

The ETAC Compiler

1 February 2019

Copyright © **Victor Vella** (2018, 2019)

All rights reserved



TM

ETAC Compiler Program

Version: 1-0-6-ena

Victor Vella

Contents

Contents

Document Conventions



1. **Introduction**
2. **Command Line Keywords**
3. **ETAC Compiler Operations**
 - 3.1 Conversion to Binary File
 - 3.2 Conversion to Text Instructions
 - 3.3 Producing Pre-processor Output
 - 3.4 Conversion to Pre-processed Packed File
 - 3.5 Conversion to Unprocessed Packed File
 - 3.6 Syntax Checking Only
4. **The Syntax Error and Warning Window**
5. **Source File Viewer**
6. **List of Error and Warning Codes**

Glossary

Other Related ETAC Documents

ETAC_Preliminaries.pdf	Preliminaries before using ETAC
ETACOverview.pdf	An Overview of ETAC
ETACProgLang(Official).pdf	The Official ETAC Programming Language
RunETAC.chm	Run ETAC Scripts Help
ETACWithCPP.pdf	ETAC: Interacting with C++
ETACCompiler.chm	ETAC Compiler Help
ETACErrorCodes.pdf	ETAC Compilation and Run-time Error Codes

Legal Information

ETAC™, the ETAC logo ™, and the **ETAC Compiler** logo ™ are unregistered trademarks 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.

This document is copyright © 2019 by Victor Vella. All rights reserved. Permission is hereby granted to make any number of exact electronic copies of this document without any remuneration whatsoever. Permission is also granted to make annotated electronic copies of this document for personal use only. Except for the permissions granted, and apart from any fair dealing as permitted under the relevant Copyright Act, no part of this document may be reproduced or transmitted in any form or by any means without the express permission of the author. The copyright of this document shall remain entirely with the original copyright holder.

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.

Document Conventions

The following symbolic conventions are used in this document.

Symbol	Meaning
$\langle x \rangle$	separates x as a unit of information from the surrounding text.
$[x]$	means that x optional.
(x)	groups x as a unit.
$x y$	means that only x or y applies, but not both (could have more than two options).
$\{y\}x$	means that if x is not present, then y is the default.
\dots	represents omitted text (as usual).
<i>text</i>	maroon coloured italic text is a link to the text's definition.

The ETAC Compiler

This document is for version **1-0-6-ena** of the **ETAC Compiler** program (ETACCompiler.exe).

(Australian English)

1. Introduction

The **ETAC Compiler** program converts *ETAC code* files from one form to another, and can do detailed syntax checking on *ETAC text script* files. Note that it is not necessary to compile *ETAC text script* files to execute them.

An *ETAC code* file can be in one of the following forms:

1. Plain text as a regular ETAC (or TAC) script file. (extensions: *.etac or *.tac)
2. Pre-processed and compressed ETAC text script file (*ETAC packed script*). (extension: *.ptac)
3. Compressed TAC binary file (containing *TAC binary instructions*). (extension: *.btac)

The **ETAC Compiler** program can convert *ETAC code* from any one of the above forms to any other. Original ETAC source code is in the form (1) above. However, converting *ETAC script* from plain text to another form can yield certain advantages. A TAC binary file, (3), can load slightly quicker than a text ETAC file, (1) or (2), since no parsing and syntax checking is done. But the main advantage of a TAC binary file is that the inclusion files would have all been processed and the results are included in the binary file. However, a TAC binary file cannot normally be debugged unless it is first converted to a text file (conversion from (3) to (1)). Such a conversion, however, will be in the form of *ETAC text instructions* rather than high-level code. Compressed ETAC text script files (2) are typically used as inclusion files. The advantage here is that the nested inclusion files specified in the original inclusion file are also compressed within the same packed file resulting in only one inclusion file. Files of the form (2) can be debugged but will have had the comments stripped out.

The **ETAC Compiler** program can also combine any number of *ETAC code* files (in any mixture of the three forms above) into a single *output file* (again, in any one of the three forms). When *ETAC code* files are combined, they are first internally converted to the output form and then merely concatenated. This requires that the contents of the original *input files* be designed for such concatenation.

The syntax checking done by the **ETAC Compiler** program is more detailed than the one done by the **Run ETAC Scripts** program, and includes warning checks. Only *ETAC code* files of the form (1) and (2) can be syntax checked. However, code files of the form (2) should have already been syntax checked. A list of errors and warnings for the code file (*input file*) being syntax checked is displayed in a dialog box. When the user selects an error or warning, the offending token is highlighted in another window which displays the whole (slightly converted) *input file* containing that token. The list of errors and warnings, along with their line numbers in the original *input file*, can be exported to an RTF (Rich Text Format) or TAB-separated text file for later viewing. The **ETAC Compiler** program can be directed to filter out warnings that have occurred with the same *input file* in the previous session. However, the **ETAC Compiler** program only remembers the error code, *input file*, and line number of a warning, and so that feature is unable to distinguish more than one warning of the same warning code on the same text line in the same *input file*. Nevertheless, this is still a useful feature, enabling the user to see only the latest warnings of an *input file*. Note that the said feature only filters warnings, not errors, since errors will cause the script to fail if run. Also note that the **ETAC Compiler** program checks at least those same errors as are checked by the **Run ETAC Scripts** program, but those errors are checked in more detail by the **ETAC Compiler** program.

The input instructions to operate the **ETAC Compiler** program are in the form of keyword-arguments (keywords and their arguments) that allow any number of groups of *ETAC code* files to be processed independently one after another. The keyword-arguments can be specified directly or in a text file.

2. Command Line Keywords

The **ETAC Compiler** program can be executed either from the **MS-DOS**[®] or **Windows**[®] environment. In either case, a command line consisting of keywords and their arguments needs to be specified.

There are two forms of the command line keywords.

Form 1

[ARGS_PATH=*arg-file-path*, ...] [{ED_ARGS} | NO_ED_ARGS] [START | ABOUT]

Form 2

[{BINARY} | PACK | EXPACK | INSTR | PPROC | (SYNTAX [F: *warn-filt-file*])] [CDIR=*dir-path*]
[INC_DIRS=*incl-dir-file*] [PPDEFS=*pp-defs*] SRC_FILES=*src-file-path*, ... [OUT_FILE=*out-file-path*]
[LIST_INCFILES] [QUIET_RUN] [{EXIT_MSG} | NO_EXIT_MSG] [{AUTO_LOG} | LOG_FILE=*log-file-path*]

The full details of the keywords and their arguments are specified in the file ETACCompiler.chm under **<Command Line Arguments → Command Syntax>**.

3. ETAC Compiler Operations

The **ETAC Compiler** program can perform six operations. The following sections describe the various operations and the corresponding types of *output file* (and their default extensions) produced by the **ETAC Compiler** program. Any of these *output files* can be used as an *input file* in another compilation session.

3.1 Conversion to Binary File

Keyword: BINARY **Extension:** btac

An *ETAC code* file can be converted to a compressed TAC binary file containing *TAC binary instructions*. The *output file* is not readable in a text editor.

3.2 Conversion to Text Instructions

Keyword: INSTR **Extension:** tac

An *ETAC code* file can be converted to a text file containing *TAC text instructions* in the form **<type:argument;>** where *type* is any one of: INT, DEC, STR, LBC, LBO, CMD, OPR, MRK, MEM, NUL, or EXE, and *argument* is an appropriate argument for *type*. The *output file* is readable in a text editor. The file can be executed by the **Run ETAC Scripts** program.

3.3 Producing Pre-processor Output

Keyword: PPROC **Extension:** etac

Produces a pre-processed file for diagnostics purposes. The *output file* may not be usable as *ETAC script* because pre-processor definition references are resolved later than the pre-processing stage. All pre-processor directives are processed. Comments are removed and white-spaces are reduced to a single space. The *output file* is readable in a text editor. If the *input file* is a TAC binary file, then the *output file* will contain *TAC text instructions*, and the default file extension will be tac.

3.4 Conversion to Pre-processed Packed File

Keyword: PACK **Extension:** ptac

Produces a pre-processed packed script (*ETAC packed script*) file. All pre-processor directives are processed and also all pre-processor definitions are moved to the beginning of the *output file*. Comments are removed and white-spaces are reduced to a single space. The *output file* is not readable

in a text editor. If the *input file* is a TAC binary file, then the *output file* will contain packed *TAC text instructions*. The file can be executed by the **Run ETAC Scripts** program, but this is not typically done.

3.5 Conversion to Unprocessed Packed File

Keyword: EXPACK	Extension: ptac
------------------------	------------------------

Produces a packed script (*ETAC packed script*) file with all ‘: :include’ instructions unconditionally expanded (thus the ‘EX’ in ‘EXPACK’). The *input files* are not pre-processed. Comments are removed and white-spaces are reduced to a single space. The *output file* is not readable in a text editor. If the *input file* is a TAC binary file, then the *output file* will contain packed *TAC text instructions*. The file can be executed by the **Run ETAC Scripts** program, but this is not typically done.

3.6 Syntax Checking Only

Keyword: SYNTAX	Extension: N/A
------------------------	-----------------------

Checks the *input files* for syntax errors and warnings only. The *input files* are pre-processed before they are checked. Shows warnings and fatal errors. Fatal errors are the errors that are checked for each time the script is run. Warnings are produced only with this option. The token causing the selected error or warning is highlighted in a window. If the *input file* is a TAC binary file, then no action will occur. No *output file* is produced.

NOTE: Some errors in an *ETAC script* file are detected when the file is executed. The **ETAC Compiler** program does not execute *ETAC code* files, so those errors will not be detected by the **ETAC Compiler** program but will be detected by the **Run ETAC scripts** program.

4. The Syntax Error and Warning Window

This window appears for the syntax checking operation (SYNTAX) only. During the syntax checking process, whenever syntax errors or warnings occur, a dialog box will appear showing the path of the source file that was syntax checked. Underneath that path, a list of the errors and warnings is shown in a grid. The programmer can select the current error (E) or warning (W) to display in the source viewer, or use the **First**, **Previous**, **Next**, or **Last** buttons to move the selection to the desired position. The check box at the beginning of each error or warning line is for the programmer to use as desired. Typically, the check boxes are used to tick errors or warnings that have been fixed by the programmer. A summary of the selected error or warning is displayed in the lower text box. A pink background indicates an error, and a blue background indicates a warning. More information about that error or warning can be displayed by clicking the **Info** button. The error and warning list can be exported to an RTF (rich text format) file or a TAB-separated text file along with the error messages (for RTF only) by clicking the **Export** button.

The diagram below illustrates the syntax and error warning window.

Glossary

A

activate

- When referring to a *script token* that creates a *stack object*, the *script token* is converted to a *stack object* by the *TAC processor* and then the object's *nominal action* is performed.
- When referring to a *script token* that does not create a *stack object*, an appropriate action is performed depending on the type of *script token*.
- When referring to a *stack object*, the *stack object* is temporarily *copied* by the *TAC processor* and then the *copied* object's *current action* is performed.

B

binary interpreter

Part of an *ETAC interpreter* that processes *TAC binary instructions*.

C

command

A *script token* having the syntax of a *comop identifier*. A *command* can be in *script form* (eg: `<FilePath>`, `<tac.var>`, `<#abc%03?>`, `<sub:>`, `<.xyz-3>`) or *instruction form* (eg: `<CMD:FilePath>`, `<CMD:tac.var>`, `<CMD:#abc%03?>`, `<CMD:sub:>`, `<CMD:.xyz-3>`).

comop

A *command* or *operator* (**com**mand **op**erator), or a *stack object* created by such a *command* or *operator*.

comop identifier

A consecutive sequence of displayable characters with the following restrictions. The sequence must **not**:

- begin with a digit or colon character,
- begin with an uppercase character and have a colon in fourth character position (eg: `<Abc:d>` is invalid),
- be in the form of an integer or decimal number (eg: `<23>`, `<+23>`, `<2.3>`, `<-2.3>`, `<+2.3e5>`, `<.3E+2>`, `<0.3>` are invalid),
- be `<+>`, `<->`, `<*>`, `</>`, `<^>`, `<=>`, `<!=>`, `<<>`, `<>>`, `<<=>`, `<>=>`, `<+>`, `<?>`,
- contain whitespaces or the characters `<'>`, `<">`, `<,>`, `<:>`, `<[>`, `<]>`, `<{>`, `<}>`, `<(>`, `<)>`.

A *comop identifier* may contain displayable characters above character code 127 (7F_H), but this is not recommended. *Comop identifiers* are case-sensitive.

Examples of *comop identifiers*: `<FilePath>`, `<tac.var>`, `<#abc%03?>`, `<sub:>`, `<.xyz-3>`.

copy (of a *stack object*)

To reproduce a *stack object* and its *embedded value* into another *stack object* (replacing that other *stack object*) such that the reproduced value and the original value are identical. The *embedded value* of a *stack object* that has a *resource value* is an internal reference to that *resource value*. Therefore, if such a *stack object* is *copied*, only its reference is reproduced not its *resource value*. Consequently, if a *stack object* that has a *resource value* is *copied* to another *stack object*, both objects will share the same *resource value*.

current action

A property of a *stack object* that indicates its current action when *activated*.

D

dictionary

A *stack object* having a *resource value* consisting of a list of internally indexed *dictionary items*. The *dictionary item* having the highest index value in its *dictionary* is called the ‘topmost’ *dictionary item*.

dictionary item

An item in a *dictionary* consisting of a label having the syntax of a *comop identifier* and a *stack object*. A *dictionary item* need not be unique to any *dictionary*; a *dictionary* can contain more than one identical *dictionary item*, and any other *dictionary* can contain the same identical item. A *dictionary item* within a *dictionary* is uniquely identified by an internal index. When a *dictionary item* is added to a *dictionary*, the item gets the next index value in the *dictionary*. The *dictionary item* having the highest index value in its *dictionary* is called the ‘topmost’ *dictionary item*.

dictionary stack

One of the three stacks in the *ETAC interpreter* that can contain only *dictionaries*.

E

embedded value (of a *stack object*)

The value of a *stack object* that is exclusively associated with that object (eg: integer, decimal, and string *stack objects* have *embedded values*). An *embedded value* is not shared with other *stack objects*, and can therefore be changed independently of the value of those other objects.

ETAC code

This is *ETAC script* or *TAC binary instructions*. A file containing *ETAC code* typically has an extension of *etac*, *tac*, *ptac*, or *btac*.

ETAC expression

A consecutive sequence of one or more *script tokens* as defined for *ETAC expression* in the document *ETACProgLang(Official).pdf*.

ETAC interpreter

A computer program that processes *ETAC code*. An *ETAC interpreter* essentially consists of a *script interpreter*, a *binary interpreter*, and a *TAC processor*.

ETAC packed script

ETAC text script that has been pre-processed or expanded, and compressed. A file containing *ETAC packed script* is a binary file, typically having an extension of *ptac*.

Note that the term “*ETAC packed script*” is used in the same sense as the word “code”, as in “*ETAC packed script code*”.

ETAC script

This is *ETAC text script* or *ETAC packed script*. A file containing *ETAC script* typically has an extension of *etac*, *tac*, or *ptac*.

Note that the term “*ETAC script*” is used in the same sense as the word “code”, as in “*ETAC script code*”.

ETAC statement

A consecutive sequence of one or more *script tokens* as defined in the document *ETACProgLang(Official).pdf* for *ETAC statement*.

ETAC text script

ETAC program code that is in human readable and writable text form. This includes *TAC text instructions*. *TAC text script* containing *comops* in the form of *variable identifiers* is also *ETAC text script*. A file containing *ETAC text script* typically has an extension of `etac` (or `<tac>` if the file contains only *TAC text script*).

Note that the term “*ETAC text script*” is used in the same sense as the word “code”, as in “*ETAC text script code*”.

I

input file

An *ETAC code* file to be processed by the **ETAC Compiler** program.

instruction form (of a *script token*)

A *script token* in the form of a *TAC text instruction*.

L

lexical analyser

Part of the *script interpreter* that converts *lexical tokens* to *logical tokens* which are then syntax checked, modified, and rearranged as necessary.

lexical parser

Part of the *script interpreter* that parses *ETAC script* into *lexical tokens*.

lexical token

The smallest unit of information, in the form of text characters, that can be identified by the *lexical parser*.

logical token

A combination of one or more *lexical tokens* and internal tokens regarded as a conceptual unit by the *lexical analyser* for the purpose of syntax checking and compiling a programming language.

N

nominal action (of a *TAC object*)

The default action of a *TAC object*.

O

object stack

One of the three stacks in the *ETAC interpreter* that can contain any type of *TAC object*. This is the main stack used by *ETAC code*.

operator

A *script token* containing the syntax of a *comop identifier*. An operator could be in *script form* qualified by a preceding `<&>` (eg: `<&AddVect>`, `<&tac.var>`, `<&#abc%03?>`, `<&add:>`, `<&.xyz-3>`) or *instruction form* (eg: `<OPR:AddVect>`, `<OPR:tac.var>`, `<OPR:#abc%03?>`, `<OPR:add:>`, `<OPR:.xyz-3>`). An operator is used in an *operator expression*.

operator expression

A consecutive sequence of *script tokens* involving an *operator* and its operands. There are two forms of *operator expressions*. The script form, where the operands are delimited by parentheses, and the instruction form, where the operands are delimited by the `start_op` and `end_op` *commands*. The *operator* of an *operator expression* can exist anywhere within its operand's delimiters.

Typically, when an *operator expression* is *activated*, its operands get *activated* first leaving the *operator* arguments on the *object stack*, then the *operator* gets *activated* and processes those arguments, returning the resultant *stack object* on the *object stack*. For example, the *operator expression* `<(3 + 4 5)>` will return 12 on the *object stack*. That *operator expression* can be written as: `<(+ 3 4 5)>`, `<(3 4 5 +)>`, `<(3 4 + 5)>`, `<end_op 3 4 &add 5 start_op>`, `<start_op; 5; 4; &add; 3; end_op;>`. Note that the *operator expressions* in all but the last example are *activated* from right to left; the *operator expression* of the last example is *activated* from left to right.

An *operator expression* can contain nested *operator expressions* as some or all of its operands, but each *operator expression* must contain exactly one *operator* at the top level.

operator stack

One of the three stacks in the *ETAC interpreter* that can contain only operator *stack objects*.

output file

The *ETAC code* file that is output by the **ETAC Compiler** program.

R

resource value

The value of a *stack object* that can be shared with other *stack objects* of the same type — sequence, procedure, dictionary, and memory *stack objects* have sharable *resource values*. A *resource value* is internally referenced by the *stack object*; that reference itself is the object's *embedded value* (the reference itself is not available to the programmer, only the value being referenced, the *resource value*, is available).

S

script form (of a *script token*)

A *script token* not written in the form of a *TAC text instruction*. This is a more natural and intuitive style of expressing *script tokens*.

script interpreter

The part of the *ETAC interpreter* that processes an *ETAC script*. The *script interpreter* consists of a *lexical parser*, a *script pre-processor*, and a *lexical analyser*.

script pre-processor

The script pre-processor is that part of the *script interpreter* that is responsible for pre-processing *ETAC text script*.

script token

A consecutive sequence of one or more *lexical tokens* regarded as a unit for the purpose of defining the syntax and semantics of the ETAC programming language.

stack object

Any one of a number of certain groups of *TAC objects*.

T

TAC binary instruction

A binary form of a *TAC text instruction*. *TAC binary instructions* exist in binary files. Any *ETAC code* can be compiled into *TAC binary instructions* by the **ETAC Compiler** program. A file containing *TAC binary instructions* typically has an extension of `btac`.

TAC object

An entity that has the capability of existing on a *TAC stack*, and consists of a type and corresponding value along with an indicator of some suitable action to perform.

TAC processor

Part of the *ETAC interpreter* that creates a *TAC object* from each *logical token* passed to it then *activates* the *TAC object* according to its type.

TAC stack

An *object stack*, *dictionary stack*, or *operator stack*.

TAC text instruction

A human readable text instruction of the form `<type:argument>` where *type* is any one of: INT, DEC, STR, LBC, LBO, CMD, OPR, MRK, MEM, NUL, or EXE, and *argument* is an appropriate argument for *type*. *TAC text instructions* may exist in *ETAC text script* files or in files containing only *TAC text instructions*. The **ETAC Compiler** program can compile *ETAC code* to *TAC text instructions*. A file containing *TAC text instructions* alone typically has an extension of `tac`.

TAC text script

TAC program code that is in human readable and writable text form. This includes *TAC text instructions*. *TAC text script* does not contain ETAC program code (*ETAC expressions* or *ETAC statements* other than assignment or allocation statements). A file containing *TAC text script* typically has an extension of `tac`.

Note that the term “*TAC text script*” is used in the same sense as the word “code”, as in “*TAC text script* code”.

V

variable identifier

A consecutive sequence of characters beginning with an alphabetic character (‘a’ to ‘z’ or ‘A’ to ‘Z’ or exotic Latin characters such as ‘Ä’), an underscore (`_`), or an ‘at’ character (`@`). The subsequent characters are alphanumeric (alphabetic or ‘0’ to ‘9’) or underscore. Note that, by convention, *variable identifiers* beginning with an ‘at’ character, or an underscore followed by an alphabetic character or underscore, are reserved for system use. An ETAC programmer, therefore, is limited to defining *variable identifiers* containing alphanumeric characters and underscores, with the first character being an alphabetic character, or the first two characters being an underscore followed by a digit character. In addition, none of the strings “if”, “then”, “else”, “endif”, “when”, “is”, “endwhen”, “do”, “repeat”, “from”, “to”, “step”, “with”, “of”, “while”, “exitdo”, “exitdo_if”, “donext”, “donext_if”, and “void” can be a *variable identifier*. *Variable identifiers* are case-sensitive.

The exotic Latin characters are: ^a, ², ³, μ , ¹, ^o, À, Á, Â, Ã, Ä, Å, Æ, Ç, È, É, Ê, Ë, Ì, Í, Î, Ï, Ð, Ñ, Ò, Ó, Ô, Õ, Ö, Ø, Ù, Ú, Û, Ü, Ý, Þ, ß, à, á, â, ã, ä, å, æ, ç, è, é, ê, ë, ì, í, î, ï, ð, ñ, ò, ó, ô, õ, ö, ø, ù, ú, û, ü, ý, þ, ÿ. Those characters should be used only if necessary.