keyword-arguments Specification** in the document "*The Official ETAC Programming Language*" (ETACProgLang(Official).pdf). ♦

@cgGetWindowsDir

@cgGetWindowsDir → *dir-str* | ?	
dir-str	A string stack object.

Details

Get the full path of the system `Windows` directory (*dir-str*) or a null stack object (`?`) if that directory could not be obtained. *dir-str* will typically contain ‹`C:\Windows`›.

Illegal UTF-16 characters (ie: unpaired Unicode surrogate code points) in *dir-str* will be replaced with '`?`' (question mark). ♦

@cgIncrSymbCount

@cgIncrSymbCount *s-name*	
s-name	A string stack object.

Details

Increments the symbol counter value of a *special symbol* (*s-name*) by one.

s-name is the name of a *special symbol* as defined in the ‹`@P=`› keyword, by the **`@SYMBOL`** *command*, or by the **`@cgAddCmdSymb`** function. Note that *s-name* is only the name of a *special symbol*; it cannot contain other text (eg: ‹`INPUT:3`› is invalid for *s-name*, but ‹`INPUT`› is valid). The *special symbol* (*s-name*) need not be currently defined, in which case the symbol counter value will become `1` (one).

Related Information

@cgSetSymbCount

Other Information

@cgGetSymbCount ▪ @SYMBOL ▪ @cgAddCmdSymb ▪ 2.1.1_Parameters (‹**`@P=`**› parameter) ♦

@cgIndentLines

@cgIndentLines *str-seq num-pos pad eolchrs*	
str-seq	A string sequence.
num-pos	A non-negative integer stack object.
pad	A string stack object.
eolchrs	A string stack object.

Details

Indents all text lines in a string sequence (*str-seq*) the specified number of positions (*num-pos*) filled with the specified *w-char* character (*pad*). Indentation also applies to the sequence elements containing specified EOL (end-of-line) characters (*eolchrs*).

An element of *str-seq* could contain more than one text line, each separated by the string in *eolchrs*. For example, the string element `"line 1\r\nline 2\r\nline 3"` (ie: ‹line 1$^{C}_{R}{}^{L}_{F}$line 2$^{C}_{R}{}^{L}_{F}$line 3›) contains three text lines if *eolchrs* is `"\r\n"` (ie: $^{C}_{R}{}^{L}_{F}$).

str-seq will have each text line within each string element modified (indented) by this function.

num-pos is a non-negative integer indicating the number of positions to indent the text lines in *str-seq*.

pad is a string, but only the first character, which must be a UCS-2 (BMP Unicode scalar value) character, is used to pad the indentation. This will typically be a space character. If *pad* is an empty string then no indentation will occur.

eolchrs is a string that separates text lines within *str-seq*. If each element of *str-seq* is a single text line, then *eolchrs* should be an empty string.

Examples

The following illustrations show how the **`@cgIndentLines`** function can be used.

```
(1) Seq := ["First line", "line 1\nline 2\nline 3"];
    @cgIndentLines(Seq 3 "*" "\n");
(2) Seq := ["First line", "line 1::line 2::line 3::"];
    @cgIndentLines(Seq 3 "!" "::");
(3) Seq := ["\r\n\r\n"];
    @cgIndentLines(Seq 3 " " "\r\n");
```

In example **(1)**, `Seq` will end up containing the two elements ‹`***First line`› and ‹`***line 1`$^{L}_{F}$`***line 2`$^{L}_{F}$`***line3`›.

In example **(2)**, `Seq` will end up containing the two elements ‹`!!!First line`› and ‹`!!!line 1::!!!line 2::!!!line3::!!!`›.

In example **(3)**, `Seq` will end up containing the single element ‹$^{S}_{P}{}^{S}_{P}{}^{S}_{P}{}^{C}_{R}{}^{L}_{F}{}^{S}_{P}{}^{S}_{P}{}^{S}_{P}{}^{C}_{R}{}^{L}_{F}{}^{S}_{P}{}^{S}_{P}{}^{S}_{P}$›, that is, three text lines each containing three spaces.

Other Information

tlIndentLines ♦

@cgIsOnlyDirPath

`@cgIsOnlyDirPath` *path* → *bool*	
path	A string stack object.
bool	An integer stack object containing a logical boolean value.

Details

Determines (*bool*) whether a path specification (*path*) is a directory path only. If the last character of *path* is a forward slash or backslash, the function returns `true`, otherwise it returns `false`.

Additional Information

Unicode File Specification ♦

@cgIsOnlyFileName

@cgIsOnlyFileName *path* → *bool*	
path	A string stack object.
bool	An integer stack object containing a logical boolean value.

Details

Determines (*bool*) whether a file path specification (*path*) contains only a file name (and extension). If the file name and extension part of *path* is equal to *path*, the function returns `true`, otherwise it returns `false`. ♦

@cgIsRelativePath

@cgIsRelativePath *path* → *bool*	
path	A string stack object.
bool	An integer stack object containing a logical boolean value.

Details

Determines (*bool*) whether a file path specification (*path*) is a relative path. If *path* does not contain a drive part and does not begin with a current directory symbol, ‹.› (dot), the function returns `true`, otherwise it returns `false`.

Note that the current directory symbol, ‹.› (dot), is regarded as an absolute path. For example, ‹.\MyFile.txt› is regarded as an absolute path.

Examples

The following illustrations show how the **@cgIsRelativePath** function can be used.

```
(1) @cgIsRelativePath("MyPath\\MyFile.txt");
(2) @cgIsRelativePath('C:\MyPath\MyFile.txt');
```

Example **(1)** returns `true`, while example **(2)** returns `false`. ♦

@cgIsStrDblQuoted

@cgIsStrDblQuoted *str* → *bool*	
str	A string stack object.
bool	An integer stack object containing a logical boolean value.

Details

Determines (*bool*) whether a string (*str*) is delimited by double quote characters ("). If the first and last characters of *str* are double quote characters (U+0022), the function returns `true`, otherwise it returns `false`. ♦

@cgIsStrInParen

`@cgIsStrInParen` *str* → *bool*	
str	A string stack object.
bool	An integer stack object containing a logical boolean value.

Details

Determines (*bool*) whether a string (*str*) is delimited by matching parentheses. If the first and last characters of *str* are the parenthesis characters ‹ (› and ‹) ›, respectively, the function returns `true`, otherwise it returns `false`. ♦

@cgIsStrInt

`@cgIsStrInt` *str* → *bool*	
str	A string stack object.
bool	An integer stack object containing a logical boolean value.

Details

Determines (*bool*) whether a string (*str*) is in the form of an integer. If *str* is of the form

[{+}|−]*digits*,

where *digits* is one or more decimal digits, the function returns `true`, otherwise it returns `false`. ♦

@cgIsStrNegInt

`@cgIsStrNegInt` *str* → *bool*	
str	A string stack object.
bool	An integer stack object containing a logical boolean value.

Details

Determines (*bool*) whether a string (*str*) is in the form of a negative integer. If *str* is of the form

−*digits*,

where *digits* is one or more decimal digits, the function returns `true`, otherwise it returns `false`. Note the minus sign before *digits*. ♦

@cgIsStrPosInt

`@cgIsStrPosInt` *str* → *bool*	
str	A string stack object.
bool	An integer stack object containing a logical boolean value.

Details

Determines (*bool*) whether a string (*str*) is in the form of a positive integer. If *str* is of the form

[{+}]*digits*,

where *digits* is one or more decimal digits, the function returns `true`, otherwise it returns `false`. Note the optional plus sign before *digits*. ♦

@cgIsStrZeroInt

@cgIsStrZeroInt *str* → *bool*	
str	A string stack object.
bool	An integer stack object containing a logical boolean value.

Details

Determines (*bool*) whether a string (*str*) is in the form of a zero integer. If *str* is of the form

[+|-]*digits*,

where *digits* is one or more zero digits, the function returns `true`, otherwise it returns `false`. For example, ‹-000› returns `true`. ♦

@cgNewTextArray

@cgNewTextArray → *ta-data*	
ta-data	A *text array* data object.

Details

Creates a new empty *text array* data object (*ta-data*).

The returned *text array* data object can be used with the various functions that require a *text array* data object. The **tsaTextLines** member of the data object is an empty string sequence to contain the actual *text array*.

See 8.3_Data Object: text array for other members of a *text array* data object.

Additional Information

8.3_Data Object: text array ♦

@cgParseString

@cgParseString *pat str* → *str-seq* | ?	
pat	A string stack object, or a string sequence.
str	A string stack object.
str-seq	A string sequence.

Details

Parses a string (*str*) based on a *pattern string* and possibly sub-patterns (*pat*), returning a string sequence (*str-seq*) corresponding to the parsed substrings of the string (*str*).

pat is a *pattern string* or a sequence containing *pattern strings* which is\are matched by the whole of *str*. Substrings within *str* matching ‘blocks’ within *pat* are captured into *str-seq* as strings. A *block* is of the form ‹<*n*...>›, where *n* is either `0` (zero) or an integer from `1` to `9`. *Blocks* can be nested. *pat* can be an empty string, resulting in *str-seq* being an empty sequence. The syntax for the strings in *pat* is as indicated under the heading **Additional Information**.

If *pat* is a sequence, the second and subsequent elements of that sequence contain custom *pattern strings* identified by the ‹<p*r*>› special characters in the elements of *pat*. The first custom pattern in *pat* (the second element of *pat*) is custom pattern number 0 (ie: pattern represented by <p**0**>);

the next custom pattern in *pat* (the third element of *pat*) is custom pattern number 1 (ie: pattern represented by <p**1**>), and so on.

Important Note

If *pat* or any element of it (if *pat* is a sequence) does not have a valid syntax, the consequence is unpredictable. **@cgParseString** does not cater for patterns with an invalid syntax.

str is the string to be parsed.

str-seq can be in either one of two formats. ‹<*n*...>› *blocks* are used in *pat* to produce the contents of *str-seq* matching those *blocks*. **Format 1**: *pat* contains only ‹<0...>› *blocks*. In that case, *str-seq* will be a flat string sequence. **Format 2**: *pat* contains only ‹<*m*...>› *blocks*, where *m* is an integer from 1 to 9, inclusive. In that case, *str-seq* will contain one level of string subsequences. *Block m* corresponds to element *m* of *str-seq* (note that *m* cannot be 0 in this case). *str-seq* will contain as many elements as the maximum *block* number in *pat*; omitted *block* numbers in *pat* will correspond to empty subsequences in *str-seq*. If a ‹<*m*...>› *block* exists in *pat* but there are no matches for that *block* then the corresponding subsequence in *str-seq* will be empty.

If the match fails completely, or no *blocks* exists in *pat*, then a null stack object (?) will be returned.

Examples

The following illustrations show how the **@cgParseString** function can be used.

```
(1) @cgParseString("%%{$?<0%%`d>}%?" "there are 120 MINUTES in 2 HOURS");
(2) @cgParseString(["%%{$?[{<p0>}{<p1>}]}%?", "<1%%`u>", "<3%%`d>"]
    "there are 120 MINUTES in 2 HOURS");
```

Example **(1)** obtains the strings of all runs of one or more digits. The function returns the sequence ‹["120", "2"]›.

Block **1** in the example **(2)** obtains the strings of all runs of one or more uppercase characters; *block* 2 does not exist so it corresponds to the empty sequence in *str-seq*; and *block* **3** obtains the strings of all runs of one or more digits. <p0> represents the pattern <1%%`u>, and <p1> represents the pattern <3%%`d>. The function returns the sequence ‹[**["MINUTES", "HOURS"], [], ["120", "2"]**]› corresponding to the three *blocks* in *pat*.

Additional Information

See **Pattern String Matching** under chapter 3 of the "*The Official ETAC Programming Language*" document, ETACProgLang(Official).pdf. ♦

@cgPathExists

@cgPathExists *path type* → *bool*	
path	A string stack object.
type	An integer stack object, or a null stack object (?).
bool	An integer stack object containing a logical boolean value.

Details

Determines (*bool*) whether a specified type of disk entity (*type*) exists for a path specification (*path*).

type can be any one of the following: `:#FP_PATH_FILE:` (the entity is a file), `:#FP_PATH_DIR:` (the entity is a directory), `:#FP_PATH_VOL:` (the entity is a volume), or `?` (the entity is a file or directory).

bool is `true` if *path* represents the entity specified by *type*, otherwise it is `false`.

Additional Information

Unicode File Specification ♦

@cgPutStrU

@cgPutStrU *str offset len rep-str* → *out-str*	
str	A string stack object.
offset	A non-negative integer stack object.
len	A non-negative integer stack object.
rep-str	A string stack object.
out-str	A string stack object.

Details

Replaces a substring at a *u-char* character offset and length (*offset*, *len*) in a given string (*str*) with a string (*rep-str*), returning the modified string (*out-str*).

offset is a zero-based *u-char* character offset into *str*. If *offset* indicates a character beyond the last character of *str*, then *out-str* will be the same as *str*.

len is the maximum number of *u-char* characters to be replaced in *str* beginning at *offset*. If *len* exceeds the remaining characters of *str*, then only the remaining characters are replaced.

rep-str is the string that replaces the substring indicated by *offset* and *length*.

out-str is the modified string after replacement.

Examples

The following illustrations show how the **@cgPutStrU** function can be used.

```
(1) @cgPutStrU("hello-ha" 5 1 " ");
(2) @cgPutStrU("hello-ha" 6 4 "*%");
(3) @cgPutStrU("hello-ha" 8 4 "*%");
(4) @cgPutStrU("hello-ha" 1 3 "");
(5) @cgPutStrU("thumbs\#1F44D#up" 6 2 "*");
(6) @cgPutStrU("thumbs\#1F44D#up" 7 2 "\#1F34F~green apple#**");
```

Example **(1)** returns the string ‹`hello ha`›.

Example **(2)** returns the string ‹`hello-*%`›.

Example **(3)** returns the string ‹`hello-ha`› because *offset* is beyond the last character of *str*.

Example **(4)** returns the string ‹`ho-ha`›.

Example **(5)** returns the string ‹`thumbs*p`›. Note that the Unicode supplementary plane code point U+1F44D (Thumbs Up Sign) is internally represented as a surrogate pair, but is only one *u-char* character wide. Because *len* (`2`) is the *u-char* character length, both *w-chars* (surrogate pairs) of the character at *offset* `6`, and the following character (`u`), are replaced.

Example **(6)** returns the equivalent of the string ‹thumbs\#1F44D#\#1F34F#**› because the substring being replaced begins at *u-char* character *offset* 7. Note that the text ‹~green apple› is ignored by ETAC (such text is an example of an "in-string comment"). ♦

@cgRemoveFileData

@cgRemoveFileData *file-path*	
file-path	A string stack object.

Details

Removes a file path (*file-path*) and its *text array* data object from the internal file list.

file-path is internally expanded to its full file path specification before being used by this function.

The **ETAC Code Generator** maintains data from files in an internal file list, identifying the data by the full file path specification of those files. This function removes *file-path* and the corresponding data from that file list. The disk file is not deleted. If *file-path* does not exist on the file list, no action occurs.

Other Information

@cgGetFileData ♦

@cgRemQuotes

@cgRemQuotes *in-str* → *out-str*	
in-str	A string stack object.
out-str	A string stack object.

Details

Trims a string (*in-str*) by removing leading and trailing single or double quotes and then spaces, leaving the result (*out-str*) on the object stack.

in-str is a string that may contain leading and\or trailing single-quote (‘'’ U+0027) or double-quote (‘"’ U+0022) characters. Those characters are removed first, then leading and trailing spaces (U+0020) are removed next.

out-str is the same as *in-str* but with the said characters removed.

Examples

The following illustrations show how the **@cgRemQuotes** function can be used.

```
(1) @cgRemQuotes("\"text str\"");
(2) @cgRemQuotes("'\"text str'\"");
(3) @cgRemQuotes("'  text str' ");
(4) @cgRemQuotes(" text str ' ");
(5) @cgRemQuotes(" text str '");
(6) @cgRemQuotes("' 'text str ' '");
```

In example **(1)**, the input string is ‹"text str"›, and the function returns ‹text str›.

In example **(2)**, the input string is ‹'"text str'"›, and the function returns ‹text str›.

In example **(3)**, the input string is ‹' text str' ›, and the function returns ‹text str'›.

In example **(4)**, the input string is ‹ text str ' ›, and the function returns ‹text str '›.

In example **(5)**, the input string is ‹ text str '›, and the function returns ‹text str›.

In example **(6)**, the input string is ‹' 'text str ' '›, and the function returns ‹'text str '›.
♦

@cgRenameDataFile

@cgRenameDataFile *path new-path*	
path	A string stack object.
new-path	A string stack object.

Details

Renames an internal data file path (*path*) to a new path (*new-path*). No action occurs if the file to be renamed (*path*) does not exist in the internal file list. *new-path* can contain a path to a different directory or disk than *path*. This function does not immediately affect disk files until the data file is written to disk at a later time.

path and *new-path* are internally expanded to their full file path specification before being used by this function. The consequence is undefined if *new-path* is the same as an existing file path in the internal file list.

The **ETAC Code Generator** maintains data from files in an internal file list, identifying the data by the full file path specification of those files. This function merely renames such a file path specification (*path*) to the specified one (*new-path*). The data is written to the disk file specified by the file path specification before the end of the current *ECG session*, or at other specified times.

Additional Information

Unicode File Specification ♦

@cgReplFileFlags

@cgReplFileFlags *file-data in-flags* → *out-flags*	
file-data	A *text array* data object for a file.
in-flags	An integer stack object containing a binary boolean value, or a null stack object (?).
out-flags	An integer stack object containing a binary boolean value.

Details

Replaces the internal flags of an individual file (*file-data*) with specified flags (*in-flags*), returning the previous flags (*out-flags*).

file-data is an ETAC data object representing the internal file data for which the file flags are to be altered. *file-data* is typically obtained from a call to the **@cgGetFileData** function.

in-flags are the new flags desired to be set for the file data. :!CGF_BACKUP: indicates that a backup of an existing file by the same name is to be made when the file is written to disk. If *file.ext* is the format of the file name specified in *file-data*, then the backup file name will be *file*~backup.*ext*. If the backup file already exists then it will be overwritten automatically without warning. Note that only files that are not empty will be backed up. :!CGF_READ_ONLY: prevents the file data from being written to disk.

If *in-flags* is a null stack object (?), then the existing flags in *file-data* are not replaced; *out-flags* will just contain the existing flags.

out-flags are the existing flags in *file-data* before possibly being replaced.

Note that the following ETAC pre-processor definitions need to be made in the script that calls this function.

```
[* Does a backup of written file. *]
::define !CGF_BACKUP 0x00000001

[* Read-only file. File is not written to disk. *]
::define !CGF_READ_ONLY 0x00000002
```

Examples

The following illustrations show how the **@cgReplFileFlags** function can be used.

```
(1) ::define !CGF_READ_ONLY 0x00000002
    FileData := @cgGetFileData('C:\MyFile.txt');
    Flags := @cgReplFileFlags(FileData ?);
    void @cgReplFileFlags(FileData (Flags &and not :!CGF_READ_ONLY:));
(2) ::define !CGF_BACKUP 0x00000001
    Flags := @cgReplFileFlags(@cgGetFileData('C:\MyFile.txt') ?);
    if ( Flags &and :!CGF_BACKUP: ) then {...} endif;
```

Example **(1)** clears the read-only flag, `:!CGF_READ_ONLY:`, within the file data object of the specified file.

Example **(2)** gets the flags within the file data object of the specified file, and does an action if the `:!CGF_BACKUP:` flag is set.

Other Information

@cgGetFileFlags ▪ @cgSetFileFlags ▪ @cgGetFileData ♦

@cgReplSubStr

@cgReplSubStr *pat repl-str in-str* → *out-str*	
pat	A string stack object, or a string sequence.
repl-str	A string stack object, or a string sequence.
in-str	A string stack object.
out-str	A string stack object.

Details

Replaces all substrings of a string (*in-str*) that match a *pattern string* or a sequence containing *pattern strings* (*pat*) with a specified string or strings (*repl-str*). Only the parts of *in-str* that match the *zero-blocks* (<0...>) of *pat* are replaced.

pat is a *pattern string* or a sequence containing *pattern strings*, but must not contain any <*n*...> *blocks* where *n* is greater than 0. If *pat* does not contain any *blocks* then it is assumed to be contained within a single *zero-block* (ie: <0...>). There can be more than one *zero-block* in *pat* but they must not be nested. If *pat* is a sequence, then the first element of that sequence is the main *pattern string*, the other elements are custom *pattern strings* beginning with custom pattern 0. The syntax for the strings in *pat* is as indicated under the heading **Additional Information**, except that those strings can contain only <0...> *blocks* (or no *blocks*).

Important Note

If *pat* or any element of it (if *pat* is a sequence) does not have a valid syntax, the consequence is unpredictable. **@cgReplSubStr** does not cater for patterns with an invalid syntax.

A match occurs when at least one substring of *in-str* matches *pat*. If no match occurs, *out-str* will be the same as *in-str*, otherwise, *out-str* will be *in-str* with the appropriate substrings matching the *zero-blocks* in *pat* replaced with *repl-str*. If *repl-str* is a string then it will replace all matching substrings. If *repl-str* is a sequence, then the strings in it must correspond to at least the number of matching substrings. Substrings in *in-str* matching the *zero-blocks* in *pat* are replaced with the corresponding strings in *repl-str* in the order in which the substrings occur from left to right.

Examples

The following illustrations show how the **@cgReplSubStr** function can be used.

```
(1) @cgReplSubStr("NUM:<0%%`d>" "10" "abcdNUM:24efgNUM:3hijNUM:40");
(2) @cgReplSubStr("NUM:<0%%`d>" ["7", "66", "204"]
    "abcdNUM:24efgNUM:3hijNUM:40");
(3) @cgReplSubStr("three" "four" "three people dug three holes");
```

Example **(1)** illustrates how **@cgReplSubStr** works when *repl-str* is a string ("10"). The first argument, *pat*, is matched by three substrings of the third argument, *in-str*. The three matching substrings are: ‹NUM:24›, ‹NUM:3›, ‹NUM:40›. The number in each of those substrings matches the *zero-block*, <0%%`d>, of *pat*. The second argument, *repl-str*, therefore replaces those numbers. That is to say, the second argument replaces those substrings matching the *zero-block*. In this case, the said numbers are replaced with 10, the second argument. The function, therefore, returns the string ‹abcdNUM:**10**efgNUM:**10**hijNUM:**10**›.

Example **(2)** is the same as example (1), except that the second argument, *repl-str*, is a sequence of three strings that correspond to the three numbers mentioned in example (1). Those three numbers, therefore, are replaced by the corresponding three strings in *repl-str*. The function returns the string ‹abcdNUM:**7**efgNUM:**66**hijNUM:**204**›.

Note that the number of elements of *repl-str* must be at least the same as the number of substrings matching the *zero-blocks*, otherwise an ETAC *error event* will occur. In some cases, the number of matching strings may not be known in advanced, so *repl-str* should be a sequence only when that number can be predicted.

The function in example **(3)** returns the string ‹four people dug four holes›, because the first argument, *pat*, is equivalent to ‹<0three>›.

Additional Information

See **Pattern String Matching** under chapter 3 of the "*The Official ETAC Programming Language*" document, ETACProgLang(Official).pdf. ♦

@cgRevLines

@cgRevLines *in-seq* | *in-data* → *in-seq* | *in-data*	
in-seq	A string sequence.
in-data	A *text array* data object.

Details

Reverses the sequential order of the text lines of a string sequence (*in-seq*) or of a *text array* data object (*in-data*) returning the same string sequence or data object with the order of the text lines reversed.

Examples

The following illustrations show how the **@cgRevLines** function can be used.

```
(1) Seq := ["morning", "afternoon", "evening"]; void @cgRevLines(Seq);
(2) void @cgRevLines(@cgGetFileData("MyTextFile.txt"));
```

Example **(1)** reverses the order of the text elements in the sequence `Seq`, resulting in the same sequence having been modified to ‹["evening", "afternoon", "morning"]›.

Example **(2)** assumes that the specified file exists, and reverses the text line order of an internal copy of that file. The file is written to disk with the order of the lines reversed before the current *ECG session* ends. Note that, in this example, the **@cgRevLines** function returns the *text array* data object of the specified file. ♦

@cgRunETACFile

@cgRunETACFile *etac-code arg-str* → *rtn-cde*	
etac-code	A string or memory stack object.
arg-str	A string stack object.
rtn-cde	An integer stack object.

Details

Runs an ETAC (or TAC) file (*etac-code*) with an argument string (*arg-str*) as it would be run from RunETAC.exe.

etac-code is either a file path specification to an ETAC code file to run, or a memory stack object containing the ETAC code. The ETAC code is such that it would normally be run from RunETAC.exe, so it therefore expects a string parameter (*arg-str*) on the TAC stack.

arg-str is the string argument for the ETAC code specified by *etac-code*.

If an ETAC *error event* occurs while the ETAC code is executing, this function returns the TAC error code (*rtn-cde*) for that *error event*, otherwise the function returns `:#TAC_RTN_SUCCESS:`.

If *rtn-cde* is not `:#TAC_RTN_SUCCESS:`, then the TAC object stack is restored to the same condition that it was before the ETAC code was executed. If *rtn-code* is `:#TAC_RTN_SUCCESS:`, then the object stack is not restored. This allows the ETAC code to return stack objects on the object stack if so designed.

This function pulls off and saves the dictionaries that are on the TAC dictionary stack, except for a replicate of the "`Main`" dictionary, before the ETAC code is run. After the ETAC code has completed, the saved dictionaries are restored. It is necessary for the function to save and restore

the dictionaries on the dictionary stack to simulate running the ETAC code from RunETAC.exe, otherwise if the current dictionaries are left on the dictionary stack, the executing ETAC code could modify them causing unpredictable behaviour after it has completed. Also, the running ETAC code could itself behave unpredictable if it links into the dictionaries existing before the call.

Examples

The following illustrations show how the **@cgRunETACFile** function can be used.

```
(1) void @cgRunETACFile("MyDir\\ShowFile.btac" "InfoFile.txt");
(2) FileData @= &1; FileData += "..."; Rtn := @cgRunETACFile(FileData "");
(3) Rtn := @cgRunETACFile((@ &1 + "...") "");
```

Example **(1)** executes ShowFile.btac directly.

Example **(2)** creates a memory stack object, initialises it with *ETAC script* (indicated by the ellipsis), then executes the *ETAC script* directly from that memory stack object. The *ETAC script* is such as would be capable of being run via RunETAC.exe.

Example **(3)** is a more concise way of writing the code in example (2).

Additional Information

Unicode File Specification ♦

@cgSeqToStrLines

@cgSeqToStrLines *str-seq* → *str*	
str-seq	A string sequence.
str	A string stack object.

Details

Converts from a string sequence (*str-seq*) to a string (*str*) with EOL characters.

str-seq is a string sequence, and each of its elements is appended to a string (*str*) with the EOL (end-of-line) characters ‹$^{C}_{R}{}^{L}_{F}$›.

Example

The following illustration shows how the **@cgSeqToStrLines** function can be used.

```
(1) @cgSeqToStrLines(["Line 1", "Line 2", "Line 3"]);
```

Example **(1)** returns the string ‹Line 1$^{C}_{R}{}^{L}_{F}$Line 2$^{C}_{R}{}^{L}_{F}$Line3› ♦

@cgSetFileFlags

@cgSetFileFlags *flags*	
flags	An integer stack object containing a binary boolean value.

Details

Sets the file data flags (*flags*) affecting all files of the current *ECG session*. A set value of :!CGF_NO_WRITES: indicates that no internal file data of the current *ECG session* is to be written to disk.

Note that the following ETAC pre-processor definition needs to be made in the script that calls this function.

```
[* No data is written to disk. *]
::define !CGF_NO_WRITES 0x00000001
```

Examples

The following illustrations show how the **@cgSetFileFlags** function can be used.

```
(1) ::define !CGF_NO_WRITES 0x00000001
    @cgSetFileFlags(:!CGF_NO_WRITES:);
(2) ::define !CGF_NO_WRITES 0x00000001
    @cgSetFileFlags((@cgGetFileFlags() &and not :!CGF_NO_WRITES:));
```

Example **(1)** prevents all files of the current *ECG session* from being written to disk.

Example **(2)** removes the `!CGF_NO_WRITES` flag.

Other Information

@cgGetFileFlags ▪ @cgRepFileFlags ♦

@cgSetSymbCount

@cgSetSymbCount *s-name cnt-val*	
s-name	A string stack object.
cnt-val	A non-negative integer stack object.

Details

Sets the symbol counter value (*cnt-val*) of a *special symbol* (*s-name*).

s-name is the name of a *special symbol* as defined in the ‹@P=› keyword, by the **@SYMBOL** *command*, or by the **@cgAddCmdSymb** function. Note that *s-name* is only the name of a *special symbol*; it cannot contain other text (eg: ‹INPUT:3› is invalid for *s-name*, but ‹INPUT› is valid). The *special symbol* (*s-name*) need not be currently defined.

cnt-val is the new counter value of the specified *special symbol* (*s-name*). *cnt-val* must not be a negative number.

Other Information

@cgGetSymbCount ▪ @cgIncrSymbCount ▪ @SYMBOL ▪ @cgAddCmdSymb ▪ 2.1.1_Parameters (‹@P=› parameter) ♦

@cgShowNewDialog

@cgShowNewDialog → *bool proc* | ?	
bool	An integer stack object containing a logical boolean value.
proc	A procedure.

Details

Shows a new uninitialised input dialog box to the user, returning the relevant information (*proc*) to generate files via the **@cgGenerate** function.

bool will be `true` if the user dismisses the dialog box via the ‘Generate’ button, otherwise if the dialog box is dismissed in any other way, *bool* will be `false`.

proc is an ETAC procedure containing the arguments from the dialog box for use with the **@cgGenerate** function. If *bool* is `false`, then a null stack object (`?`) is returned instead of *proc*.

The dialog box does not actually process *template files*; it merely allows the user to enter data for later processing via the **@cgGenerate** function. Some fields are therefore disabled in the dialog box.

Example

The following illustration shows how the **@cgShowNewDialog** function can be used.

```
(1) Pars := Success := @cgShowNewDialog();
    if Success then {... @cgGenerate(Pars);} endif;
```

Example **(1)** obtains *input arguments* from the user via a dialog box, which is the same as the main input dialog box. If the user clicks the 'Generate' button on the dialog box, the **@cgShowNewDialog** function returns `true` in `Success`, and also returns a procedure (in `Pars`) containing the user-entered arguments. That procedure can then be used with the **@cgGenerate** function to generate the desired files. Note that if `Pars` is activated, then the top stack objects will be the raw arguments for the **@cgGenerate** function; those arguments may be modified if desired before being used with the **@cgGenerate** function, although that is not typically done.

Related Information

@cgGenerate ♦

@cgSortLines

@cgSortLines *in-seq* | *in-data* → *in-seq*$_1$ | *in-data*$_1$	
in-seq	A string sequence.
in-data	A *text array* data object.
in-seq$_1$	A string sequence.
in-data$_1$	A *text array* data object.

Details

Sorts the text lines of a string sequence (*in-seq*) or of a *text array* data object (*in-data*) returning the same string sequence or data object with the text lines in ascending order. *in-seq*$_1$ and *in-data*$_1$ are *in-seq* and *in-data*, respectively, with possibly modified content.

This function uses the "insertion sort" algorithm, and is efficient for an initially nearly sorted sequence.

To sort the text lines in descending order, call **@cgRevLines** with the returned value of this function as the argument.

Examples

The following illustrations show how the **@cgSortLines** function can be used.

```
(1) Seq := ["morning", "afternoon", "evening"]; void @cgSortLines(Seq);
(2) Seq := ["morning", "afternoon", "evening"];
    void @cgRevLines(@cgSortLines(Seq));
(3) void @cgSortLines(@cgGetFileData("MyTextFile.txt"));
```

Example **(1)** sorts the text elements in the sequence `Seq` in ascending order, resulting in the same sequence having been modified to ‹["afternoon", "evening", "morning"]›.

Example **(2)** sorts the text elements in the sequence `Seq` in descending order, resulting in the same sequence having been modified to ‹`["morning", "evening", "afternoon"]`›.

Example **(3)** assumes that the specified file exists, and sorts the order of the text lines of an internal copy of that file. The file is written to disk with the order of the lines sorted in ascending order before the current *ECG session* ends. Note that, in this example, the **`@cgSortLines`** function returns the (ignored) *text array* data object of the specified file. ♦

@cgStrLinesToSeq

`@cgStrLinesToSeq` *str eolchrs* → *str-seq*	
str	A string stack object.
eolchrs	A string stack object.
str-seq	A string sequence.

Details

Converts from a string (*str*) containing text lines separated by EOL (end-of-line) characters (*eolchrs*) to a sequence of text lines (*str-seq*) without the EOLs.

str is a string that typically contains text lines separated by the string in *eolchrs*.

eolchrs is a *pattern string* that indicates how the text lines within *str* are separated. For example, the string `"line 1\r\nline 2\r\nline 3"` (ie: ‹`line 1`$^{C}_{R}{}^{L}_{F}$`line 2`$^{C}_{R}{}^{L}_{F}$`line 3`›) contains three text lines if *eolchrs* is `"\r\n"` (ie: $^{C}_{R}{}^{L}_{F}$). Note that *eolchrs* can be ‹`"[{\r\n}\r\n]"`› which checks for $^{C}_{R}{}^{L}_{F}$, $^{C}_{R}$, and $^{L}_{F}$ EOL characters.

str-seq is a sequence containing the separate text lines within *str*, but without the EOL characters. If *str* is an empty string, then *str-seq* will be an empty sequence. If *eolchrs* is an empty string, then *str-seq* will contain the single element *str* if *str* is not an empty string.

This function is typically used to extract a sequence of text lines from a text file, as in the following example:

```
@cgStrLinesToSeq(mem_to_str read_file "MyTextFile.txt" "\r\n");
```

Examples

The following illustrations show how the **`@cgStrLinesToSeq`** function can be used.

```
(1) @cgStrLinesToSeq("line 1line 2line 3" "");
(2) @cgStrLinesToSeq("line 1line 2line 3" "\r\n");
(3) @cgStrLinesToSeq("" "");
(4) @cgStrLinesToSeq("" "\r\n");
(5) @cgStrLinesToSeq("line 1\r\nline 2\r\nline 3" "\r\n");
(6) @cgStrLinesToSeq("line 1:line 2:line 3:" ":");
(7) @cgStrLinesToSeq("line 1\rline 2\r\nline 3" "[{\r\n}\r\n]");
```

Examples **(1)** and **(2)** return a sequence with a single string which is the same as the first argument to the function (ie: ‹`["line 1line 2line 3"]`›).

Examples **(3)** and **(4)** return an empty sequence because the first argument is an empty string.

Examples **(5)** to **(7)** return the sequence ‹`["line 1", "line 2", "line 3"]`›.

Other Information

@cgSeqToStrLines ♦

@cgTrimStrEOL

@cgTrimStrEOL *in-str* → *out-str*	
in-str	A string stack object.
out-str	A string stack object.

Details

Trims a string (*in-str*) by removing trailing EOL (end-of-line) characters, leaving the result (*out-str*) on the object stack.

The EOL characters that are removed from the end of *in-str* are any sequence of: ᶜR and ᴸF.

Example

The following illustration shows how the **@cgTrimStrEOL** function can be used.

```
(1) @cgTrimStrEOL("string\r\n\n\r\r");
```

In example **(1)**, the input string is ‹stringᶜRᴸFᴸFᶜRᶜR›, and the function returns ‹string›. ♦

@cgTrimStrSpaces

@cgTrimStrSpaces *in-str* → *out-str*	
in-str	A string stack object.
out-str	A string stack object.

Details

Trims a string (*in-str*) by removing leading and trailing spaces, leaving the result (*out-str*) on the object stack.

in-str is a string that may contain leading and\or trailing space characters (U+0020) which are removed.

out-str is the same as *in-str* but with the said characters removed. ♦

@cgWriteAllToOne

@cgWriteAllToOne *file-path*	
file-path	A string stack object.

Details

Writes the data of all files on the internal file list to the specified (*file-path*) disk file.

file-path is internally expanded to its full file path specification before being used by this function.

The **ETAC Code Generator** maintains data from files in an internal file list, identifying the data by the full file path specification of those files. This function writes all the *file-data* of all the files on that file list to the single disk file specified by *file-path*. Each block of written data will be preceded by a text line of the form ‹****** *path*›, where *path* is the file path specification on the internal file list associated with the written file data. If the disk file specified by *file-path* already exists, it will be overwritten. Note that “no write” flags associated with file data on the file list are ineffective. The file will be written as a UTF-8 file (with a BOM signature), unless

the file characters are all a subset of the Windows-1252 character set, in which case the file will be written as a Windows-1252 file.

This function is typically used for diagnostic purposes to determine the contents of all the files on the internal file list.

Additional Information

Unicode File Specification

Other Information

@cgWriteFile ♦

@cgWriteCon

@cgWriteCon *msg*	
msg	A string stack object.

Details

Displays a message (*msg*) to the console window.

The console window can be closed via the ETAC `close_con` command. ♦

@cgWriteFile

@cgWriteFile *file-path*	
file-path	A string stack object.

Details

Writes the specified (*file-path*) internal data file to disk.

file-path is internally expanded to its full file path specification before being used by this function.

The **ETAC Code Generator** maintains data from files in an internal file list, identifying the data by the full file path specification of those files. This function writes the file data associated with *file-path* on the file list to the disk file specified by that file path. If all the characters of the file data are a subset of the Windows-1252 character set, then the file data will be written as a Windows-1252 file. Otherwise (if not all a subset), the file data will be written in the same UTF encoding scheme as the original file on disk, or if the file data was not obtained from disk or the original file was a Windows-1252 file, then the file data will be written as a UTF-8 file with a BOM signature.

No action occurs if `:!CGF_NO_WRITES:` is set (via the **@cgSetFileFlags** function) or `:!CGF_READ_ONLY:` is set (via the **@cgReplFileFlags** function) for *file-path*.

If `:!CGF_BACKUP:` is set for *file-path*, and the file to be written already exists on disk, a backup of that disk file is made before the specified file is written. If *file.ext* is the format of file name specified in *file-path*, then the backup file name will be *file*~backup.*ext*. If the backup file already exists then it will be overwritten automatically without warning. `:!CGF_BACKUP:` is set via the **@cgReplFileFlags** function.

Additional Information

Unicode File Specification

Other Information

@cgWriteAllToOne ♦

8.3 Data Object: *text array*

This section describes the ETAC functions that can be used directly with *text arrays*. The names of all such functions begin with `tl`. "`tl`" stands for "text lines". These functions can only be accessed via a *text array* data object. There are also two data members that begin with "`tsa`". "`tsa`" stands for "text string array". Note that the data object contains other private members that are undefined for the user and must not be accessed.

The data from text files is stored in *text array* data objects which are kept on an internal file list indexed by the full file specification of the text file. *Text array* data objects can also be created by the template designer for their own purposes.

ECGL functions that involve *text array* data objects also exist (see the **Text Array** heading under 8.2.1_Functions by Category).

8.3.1 Data Members

The following boxes contain a description of all the data members of the *text array* data object. **R** means that the member can be read from, and **W** means that the member can be written to.

tsaEOLChars

tsaEOLChars	
value	A string stack object. (**RW**)

Details

Contains the EOL (end-of-line) characters (*value*) of the text lines in the *text array*.

value is a string of characters that that is used to delimit the text lines in text data (typically from a text file). The string can be any characters that are not part of the text lines. The default for *value* is the string $^{C}_{R}{}^{L}_{F}$. *value* cannot contain U+0000 (which is an internal ETAC string delimiter).

The EOL characters are set to **`tsaEOLChars`** automatically when the data of a text file is read into the *text array* data object (only $^{C}_{R}{}^{L}_{F}$, $^{C}_{R}$, and $^{L}_{F}$ are recognised as EOL characters when reading file data). When the *text array* of a file is written to disk, each text line in that file will be terminated by *value*.

When a new *text array* data object is programmatically created, the programmer should set *value* as appropriate.

Other Information

tsaTextLines ♦

tsaTextLines

tsaTextLines	
value	A string sequence. (**RW**)

Details

Contains the *text array* itself (*value*), which is an ETAC string sequence.

value contains the actual text lines of the *text array*. The elements of the string sequence are not delimited by the EOL (end-of-line) characters contained in **tsaEOLChars**.

Other Information

tsaEOLChars ♦

8.3.2 Function Summary

The table below contains an alphabetical list of the function members used with *text arrays*.

Text Array Function Summary

Function	Description
tlAppendLines	Appends text lines to the end of the *text array*.
tlDeleteLines	Deletes a number of contiguous text lines of the *text array*.
tlFindMark	Returns the line number of a specified line within the *text array*.
tlGetLines	Gets a copy of the text lines of the *text array*.
tlIndentLines	Indents all text lines of the *text array*.
tlInsertLines	Inserts text lines into the *text array*.

8.3.3 Function Members

The following boxes contain a description of all the function members of the *text array* data object. The function members themselves should not be reassigned.

tlAppendLines

tlAppendLines *str-seq*

str-seq A string sequence.

Details

Appends text lines (*str-seq*) to the end of the *text array*.

Example

The following illustration shows how the **tlAppendLines** function can be used.

```
(1) TArr := @cgGetFileData(...); [* Should check for valid TArr. *]
    TArr.tlAppendLines(["yesterday", "today", "tomorrow"]);
```

In example **(1)**, the internal file data of the file indicated by the ellipsis will have the three text lines ‹yesterday›, ‹today›, and ‹tomorrow› appended to it.

Other Information

@cgGetFileData ♦

tlDeleteLines

tlDeleteLines *start-line amount*	
start-line	A positive integer stack object.
amount	An integer stack object.

Details

Deletes a number (*amount*) of contiguous text lines of the *text array* beginning at and including a specified line number (*start-line*). If *amount* is `-1` then the rest of the lines from *start-line* (inclusive) are deleted. The first line is line number `1`.

Example

The following illustration shows how the **tlDeleteLines** function can be used.

```
(1) TArr := @cgGetFileData(...); [* Should check for valid TArr. *]
    TArr.tlDeleteLines(5 3]);
```

In example **(1)**, the internal file data of the file indicated by the ellipsis will have the fifth, sixth, and seventh text lines deleted.

Other Information

@cgGetFileData ♦

tlFindMark

tlFindMark *start-line pat-type mark-pat cust-pat match-num offset* → *line-num*	
start-line	A positive integer stack object.
pat-type	A string stack object, or a null stack object (`?`).
mark-pat	A string stack object.
cust-pat	A string sequence, or a null stack object (`?`).
match-num	A non-zero integer stack object.
offset	An integer stack object.
line-num	An integer stack object.

Details

Returns the line number (*line-num*) plus an offset (*offset*) of a specified line (*pat-type*, *mark-pat*, *cust-pat*, *match-num*) within the *text array*.

This function searches the *text array* for a text line based on a *pattern string* (or plain string) specified by *mark-pat*, along with *cust-pat* if required. *offset* is added to the line number of the found text line, and the sum is returned to the caller.

start-line is a one-based line number of a text line within the *text array*, indicating that the search is to be between that line number and the last text line of the *text array* inclusively.

pat-type is a string containing a single character (or a null stack object), and determines what part of each text line of the *text array* is to be searched. *pat-type* is as follows (*M* is *mark-pat*, and *T* is the current text line in the *text array*):

Pattern String Types

pat-type	Meaning
P	implies that *M* contains a *pattern string*. The match is for any part of *T*.
S	implies that *M* contains a *pattern string*. The match is for the start (initial) part of *T*.
E	implies that *M* contains a *pattern string*. The match is for the end part of *T*.
A	implies that *M* contains a *pattern string*. The match is for all of *T*.
?	a null stack object implies that *M* is a plain string. The match is for all of *T*.

The consequence is undefined for any other value of *pat-type* than shown above.

mark-pat (along with *cust-pat*) is a *pattern string* or a plain string which is to be matched by a text line within the *text array* between and including the text line indicated by *start-line* and the last text line of the *text array*.

cust-pat is a string sequence of custom *pattern strings* indicated within *mark-pat*, or a null stack object if *mark-pat* does not require custom *pattern strings*.

match-num is a positive or negative integer. It determines which matched text line is to be the matching line, searching from the text line indicated by *start-line* (if *match-num* is positive), or searching backwards from the end of the *text array* (if *match-num* is negative). For example, if *match-num* is ‹-3› then the matching line will be the third last match of *mark-pat* within the *text array*. A value of zero for *match-num* is invalid.

offset is internally added to the matching line number if there was a match. Note that *offset* can be a negative number. *offset* is typically zero.

line-num will be the one-based line number of the matched text line within the *text array* plus *offset*, or -1 if there was no match. If there was a match, *line-num* will be limited to the value of *start-line* minus one and the number of text lines in the *text array*, inclusively.

Examples

The following illustrations show how the **tlFindMark** function can be used. TArr is assumed to contain a *text array* data object.

```
(1) LineNum := TArr.tlFindMark(10 ? "//+MARKER+//" ? 1 0);
(2) LineNum := TArr.tlFindMark(10 ? "//+MARKER+//" ? -1 0);
(3) LineNum := TArr.
    {tlFindMark(max 1 (|tsaTextLines| - 4) "S" "%%`d" ? -3 0);};
(4) LineNum := TArr.tlFindMark(1 "P" "%%{%<p1><p0>}$<p1>" ["`<%
    [<p1><p0>]`>", "[~[`<`>]]"] 3 -1);
```

In example **(1)**, the first text line within TArr containing the string ‹//+MARKER+//› is searched for, beginning with the tenth line inclusively. If the text line is found, LineNum will contain its line number, otherwise it will contain the number -1. Note that, because the second argument (*pat-type*) is a null stack object, the third argument, *mark-pat*, is a plain string (not a *pattern string*).

Example **(2)** is the same as example (1), except that the last text line matching ‹//+MARKER+//› is to be searched for.

In example **(3)**, the third last text line within TArr beginning with at least one digit character is searched for, beginning with the fifth last line inclusively. If the text line is found, LineNum will contain its line number, otherwise it will contain the number -1. Note that the first argument of **tlFindMark** must not evaluate to an integer less than one. The fifth last line is therefore

expressed by ‹max 1 (|tsaTextLines| - 4)› to cater for the number of text lines being less than five.

In example **(4)**, the third text line within TArr containing nested and balanced ‹<› and ‹>› blocks is searched for, beginning with the first line inclusively. If the text line is found, LineNum will contain its line number minus one, otherwise it will contain the number -1. Note that the *pattern string*, *mark-pat*, uses a custom *pattern string* sequence, *cust-pat*.

Additional Information

See **Pattern String Matching** under chapter 3 of the "*The Official ETAC Programming Language*" document, ETACProgLang(Official).pdf.

Other Information

@OUTPUT ▪ @INSERT ▪ @DELETE ♦

tlGetLines

tlGetLines *start-line end-line* → *out-seq*	
start-line	A positive integer stack object.
end-line	A positive integer stack object.
out-seq	A string sequence.

Details

Gets a copy (*out-seq*) of the text lines of the *text array* between two line numbers (*start-line*, *end-line*) inclusively.

start-line is a one-based line number of a text line within the *text array* indicating the first text line to get. The consequence is undefined if *start-line* indicates a line number before the first line or after the last line of the *text array*.

end-line is a one-based line number of a text line within the *text array* indicating the last text line to get. The consequence is undefined if *end-line* indicates a line number before the first line or after the last line of the *text array*.

out-seq contains a copy of the text lines between the first and last specified ones inclusively. ♦

tlIndentLines

tlIndentLines *num-pos pad*	
num-pos	A non-negative integer stack object.
pad	A string stack object.

Details

Indents all text lines of the *text array* a number of positions (*num-pos*) filled with a character (*pad*). Indentation also applies to the text lines containing EOL (end-of-line) characters as defined by the **tsaEOLChars** member of the *text array* data object. Typically, the EOL character string is ‹$^{C}_{R}{}^{L}_{F}$›.

An element of the *text array* could contain more than one text line, each separated by the EOL characters. For example, the string element "line 1\r\nline 2\r\nline 3" (ie: ‹line 1$^{C}_{R}{}^{L}_{F}$line 2$^{C}_{R}{}^{L}_{F}$line 3›) contains three text lines if the EOL character string is "\r\n" (ie: $^{C}_{R}{}^{L}_{F}$).

num-pos is a non-negative integer indicating the number of positions to indent the text lines in the *text array*.

pad is a string, but only the first character, which must be a UCS-2 (BMP Unicode scalar value) character, is used to pad the indentation. This will typically be a space character. If *pad* is an empty string then no indentation will occur.

Example

The following illustration shows how the **tlIndentLines** function can be used.

```
(1) TArr.tlIndentLines(3 "*");
```

In example **(1)**, assume that the *text array* [‹First line›, ‹line 1$^{C_R}{}_{L_F}$line 2$^{C_R}{}_{L_F}$line 3›] is contained in the *text array* data object, TArr, and that the EOL character string of the data object is ‹$^{C_R}{}_{L_F}$›. The *text array* will end up containing the two elements ‹***First line› and ‹***line 1$^{C_R}{}_{L_F}$***line 2$^{C_R}{}_{L_F}$***line3›.

Other Information

tsaEOLChars ▪ @cgIndentLines ♦

tlInsertLines

tlInsertLines *start-line in-seq* → *num-lines*	
start-line	A positive integer stack object.
in-seq	A string sequence.
num-lines	A positive integer stack object.

Details

Inserts text lines (*in-seq*) into the *text array* at a text line position (*start-line*), returning the number of lines inserted (*num-lines*).

start-line is a one-based line number of a text line within the *text array* indicating the position of the first text line to be inserted. The consequence is undefined if *start-line* indicates a line number before the first line or after the last line of the *text array*.

in-seq contains the text lines to insert into the *text array*. The existing text lines beginning at position *start-line* in the *text array* are moved to subsequent positions to make room for the inserted text lines.

num-lines will be the number of lines inserted, that is, the size of *in-seq*. ♦

Appendix A

HTML Template Files

This appendix is about using HTML code in *template files*.

A.1 Introduction

The *meta-codes* in a *template file* are delimited by angle brackets ("less than" and "greater than" characters, ‹<› and ‹>›, respectively). This situation may cause a problem if HTML text is included in a *template file* because HTML text also uses the same angle brackets for its tags, and the **ETAC Code Generator** may confuse HTML tags with *meta-codes*. The following solution is recommended for using HTML (and other files that use angle brackets as delimiters) as *template files*.

A.2 HTML in a Template File

HTML text can be used directly in a *template file*, provided that the follow methods are used. These methods are not suitable for rendering the *template file* in an HTML browser.

For a small amount HTML text, the angle brackets that delimit HTML tags need to be replaced with the corresponding *instructions*; *meta-codes* remain unchanged. The following example illustrates how the HTML text needs to be modified in a *template file*.

The text below shows an HTML text fragment (with proper HTML tags shown in bold **blue**) that is desired to exist in a *template file*.

```
<p>I have some
<em><FRUITS:#1></em>,
and I like them all.</p>
```

The text above is rewritten in the *template file* as follows.

```
<&lt;p<&gt;I have some
<&lt;em<&gt;<FRUITS:#1><&lt;/em<&gt;,
and I like them all.<&lt;/p<&gt;
```

Notice that the HTML tag angle brackets have been replaced with the corresponding *instructions* (<< and <>). Assuming that <FRUITS> has values apples, oranges, bananas, the text above generates

```
<p>I have some
<em>apples</em>,
<em>oranges</em>,
<em>bananas</em>,
and I like them all.</p>
```

which is the desired HTML code.

An alternative to the method above is to retain the angle brackets of the original HTML text, but specify the option IGNORE_BAD_SYMB in the *header block* at the ‹@S=› keyword. The IGNORE_BAD_SYMB option ignores *special symbols* that are undefined or invalid, such as <p> and . If this alternative is used, *special symbols* must not be identical to HTML tags. For example, a *special symbol* must not be , otherwise the will be replaced by its value; however, <EM> can be used for the *special symbol* if it is not to be used as an HTML tag.

A.3 HTML in an External File

Substantial HTML text should be put in a separate file, and included into the *template file* via the **@INSERT** *command*. The following method allows a separate HTML file, for use in a *template file*, to be display in an HTML browser. The HTML file will contain (altered) *meta-codes*, and so may not be displayed as expected in the browser.

With this method, all *meta-codes* within the HTML file are enclosed within ‹[~› and ‹~]› instead of the angle brackets. The HTML file is then automatically converted via the **cgCvtToAngBraks** function when it is included into the *template file*.

As an illustration, assume that the text below exists in an HTML file. Note especially that the angle brackets ‹<› and ‹>› of the *meta-codes* have been replaced with ‹[~› and ‹~]›, respectively. This allows the HTML file to be displayed in an HTML browser; if the angle brackets of the *meta-codes* were not replaced, the file would not have displayed properly in the browser, if at all.

```
<HTML>
<BODY>
[~@JOIN:[]~]
<p>I have some
 <em>[~FRUITS:#1~]</em>,
 and I like them all.</p>
[~@END:[JOIN]~]
</BODY>
</HTML>
```

Assuming that the text above exists in a file called MyHTML.html, the following illustration shows how to include that file into a *template file*.

```
@ECG V1@
...
@S=IGNORE_BAD_SYMB
...
@endhead@
...
<@INSERT:[PATH="MyHTML.html" SCRIPT=(@cgCvtToAngBraks())]>
...
```

The option IGNORE_BAD_SYMB in the *header block* at the ‹@S=› keyword ignores *special symbols* that are undefined or invalid within the inserted HTML file, such as <p> and . The **cgCvtToAngBraks** function respectively converts ‹[~› and ‹~]› back to ‹<› and ‹>› before the HTML text is inserted via the **@INSERT** *command*.

Note that the **@INSERT** *command* is processed at stage 1. There may be situations where other processing needs to be done before the HTML file is inserted, therefore, the **@INSERT** *command* may need to be activated at a later stage. This is accomplished by using the DEFER keyword with the **@INSERT** *command*, as follows.

```
@ECG V1@
...
@S=IGNORE_BAD_SYMB
...
@endhead@
...
<@REPROCESS:[]>
<@INSERT:[PATH="MyHTML.html" SCRIPT=(@cgCvtToAngBraks()) DEFER]>
<@END:[REPROCESS]>
...
```

Because the DEFER keyword forces the **@INSERT** *command* to be activated at stage 12, *meta-codes* within the HTML file that need activation prior to stage 12 will not be activated unless the **@INSERT** *command* is placed within the **@REPROCESS** and **@END**[REPROCESS] *commands*. The **@REPROCESS** *command* processes the inserted HTML file from stages 4 to 17. In some cases, even using the **@REPROCESS** *command* may not produce the desired result with the DEFER keyword. In that case a different strategy needs to be undertaken.

Note that if the inserted HTML file contains *commands* that are spread over more than one line, then the **@cgCvtTmplData** function will need to be called at ‹SCRIPT=›, as shown in the following illustration.

```
<@INSERT:[PATH="..." SCRIPT=(@cgCvtToAngBraks(); @cgCvtTmplData();)]>
```

If all the *commands* in the HTML file are already on a single text line, then **@cgCvtTmplData** need not be called.

To prevent ‹[~› and ‹~]› from being converted by **@cgCvtTmplData**, use ‹[[~&~]~› and ‹~[~&~]]›, respectively. **@cgCvtTmplData** will convert ‹[**[~&~]**~› to ‹[**<&>**~› and ‹~**[~&~]**]› to ‹~**<&>**]›, and the **ETAC Code Generator** will then convert ‹[<&>~› to ‹[~› and ‹~<&>]› to ‹~]›. Alternatively, ‹[<&>~› and ‹~<&>]› can be used directly in the HTML file if desired.

Appendix B

Self-contained ETAC Code Generator

Officially, the **ETAC Code Generator** is released as a compiled TAC binary instruction file named ETACCodeGen.btac, along with the ETAC source files, which is part of (and depends upon) the **Run ETAC Scripts** package release. The official release of the **ETAC Code Generator** requires, therefore, that the **Run ETAC Scripts** package be already installed. This appendix is about the self-contained release of the **ETAC Code Generator**, which does not require **Run ETAC Scripts** to be already installed. The self-contained implementation of the **ETAC Code Generator** itself is named ETACCodeGen.exe.

LICENCE NOTE

The licence agreement, included within the **ETAC Code Generator** installation folder, must be accepted by the person who installed the **ETAC Code Generator** before the **ETAC Code Generator** is permitted to be used or copied in any way. If the licence agreement is not accepted, then the **ETAC Code Generator** installation folder must be deleted.

B.1 Introduction

The self-contained implementation of the **ETAC Code Generator**, ETACCodeGen.exe, can be run where **Run ETAC Scripts** is not installed. ETACCodeGen.exe is actually a silent installer that temporarily installs a TAC binary instruction file (ETACCodeGen.btac) containing the **ETAC Code Generator**, along with a portable implementation of **Run ETAC Scripts** (RunETAC.exe), and executes that instruction file. The installation is automatically put into a temporary folder, which is then automatically deleted after the **ETAC Code Generator** has finished executing. The temporary installation does not affect the system registry.

The self-contained **ETAC Code Generator** package is released as a self-extracting ZIP file, ‹ETACCodeGenEXE_...Installer.exe›, which creates a user-specified folder containing ETACCodeGen.exe and other relevant files, including the licence agreement, documentation, and some *template files*.

B.2 System Requirements

The self-contained **ETAC Code Generator** has the same system requirements as does the **Run ETAC Scripts** package, and is therefore only released for the **Windows**® operating system using the x86 (32-bit) architecture (it can also run on the x64 (64-bit) architecture) beginning with **Windows**® **XP**. Note that the self-contained **ETAC Code Generator** package is not released for other platforms and non-**Windows**® operating systems. It is expected to operate correctly on any **Windows**® operating system compatible with the one mentioned. Also, the **ETAC Code Generator** operates with Unicode files.

B.3 Self-contained Installation

The file ‹ETACCodeGenEXE_...Installer.exe› will create a directory tree structure in a directory specified by the user as follows.

```
ETACCodeGenEXE (default installation directory name, can be renamed during installation)
    Documents
    |   ECGTemplates_Licence.pdf (licence agreement for the ETAC Code Generator templates)
    |   ETACCodeGenerator.pdf (this document)
    |   ETACCodeGenEXE_Licence.pdf (licence agreement must be accepted to use the ETAC Code Generator)
```

```
| ETACCodeGenTemplates.pdf (describes the files in the Templates directory)
| ETACErrorCodes.pdf (lists the ETAC programming language syntax error codes)
| ETACOverview.pdf (overview of the ETAC programming language)
| ETACProgLang(Official).pdf (the official definition of the ETAC programming language)
Other
| ECGTdef.xml (for use with Notepad++)
Templates
| ECGData
| | ExternTACLibECG.cpp
| | ExternTACLibECG.def
| ECGTSourceFile.ecgt
| CPPModsRec.ecgt
| CPPSourceFiles.ecgt
| DataFileReg.ecgt
| ETACForApp.ecgt
| ETACMainApp.ecgt
| ETACModsRec.ecgt
| ExternTACLib.ecgt
| MakeEXE.ecgt
| MakeExtractor.ecgt
ECGTFile.ico (can be used with template files)
ETACCodeGen.exe (is the self-contained ETAC Code Generator)
ETACCodeGen.ini
MakeECGIni.cmd (creates ETACCodeGen.ini)
ReadMe.txt (read this first)
```

Documents

ETACCodeGenEXE_Licence.pdf and ECGTemplates_Licence.pdf contain the licence agreement which must be accepted by the person who performed the installation before the **ETAC Code Generator** can be used. If the licence agreement is not accepted, the **ETAC Code Generator** installation must be deleted. ETACCodeGenTemplates.pdf describes the files in the `Templates` folder. ETACOverview.pdf contains an overview of the ETAC™ programming language, and ETACProgLang(Official).pdf is the official definition of the ETAC programming language.

Other

ECGTdef.xml is the **Notepad++** definition file for syntax highlighting the contents of *template files* displayed in the **Notepad++** editor window. See chapter 3, **Editing Template Files**, in the "*ETAC Code Generator Templates*" document, ETACCodeGenTemplates.pdf.

Templates

This folder contains the *template files* which are described in the "*ETAC Code Generator Templates*" document, ETACCodeGenTemplates.pdf.

Files

The installer program runs MakeECGIni.cmd automatically which creates the initialisation file ETACCodeGen.ini. ReadMe.txt contains some initial information for the users of the **ETAC Code Generator**. ECGTFile.ico can be used as an icon for *template files*; the user will need to set up such usage himself. ETACCodeGen.exe is the **ETAC Code Generator** program that the user runs to generate text files.

B.3.1 The Initialisation File

The initialisation file, ETACCodeGen.ini, is initially created automatically by the installation process to contain references to the installation directory (see 5.2 Initialisation File for more details about the content of the initialisation file). Whenever the user changes the installation directory name or path, the user should execute MakeECGIni.cmd to create new references in the

initialisation file to the changed name or path. MakeECGIni.cmd will make a backup of an existing ETACCodeGen.ini file before changing it, overwriting a previous backup. ETACCodeGen.ini can be moved to the system `Windows` directory if desired.

B.4 ETAC Code Generator Execution

ETACCodeGen.exe is actually a silent installer, and when executed it installs and runs a portable version of the **ETAC Code Generator** and **Run ETAC Scripts** in a temporary directory ("folder"), which is then automatically deleted before ETACCodeGen.exe terminates. The current directory, while ETACCodeGen.exe is running, is the said temporary directory. Therefore, any files generated into that temporary directory will be subsequently deleted.

The **ETAC Code Generator** program, ETACCodeGen.exe, can be run directly, or via a shortcut file. Either way, ETACCodeGen.exe, will always create an input dialog box.

B.4.1 Direct Execution

ETACCodeGen.exe is executed directly by double clicking it from the **Windows**® environment. The *input arguments* string for the internal ETACCodeGen.btac is:

```
'INI_DIR="exe-dir" AUTOLOG PROMPT TEMPLATE="" ARGS=()'
```

where *exe-dir* is the directory containing ETACCodeGen.exe. When executed, ETACCodeGen.exe, will display the input dialog box. Note that the 'Output Folder' field displayed in the input dialog box will be the temporary directory of the **ETAC Code Generator**. The user should select a different directory for *generated files*, otherwise, if the files are generated into the temporary directory, they will be automatically deleted. To locate the temporary directory containing the **ETAC Code Generator**, click the 'Output Folder' button on the input dialog box when it is first presented.

B.4.2 Command Line Execution

ETACCodeGen.exe can be executed on a command line via a shortcut file or a command file. In this case, the *input arguments* string for the internal ETACCodeGen.btac is

```
'INI_DIR="exe-dir" AUTOLOG PROMPT cmd-str'
```

From within **Windows**®, the command line is typically entered in a shortcut file to the **ETAC Code Generator** (ETACCodeGen.exe) as follows. In the `Target` entry of the shortcut properties, enter the following:

```
"exe-dir\ETACCodeGen.exe" [cmd-str]
```

where *exe-dir* is the directory containing ETACCodeGen.exe, and *cmd-str* is the command line string arguments specified by the user on the command line. The arguments on the command line string must include the ‹TEMPLATE=› and ‹ARGS=› (or ‹ARG_FILE=›) keywords, unless no arguments are supplied (in which case, ‹TEMPLATE="" ARGS=()› is the default). In addition, the arguments for the keywords ‹GEN_DIR=›, ‹OUTPUT=›, and ‹LOG=› (if present) should not be specified as relative paths to avoid *generated files* or the log file from being created in the temporary directory. See Command Line Input Arguments for details of *cmd-str*.

An example of a command line specified by the user is:

```
...\ETACCodeGen.exe TEMPLATE="ETACMainApp.ecgt" ARGS=() GEN_DIR="C:\..."
```

which will display the input dialog box with the specified parameters.

B.4.3 ETAC Script Execution

The **ETAC Code Generator** can be executed directly from *ETAC script* within a *template file*. The current directory will be the temporary directory containing ETACCodeGen.btac (existing internally to ETACCodeGen.exe), and so paths relative to the current directory should not be specified.

> **Important Note**
> Do not use `exec_tac` to run the **ETAC Code Generator** — the consequence is unpredictable.

The following example illustrates how to execute the **ETAC Code Generator** from *ETAC script* inside a *template file*. The second argument of **`@cgRunETACFile`** is only for illustration.

```
<@SCRIPT:[]>
RtnCode :- ?; IniDir :- ?;
   ...
   IniDir := (+ "INI_DIR='" @cgGetCmdLineArgs().claIniDirPath "'");
   RtnCode := @cgRunETACFile("ETACCodeGen.btac"
      (IniDir + " TEMPLATE='...' ARGS=(...) OUTPUT='C:\\...'"));

   if ( RtnCode = :#TAC_RTN_SUCCESS: ) then {...} endif;
   ...
<@END:[SCRIPT]>
```

The second argument of **`@cgRunETACFile`** in the example above is the command line *input arguments*, which must include the ‹TEMPLATE=› and ‹ARGS=› keywords (see Command Line Input Arguments). In addition, the arguments for the keywords ‹GEN_DIR=› and ‹OUTPUT=› (if present) should not be specified as relative paths to avoid *generated files* from being created in the temporary directory. In the example above, `IniDir` is assigned the ‹INI_DIR=› keyword with its argument being the directory containing ETACCodeGen.exe as the directory for the initialisation file. A different initialisation file can be specified if desired.

An *ECG session* can be executed from within a *template file* to produce the *generated lines* into an ETAC sequence (`OutSeq`) as illustrated in the following example. As in the preceding example, file and directory paths should not be specified as relative paths because the current directory will be the temporary directory. In particular, the fourth argument should not be the null stack object (`?`).

```
<@SCRIPT:[]>
OutSeq :- []; Success :- ?;
   ...
   Success := @cgGenerate("C:\\..." OutSeq "..." "C:\\..." 0);

   if Success then {...} endif;
   ...
<@END:[SCRIPT]>
```

Note the double backslash (`\\`) used in the file paths of the preceding examples. The double backslash represents a single backslash, and is necessary because the backslash is an escape character in regular ETAC strings. Alternatively, if single-quoted strings (“raw” strings) are used, the backslashes must not be doubled.

B.5 ETAC Script Debugging

ETAC script, whether existing in a **`&FNT`** *instruction*, on in an **`@INSERT`**, **`@SCRIPT`**, **`@IF`**, or **`@EDIT`** *command*, is debugged as described in chapter 2, **ETAC Debugger**, of the “*The Official ETAC Programming Language*” document, ETACProgLang(Official).pdf.

To debug *ETAC script*, the **ETAC Code Generator** needs to be started in debug mode via the ETACCodeGen.exe program after placing breakpoints at suitable positions within the *ETAC script*. In the debug window, the 'Silent Continue' button needs to be clicked repeatedly until the debugger pauses at a set break point. Debugging the *ETAC script* can commence from that point as described in the aforementioned chapter. Note that breakpoints can be placed within the *ETAC script* of **&FNT** *instructions* and conditions of **@IF** *commands*.

To start the **ETAC Code Generator** in debug mode, the special keyword (-debug-) needs to be specified as the first argument on the command line, as in the following illustration.

```
"...\ETACCodeGen.exe" -debug- TEMPLATE="ETACMainApp.ecgt" ARGS=(...)
```

During the debugging session, there may be occasions when the response is extremely slow; this is currently unavoidable.

B.6 Uninstalling the ETAC Code Generator

Because the **ETAC Code Generator** does not alter the system registry, uninstallation is achieved simply by deleting the installation directory. Note that if the user associated ECGTFile.ico with *template files*, then that association must be broken by the user.

Bibliography

An Overview of ETAC copyright © Victor Vella (2020)

The Official ETAC Programming Language copyright © Victor Vella (2020).

Glossary

C

command
Specially marked text in the *template line block* that processes groups of *template lines*, or groups of text lines that exist in an external file.

See 2.2.5_Commands for more information.

command symbol
A *special symbol* defined by the **@SYMBOL** *command* or by the **@cgAddCmdSymb** function. A *command symbol* and its values exists in an internal list of such symbols separate from the *special symbols* specified at the ‹@P=› keyword of the *header block*. A *command symbol* can have more than one value, and is used like any other *special symbol*.

continued line
A text line ending with ‹\CRLF›, ‹\CR›, or ‹\LF› in a *template line block*. A series of *continued lines* and the following text line are treated as a single *template line*.

E

ECG session
The full processing of a single *template file*. An *ECG session* can be started via a command line, the **@cgGenerate** function, or the **@GEN** or **@POSTGEN** *commands*.

See 5_Operating the ETAC Code Generator for more information.

ECGL
ECGL stands for "**E**TAC **C**ode **G**enerator **L**anguage". *ECGL* is a unique sophisticated declarative *template file* language, used by the **ETAC Code Generator**, having capabilities extended by *ETAC scripts*. *ECGL* consists of *meta-codes* and a standard function library of *ECGL functions*.

ECGL function
An ETAC function intrinsically defined as part of a standard function library for *ECGL*. An *ECGL function* begins with the characters @cg.

error event
The situation that occurs when the action of an ETAC command or operator can no longer proceed. In such a case, the ETAC interpreter intercepts the action and takes appropriate action which typically consists of ending the main *ECG session*, unless the *error event* is trapped by appropriate ETAC code.

ETAC comment
A comment token as defined for the ETAC™ programming language. An *ETAC comment* consists of the two sequences of characters ‹[*···*› and ‹*···*]›, and the text between them. *ETAC comments* are ignored by the **ETAC Code Generator**.

ETAC script
ETAC program code that is in human readable and writable text form. A file containing only *ETAC script* typically has an extension of 'etac'. Note that the term "ETAC script" is used in the same sense as the word "code", as in "ETAC script code".

G

generated file
A disk file generated or modified by the **ETAC Code Generator**.

generated line
A final text line generated by the **ETAC Code Generator** requiring no more processing.

H

header block
The *header block* is the first section of a *template file* containing information for the **ETAC Code Generator** for processing the *template line block* following that first section. A *header block* ends with the text line ‹@endhead@›.

I

input arguments
Operational information supplied by the user (typically via a dialog box or the command line) for the **ETAC Code Generator** to initiate the creation of *generated files*.

See 5.1_Command Line for details of the *input arguments*.

input point
An imaginary point that exists between characters, before the first character, or after the last character in a *template line* during the processing of that *template line*.

instruction
Specially marked text in the *template line block* that modifies (an internal copy of) the *template line* in which the *instruction* exists.

See 2.2.2_Instructions for more information.

K

keyword template
A *keyword template* is a template specification, against which *template arguments* are matched and parsed, as defined by the keyword-arguments system. The *keyword template* for a *template file* is specified at the ‹@T=› keyword of the *header block*.

See **Appendix A: Keyword-arguments Specifications** in the document "*The Official ETAC Programming Language*" (ETACProgLang(Official).pdf) for details of a template specification.

L

line continuation character
The backslash (\) at the end of a *continued line* is called the *line continuation character*. During processing of a *template file*, the *line continuation characters* are removed (but the end-of-line characters remain), and a contiguous series of *continued lines*, along with the text line that follows, are treated as a single *template line*.

M

meta-code
The *special symbols*, *instructions*, and *commands* in a *template file* are collectively called *meta-codes*.

multi-line
A *template line* that can generate more than one *output line*. A *multi-line* is identified by the existence of hash (#) characters in the *special symbols* of a *template line*.

O

output file
The file that receives the default *generated lines* of a *template file*, including the *generated lines* of **@OUTPUT** *commands* that do not specify an *output file* (at ‹PATH=›). This is the main file produced by the **ETAC Code Generator**. The generated text produced by the **ETAC Code Generator** may exist entirely within the *output file*, or text may also be generated to other files as specified by the *commands* within the *template files*. The data in the *output file* is, in fact, a modified copy of the *template line block* after all processing has been completed.

The *output file* will be written as a UTF-8 file (with a BOM signature), unless the file characters are all a subset of the Windows-1252 character set, in which case the file will be written as a Windows-1252 file.

output line
A resulting text line of a particular operation of a *meta-code*. An *output line* may undergo further processing before becoming a *generated line*.

output point
An imaginary point that exists between characters, before the first character, or after the last character in an *output line* during the processing of a *template line*.

P

pattern string
A *pattern string* is a string which is composed of characters to be matched literally and special characters that indicate predefined patterns to be matched or used to indicated how the pattern matching process is to be performed. *Pattern strings* are unique to, and defined by, the ETAC™ programming language. *Pattern strings* are analogous to “regular expressions” in other programming languages.

See **Pattern String Matching** under chapter 3 of the “*The Official ETAC Programming Language*” document, ETACProgLang(Official).pdf, for more information including the syntax of a *pattern string*.

protection instruction
An *instruction* that immediately encloses a *meta-code* so that the *meta-code* is not processed when it would normally be processed; the *meta-code* is thus “protected”. When a *protection instruction* is itself activated, it transforms itself into the enclosed *meta-code*.

S

special symbol
Text within the *template line block* that conforms to a certain syntax, and which gets replaced with other text (the *special symbol’s* ‘value’) during the processing of *template lines*.

See 2.2.1_Special Symbols for more information, including the syntax of a *special symbol*.

string block

This is any part of a text string delimited by matching parentheses ‹()›, square brackets ‹[]›, braces ‹{}›, double quotes ‹"›, or single quotes ‹'›. A *string block* includes the said delimiter characters. The three bracket delimiters can be nested. Quoted substrings within bracket delimiters are skipped. Backslash escaped quote characters within single and double quoted *string block* are ignored. For example, the highlighted substrings within
‹is there ((anybody) "("out) there, tell 'me\' if' there is›
are *string blocks*.

T

template arguments

Template arguments are keywords and their arguments, supplied by the user, that match the *keyword template* of a *template file*. *Template arguments* are used as values for *special symbols*.

ETAC comments within *template arguments* are logically replaced with one space, unless the comments are within a pair of double quotes or double-angle quotes. For example, the *template arguments* string, ‹KW=my[*comment*]arg1, arg2"[*cmt*]"›, is logically equivalent to ‹KW=my$_P$arg1, arg2"[*cmt*]"›. However, ‹KW=my«[*comment*]»arg1, arg2"[*cmt*]"› is logically equivalent to ‹KW=my[*comment*]arg1, arg2"[*cmt*]"›. Note that the outer double-angle quotes are ignored, but the text in-between those quotes remains.

Backslashes (‹\›) in *source arguments* that are outside of *string blocks* are ignored and the character following a backslash is accepted literally, for example, escaped *ETAC comments* outside of *string blocks* are retained as in this example with only one *source argument*, ‹KW=**my\[*comment*\]one\,arg**›. The backslashes are automatically removed, leaving ‹KW=**my[*comment*]one,arg**› as the effective *source argument*. Without the backslashes in the example, the comment is replaced with a single space, and the keyword will have two *source arguments* instead of one.

See **A.3 Source String Syntax** under **Appendix A: Keyword-arguments Specification** in the document "*The Official ETAC Programming Language*" (ETACProgLang(Official).pdf) for more details about the syntax of *source arguments*.

template file

A special text file, for use with the **ETAC Code Generator**, acting as a model for generating or modifying one or more text files for desired purposes. A *template file* contains fragments of the generated text, *meta-codes*, and possibly *ETAC script*, that describe how those fragments are to be produced into the *generated files*.

A *template file* can be a Windows-1252, UTF-8, UTF-16, or UTF-32 file. If a *template file* is a UTF file, it is highly recommended that the file has a BOM signature (including for UTF-8).

See 1.5_Overview of a Template File and 2_The Template File for more information.

template line

A text line within the *template line block*, that directly participates in producing *generated lines*. *Commands* can also be considered to be *template lines* in some cases. **@SCRIPT** *command* blocks are not usually considered to be *template lines* for the purpose of producing *generated lines*. A *template line* can contain *meta-codes*, or it may be a plain text line.

template line block

All the text lines in a *template file* that are beneath the ‹@endhead@› text line (ie: that follow the *header block*).

See 2.2_The Template Line Block for more information.

text array
An ETAC string sequence (**tsaTextLines**) encapsulated by a particular ETAC data object (the "*text array* data object") that contains members for manipulating that sequence.

See 8.3_Data Object: text array for more information.

U

u-char
A Unicode scalar value. A *u-char* is equivalent to a UTF-32 code unit. The size of a *u-char* in an ETAC string is two or four bytes (one or two *w-chars*, respectively). However, a *u-char* size as a character is considered to be one unit in length. Note that a surrogate pair is one *u-char* (even though it is two *w-chars*). A surrogate code point is not a *u-char* (it is a *w-char*).

W

w-char
A Unicode code point in the BMP (Basic Multilingual Plane). A *w-char* is equivalent to a UTF-16 code unit. The size of a *w-char* in an ETAC string is two bytes. However, a *w-char* size as a character is considered to be one unit in length. Note that a surrogate code point is a *w-char*.