ETAC: Interacting with C++

TM

ETAC Interface Version: **1.2** (beta)

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
ETACCompiler.pdf	The ETAC Compiler
ETACCompiler.chm	ETAC Compiler Help
ETACErrorCodes.pdf	ETAC Compilation and Run-time Error Codes
FunctionsETACScriptLib.pdf	Functions ETAC Script Library

Legal Information

ETAC: Interacting with C++

Victor Vella

Published by Victor Vella
(1 August 2020)

First Published: 28 April 2018
Revised: 1 February 2019
Second Edition: 1 August 2020

Preface

This document is part of the main ETAC document (ETACProgLang(Official).pdf) for describing how to write C++ code that communicates with code written in ETAC™ (pronounced: E-tack). Since ETAC is released only on the **Windows**® operating system, I am assuming that the reader is confident in writing **Windows**® programs in C++. I have chosen C++ for the communication code rather than C because ETAC itself is written in C++, and also because C++ offers certain advantages over C. However, there is one drawback in the particular way that C++ class instances are used for communicating with ETAC. Because of the enormous advantages, I decided to use the C++ class virtual table system for implementing the communication from C++ code into the *ETAC interpreter*. This requires that any C++ compiler used must be compatible with the one used to compile ETAC itself (Microsoft® Visual C++® compiler version 7.1). Fortunately, at the time of this writing, Microsoft® provided a compatible compiler free for use (subject to terms and conditions).

This is the second beta release of the intercommunication system between C++ and ETAC, so naturally some adjustments will probably need to be made in the first few releases. This document describes the first Unicode® release of the communication code mentioned above with basic support for the full Unicode codespace. Normally, the ETAC interface version of the communication code would be increased, but for practical reasons I decided to retain the same interface version number as before (version 1) — the previous version 1 is now deemed defunct. This version 1 of the ETAC interface is different to, and not compatible with, the previous version 1. I have introduced some new functions, and modified some previous ones, in this new interface.

Victor Vella

Perth, Western Australia
1 August 2020

Contents

Tables and Diagrams

Document Conventions

The following conventions are used in this document.

Document Conventions

Symbol	Meaning
‹x›	separates x as a unit of information from the surrounding text.
…	ellipsis represents omitted text (as usual).
U+x	represents a Unicode code point where x is in hexadecimal notation.
$X \cdots X_{\mathrm{H}}$	$X \cdots X$ represents a number in hexadecimal notation (X is a hexadecimal digit).
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 redefines version **1** of the **ETAC™ Interface**, which is compatible with the **ETAC Programming Language** version **1-1** implemented in programs RunETAC.exe and AppETAC.dll version **3-0-6-ena**.

(Australian English)

Important Note

Version **1** of the **ETAC Interface** has now been redefined for compatibility with Unicode®. The previous version **1** of the **ETAC Interface** (non-Unicode) is now defunct and incompatible with RunETAC.exe and AppETAC.dll version **3-0-6-ena** (Unicode). Programs written with the previous version need to be recompiled for Unicode (and perhaps modified) for use with the redefined **ETAC Interface** version **1**.

ETAC™ (pronounced: E-tack) is a syntactically simple but extremely versatile dictionary and stack based interpreted script programming language. The details of that language are described in the document titled "The Official ETAC Programming Language" (ETACProgLang(Official).pdf) which is a prerequisite for understanding this document. In addition, the reader needs to be familiar with **Windows**® programming in the C++ language since ETAC is released only on the **Windows**® operating system using the x86 architecture (can also run on the x64 architecture).

This document is for C++ programmers who want to implement new *comops* in C++ and\or who want existing or new C++ application programs or dynamic linked libraries (DLLs) to interact in both directions with the *ETAC interpreter*.

A computer programmer familiar with the C++ programming language can extend the native set of *comops* via *external TAC libraries* (implemented as dynamic linked libraries) by creating library functions (*ETL functions*) written in the C++ programming language that are *activated* by the said *comops*. This allows an ETAC programmer to be able to create *TAC objects* with any desired data, based on the values of other *TAC objects*. It also allows an ETAC programmer to be able to manipulate and access data in the operating system via the *external TAC library*.

ETAC is also designed with the capability of controlling the internals of an application program originally written in the C++ programming language. The application program needs to be constructed to incorporate that capability. For example, when linked to an application program via a special ETAC dynamic linked library (AppETAC.dll), the *ETAC interpreter* can communicate with that program via *comops* to create data structures and carry out functions in that program. Those *comops*, which exist in *ETAC code*, can execute subroutines within the application program. The advantage of this is that an *ETAC text script* file can be created, by a user, to use the functionality existing within an application program. A typical example of where such a system can be used is the creation of macro-like instructions, written in *ETAC text script*, by a user to control a word processor or text editor designed for that purpose. Thus the ETAC programming language can be used as a macro language for suitably designed application programs. An application program can also be designed to execute ETAC instructions via C++ interfaces, that includes executing *ETAC code* files.

ETAC 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 supports only strict conformance to the UTF-8, UTF-16, and UTF-32 encoding schemes; unpaired surrogate code points are not supported (character strings must be well-formed Unicode strings). 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

strAppend, **strUCharCount**, **strWCharCount**, **mbApplyBOM**, **mbCvtDataTo**, **mbGetErrCode**, **mbRepDataForm**, **mbRepDstPath**, **mbRepSrcPath**.

Modified Items

strAppend, **strAssign**, **strDeleteStr**, **strGetChar**, **strInsertStr**, **strPutChar**, **mbExport**, **mbImport**, **etacSetDebug**.

Enhanced Items

strStrip, **mbReadWholeFile**, **mbWriteWholeFile**.

1

The Principles of ETAC and C++ Interaction

This chapter presents an overview of the principles of the interaction between programmer-defined C++ code and *ETAC code*. The purpose of this chapter is to present the concepts involved in the interaction, not the actual methods or techniques. The diagrams in this chapter do not necessarily represent the actual implementation of the intercommunication between ETAC and C++.

1.1 The ETAC Interface

Apart from a few management functions, the *ETAC interface* is the only means by which C++ code can initially interact with the *ETAC interpreter*. The *ETAC interface* is defined by a C++ class called `ccTAC`, containing only virtual member functions which point directly into the *ETAC interpreter* which exists in AppETAC.dll. An instance of the *ETAC interface* is created in the *ETAC interpreter*, and passed to C++ code either as a pointer argument of a C++ function, or returned as a pointer from AppETAC.dll via the function `etacSetAppETAC()` (pre-existing in AppETAC.dll). C++ functions that receive a pointer to an *ETAC interface* exist either in an *external TAC library* (such functions are called "*ETL functions*") or in a programmer designed main application program (there is only one such function called the "*call-back function*"). *ETL functions* or the *call-back function* can be called either from *ETAC code*, or from C++ code; in both cases the functions access the *ETAC interpreter* through the pointer to the *ETAC interface*. The *ETAC interface* includes the ability to push and pull the value of the topmost *stack object* on the *object stack* from or into C++ variables.

Some *stack objects* have a *resource value*. The data in the *resource value* is maintained by the *ETAC interpreter*, but can be accessed via an appropriate *resource interface*. An instance of a *resource interface* is created in the *ETAC interpreter*, and passed to C++ code as a C++ pointer via some member functions of the *ETAC interface* or other *resource interfaces*. A *resource interface* is defined by a C++ class containing only virtual member functions which point directly into the *ETAC interpreter*. A *resource interface* also contains an internal reference (not available to the C++ programmer) to a *resource value*. Those member functions internally access the *resource value* via the internal reference (called a "*managed reference*"). A *resource interface* for a string *stack object* does not contain a *managed reference*, but contains the actual string itself.

Sequence, procedure, dictionary, and memory *stack objects* have *resource values*, and therefore corresponding *resource interfaces*. A string *stack object* also has a *resource interface* but no *resource value* as such (although a string value requires memory that is managed by the *ETAC interpreter*, which is the reason that it needs a *resource interface*).

The C++ class names of the *resource interfaces* are: `ccStackObj` (for a *TAC object*), `ccString` (for string, *command*, and *operator stack objects*), `ccSequence` (for *sequence* and *procedure stack objects*), `ccMemoryBlock` (for a memory *stack object*), and `ccDictionary` (for a *dictionary stack object*). The value of other types of *stack objects* (which do not have corresponding *resource interfaces*) are contained in C++ variables of the following types: `ccINT` (for integer, *intrinsic command*, *intrinsic operator*, mark, null, and `EXE` *stack objects*) and `ccDEC` (for a decimal *stack object*).

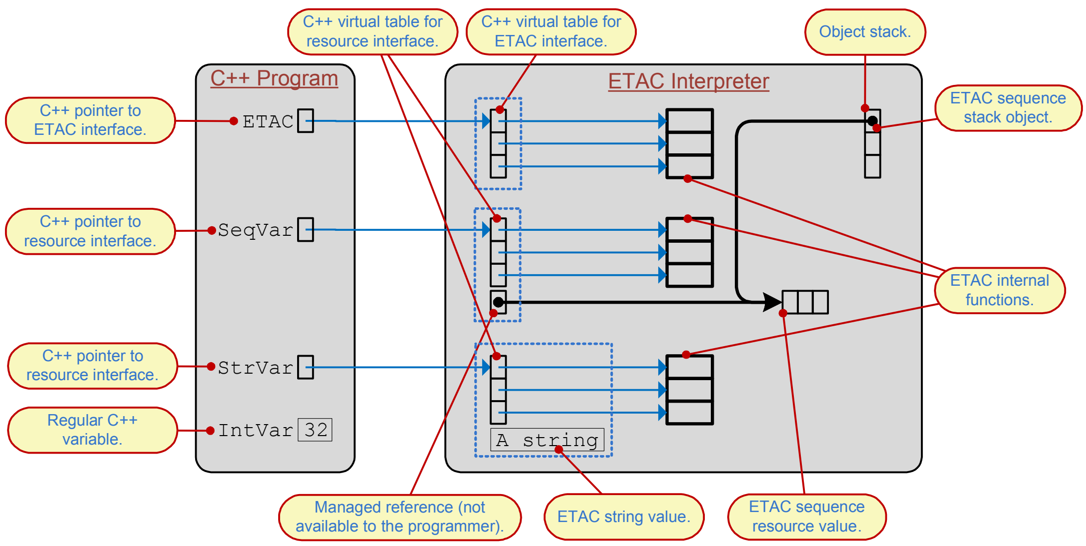
A *managed reference* is an internal reference to a *resource value* using a reference counting system. The system does not detect circular references — *resource values* directly or indirectly containing references to themselves are not internally deleted until AppETAC.dll is unloaded. Circular references should therefore not be used unless the circular reference loop is broken before the *resource values* are destroyed. (It is intended that some future version of AppETAC.dll automatically detect circular references and release the relevant *resource values* if required.)

For convenience, a *resource interface* can be conceptualised by the C++ programmer as containing an internal *compound stack object* rather than a *managed reference* — the effect is the same since a *compound stack object* contains a *resource value* via a *managed reference*. The exception is a *resource interface* for an ETAC string which does not contain a *managed reference* but the actual string itself (which can be conceptualised as a string *stack object*). The said conceptual *stack object* can be called a "*resource object*".

Various operations can be performed on a *resource object* via its *resource interface*. For example, a *resource object* can be *copied* and *duplicated*, and its data can be modified, extracted, and new data can be inserted, appended, and deleted.

The diagram below illustrates the implementation of the structure of an *ETAC interface* and *resource interface*. A thick black arrow represents a *managed reference*, and a thin blue arrow represents a C++ pointer. The C++ program could be an *external TAC library* (a DLL) or an application program. The ETAC *sequence resource value* has two *managed references* pointing to it. The *resource value* is destroyed only when all *managed references* pointing to it have been released.

The ETAC and Resource Interfaces



In the illustration above, a C++ variable ("ETAC") points to (an instance of) an *ETAC interface*, from which pointers to other *resource interfaces* ("SeqVar" and "StrVar") can be obtained. Integer ("IntVar") and decimal variables do not point to a *resource interface* because they do not require allocated resources (which are maintained by the *ETAC interpreter*). String variables ("StrVar") point to a *resource interface* containing its own individual string (which is maintained by the *ETAC interpreter*) rather than a *managed reference*. A variable for a *stack object* also points to a *resource interface* containing its own individual *stack object* rather than a *managed reference*. However, the *stack object* itself may contain a *resource value*.

Notice that in the diagram, the *resource interface* of SeqVar contains a *managed reference* to the *sequence* (*resource value*) that is also referenced by a *stack object* (only for illustration). If SeqVar were *copied* to another variable, then a new *resource interface* would be created with a *managed reference* pointing to the <u>same</u> *sequence*. If SeqVar were *duplicated* to another variable, then the new *resource interface* would contain a *managed reference* to a *duplicate* of the *sequence*. This behaviour is the same as for a *stack object* under the same circumstances. Therefore, all *resource interfaces* act as though they each directly contain the value of a *stack object*.

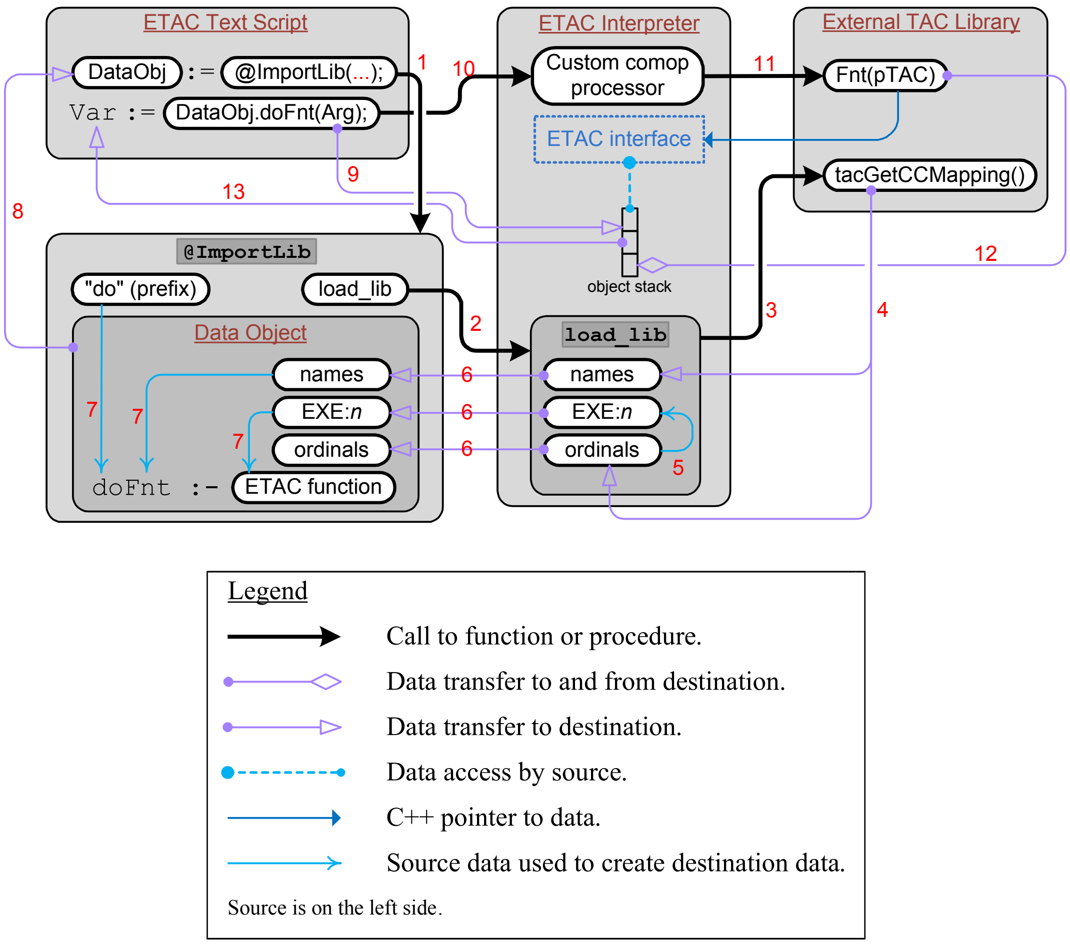
Since the *ETAC interface* and *resource interfaces* use the virtual table produced by the C++ compiler, such interfaces may fail to operate if a different C++ compiler produces a virtual table not compatible with the one with which the *ETAC interpreter* was compiled. See Appendix A: Compatibility Issues for more details.

1.2 ETAC Code and External TAC Library Interaction

To call an *ETL function* from *ETAC code*, the *external TAC library* containing the *ETL function* must first be loaded, typically via the **@ImportLib** *command* or the **load_lib** *command* for more refined processing. Those two *commands* return the capability of calling the *ETL functions*.

The following diagram illustrates the essentials of the process of importing an *external TAC library* and calling an *ETL function* (Fnt()) within that library. The red numbers indicate the order of the events. Error checking code, variable allocations, and other irrelevant items are not shown in the illustration.

Illustration of ETAC Code and ETL Interaction



1. **@ImportLib** is called from *ETAC code* (1).
2. **@ImportLib** then calls **load_lib** which obtains a handle to the *external TAC library* (2).
3. **load_lib** then calls `tacGetCCMapping()` defined in the *external TAC library* (3).
4. `tacGetCCMapping()` returns a string list of the requested *ETL function* names and also a corresponding list of the ordinal numbers of the *ETL functions* (4). The version number of the *ETAC interface* with which the *external TAC library* was compiled is also returned. An *ETL function* name is a string name in the form of a *variable identifier* that represents the *ETL function*. *ETL function* names are created by the *external TAC library* designer.
5. **load_lib** returns three corresponding *sequences* to **@ImportLib** (6): **(1)** a *sequence* of the requested *ETL function* names ("names"), **(2)** a *sequence* containing EXE *comops* constructed

from the returned ordinal numbers in such a way that they are not confused with the same ordinal numbers in a different *external TAC library* ("EXE:*n*") (5), and **(3)** a *sequence* of the returned ordinal numbers ("ordinals").

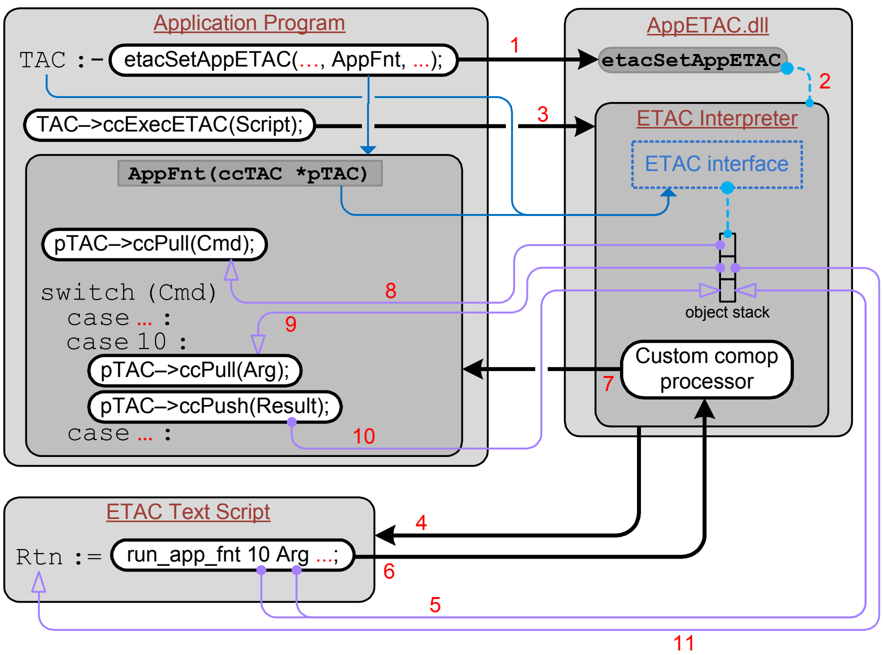
6. **`@ImportLib`** creates a *data object* and privately stores the three *sequences* in it, but also creates *member variables* in the form of *ETAC functions* ("ETAC function") (or in a form that act as *operators*) from the *ETL function* names, (1), and the `EXE` *comops*, (2). The *member variables* will have names ("doFnt") beginning with the requested prefix passed to **`@ImportLib`** (7). The created *data object* is also initialised with other data before being returned ("DataObj") to the caller of **`@ImportLib`** (8).
7. When the *ETAC code activates* a *member variable* in the *data object*, the embedded `EXE` *comop* gets *activated* (10) after being passed any required arguments ("Arg") to the *object stack* (9).
8. The *ETAC interpreter* obtains the ordinal number from the *custom comop number* of the `EXE` *comop* to call the corresponding *ETL function* ("Fnt"), passing a pointer ("pTAC") to the appropriate version of the *ETAC interface* to that *ETL function* (11).
9. The *ETL function* retrieves the *member variable's stack object* arguments via the passed *ETAC interface* (12), and can push *stack objects* onto the *object stack* (via the *ETAC interface*) to be returned ("Var") to the ETAC caller of the *member variable* (13).

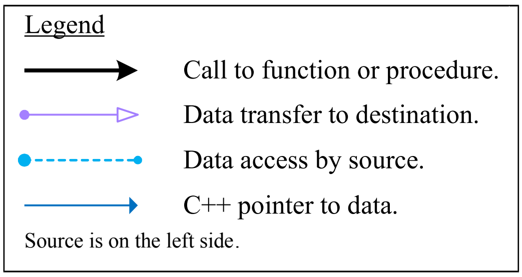
1.3 ETAC Code and Application Program Interaction

To use the *ETAC interpreter* from an application program, the application program must first load AppETAC.dll (this is done automatically if the import library AppETAC.lib is linked into the application program). The application program then calls `etacSetAppETAC()` to set up the *ETAC interpreter* for use. The application program interacts with the *ETAC interpreter* via the returned *ETAC interface*.

The following diagram illustrates the essentials of the process of starting the *ETAC interpreter* from within an application program, and setting the *call-back function* for use by *ETAC code*. The red numbers indicate the order of the events. Error checking code, variable allocations, and other irrelevant items are not shown in the illustration.

<u>Illustration of ETAC Code and Application Program Interaction</u>





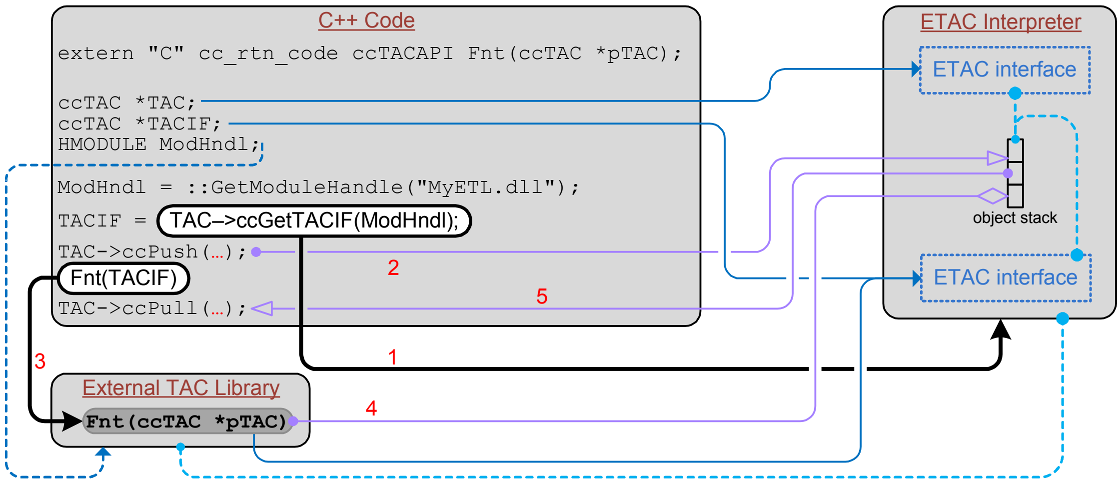
1. The application program calls `etacSetAppETAC()`, passing a pointer to the (optional) *call-back function* ("AppFnt") (1) which is stored in the *ETAC interpreter* for later use.
2. `etacSetAppETAC()` initialises the *ETAC interpreter* for use (2), returning a pointer to the *ETAC interface*.
3. `ccExecETAC()` is called with an argument ("Script") to execute *ETAC code* (3).
4. The *ETAC interpreter activates* the requested *ETAC code* (4).
5. The *ETAC code activates* **run_app_fnt** (which is processed by the *ETAC interpreter*) (6), optionally pushing a command number ("10") and corresponding arguments ("Arg") onto the *object stack* (5). The command number is only a convention so that the *call-back function* can distinguish among different operations. If a command number is used, **run_app_fnt** and its command number are typically both enclosed in a *procedure*, and that *procedure* is assigned to an *ETAC variable* to be used as a *command* (not depicted in this illustration).
6. The *ETAC interpreter* calls the *call-back function* ("AppFnt") with a pointer to the *ETAC interface* ("pTAC") (7).
7. The *call-back function* pulls the command number (into "Cmd") off the *object stack* (8), and uses it to execute the appropriate code via the `switch` statement.
8. The appropriate part of the `switch` statement pulls the rest of the arguments ("Arg") off the *object stack* (9) and performs the desired action (possibly interacting with the *ETAC interpreter* via the *ETAC interface*), optionally pushing the return values ("Result") onto the *object stack* (10).
9. The returned values are assigned to the appropriate *ETAC variables* ("Rtn") or processed in some other way (11).

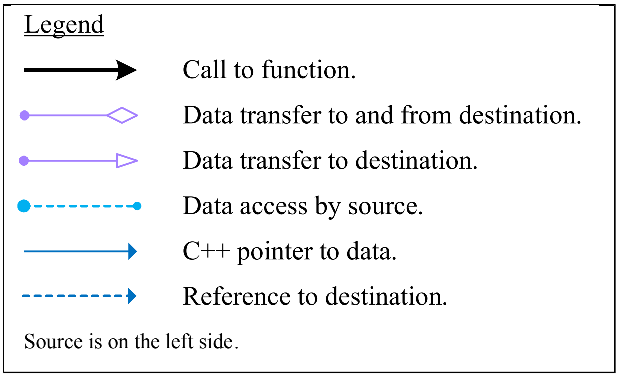
1.4 C++ Code and External TAC Library Interaction

ETL functions can be called from the main application program, the *call-back function* in an application program, or other *ETL functions* existing in a different *external TAC library*. Of course, calling an *ETL function* existing in the same *external TAC library* involves only a C++ function call. Calling an *ETL function* from an application program requires that the *ETAC interpreter* has been initialised beforehand, and that the *external TAC library* has been loaded.

The following diagram illustrates the essentials of the process of C++ code interacting with an *external TAC library*. The diagram assumes that the C++ code calling the *ETL functions* has been linked to the import library of the *external TAC library*. Also, AppETAC.dll is assumed to have been loaded and initialised if the C++ code exists in an application program. The name of the *ETL function* being called in this illustration is "Fnt" existing in an *external TAC library* called "MyETL.dll". Error checking code and other irrelevant items are not shown in the illustration.

Illustration of C++ Code and ETL Interaction





1. Before C++ code can call *ETL functions*, a pointer to the *ETAC interface* (“TAC”) must first be obtained. That pointer can either be returned via the `etacSetAppETAC()` function defined in AppETAC.dll, or automatically passed as a C++ argument to the C++ function that will call the *ETL functions*.
2. The prototypes of the *ETL functions* need to be present (“`extern "C" cc_rtn_code Fnt(ccTAC *pTAC);`”).
3. A handle (“ModHndl”) to the *external TAC library* (“MyETL.dll”) is obtained via `GetModuleHandle()` (if the library has already been loaded) or `LoadLibrary()` (if the library has not yet been loaded). If `LoadLibrary()` is called then `FreeLibrary()` must be called when the *external TAC library* is no longer to be accessed.
4. The *ETAC interface* (“TACIF”) for the *external TAC library* is obtained for passing to the *ETL functions* in that library (1). `ccGetTACIF()` internally gets the *ETAC interface* version number from `tacGetCCMapping()` which is defined in the *external TAC library*. `ccGetTACIF()` uses that number to determine the corresponding *ETAC interface*, which is returned to the caller. Note that the *ETAC interface* (“TACIF”) for the *external TAC library* may be of a different version than the originally obtained *ETAC interface* (“TAC”).
5. Required arguments for the *ETL function* are pushed onto the *object stack* (2).
6. The *ETL function* (“Fnt”) is called with the appropriate *ETAC interface* (“TACIF”) (3). The *ETL function* receives a pointer to that *ETAC interface* in the parameter “pTAC”.
7. The *ETL function* pulls *stack object* arguments, and pushes returned *stack objects* to the *object stack* (4).
8. Any return values from the call to the *ETL function* are pulled off the *object stack* (5).

2

Programming Guide

This chapter is intended for C++ programmers who want to create C++ code to interact with the *ETAC interpreter*. It is assumed that the C++ programmer is familiar with **Windows**® programming, and in particular, how to create a **Windows**® 32-bit DLL (required for creating an *external TAC library*). Secondly, it is assumed that the C++ programmer is familiar with the document entitled "The Official ETAC Programming Language" (ETACProgLang(Official).pdf).

Note that ETAC is released only for the **Windows**® 32-bit platform (however, ETAC can run on the **Windows**® 64-bit platform) beginning with **Windows**® **XP**. Also note that ETAC is not multi-threading and must not be regarded as such if multi-threading is used with an *external TAC library*.

2.1 C++ Compiler Requirements

The *ETAC interpreter* was compiled with MSVC 7.1 (Microsoft® Visual C++® compiler version 7.1). A C++ programmer wanting to write C++ code to interact with the *ETAC interpreter* needs to compile their code with a 32-bit C++ compiler compatible with MSVC 7.1 mainly because the intercommunication between C++ and the *ETAC interpreter* uses the hidden virtual table of the virtual member functions of the *ETAC interface* and *resource interfaces* (see Appendix A: Compatibility Issues for more information). In addition, all function calls to and from AppETAC.dll and *external TAC libraries* use the native C calling convention known as ‹`__cdecl`› within MSVC. A compiler compatible with MSVC 7.1 needs to be able to generate code that is exactly compatible with the ‹`__cdecl`› calling convention. At the time of this writing, Microsoft® provided an MSVC compiler free for use (subject to terms and conditions). The *ETAC interpreter* is a 32-bit program that can interact only with 32-bit application programs, DLLs, and *external TAC libraries*.

2.2 Creating an External TAC Library

ETAC is distributed with the *standard TAC library* which is intended to be a minimal library of *commands* built into the *ETAC interpreter*. An *external TAC library* allows a C++ programmer to extend the set of *comops* using C++ code when the desired *comops* cannot be implemented in *ETAC text script*. Such a case arises, for example, when *ETAC code* is desired to perform more specialised functionality than is currently available, or when *ETAC code* is desired to interact with the operating system.

To create an *external TAC library*, a C++ programmer needs to create a DLL and implement the function `tacGetCCMapping()` which returns required information to the *ETAC interpreter*. The C++ programmer also needs to implement the desired *ETL functions*. The **`@ImportLib`** *command* (via the **`load_lib`** *command*) uses the information returned by `tacGetCCMapping()` to construct a *data object* containing *function members* that call the appropriate *ETL functions* with an *ETAC interface* to the *ETAC interpreter*. The *ETAC interface* is an instance of a C++ class containing only virtual member functions that are implemented in the *ETAC interpreter*. An *ETL function* receives the *ETAC interface* as the only C++ argument. Other arguments that an *ETL function* may need are passed as *TAC objects* on the *object stack*, which are retrieved via the *ETAC interface*. An *ETL function* can push returned *TAC objects* onto the *object stack* before completing.

2.2.1 Requirements for Creating an External TAC Library

The following requirements need to be satisfied when creating an *external TAC library*.

1. A mapping function (`tacGetCCMapping()`) needs to be implemented in the *external TAC library*. (**Ref**: 3.13.1_Mapping Function)
2. The function names of the desired *ETL functions* need to be established. Those names will be exported by the *external TAC library* to be called directly from external C++ code if required. The *ETAC interpreter* uses only the exported name "`tacGetCCMapping`".
3. A unique name satisfying the syntax of a *variable identifier* needs to be designated for each *ETL function*. The name is typically based on the name of the *ETL function*, and will be used as a base name for calling the *ETL function* from *ETAC code*. A name of an *ETL function* for use as an *operator* must be prefixed with an ampersand character (`&`).
4. A DLL ordinal number for each *ETL function* needs to be defined. The ordinal numbers will typically be in sequential order begin with `2` (ordinal number `1` will typically be the ordinal number of `tacGetCCMapping()`). The *ETAC interpreter* uses the ordinal numbers to call the *ETL functions*.
5. A module-definition file (`.def`) needs to be created for the *external TAC library*. The module-definition file will contain the names of the exported *ETL functions* along with their ordinal numbers. The module-definition file should be released along with the *external TAC library* if the library is to be release to the public.
6. An import library needs to be created from the module-definition file. The import library is not used by the *ETAC interpreter*, but can be linked to other C++ code to directly call the *ETL functions*. With MSVC, the import library can be specified to be created automatically from the module-definition file. The import library should be released along with the *external TAC library* if the library is to be release to the public.
7. The inclusion file ExternTACLib_***n***.h, where ***n*** which indicates the version of the *ETAC interface* and *resource interfaces* currently in use, needs to be included once in each source file of the *external TAC library* that interacts with the *ETAC interpreter*. That inclusion file resides in the path ‹…\Victella\AppETAC\Include›. For a new *external TAC library*, the inclusion file with the latest version, ***n***, can be used. If pre-processor operational definitions need to be made, they must be defined before the inclusion of ExternTACLib_***n***.h. (**Ref**: 3.3.8_Operational Definitions)
8. `tacGetCCMapping()` and the *ETL functions* are required to use the C programming language linkage conventions (‹`extern "C"`›).
9. The following assertions must not fail:
```
assert(sizeof(char) == 1);
assert(sizeof(bool) == 1);
assert(sizeof(short) == 2);
assert(sizeof(int) == 4);
assert(sizeof(long) == 4);
assert(sizeof(void *) == 4);
assert(sizeof(double) == 8);
assert(sizeof(ccCHAR) == sizeof(unsigned long));
```

The following code illustrates the essentials of the structure of an *external TAC library*. The code would be compiled into a DLL (ExternTACLib.dll). The function `tacGetCCMapping()` must be defined to process the class list and return the current version of the *ETAC interface*. `pReserved1`, `pReserved2`, and `pReserved3` are ignored. In this illustration, there are three *ETL functions*: Command1(), Command2(), and Operator(). The three *ETL functions* correspond, respectively, with the programmer-defined *comop* names "command_1", "command_2", and "operator", and with the *ETL function* ordinal numbers 2, 3, and 4, respectively. Note that the last *ETL function* is to be used as an *operator*, which is indicated by the ampersand (`&`) prefixed to the *comop* name ("&operator"). The `tacGetCCMapping()` function recognises three programmer-defined classes: "Command1", "Class1", and "Class2". Any of those class names are specified by the caller via **@ImportLib** or **load_lib**. The class names determine which *ETL functions* are recognised by the *ETAC interpreter*. "command_1" indicates Command1(), "Class1" indicates Command2() and Operator(), and "Class2" indicates Command1() and Operator().

```
#include <assert.h>
#include <string.h>

#include "ExternTACLib_1.h"

BOOL APIENTRY DllMain(...)
{
   ...
}

extern "C"
{
long ccTACAPI
   tacGetCCMapping(ccSTR *&pComopNameList, ccULONG *&pComopOrdList, ccSTR *pClasses,
                   void *pReserved1, void *pReserved2, void *pReserved3)
{
unsigned long           Idx;
unsigned long           LstIdx;

/* Comop name list - ends with NULL. */
static ccSTR            CNameLst[] = {L"command_1", L"command_2", L"&operator", NULL};

/* ETL function ordinal number list - ends with 0. */
static ccULONG          COrdLst[] = {2, 3, 4, 0};

/* Comop and ordinal lists to pass back to caller. */
static ccSTR            OutNameLst[4];
static ccULONG          OutOrdLst[4];

   /* Type size assertions. */
   assert(sizeof(char) == 1);
   assert(sizeof(bool) == 1);
   assert(sizeof(short) == 2);
   assert(sizeof(int) == 4);
   assert(sizeof(long) == 4);
   assert(sizeof(void *) == 4);
   assert(sizeof(double) == 8);
   assert(sizeof(ccCHAR) == sizeof(unsigned long));
```

continued on next page …

```
   if ( pClasses == NULL || pClasses[0] == NULL )
   {
      /* All ETL functions requested. */
      pComopNameList = CNameLst;
      pComopOrdList = COrdLst;
   }
   else
   {
      /* Construct the name and ordinal lists corresponding to the specified classes. */
      LstIdx = 0;
      for (Idx = 0; pClasses[Idx] != NULL; Idx++)
      {
         if ( wcscmp(pClasses[Idx], L"command_1") == 0 )
         {
            OutNameLst[LstIdx] = CNameLst[0];
            OutOrdLst[LstIdx] = COrdLst[0];
            LstIdx++;
         }
         else
         if ( wcscmp(pClasses[Idx], L"Class1") == 0 )
         {
            OutNameLst[LstIdx] = CNameLst[1];
            OutOrdLst[LstIdx] = COrdLst[1];
            LstIdx++;
            OutNameLst[LstIdx] = CNameLst[2];
            OutOrdLst[LstIdx] = COrdLst[2];
            LstIdx++;
         }
         else
         if ( wcscmp(pClasses[Idx], L"Class2") == 0 )
         {
            OutNameLst[LstIdx] = CNameLst[0];
            OutOrdLst[LstIdx] = COrdLst[0];
            LstIdx++;
            OutNameLst[LstIdx] = CNameLst[2];
            OutOrdLst[LstIdx] = COrdLst[2];
            LstIdx++;
         }
      }

      /* Pass the name and ordinal lists back to the caller.  */
      OutNameLst[LstIdx] = NULL;
      OutOrdLst[LstIdx] = 0;
      pComopNameList = OutNameLst;
      pComopOrdList = OutOrdList;
   }

   return (ccTAC_VRSN); // Returns version number.
}

cc_rtn_code ccTACAPI Command1(ccTAC *pTAC)
{
   ...
}

cc_rtn_code ccTACAPI Command2(ccTAC *pTAC)
{
   ...
}

cc_rtn_code ccTACAPI Operator(ccTAC *pTAC)
{
   ...
}
} ♦
```

The code above would be compiled with the module-definition file as shown below. An import library should automatically be created by the compiler. The *external TAC library* (ExternTACLib.dll), module-definition file (ExternTACLib.def), and the import library (ExternTACLib.lib) should be distributed together if the *external TAC library* is to be publicly released.

```
LIBRARY "ExternTACLib"
EXPORTS
   tacGetCCMapping @1 PRIVATE
   Command1 @2
   Command2 @3
   Operator @4
```

It is **important** that the ordinal numbers specified in the module-definition file correspond with the *comop* names and ordinal numbers specified in `tacGetCCMapping()`.

2.3 Creating an Application Program to Use ETAC

The *ETAC interpreter* can be integrated with a C++ application program or DLL (dynamic linked library) via the AppETAC.dll implementation of the *ETAC interpreter*. A *call-back function* can be implemented in the application program allowing *ETAC code* to execute code in that application program. The *ETAC interpreter* can also be indirectly integrated with a program written in any other programming language, such as Visual Basic, that can load DLLs. This can be achieved by creating such a DLL and incorporating the *ETAC interpreter* within it. Thus the said DLL would be the interface between the said program and the *ETAC interpreter*.

To incorporate the *ETAC interpreter* into an application program or DLL, a C++ programmer needs to load AppETAC.dll and call the DLL function `etacSetAppETAC()` which returns an *ETAC interface*. `etacSetAppETAC()` is passed a single optional *call-back function* (defined in the application program) which is called by the **`run_app_fnt`** *command* within *ETAC code*. The *call-back function* receives the *ETAC interface* as the only C++ argument. Other arguments that the *call-back function* may need are passed by the caller as *TAC objects* on the *object stack*, which are retrieved via the *ETAC interface*. The *call-back function* can push returned *TAC objects* onto the *object stack* before completing. The first *TAC object* argument received by the *call-back function* typically contains an integer that determines the action of that *call-back function*. The *call-back function* can therefore call different C++ functions (with the *ETAC interface* as an argument) based on the value of that integer to perform various activities.

2.3.1 Preliminaries for an Application Program to Use ETAC

To execute *ETAC code* from an application program, AppETAC.dll requires the location of the file containing the *loader script* and the directories containing the ETAC inclusion files. However, inclusion files are not required for executing *ETAC code* files containing *TAC binary instructions*.

The *loader script* file (RunETAC.btac) is the same as the one used with the **Run ETAC Scripts** program. The initialisation function (`etacSetAppETAC()`) within AppETAC.dll searches for the *loader script* file in the following order.

1. The directory of the application program.
2. The current directory.
3. The directory specified at ‹`LoadDir=`› in the initialisation file (RunETAC.ini) used with the **Run ETAC Scripts** program.
4. Prompts the user to enter the *loader script* file. If the user cancels the prompt, then a *loader script* is not used.

The initialisation function also obtains the file path of the text file containing the list of actual inclusion directories used for relative inclusion paths within *ETAC script*. Only the first file path specified is used, as follows.

1. The file path specified in the aeInclDirs parameter of the start-up parameter structure (aeAppETACPars) passed to the initialisation function.
2. The file specified at ‹IncDirs=› in the initialisation file (RunETAC.ini) used with the **Run ETAC Scripts** program.

Note that aeInclDirs and ‹IncDirs=› specify the file path of the file containing the inclusion directory paths, not the inclusion directory paths themselves. If the said file path is not specified, then the current directory is used as the directory for the inclusion files themselves.

The initialisation file (RunETAC.ini) is searched for in the following order.

1. The directory of the application program.
2. The system Windows directory.

If **Run ETAC Scripts** has not been installed, and the application programmer does not want to use the initialisation file (RunETAC.ini), then the *loader script* (RunETAC.btac) can be placed in the directory of the application program, and the file path of the file containing the list of inclusion files can be specified in the aeInclDirs parameter of the start-up parameter structure (aeAppETACPars).

(**Ref**: 3.14.1_Initialisation Function)

2.3.2 Requirements for Incorporating ETAC into an Application Program

The following requirements need to be satisfied when incorporating the *ETAC interpreter* into an application program or DLL.

1. A single *call-back function* needs to be created if required. The *ETAC interpreter* calls that *call-back function* directly through a pointer provided via the initialisation function. (**Ref**: 3.15.1_Call-back Function)
2. The inclusion file AppETAC_***n***.h, where ***n*** which indicates the version of the *ETAC interface* and *resource interfaces* currently in use, needs to be included once in each source file of the application program that interacts with the *ETAC interpreter*. That inclusion file resides in the path ‹**...**\Victella\AppETAC\Include›. For a new application program, the inclusion file with the latest version, ***n***, can (and should) be used. If pre-processor operational definitions need to be made, they must be defined before the inclusion of AppETAC_***n***.h. AppETAC_***n***.h automatically includes ExternTACLib_***n***.h and aeAppETACPars_***v***.h, where ***v*** is the version of the start-up parameter structure (aeAppETACPars). (**Ref**: 3.3.8_Operational Definitions)
3. The initialisation function (etacSetAppETAC()) must be called before any interaction with the *ETAC interpreter* can be made. The initialisation function, declared in the AppETAC_***n***.h file, is required to use the C programming language linkage conventions (‹extern "C"›). (**Ref**: 3.14.1_Initialisation Function)
4. The application program can be linked with AppETAC.lib (the import library for AppETAC.dll).
5. All strings used with a *resource interface* must be well-formed Unicode strings.
6. The following assertions must not fail:
```
assert(sizeof(char) == 1);
assert(sizeof(bool) == 1);
assert(sizeof(short) == 2);
assert(sizeof(int) == 4);
assert(sizeof(long) == 4);
```

```
assert(sizeof(void *) == 4);
assert(sizeof(double) == 8);
assert(sizeof(ccCHAR) == sizeof(unsigned long));
```

When loading AppETAC.dll directly, the application programmer can use either a full path to the DLL or let **Windows**® try to find it. If AppETAC.dll is loaded via AppETAC.lib, a path for the DLL cannot be specified, therefore **Windows**® will need to find it. The standard **Windows**® search order for finding DLLs will be used. That search order (for **Windows**® **XP**) is:

1. The directory from which the application was loaded.
2. The current directory.
3. The system directory.
4. The 16-bit system directory.
5. The `Windows` directory.
6. The directories that are listed in the `PATH` environment variable.

In addition, a programmer familiar with the system registry can use the registry key "`HKLM\Software\Microsoft\Windows\CurrentVersion\App Paths\`*`appfile.exe`*" (for **Windows**® **XP**) with a value name of "`Path`" and value data being the full path of AppETAC.dll. By default, AppETAC.dll exists in the directory "**...**\Victella\AppETAC". **Windows**® will automatically use the path specified in the said registry key to locate the DLL.

Note that AppETAC.dll is a 32-bit program and is compatible only with 32-bit application programs. 32-bit application programs can run on 64-bit platforms.

The following code illustrates the essentials of the structure of an application program (or a DLL if appropriate modifications are made). The *call-back function* (CallbackFnt()) is optional. The initialisation function (`etacSetAppETAC()`) must be called before any interaction with the *ETAC interpreter* can occur via the returned *ETAC interface* (TAC). In this illustration, the *ETAC interpreter* is set to operate in debug (`aeDEBUG`) and silent (`aeSILENT`) mode. A log file is specified for error messages (MainAppLog.log). Two *ETAC code* files (ETACScript1.etac and ETACScript2.btac) are executed before `etacSetAppETAC()` returns. Finally, the application program interacts with the *ETAC interpreter* before the program's main message loop is executed. Note that `etacRelease()` needs to be called before the main window is destroyed, otherwise the *ETAC interpreter* may not be fully released.

```
#include <assert.h>

/* Define the ETAC interface variable. */
#define ccETAC_REF                      TAC

#include "AppETAC_1.h"

/* Declare the call-back function. */
cc_rtn_code ccTACAPI MyCallbackFnt(ccTAC *pTAC);

int WinMain(...)
{
ccRTNCODE; /* Declare the return code variable. */
aeAppETACPars            Pars;
ccTAC                   *TAC;
ccNEW_STRING            (ArgStr); /* Stack argument for ScriptTest.etac. */
aeSTR                    Scripts[3];
int                      RtnVal = 0;

   /* Type size assertions. */
   assert(sizeof(char) == 1);
   assert(sizeof(bool) == 1);
   assert(sizeof(short) == 2);
   assert(sizeof(int) == 4);
```

```
   assert(sizeof(long) == 4);
   assert(sizeof(void *) == 4);
   assert(sizeof(double) == 8);
   assert(sizeof(ccCHAR) == sizeof(unsigned long));

   /* Set required parameters. */
   Pars.aeFlags |= aeDEBUG | aeSILENT; // Optional startup flags.
   Pars.aeLogFilePath = L"MainAppLog.log"; // Optional log file.
   Scripts[0] = L"ETACScript1.etac"; // Optional ETAC code file.
   Scripts[1] = L"ETACScript2.btac"; // Optional ETAC code file.
   Scripts[2] = NULL; // List must be NULL terminated.
   Pars.aeScripts = Scripts;

   /* Call the initialisation function. */
   TAC = ::etacSetAppETAC(ccTAC_VRSN, MyCallbackFnt, &Pars);

   if ( TAC != NULL )
   {
      /* The following is just for demonstration. */
      ccPUSH((0L));
      ccPUSH((L""));
      ccPUSH((L"AppETAC demo."));
      ccCALLTAC(ccExecCmd(L"msg_box"));
      TAC->ccPop();

      /* The following is just for demonstration. */
      ArgStr->strAssign(L"I am all string.");
      ccPUSH((ArgStr));
      ccCALLTAC(ccExecETAC(L"ScriptTest.etac"));

      /* Main message loop. */
      ...
   }

ccEXITLBL:
   if ( ccERROR )
   {
      ::etacProcessTACError(ccRTNCDE_VAR);
   }

   /* Release resource interfaces. */
   ccFREE(ArgStr);

   /* This also needs to be called just before the main window is destroyed. */
   ::etacRelease();

   return (RtnVal);
}

/* Redefine the ETAC interface variable. */
#undef ccETAC_REF
#define ccETAC_REF                    pTAC

/* Define the call-back function. */
cc_rtn_code ccTACAPI MyCallbackFnt(ccTAC *pTAC)
{
   ...
} ♦
```

The illustration above demonstrates the following items: ccCALLTAC, ccERROR, ccETAC_REF (Operational Definitions), ccExecCmd, ccExecETAC, ccEXITLBL (Operational Definitions), ccFREE, ccNEW_STRING, ccPUSH, ccRTNCDE_VAR (Operational Definitions), ccRTNCODE, ccPop, etacSetAppETAC, strAssign.

2.4 Interacting with the ETAC Interpreter

Whether creating an application program or an *external TAC library*, the same method is used for interacting with the *ETAC interpreter*. There are two ways that C++ code can interact with the *ETAC interpreter*. The first, preferred, method is to use pre-processor macro definitions. Using the said macros removes the need to create extraneous coding, thus simplifying the interaction with the *ETAC interpreter*. The second method is not to use the macros but to produce full explicit code that interacts with the *ETAC interpreter*. (**Ref**: 3.4_Macro Definitions)

The most common interaction with the *ETAC interpreter* is to access and modify *TAC objects*. A *resource interface* is required for each non-numerical *TAC object* to be modified. To use a *resource interface*, the C++ programmer defines a NULL pointer to the required *resource interface* which is initialised by appropriate member functions of the *ETAC interface* or member functions of some other already initialised *resource interface*. A *resource interface* must be explicitly released by the C++ programmer when it is no longer to be used, otherwise the internal memory for that interface will not be released until AppETAC.dll is unloaded.

In this section, the macro method mentioned above will be used in the illustrations. To use the explicit method, a C++ programmer can inspect the body of the macro definitions.

The following code illustrates an *ETL function* (MyETLFnt) which receives a string argument on the *object stack*, displays the string to the console prefixed with a number, and waits for the user to type in a response. The string response is pushed onto the *object stack* as the returned value for the *ETL function* caller. For example, if the string argument is "what's up?", the *ETL function* displays "1 what's up? :>" on the console and waits for the user to type a response (followed by hitting the ENTER key). If the user response is "Nothing" then the returned string on the *object stack* would be "Response: Nothing". If the string argument on the next call of the *ETL function* is "Why Nothing?", then the *ETL function* displays "2 Why Nothing :>", and the user's response is returned to the caller. Note that the string response is constructed in a memory block for illustration purposes only; a string variable could have been used to achieve the same result.

```
#include <stdlib.h>

#include "ExternTACLib_1.h"

extern "C" cc_rtn_code ccTACAPI MyETLFnt(ccTAC *pTAC)
{
ccRTNCODE; /* Declare the return code variable. */
ccSTRING               (ArgStr); /* Stack argument for library function. */
ccSTRING               (RtnStr); /* String value to return to caller. */
ccNEW_STRING           (NewStr);
ccNEW_MEMORY           (StrMem);
wchar_t                 StrBuff[33];
static int              Num = 0;

   assert(sizeof(wchar_t) == ccSTR_CHAR_SZ)

   /* Pull the string argument off the object stack. */
   ccPULL((ArgStr));

   /* Construct the next number and prefix it to the string argument. */
   (void)_itow(++Num, StrBuff, 10);
   NewStr->strAssign(StrBuff);
   NewStr->strAppend(L" ");
   NewStr->strAppend(ArgStr);
   NewStr->strAppend(L" :>");

   /* Display the new string to the console and wait for a response. */
   ccPUSH((NewStr));
   ccCALLTAC(ccExecCmd(L"read_con"));
```

```
   /* Get the response off the object stack. */
   ccPULL((ArgStr)); // ArgStr can be used again.

   /* Construct the string in a memory block to return on the object stack. */
   (void)StrMem->mbSetDataSize(0); // Not needed in this case (data size is already 0).
   StrMem->mbImport(L"Response: ");
   StrMem->mbImport(ArgStr);
   StrMem->mbImport(L"\r\n");

   /* Transfer the string in the memory block to a string resource. */
   StrMem->mbExport(RtnStr);

   /* Return the response string on the object stack for the caller. */
   ccPUSH((RtnStr));

ccEXITLBL:
   /* Release resource interfaces. */
   ccFREE(StrMem);
   ccFREE(NewStr);
   ccFREE(RtnStr);
   ccFREE(ArgStr);

   return (ccRTNCDE_VAR);
} ♦
```

The illustration above demonstrates the following items: ccCALLTAC, ccEXITLBL (Operational Definitions), ccFREE, ccNEW_MEMORY, ccNEW_STRING, ccPULL, ccPUSH, ccRTNCDE_VAR (Operational Definitions), ccRTNCODE, ccSTRING, mbExport, mbImport, strAppend, strAssign.

The following code illustrates an *ETL function* (OperETLFnt) to be called from *ETAC code* as an *operator*. The *ETL function* appends two or more string arguments on the *object stack*, returning the concatenated string.

```
#include "ExternTACLib_1.h"

extern "C" cc_rtn_code ccTACAPI OperETLFnt(ccTAC *pTAC)
{
ccRTNCODE; /* Declare the return code variable. */
ccSTRING               (ArgStr); /* Stack argument for library function. */
ccNEW_STRING           (RtnStr); /* String value to return to caller. */
ccNEW_SEQUENCE         (ErrSeq); /* Sequence for error message. */
ccULONG                 NumArgs;
unsigned long           Idx;

   /* Determine the number of string arguments on the object stack. */
   NumArgs = pTAC->ccCountToMark();

   if ( NumArgs >= 2 )
   {
      /* Correct number of arguments - proceed. */
      for (Idx = 0; Idx < NumArgs; Idx++)
      {
         /* Pull the next string argument off the object stack and append it to the
            previous one. */
         ccPULL((ArgStr));
         RtnStr->strAppend(ArgStr);
      }

      /* Pop the mark 0 stack object off the object stack. */
      pTAC->ccPop();

      /* Return the concatenated string on the object stack for the caller. */
      ccPUSH((RtnStr));
   }
   else
```

```
   {
      /* Incorrect number of arguments. */
      /* Pop the arguments and mark 0 off the object stack. */
      pTAC->ccPop(NumArgs + 1); // 1 is for the mark 0 stack object.

      /* Return a programmer-defined error. */
      (void)ErrSeq->sPut(L"Wrong number of arguments."); // Ignore boolean return value.
      ccPUSH((ErrSeq)); // Push the programmer-defined error message.
      ccPUSH((10L)); // Push the programmer-defined error number 10.
      ccSET_LIBERR; // Indicates programmer-defined error for the ETAC interpreter.
   }

ccEXITLBL:
   /* Release resource interfaces. */
   ccFREE(ErrSeq);
   ccFREE(RtnStr);
   ccFREE(ArgStr);

   return (ccRTNCDE_VAR);
} ♦
```

The illustration above demonstrates the following items: ccEXITLBL (Operational Definitions), ccFREE, ccNEW_SEQUENCE, ccNEW_STRING, ccPULL, ccPUSH, ccSET_LIBERR, ccRTNCDE_VAR (Operational Definitions), ccRTNCODE, ccSTRING, ccCountToMark, ccPop, sPut, strAppend.

The following code illustrates a *call-back function* (CallbackFnt) that performs various operations. The code uses a command number (CmdNum) pulled from the *object stack* to determine which operation to perform. However, such a method of determining which operation to perform is decided by the programmer — a different method can be used if desired.

```
#include "AppETAC_1.h"

extern "C" cc_rtn_code ccTACAPI CallbackFnt(ccTAC *pTAC)
{
ccRTNCODE; /* Declare the return code variable. */
ccINT                   CmdNum; /* Command number to perform various activities. */
... /* Other declarations. */

   ccPULL((CmdNum));

   switch (CmdNum)
   {
   case 1 : /* Operation 1. */
      /* Code. */
      ccRTNCDE_VAR = Fnt1(pTAC); // Fnt1() is an example.
      break;

   case 2 : /* Operation 2. */
      /* Code. */
      break;

   ...

   default:
      /* Return a programmer-defined error. */
      (void)ErrSeq->sPut(L"Unaccounted command number for 'CallBackFnt'.");
      ccPUSH((ErrSeq)); // Push the programmer-defined error message.
      ccPUSH((5L)); // Push the programmer-defined error number 5.
      ccSET_LIBERR; // Indicates programmer-defined error for the ETAC interpreter.
   }

   ...
```

```
ccEXITLBL:
   /* Release resource interfaces. */
   ...

   return (ccRTNCDE_VAR);
} ♦
```

The illustration above demonstrates the following items: ccEXITLBL (Operational Definitions), ccPULL, ccPUSH, ccSET_LIBERR, ccRTNCDE_VAR (Operational Definitions), ccRTNCODE, sPut.

2.5 Calling ETL Functions from C++

An *ETL function* can be called directly from C++ code wherever a pointer to the *ETAC interface* is available. The general procedure to call an *ETL function* from C++ code is as follows. The procedure applies to each *external TAC library*.

1. Load the *external TAC library*, either explicitly or via the corresponding import library. In either case, a handle to the *external TAC library* (a DLL) is obtained.
2. Call TAC->ccGetTACIF() with the *external TAC library* handle (from step (1)) to obtain a pointer to the *ETAC interface* for the *ETL functions* within that *external TAC library*. TAC is a pointer to the existing *ETAC interface*. Note that the obtained *ETAC interface* for the *external TAC library* may be of a different version than the existing *ETAC interface* ("TAC").
3. Call the desired *ETL functions* with the *ETAC interface* obtained from step (2).

The following code illustrates how to call an *ETL function*, StringTest(), from C++ code. The example assumes that the C++ code has been linked with the import library of the *external TAC library* (ExternTACLib.dll). The example also assumes that pTAC already points to an existing *ETAC interface*. StringTest() takes a string *stack object* from the *object stack*, converts it to a different string, and returns the converted string onto the *object stack*. Error checking code and other irrelevant items are not shown in the illustration.

```
#include "ExternTACLib_1.h" /* or AppETAC_1.h if appropriate. */

extern "C" cc_rtn_code ccTACAPI StringTest(ccTAC *TACIF); /* Declare library function. */

cc_rtn_code Function(ccTAC *pTAC) /* Some function. */
{
ccRTNCODE; /* Declare the return code variable. */
ccTAC                   *TACIF; /* ETAC interface for all the ETL functions. */
HMODULE                  ModHndl;
ccNEW_STRING            (ArgStr); /* Stack argument for StringTest(). */
ccSTRING                (RtnStr); /* Returned value from StringTest(). */

   /* Get library handle (needs only be done once for the same external TAC library). */
   ModHndl = ::GetModuleHandle(L"ExternTACLib.dll");

   /* Get the ETAC interface for all the ETL functions (needs only be done once for the
      same external TAC library). */
   TACIF = pTAC->ccGetTACIF(ModHndl);

   /* Call the ETL function after pushing the required argument onto the object stack. */
   ArgStr->strAssign(L"I am all string.");
   ccPUSH((ArgStr)); /* OR: ccPUSH((L"I am all string.")). */
   ccCALL(StringTest(TACIF)); /* Use TACIF to call the StringTest() ETL function. */
   ccPULL((RtnStr)); /* Get the value returned by StringTest() from the object stack. */

   /* Do something nice here. */
   ...
```

```
ccEXITLBL:
   /* Release resource interfaces. */
   ccFREE(RtnStr);
   ccFREE(ArgStr);

   return (ccRTNCDE_VAR);
} ♦
```

3

Programming Reference

This chapter contains a reference to the C++ functions required for communication between the *ETAC interpreter* and C++ code. The chapter is intended for programmers who are familiar with the ETAC programming language and the C++ programming language.

3.1 The ETAC Interface

The *ETAC interface* is the means by which C++ code communicates with the *ETAC interpreter*, whether the C++ code exists in an application program or in an *external TAC library*. The *ETAC interface* is implemented as an instance of a C++ class named `ccTAC` which is obtained from, and owned by, the *ETAC interpreter*. *Resource interfaces* represent the values of *TAC objects*, and are obtained from the *ETAC interface* and other *resource interfaces*, and owned by the *ETAC interpreter*.

3.2 Resource Interfaces

A *resource interface* internally contains the *resource value* of a *compound stack object*. That *resource value* is accessed by the C++ programmer by member functions of the *resource interface*. The *resource value* itself is maintained by the *ETAC interpreter*, and not directly accessible by the C++ programmer (except where noted). The value of a *TAC object* with an *embedded value* does not use a *resource interface*. Instead, the value of such a *TAC object* (decimal and integer based *TAC objects*) exists directly in a C++ variable.

An instance of a *resource interface* is accessed via a C++ variable containing a pointer to the *resource interface*. The variable must initially be initialised with `NULL` before the *resource interface* is used. A *resource interface* must be released when no longer used. Once a *resource interface* is released, all existing pointers to it will no longer be valid. A *resource interface* can be created via a `ccNEW_*()` macro (where `*` represents the kind of *resource interface* to be created), and released by the `ccFREE()` macro.

If an output variable referenced by a member function contains a `NULL` pointer to a *resource interface*, then a new *resource interface* will be allocated to that variable, otherwise the existing *resource interface* pointed to by that variable will automatically be released and a new one will be allocated.

All strings used with a *resource interface* must be well-formed Unicode strings (unpaired surrogate code points are not supported), otherwise failures will occur.

3.3 Pre-processor Definitions

There are several pre-processor definitions used with the *ETAC interface* and *resource interfaces* as described in the following sections. The pre-processor definitions exist in the file `ExternTACLib_n.h`, where ***n*** is the version number of the *ETAC interface* and *resource interfaces* in use. The following sections lists the pre-processor names and their meaning.

3.3.1 TAC Object Types

The following are the pre-processor names for the *TAC object* types.

TAC Object Types

Name	Meaning
ccTAC_INT	Integer (–2,147,483,648 to 2,147,483,647).
ccTAC_DEC	Decimal ($\pm 2.2250738585072014 \times 10^{-308}$ to $\pm 1.7976931348623158 \times 10^{308}$ or zero).
ccTAC_STR	Unicode® string.
ccTAC_SEQ	*Sequence*.
ccTAC_PROC	*Procedure*.
ccTAC_CMD	Named *command*.
ccTAC_CMDL	*Linked command*.
ccTAC_CMDI	*Intrinsic command*.
ccTAC_OPR	Named *operator*.
ccTAC_OPRL	*Linked operator*.
ccTAC_OPRI	*Intrinsic operator*.
ccTAC_DICT	*Dictionary*.
ccTAC_MARK	Mark (0 to 7).
ccTAC_MEM	Memory.
ccTAC_NULL	Null.
ccTAC_EXE	Executable.

3.3.2 Intrinsic Command Codes

The following are the pre-processor names for the *intrinsic commands*.

Intrinsic Command Codes

Name	Meaning
ccTAC_I_START_SEQ	Start *sequence expression* construction ([).
ccTAC_I_END_SEQ	End *sequence expression* construction (]).
ccTAC_I_START_PROC	Start *procedure expression* construction ({).
ccTAC_I_END_PROC	End *procedure expression* construction (}).
ccTAC_I_START_OP	Start *operator expression* ()).
ccTAC_I_END_OP	End *operator expression* (().
ccTAC_I_ASN_DICT_ITEM	Assign *dictionary* item (:=).
ccTAC_I_NEW_DICT_ITEM	New *dictionary* item (:-).
ccTAC_I_NEW_DICT	New *dictionary*.
ccTAC_I_NEG	Negate number.
ccTAC_I_POP	Pop off *TAC stack*.
ccTAC_I_SWAP	Swap first two *stack objects*.
ccTAC_I_DUP_TOP	*Duplicate* topmost *stack object*.
ccTAC_I_COPY_TOP	*Copy* topmost *stack object*.
ccTAC_I_COPY	*Copy stack objects* from top of *TAC stack*.

ccTAC_I_COPY_ANY	*Copy* any *stack object*.
ccTAC_I_COPY_REL	*Copy* relative *stack object*.
ccTAC_I_ROLL	Roll *stack objects*.
ccTAC_I_COUNT	Count *stack objects*.
ccTAC_I_CLEAR	Remove *stack objects*.
ccTAC_I_CNT_TO_MARK	Count to mark.
ccTAC_I_GET_TYPE	Get *stack object* type.
ccTAC_I_CUSTOM	Execute custom *comop*.
ccTAC_I_ASK_STACK	Determine current *TAC stack*.
ccTAC_I_SET_STACK	Set current *TAC stack*.
ccTAC_I_EXEC	Execute *stack object*.
ccTAC_I_IF_THEN	'If then' construct.
ccTAC_I_SWITCH	'Switch' construct.
ccTAC_I_DO_FOR	'Do for' iteration.
ccTAC_I_DO_REPEAT	'Do repeat' iteration.
ccTAC_I_DO_LOOPS	'Do loops' iteration.
ccTAC_I_DO_WITH	'Do with' iteration.
ccTAC_I_DO_WHILE	'Do while' iteration.
ccTAC_I_DO_UNTIL	'Do until' iteration.
ccTAC_I_BREAK	Break from *procedure*.
ccTAC_I_GO_END	Go to end of *procedure*.
ccTAC_I_END	End *ETAC session*.
ccTAC_I_NOT	Binary inversion of integer.
ccTAC_I_GET	Get *sequence* or *procedure* element.
ccTAC_I_PUT	Put *sequence* or *procedure* element.
ccTAC_I_SIZE	Get size of *compound stack object*.
ccTAC_I_TRUNC	Truncate decimal, integer, or string.
ccTAC_I_SET_CUR_ACT	Set current action of *stack object*.
ccTAC_I_GET_CUR_ACT	Get current action of *stack object*.

3.3.3 Intrinsic Operator Codes

The following are the pre-processor names for the *intrinsic operators*.

Intrinsic Operator Codes

Name	Meaning
ccTAC_I_ADD	Add (+).
ccTAC_I_SUB	Subtract (−).
ccTAC_I_DIV	Divide (/).
ccTAC_I_MULT	Multiply (*).
ccTAC_I_POWER	Power (^).
ccTAC_I_EQUAL	Equal (=).
ccTAC_I_N_EQUAL	Not equal (!=).
ccTAC_I_LESS	Less than (<).
ccTAC_I_GREAT	Greater than (>).
ccTAC_I_LESS_EQ	Less than or equal (<=).
ccTAC_I_GREAT_EQ	Greater than or equal (>=).
ccTAC_I_AND	Binary 'AND'.
ccTAC_I_OR	Binary 'OR'.
ccTAC_I_COMBINE	Combine *sequences* or *procedures* (++).

3.3.4 TAC Object Actions

The following are the pre-processor names of *stack object* actions for `ccTAC_I_SET_CUR_ACT` and `ccTAC_I_GET_CUR_ACT`.

TAC Object Actions

Name	Meaning
ccSO_NOM_ACT	Set nominal action for *stack object*.
ccSO_EXEC_INT	Execute internal process.
ccSO_EXEC_DICT	Execute *stack object* found in *dictionary*.
ccSO_EXEC_MEMBS	Execute *members* of *procedure*.
ccSO_PUSH_ON_STACK	Push *stack object* onto the *object stack*.
ccSO_PUSH_ON_OSTACK	Push *stack object* onto the *operator stack*.
ccSO_PUSH_ON_DSTACK	Push *stack object* onto the *dictionary stack*.

3.3.5 TAC Object Indicators

The following are the pre-processor names of *TAC stack* indicators for `ccTAC_I_ASK_STACK` and `ccTAC_I_SET_STACK`.

TAC Stack Indicators

Name	Meaning
ccOBJ_STACK	*Object stack*.
ccOPR_STACK	*Operator stack*.
ccDICT_STACK	*Dictionary stack*.

3.3.6 Logical Boolean Values

The following are the pre-processor names of the logical *boolean values* for *stack objects* of type ccTAC_INT.

Boolean Values

Name	**Meaning**
ccTAC_TRUE	'true' value.
ccTAC_FALSE	'false' value.

3.3.7 Dictionary Binary Flags

The following are the pre-processor names of the binary flags associated with a *dictionary resource value*.

Dictionary Binary Flags

Name	**Meaning**
ccD_LINKAGES	The *dictionary resource value* has at least one *link* into it.

3.3.8 Operational Definitions

The following pre-processor names are re-definable by the C++ programmer (unless stated otherwise). Redefinitions must be made before the inclusion of ExternTACLib_***n***.h or AppETAC_***n***.h.

Operational Definitions

Name	**Meaning**
ccETAC_REF	The variable name used for the pointer (ccTAC *) to the *ETAC interface* (default: pTAC).
ccRTNCDE_VAR	The variable name used for the return code (cc_rtn_code) for the *ETAC interface* and *resource interface* member functions (default: ccRtnCode).
ccEXITLBL	The name of the C++ label used for releasing *resource interfaces* and other actions when an *ETAC interface* or *resource interface* member function returns a non-success return code (default: ccExit).
ccTACAPI	The low-level function calling convention used for communication with all functions defined in an *external TAC library* or the *call-back function* in an application program. The value must be exactly equivalent to __cdecl as defined for the Microsoft® Visual C++® compiler (default: __cdecl).
ccVFI	The low-level function calling convention used for communication with the *ETAC interpreter* defined in the member functions of the *ETAC interface* and *resource interfaces*. The value must be exactly equivalent to __cdecl as defined for the Microsoft® Visual C++® compiler (default: __cdecl).
ccTAC_VRSN	Specifies the version of the *ETAC interface* and the *resource interfaces* in use. The value of this definition must not be modified.

The first three definitions are used by some of the 'cc' macros.

3.3.9 Data Form Indicators

The following names are defined for the *data forms* of a memory *stack object*.

Data Form Indicators

Symbol	Meaning
ccMO_BIN	Non-text data or data regarded as binary (default).
ccMO_TXT	Standard (8-bit) character width text data (Windows-1252).
ccMO_U8	UTF-8 text data.
ccMO_U16_LE	UTF-16 text data (little-endian).
ccMO_U16_BE	UTF-16 text data (big-endian).
ccMO_U32_LE	UTF-32 text data (little-endian).
ccMO_U32_BE	UTF-32 text data (big-endian).
ccMO_NATIVE	Native *data form* for Unicode® strings on the **Windows**® operating system (same as ccMO_U16_LE).

Note that the UTF formats must be well-formed code unit sequences, otherwise an error may occur or the consequence is unpredictable. When verifying memory data for conformance with UTF-16 or UTF-32, ETAC regards UTF null terminator characters (U+0000) as being invalid.

3.4 Macro Definitions

There are several pre-processor macro definitions used with the *ETAC interface* and *resource interfaces* as described in the following sections. The pre-processor definitions exist in the file ExternTACLib_***n***.h, where ***n*** is the version number of the *ETAC interface* and *resource interfaces* in use.

3.4.1 Macro Summary

The following list is a summary of the pre-processor macro definitions.

Pre-processor Macro Summary

Function	Description
ccCALL	Calls an *ETAC interface* or a *resource interface* member function that returns a return code.
ccCALLTAC	Calls an *ETAC interface* member function that returns a return code.
ccDATAOBJECT	Defines and initialises a variable to later hold a pointer to a *data object resource interface*.
ccDICTIONARY	Defines and initialises a variable to later hold a pointer to a *dictionary resource interface*.
ccERROR	Evaluates to true if the return code variable indicates an error.
ccFREE	Releases a *resource interface* allocated to a variable.
ccMEMORY	Defines and initialises a variable to later hold a pointer to a memory *resource interface*.
ccNEW	Allocates a new empty *resource interface* to a variable.
ccNEW_DATAOBJECT	Defines and allocates a variable containing a new empty *data object resource interface*.
ccNEW_DICTIONARY	Defines and allocates a variable containing a new empty *dictionary resource interface*.

ccNEW_MEMORY	Defines and allocates a variable containing a new empty memory *resource interface*.
ccNEW_SEQUENCE	Defines and allocates a variable containing a new empty *sequence resource interface*.
ccNEW_STACKOBJ	Defines and allocates a variable containing a new null *stack object resource interface*.
ccNEW_STRING	Defines and allocates a variable containing a new empty string *resource interface*.
ccPULL	Obtains a specified item from the top of the *object stack* or *dictionary stack*, removing that item.
ccPUSH	Pushes a *stack object* containing a specified item to the top of the *object stack* or *dictionary stack*.
ccRTNCODE	Defines and initialises the return code variable.
ccSEQUENCE	Defines and initialises a variable to later hold a pointer to a *sequence resource interface*.
ccSET_LIBERR	Assigns the programmer initiated *external TAC library* error return code to the return code variable.
ccSPCHAR	Converts a supplementary plane character to the corresponding Unicode® code point.
ccSTACKOBJ	Defines and initialises a variable to later hold a pointer to a *stack object resource interface*.
ccSTRING	Defines and initialises a variable to later hold a pointer to a string *resource interface*.
ccSUCCESS	Evaluates to `true` if the return code variable indicates success.

3.4.2 Return Code Macros

The following pre-processor definitions involve return codes from the *ETAC interface* and *resource interface* member functions.

ccERROR

ccERROR

Dependency
`ccRTNCDE_VAR`.

Action
Evaluates to `true` if the return code variable (defined by `ccRTNCDE_VAR`) indicates an error.

Example

```
ccRTNCODE; /* Declare the return code variable. */

   ...

   ccCALLTAC(ccExecCmd(L"msg_box")); /* Could cause an error. */

   ...

ccEXITLBL:
   if ( ccERROR ) /* Detect and process error. */
   {
      ::etacProcessTACError(ccRTNCDE_VAR);
   }
```

Additional Information

Operational Definitions (ccRTNCDE_VAR)

ccSUCCESS

ccSUCCESS

Dependency

ccRTNCDE_VAR.

Action

Evaluates to true if the return code variable (defined by ccRTNCDE_VAR) indicates success.

Example

```
ccRTNCODE; /* Declare the return code variable. */

   ...

   ccRTNCDE_VAR = ccETAC_REF->ccExecCmd(L"msg_box"); /* Could cause an error. */


   if ( ccSUCCESS )  /* Uses ccRTNCDE_VAR. */
   {
      /* Do something nice. */
   }
   else
   {
      /* Process error. */
      ::etacProcessTACError(ccRTNCDE_VAR);
   }
```

Additional Information

Operational Definitions (ccRTNCDE_VAR)

ccRTNCODE

ccRTNCODE

Dependency

ccRTNCDE_VAR.

Action

Defines and initialises the return code variable (defined by ccRTNCDE_VAR).

Example

```
ccRTNCODE; /* Declare and initialise the return code variable with ccTAC_RTN_SUCCESS. */
... /* Rest of code. */
```

Additional Information
Operational Definitions (ccRTNCDE_VAR)

ccSET_LIBERR

ccSET_LIBERR

Dependencies
ccRTNCDE_VAR.

Action
Assigns the programmer initiated *external TAC library* error return code (ccETP_RTN_LIBERR) to the return code variable (defined by ccRTNCDE_VAR). If an *ETL function* or the *call-back function* returns ccETP_RTN_LIBERR to the *ETAC interpreter* (by use of this macro or not), then before returning, the topmost *stack object* on the *object stack* must be an integer *stack object* containing a programmer-defined error number, followed by a *sequence*. The first element of the *sequence* can be a string that is automatically displayed for the default **@OnLibErr** and **@OnAppErr** *ETAC functions*. The *sequence* can be empty, or contain any required elements for custom redefinitions of **@OnLibErr** or **@OnAppErr**.

Example
```
   /* Return a programmer-defined error. */
   (void)ErrSeq->sPut(L"Wrong number of arguments.");
   ccPUSH((ErrSeq)); // Push the programmer-defined error message.
   ccPUSH((5L)); // Push the programmer-defined error number 5.
   ccSET_LIBERR; // Indicates programmer-defined error for the ETAC interpreter.
```

Additional Information
Operational Definitions (ccRTNCDE_VAR)

3.4.3 Resource Interface Definition Macros

These macros define and initialise a variable for use with the specified *resource interface*. The macros do not actually create a *resource interface*, but only prepare the variable for use (by initialising the variable with NULL).

ccSTACKOBJ

ccSTACKOBJ (**mVar**)

Output
mVar The variable name to hold a *stack object resource interface* (ccStackObj *).

Action
Defines and initialises a variable (mVar) to later hold a pointer to a *stack object resource interface*. The macro does not allocate the *resource interface*.

Example
```
ccSTACKOBJ (StkObj); /* Declare the StkObj variable to be allocated later. */
... /* Rest of code. */
```

ccSTRING

ccSTRING (**mVar**)

Output
mVar The variable name to hold a string *resource interface* (ccString *).

Action
Defines and initialises a variable (`mVar`) to later hold a pointer to a string *resource interface*. The macro does not allocate the *resource interface*.

Example
```
ccSTRING (MyStr); /* Declare the MyStr variable to be allocated later. */
... /* Rest of code. */
```

ccMEMORY

ccMEMORY (mVar)

Output

`mVar` The variable name to hold a memory *resource interface* (`ccMemoryBlock *`).

Action
Defines and initialises a variable (`mVar`) to later hold a pointer to a memory *resource interface*. The macro does not allocate the *resource interface*.

Example
```
ccMEMORY (FileData); /* Declare the FileData variable to be allocated later. */
... /* Rest of code. */
```

ccSEQUENCE

ccSEQUENCE (mVar)

Output

`mVar` The variable name to hold a *sequence resource interface* (`ccSequence *`).

Action
Defines and initialises a variable (`mVar`) to later hold a pointer to a *sequence resource interface*. The macro does not allocate the *resource interface*.

Example
```
ccSEQUENCE (ErrSeq); /* Declare the ErrSeq variable to be allocated later. */
... /* Rest of code. */
```

ccDICTIONARY

ccDICTIONARY (mVar)

Output

`mVar` The variable name to hold a *dictionary resource interface* (`ccDictionary *`).

Action
Defines and initialises a variable (`mVar`) to later hold a pointer to a *dictionary resource interface*. The macro does not allocate the *resource interface*.

Example
```
ccDICTIONARY (MainDict); /* Declare the MainDict variable to be allocated later. */
... /* Rest of code. */
```

ccDATAOBJECT

ccDATAOBJECT(mVar)

Output

mVar The variable name to hold a *data object resource interface* (ccDataObject *).

Action

Defines and initialises a variable (mVar) to later hold a pointer to a *data object resource interface*. The macro does not allocate the *resource interface*. Note that, currently, a *data object* is implemented as a *sequence*, but should not be used as such. A *data object resource interface* is used with the ccMakeDataObj() and ccGetDataDict() helper functions.

Example

```
ccDATAOBJECT (DataObj); /* Declare the DataObj variable to be allocated later. */
... /* Rest of code. */
```

3.4.4 Resource Interface Allocation and Release Macros

These macros define and allocate a variable with a specified *resource interface* (the variable need not have been initialised, except for the ccNEW() macro). The macros actually create an empty *resource interface* and initialise the variable with a pointer to that *resource interface*. For the ccNEW() macro, the variable must have already been defined and initialised with NULL. The ccFREE() macro releases the *resource interface* pointed to by a variable.

ccNEW

ccNEW(mVar)

Output

mVar The variable name to hold a new *resource interface*.

Dependencies

ccETAC_REF.

Action

Allocates a new empty *resource interface* to a variable (mVar), releasing the previous *resource interface* if there was one. mVar must be correctly defined and initialised for the desired *resource interface* before this macro is called. The *resource interface* must be released (ccFREE(mVar)) when no longer in use.

Example

```
ccSTRING (MyStr); /* Declare the MyStr variable to be allocated later. */
   ...
   ccNEW(MyStr); /* Allocate a new string resource interface into MyStr. */
   ...
   ccNEW(MyStr); /* Release existing resource interface in MyStr and allocate a new one. */
```

Additional Information

Operational Definitions (ccETAC_REF)

Related Information

ccSTACKOBJ ▪ ccSTRING ▪ ccMEMORY ▪ ccSEQUENCE ▪ ccDICTIONARY ▪ ccFREE

Other Information

ccNEW_STACKOBJ ▪ ccNEW_STRING ▪ ccNEW_MEMORY ▪ ccNEW_SEQUENCE ▪ ccNEW_DICTIONARY

ccNEW_STACKOBJ

ccNEW_STACKOBJ(mVar)

Output

mVar The variable name to hold a new *stack object resource interface* (ccStackObj *).

Dependencies

ccSTACKOBJ(), ccNEW().

Action

Defines and allocates a variable (mVar) containing a new null *stack object resource interface*. The *resource interface* must be released (ccFREE(mVar)) when no longer in use.

Example

```
ccNEW_STACKOBJ (StkObj); /* Declare and allocate the StkObj variable with a null stack
                            object resource interface. */
... /* Rest of code. */
```

ccNEW_STRING

ccNEW_STRING(mVar)

Output

mVar The variable name to hold a new string *resource interface* (ccString *).

Dependencies

ccSTRING(), ccNEW().

Action

Defines and allocates a variable (mVar) containing a new empty string *resource interface*. The *resource interface* must be released (ccFREE(mVar)) when no longer in use.

Example

```
ccNEW_STRING (MyStr); /* Declare and allocate the MyStr variable with an empty string
                         resource interface. */
... /* Rest of code. */
```

ccNEW_MEMORY

ccNEW_MEMORY(mVar)

Output

mVar The variable name to hold a new memory *resource interface* (ccMemoryBlock *).

Dependencies

ccMEMORY(), ccNEW().

Action

Defines and allocates a variable (mVar) containing a new empty memory *resource interface*. The *resource interface* must be released (ccFREE(mVar)) when no longer in use.

Example

```
ccNEW_MEMORY (FileData); /* Declare and allocate the FileData variable with an empty memory
                            resource interface. */
... /* Rest of code. */
```

ccNEW_SEQUENCE

ccNEW_SEQUENCE(**mVar**)

Output

mVar The variable name to hold a new *sequence resource interface* (ccSequence *).

Dependencies

ccSEQUENCE(), ccNEW().

Action

Defines and allocates a variable (mVar) containing a new empty *sequence resource interface*. The *resource interface* must be released (ccFREE(mVar)) when no longer in use.

Example

```
ccNEW_SEQUENCE (ErrSeq); /* Declare and allocate the ErrSeq variable with an empty sequence
                            resource interface. */
... /* Rest of code. */
```

ccNEW_DICTIONARY

ccNEW_DICTIONARY(**mVar**)

Output

mVar The variable name to hold a new *dictionary resource interface* (ccDictionary *).

Dependencies

ccDICTIONARY(), ccNEW().

Action

Defines and allocates a variable (mVar) containing a new empty *dictionary resource interface*. The *resource interface* must be released (ccFREE(mVar)) when no longer in use.

Example

```
ccNEW_DICTIONARY (MainDict); /* Declare and allocate the MainDict variable with an empty
                                dictionary resource interface. */
... /* Rest of code. */
```

ccNEW_DATAOBJECT

ccNEW_DATAOBJECT(**mVar**)

Output

mVar The variable name to hold a new *data object resource interface* (ccDataObject *).

Dependencies

ccDATAOBJECT(), ccNEW().

Action

Defines and allocates a variable (mVar) containing a new empty *data object resource interface*. The *resource interface* must be released (ccFREE(mVar)) when no longer in use. Note that, currently, a *data object* is implemented as a *sequence*, but should not be used as such. A *data object resource interface* is used with the ccMakeDataObj() and ccGetDataDict() helper functions.

Example

```
ccNEW_DATAOBJECT (DataObj); /* Declare and allocate the DataObj variable with an empty
                               data object resource interface. */
... /* Rest of code. */
```

ccFREE

ccFREE (mVar)

Input

mVar The variable name that holds a *resource interface* to be released.

Dependencies

ccETAC_REF.

Action

Releases a *resource interface* allocated to a variable (mVar). Note that all *resource interfaces* must be released when no longer in use.

Example

```
ccRTNCODE; /* Declare the return code variable. */
ccNEW_STRING (MsgStr); /* Declare and allocate the MyStr variable. */

   ...

   MsgStr->strAssign(L"I'm a pretty string"); /* Use the MsgStr variable. */
   ccPUSH((0L));
   ccPUSH((L""));
   ccPUSH((MsgStr));
   ccCALLTAC(ccExecCmd(L"msg_box"));
   ccETAC_REF->ccPop();

   ...

ccEXITLBL:
   ccFREE(MsgStr); /* Free the resource interface in the MsgStr variable. */
```

Additional Information

Operational Definitions (ccETAC_REF)

3.4.5 Member Function Execution Macros

These macros call member functions for the *ETAC interface* and *resource interfaces* that return a return code (cc_rtn_code). They automatically handle returned error code conditions.

ccCALL

ccCALL (mFnt)

Input

mFnt The expression of a member function of the *ETAC interface* or a *resource interface* to be called, or the name and arguments of an *ETL function* to be executed.

Dependencies

ccRTNCDE_VAR, ccERROR, ccEXITLBL.

Action

Calls an *ETAC interface* or a *resource interface* member function (mFnt) that returns a return code (cc_rtn_code). If the member function returns an error (ccERROR) then control will jump to the label defined by ccEXITLBL. Note that for member functions of the *ETAC interface*, ccCALLTAC() can be used instead of this macro.

Example

```
ccRTNCODE; /* Declare the return code variable. */
ccNEW_SEQUENCE (MySeq); /* Declare and allocate the MySeq variable. */
ccINT MyInt = 10;
   ...
   (void)MySeq->sPut(MyInt); /* Append an integer to the sequence. */
   ...
   ccCALL(MySeq->sGet(MyInt)); /* Retrieve the same integer. */
   ccCALL(ccETAC_REF->ccPush(MyInt)); /* Push the integer to the object stack. */
   ccCALL(MyETLFnt(MyInt)); /* Call an ETL function (needs appropriate setup). */
   ...

ccEXITLBL:
   if ( ccERROR ) /* Detect and process error. */
   {
      ::etacProcessTACError(ccRTNCDE_VAR);
   }
```

Additional Information

Operational Definitions (ccRTNCDE_VAR, ccEXITLBL)

Other Information

ccCALLTAC

ccCALLTAC

ccCALLTAC(**mFnt**)

Input

mFnt The expression of a member function of the *ETAC interface* to be called.

Dependencies

ccCALL(), ccETAC_REF.

Action

Calls an *ETAC interface* member function (mFnt) that returns a return code (cc_rtn_code). If the member function returns an error (ccERROR) then control will jump to the label defined by ccEXITLBL.

Example

```
ccRTNCODE; /* Declare the return code variable. */
   ...
   ccPUSH((L"Tap a key to exit.")); /* A prompt string to display to the console. */
   ccCALLTAC(ccExecCmd(L"read_con")); /* Read a line from the console. */
   ccETAC_REF->ccPop(); /* Discard the response from the user. */
   ...

ccEXITLBL:
   if ( ccERROR ) /* Detect and process error. */
   {
      ::etacProcessTACError(ccRTNCDE_VAR);
   }
```

Additional Information

Operational Definitions (ccETAC_REF)

3.4.6 Stack Access Macros

These macros pull and push values as *TAC objects* from and onto the *object stack* (typically). They automatically handle returned error code conditions.

ccPULL

ccPULL(**mArgs**)

Output

mArgs The arguments for `ccPull()` enclosed in parentheses.

Dependencies

`ccCALLTAC()`.

Action

Obtains the specified item (`mArgs`) from the top of the *object stack* (or *dictionary stack* as appropriate), and pops (deletes) that topmost *stack object* from the said stack. If the macro returns an error (`ccERROR`) then control will jump to the label defined by `ccEXITLBL`. Note that `mArgs` are the arguments to `ccPull()` and must be enclosed in parentheses, for example,

`ccPULL((MyInt))`

and

`ccPULL((MyCmd, ccTAC_CMD))`.

Additional Information

ccPull

ccPUSH

ccPUSH(**mArgs**)

Input

mArgs The arguments for `ccPush()` enclosed in parentheses.

Dependencies

`ccCALLTAC()`.

Action

Pushes a *stack object* containing the specified item (`mArgs`) to the top of the *object stack* (or *dictionary stack* as appropriate). If the macro returns an error (`ccERROR`) then control will jump to the label defined by `ccEXITLBL`. Note that `mArgs` are the arguments to `ccPush()` and must be enclosed in parentheses, for example,

`ccPUSH((MyInt))`

and

`ccPUSH((MyCmd, ccTAC_CMD))`.

Additional Information

ccPush

3.4.7 Miscellaneous Macros

The following are miscellaneous macros.

ccSPCHAR

ccSPCHAR(**mCharStr**)

Input

mCharStr	A non-empty wide-character (wchar_t) string. Only the first character is used, which must be a Unicode supplementary plane character.

Action

Converts a Unicode® supplementary plane character to the corresponding Unicode code point compatible with ccCHAR. Only the first character of mCharStr is converted.

Example

```
ccCHAR CodePoint; /* Will contain the code point of the specified character. */
   ...
   CodePoint = ccSPCHAR(L"\x3CD8\x4FDF"); /* Green Apple (U+1F34F) as a surrogate pair. */
   /* CodePoint will contain the value 127823 (U+1F34F). */
   ...
```

3.5 ccTAC Class

The ccTAC C++ class represents the definition of the *ETAC interface*. The *ETAC interface* contains only C++ virtual functions that point directly into the *ETAC interpreter*, and is the initial means by which an application program and an *external TAC library* can communicate with the *ETAC interpreter*. Some member functions of the ccTAC class provide the means by which other *resource interfaces* can be created. The *ETAC interface* is not allocated or released by the C++ programmer.

An instance of a ccTAC class (the *ETAC interface*) can be obtained in the following ways:

1. Automatically passed to an *ETL function* by the *ETAC interpreter*.
2. Returned by the ccGetTACIF() member function of ccTAC, and passed to an *ETL function*.
3. Returned by the etacSetAppETAC() function defined in AppETAC.dll (for use in an application program).
4. Automatically passed to the *call-back function* of an application program.

The ccTAC class is defined in the file ExternTACLib_***n***.h, which must be included in C++ source code for creating *external TAC libraries*. The pre-processor definition ccTAC_VRSN (defined in ExternTACLib_***n***.h) defines the number ***n*** which indicates the version of the ccTAC class in use. That number is the same as the ***n*** in AppETAC_***n***.h, which must be included in C++ source code for application programs. The inclusion of those two files ensures that the appropriate version of the ccTAC class is issued by the *ETAC interpreter*.

3.5.1 Function Summary

The following list is a summary of the member functions of the ccTAC class.

ccTAC Class Function Summary

Function	Description
ccCountToMark	Determines the number of top *stack objects* on the *object stack* up to but not including the mark *stack object*.
ccDeleteDict	Deletes a number of *dictionaries* on the *dictionary stack* at a position.
ccExecCmd	Executes a *command*.
ccExecETAC	Executes *ETAC code* directly from a disk file or memory.
ccGetDict	Gets the named *dictionary* and its position.
ccGetDictOfItem	Gets the *dictionary* (and its position) containing the named item.
ccGetObjType	Gets the *stack object* type of the top *stack object* on the *object stack*.
ccGetTACIF	Gets the appropriate *ETAC interface* for use with calling functions in a loaded *external TAC library*.
ccNew	Creates a *resource interface* to contain the *resource value* of a *stack object*.
ccPop	Pops (deletes) a number of top *stack objects* on the *object stack*.
ccPull	Pulls the value of a *stack object* from the *object stack* into a variable.
ccPush	Pushes the value of a variable onto the *object stack*.
ccRelease	Releases a *resource interface*.

3.5.2 Member Functions

The following points apply to the member functions of the `ccTAC` class.

- For functions that return a `cc_rtn_code` type, a success return code has the value of `ccTAC_RTN_SUCCESS`.
- For functions that return a `cc_rtn_code` type, the return code should be returned to the function's caller.
- The ccCALLTAC macro can be used for functions that return a `cc_rtn_code` type.
- *Stack object* types are automatically checked when being pulled from a *TAC stack* into the corresponding variable — the return code will indicate an error if the *stack object* is not of the type requested in the program.
- A pointer variable to a *resource interface* must initially be initialised with `NULL`.
- Variables for integral *stack object* values do not use *resource interfaces*; such variables are defined as `ccINT` (these are *stack objects* of type: `ccTAC_INT`, `ccTAC_CMDI`, `ccTAC_OPRI`, `ccTAC_MARK`, `ccTAC_NULL`, `ccTAC_EXE`).
- A variable for a decimal *stack object* value does not use a *resource interface*; such a variable is defined as `ccDEC` (*stack object* type is: `ccTAC_DEC`).
- The order of the member functions declared in class `ccTAC` must not be altered except as explained in Appendix A: Compatibility Issues.

The definitions of the `ccTAC` class member functions are as follows.

ccTAC::ccCountToMark

`ccULONG` **ccCountToMark**`(ccULONG` **pMarkNum** `= 0)`

Input

pMarkNum	The number (from 0 to 7) of a mark *stack object* on the *object stack*.

Return
The number of top *stack objects* on the *object stack*.

Action
Determines the number of top *stack objects* on the *object stack* up to but not including the specified (`pMarkNum`) mark *stack object*. The input number (`pMarkNum`) is not included in the count.

ccTAC::ccDeleteDict

```
void ccDeleteDict(ccULONG pAmount = 1, ccULONG pPosition = 1)
```

Input

pAmount	The number of *dictionaries* to delete from the *dictionary stack*. This number can be 0.
pPosition	The position of the first *dictionary* to delete from the top of the *dictionary stack*. The topmost *dictionary* is at position 1.

Action
Deletes a number (`pAmount`) of *dictionaries* on the *dictionary stack* at a position (`pPosition`). If `pAmount` is 0, or `pPosition` is 0 or greater than the number of *dictionaries* on the *dictionary stack* then no action occurs. If `pAmount` is greater than the number of *dictionaries* at and after the specified position, then the remaining *dictionaries* at and after the specified position are deleted.

ccTAC::ccExecCmd

```
cc_rtn_code ccExecCmd(ccSTR pCommand)
cc_rtn_code ccExecCmd(cc_tac_val pIntrCmd)
```

Input

pCommand	The name of the *command* to execute, or NULL.
pIntrCmd	The code (ccTAC_I_*) for an *intrinsic command*.

Return
Return code indicating the success or otherwise of the function.

Action
Syntax 1
Executes a named *command* (`pCommand`) as though it were *activated* from *ETAC code*. If `pCommand` is an empty string or NULL, no action occurs and the return code will indicate success.

Syntax 2
Executes an *intrinsic command* (`pIntrCmd`) as though it were *activated* from *ETAC code*. The codes for *intrinsic commands* are specified in Intrinsic Command Codes.

Additional Information
Intrinsic Command Codes

ccTAC::ccExecETAC

```
cc_rtn_code ccExecETAC(ccSTR pETACCodeFile)
cc_rtn_code ccExecETAC(ccMemoryBlock *pMemory)
```

Input

pETACCodeFile	The file path specification of a file containing *ETAC code*, or NULL.

pMemory The memory block containing *ETAC code*, or NULL.

Return
Return code indicating the success or otherwise of the function.

Action
Syntax 1
Executes *ETAC code* directly from a disk file (pETACCodeFile) as though it were *activated* by the **exec_tac** *command*. If pETACCodeFile is an empty string or NULL, no action occurs and the return code will indicate success.

Syntax 2
Executes *ETAC code* existing in a memory block (pMemory) as though it were *activated* by the **exec_tac** *command*. If pMemory is NULL, no action occurs and the return code will indicate success.

ccTAC::ccGetDict

```
ccULONG ccGetDict(ccDictionary *&pDictionary, ccSTR pDictName,
    int pInstance = 1)
```

Input
pDictName The name of the *dictionary* to get from the *dictionary stack*, or NULL.
pInstance A positive or negative integer indicating which instance of the named *dictionary* to find (must not be 0).

Output
pDictionary The requested *dictionary*.

Return
Returns the position of the found *dictionary* from the top of the *dictionary stack* (topmost *dictionary* is at position 1). Otherwise returns 0 if the *dictionary* was not found.

Action
Gets the named (pDictName) *dictionary* (pDictionary) and its position. If pInstance is positive (n), the function will search for the n^{th} instance of the *dictionary* with name pDictName from the top of the *dictionary stack*. If pInstance is negative ($-n$), the function will search for the n^{th} instance of the *dictionary* with name pDictName from the bottom of the *dictionary stack*.

If pDictName is an empty string or NULL, an unnamed *dictionary* will be searched for. If pInstance is 0, the consequence is undefined.

ccTAC::ccGetDictOfItem

```
ccULONG ccGetDictOfItem(ccDictionary *&pDictionary, ccSTR pItemName,
    int pInstance = 1)
```

Input
pItemName The name of the *dictionary item* to search for on the *dictionary stack*, or NULL.
pInstance A positive or negative integer indicating which instance of the named *dictionary item* to find on the *dictionary stack*.

Output
pDictionary The requested *dictionary* containing the specified instance of the *dictionary item*.

Return
Returns the position of the found *dictionary* from the top of the *dictionary stack* (topmost *dictionary* is position 1). Otherwise returns 0 if the *dictionary item* was not found.

Action
Gets the *dictionary* (pDictionary) containing the named *dictionary item* (pItemName) and the position of that *dictionary*. If pInstance is positive (n), the function will search for the n^{th} instance of the *dictionary item* with name pItemName from the top of the *dictionary stack*. If pInstance is negative ($-n$), the function will search for the n^{th} instance of the *dictionary item* with name pItemName from the bottom of the *dictionary stack*.

If pItemName is an empty string or NULL, no action occurs and the returned value will be 0. If pInstance is 0, the consequence is undefined.

ccTAC::ccGetObjType

`cc_rtn_code ccGetObjType(cc_tac_type &pType)`

Output

pType	Type of topmost *stack object* on the *object stack*.

Return
Return code indicating the success or otherwise of the function.

Action
Retrieves the *stack object* type (pType) of the topmost *stack object* on the *object stack*. The topmost *stack object* remains on the *object stack*.

Additional Information
TAC Object Types

ccTAC::ccGetTACIF

`ccTAC *ccGetTACIF(HMODULE pTACLibHndl, long pTACIFVrsn = 0)`

Input

pTACLibHndl	The handle of a loaded *external TAC library*, or NULL.
pTACIFVrsn	The version number of an *ETAC interface*, or 0. Version numbers begin with 1.

Return
A pointer to the requested *ETAC interface*, or NULL if the *ETAC interface* could not be obtained.

Action
Gets the appropriate *ETAC interface* for use with calling *ETL functions* in a loaded *external TAC library*. If pTACLibHndl is NULL and pTACIFVrsn is 0 then no action occurs and NULL is returned. If pTACLibHndl is not NULL then pTACIFVrsn is ignored. pTACLibHndl contains the handle to the *external TAC library* obtained from the operating system functions GetModuleHandle() or LoadLibrary(). If the version number of the *ETAC interface* for use with the *external TAC library* is known (for certain), then pTACIFVrsn can specify that version number and pTACLibHndl is set to NULL. The returned *ETAC interface* is used to call *ETL functions* in the specified *external TAC library*.

Additional Information
2.5 Calling ETL Functions from C++

ccTAC::**ccNew**

```
void ccNew(ccStackObj *&pStackObj)
void ccNew(ccString *&pString)
void ccNew(ccSequence *&pSequence)
void ccNew(ccMemory *&pMemory)
void ccNew(ccDictionary *&pDictionary)
```

Output

pStackObj	A pointer to a *resource interface* for a *stack object*.
pString	A pointer to a *resource interface* for a string value.
pSequence	A pointer to a *resource interface* for a *sequence resource value*.
pMemory	A pointer to a *resource interface* for a memory *resource value*.
pDictionary	A pointer to a *resource interface* for a *dictionary resource value*.

Action
Creates a new *resource interface* and corresponding empty *resource value* for the specified item, releasing the previous *resource interface* if there was one. The new *resource value* for `pStackObj` will be a null *stack object*. A variable designated to be used as a *resource interface* is declared as a pointer to that *resource interface*. That variable must be initialised with `NULL` before its first use (including before calling this function). The *resource interface* of the variable must be released (via `ccRelease()`) when no longer in use.

Related Information
ccRelease

Other Information
ccNEW ▪ ccNEW_STACKOBJ ▪ ccNEW_STRING ▪ ccNEW_MEMORY ▪ ccNEW_SEQUENCE ▪ ccNEW_DICTIONARY

ccTAC::**ccPop**

```
void ccPop(ccULONG pNumObjs = 1)
```

Input

pNumObjs	The number of top *stack objects* to delete from the *object stack*. This number can be 0.

Action
Pops (deletes) a number (`pNumObjs`) of top *stack objects* on the *object stack*.

ccTAC::**ccPull**

```
cc_rtn_code ccPull(ccStackObj *&pStackObj)
cc_rtn_code ccPull(ccINT &pInteger, cc_tac_type pType1 = ccTAC_INT)
cc_rtn_code ccPull(ccDEC &pDecimal)
cc_rtn_code ccPull(ccString *&pString, cc_tac_type pType2 = ccTAC_STR)
cc_rtn_code ccPull(ccSequence *&pSequence, cc_tac_type pType3 = ccTAC_SEQ)
cc_rtn_code ccPull(ccMemoryBlock *&pMemory)
cc_rtn_code ccPull(ccDictionary *&pDictionary, ccBOOL pDictStack = true)
```

Input

pType1	Type of topmost *stack object* expected on the *object stack*. Possible values are: ccTAC_INT, ccTAC_CMDI, ccTAC_OPRI, ccTAC_MARK, ccTAC_NULL, ccTAC_EXE.
pType2	Type of topmost *stack object* expected on the *object stack*. Possible values are: ccTAC_STR, ccTAC_CMD, ccTAC_OPR.
pType3	Type of topmost *stack object* expected on the *object stack*. Possible values are: ccTAC_SEQ, ccTAC_PROC.
pDictStack	Determines from which *TAC stack* the *dictionary resource value* is to be retrieved. A value of true indicates the *dictionary stack*, otherwise it indicates the *object stack*.

Output

pStackObj	Receives the topmost *stack object* on the *object stack*.
pInteger	Receives the integer value of the topmost *stack object* on the *object stack*.
pDecimal	Receives the decimal value of the topmost *stack object* on the *object stack*.
pString	Receives the string value of the topmost *stack object* on the *object stack*.
pSequence	Receives the *sequence resource value* of the topmost *stack object* on the *object stack*.
pMemory	Receives the memory *resource value* of the topmost *stack object* on the *object stack*.
pDictionary	Receives the *dictionary resource value* of the topmost *stack object*.

Return
Return code indicating the success or otherwise of the function.

Action
Each function obtains the specified item from the top of the specified *TAC stack*, and pops (deletes) that topmost *stack object* from that *TAC stack*. For output variables that point to a *resource interface*, that *resource interface* is automatically released prior to receiving the specified item in a new *resource interface*. If the topmost *stack object* is not of the same type as specified or implied by the function, then the function returns a non-success return code.

Additional Information
TAC Object Types

Other Information
ccPULL

ccTAC::ccPush

```
cc_rtn_code ccPush(ccStackObj *pStackObj)
cc_rtn_code ccPush(ccINT pInteger, cc_tac_type pType1 = ccTAC_INT)
cc_rtn_code ccPush(ccDEC pDecimal)
cc_rtn_code ccPush(ccSTR pString, cc_tac_type pType2 = ccTAC_STR)
cc_rtn_code ccPush(ccString *pString, cc_tac_type pType2 = ccTAC_STR)
cc_rtn_code ccPush(ccSequence *pSequence, cc_tac_type pType3 = ccTAC_SEQ)
cc_rtn_code ccPush(ccMemoryBlock *pMemory)
cc_rtn_code ccPush(ccDictionary *pDictionary, ccBOOL pDictStack = true)
```

Input

pStackObj	Contains a *stack object* to push onto the *object stack* (must not be NULL).
pInteger	Contains an integer value to push as a *stack object* onto the *object stack*.
pDecimal	Contains a decimal value to push as a *stack object* onto the *object stack*.
pString	Contains a string value to push as a *stack object* onto the *object stack*, or NULL (only if pType2 is ccTAC_STR).
pSequence	Contains a *sequence resource value* to push as a *stack object* onto the *object stack*, or NULL.
pMemory	Contains a memory *resource value* to push as a *stack object* onto the *object stack*, or NULL.
pDictionary	Contains a *dictionary resource value* to push as a *stack object* onto a *TAC stack*, or NULL.
pType1	Type of *stack object* to be pushed onto the *object stack*. Possible values are: ccTAC_INT, ccTAC_CMDI, ccTAC_OPRI, ccTAC_MARK, ccTAC_NULL, ccTAC_EXE.
pType2	Type of *stack object* to be pushed onto the *object stack*. Possible values are: ccTAC_STR, ccTAC_CMD, ccTAC_OPR.
pType3	Type of *stack object* to be pushed onto the *object stack*. Possible values are: ccTAC_SEQ, ccTAC_PROC.
pDictStack	Determines to which *TAC stack* the *dictionary* is to be pushed. A value of true indicates the *dictionary stack*, otherwise it indicates the *object stack*.

Return

Return code indicating the success or otherwise of the function.

Action

Each function pushes a *stack object* containing the specified item to the top of the specified *TAC stack*. For input *resource variables*, a *copy* of the *resource object* is pushed onto the specified *TAC stack*. For input *resource variables* that contain NULL, the value of the *stack object* pushed onto the *TAC stack* will be empty, except for a *stack object resource variable* (syntax 1) and for string variables if pType2 is not ccTAC_STR (syntaxes 4 and 5). In those cases, the function returns a non-success return code.

Additional Information

TAC Object Types

Other Information

ccPUSH

ccTAC::ccRelease

```
void ccRelease(ccStackObj *&pStackObj)
void ccRelease(ccString *&pString)
void ccRelease(ccSequence *&pSequence)
void ccRelease(ccMemory *&pMemory)
void ccRelease(ccDictionary *&pDictionary)
```

Input

pStackObj	A pointer to a *resource interface* for a *stack object*.
pString	A pointer to a *resource interface* for a string value.
pSequence	A pointer to a *resource interface* for a *sequence resource value*.
pMemory	A pointer to a *resource interface* for a memory *resource value*.
pDictionary	A pointer to a *resource interface* for a *dictionary resource value*.

Output

pStackObj	Set to NULL.
pString	Set to NULL.
pSequence	Set to NULL.
pMemory	Set to NULL.
pDictionary	Set to NULL.

Action
Releases the *resource interface* and its *resource object* for the specified item. No action occurs if the *resource variable* contains NULL. The *resource variable* is set to NULL before this function returns to the caller. The *resource interface* of a *resource variable* must be released when no longer in use.

Related Information
ccNew

Other Information
ccFREE

3.6 ccStackObj Class

The ccStackObj C++ class is the *resource interface* containing an internal *stack object*. The contained *stack object* is treated as a whole; the properties of the *stack object* are inaccessible. Each *stack object resource interface* always contains its own individual *stack object*, but the value of that *stack object* could be a *resource value*, meaning that it can be shared with the *resource values* in other *resource interfaces* and *stack objects* of the same type. This class exposes only C++ virtual functions that point directly into the *ETAC interpreter*.

The ccStackObj class is defined in the file ExternTACLib_***n***.h which must be included in C++ source code for creating *external TAC libraries*. The pre-processor definition ccTAC_VRSN (defined in ExternTACLib_***n***.h) defines the number ***n*** which indicates the version of the ccStackObj class in use. That number is the same as the ***n*** in AppETAC_***n***.h which must be included in C++ source code for application programs. The inclusion of those two files ensures that the appropriate version of the ccStackObj class is created by the *ETAC interpreter*.

3.6.1 Function Summary

The following list is a summary of the member functions of the ccStackObject class.

ccStackObject Class Function Summary

Function	Description
soCopyObj	Copies the *stack object* to another.
soDuplicateObj	Duplicates the *stack object* to another.

3.6.2 Member Functions

The definitions of the `ccStackObject` class member functions are as follows.

ccStackObj::soCopyObj

`cc_rtn_code` **soCopyObj**`(ccStackObj *&`**pStackObj**`)`

Output

`pStackObj` Receives a *copy* of the *stack object* contained in this *resource interface*.

Return

Return code indicating the success or otherwise of the function.

Action

Copies the *stack object* contained in this *resource interface* to the *stack object* contained in another *resource interface* (`pStackObj`). If `pStackObj` contains `NULL`, a *resource interface* for `pStackObj` is allocated before the *copy*.

ccStackObj::soDuplicateObj

`cc_rtn_code` **soDuplicateObj**`(ccStackObj *&`**pStackObj**`)`

Output

`pStackObj` Receives a *duplicate* of the *stack object* contained in this *resource interface*.

Return

Return code indicating the success or otherwise of the function.

Action

Duplicates the *stack object* contained in this *resource interface* to the *stack object* contained in another *resource interface* (`pStackObj`). If `pStackObj` contains `NULL`, a *resource interface* for `pStackObj` is allocated before the *duplication*.

3.7 ccString Class

The `ccString` class is the *resource interface* containing an internal *stack object* Unicode string value. The contained string is modified by the *resource interface* member functions. Each string *resource interface* always contains its own individual string (strings are not a shared resource). This class exposes only C++ virtual functions that point directly into the *ETAC interpreter*.

The `ccString` class is defined in the file `ExternTACLib_`***n***`.h` which must be included in C++ source code for creating *external TAC libraries*. The pre-processor definition `ccTAC_VRSN` (defined in `ExternTACLib_`***n***`.h`) defines the number ***n*** which indicates the version of the `ccString` class in use. That number is the same as the ***n*** in `AppETAC_`***n***`.h` which must be included in C++ source code for application programs. The inclusion of those two files ensures that the appropriate version of the `ccString` class is created by the *ETAC interpreter*.

3.7.1 Function Summary

The following list is a summary of the member functions of the `ccString` class.

ccString Class Function Summary

Function	Description
strAppend	Appends a specified substring to the string.
strAssign	Assigns a specified substring to the string.
strDeleteStr	Deletes a substring from the string.
strFindAndRepStr	Replaces each occurrence of a specified substring with a string.
strGetChar	Gets a character from the string.
strGetStrBuff	Gets a pointer to the string buffer which can be modified.
strGetStrPtr	Gets a temporary pointer to the raw string.
strInsertStr	Inserts a specified string into the string.
strLength	Gets the length of the string.
strPutChar	Replaces a character in the string.
strReleaseStrBuff	Releases the string buffer obtained via `strGetStrBuff()`.
strStrip	Removes specified characters from the string.
strUCharCount	Determines the number of *u-char* characters in the string.
strWCharCount	Determines the number of *w-char* characters for a given number of *u-char* characters the string.

3.7.2 Member Functions

The definitions of the `ccString` class member functions are as follows.

ccString::strAppend

```
cc_rtn_code strAppend(ccSTR pString, ccULONG pOffset = 0,
    ccULONG pLength = ccS_REMAINING_LENGTH)
cc_rtn_code strAppend(ccString *pString, ccULONG pOffset = 0,
    ccULONG pLength = ccS_REMAINING_LENGTH)
cc_rtn_code strAppend(ccCHAR pCharacter)
```

Input

`pString`	Contains the string value (or part thereof) to append, or `NULL`.
`pOffset`	Zero-based *w-char* character offset into the string to append.
`pLength`	Length of the substring (in *w-char* characters) to append, or `ccS_REMAINING_LENGTH`.
`pCharacter`	*u-char* character, as a Unicode scalar value, to append.

Return

Return code indicating the success or otherwise of the function.

Action

Appends a substring of a string (`pString`) beginning at an offset (`pOffset`) and of a specified length (`pLength`) to the end of the string value contained in this *resource interface*. If `pString` is `NULL`, or `pOffset` is out of range, then no action occurs. A value of `ccS_REMAINING_LENGTH` for `pLength` indicates the rest of the string from the offset.

Also appends a *u-char* character to the end of the string value contained in this *resource interface*.

Note that the substring specified by `pOffset` and `pLength` must be a well-formed Unicode substring, and `pCharacter` must be a valid Unicode scalar value, otherwise `ccTAC_RTN_BAD_OBJ_VAL` will be returned.

ccString::strAssign

```
cc_rtn_code strAssign(ccSTR pString, ccULONG pOffset = 0,
    ccULONG pLength = ccS_REMAINING_LENGTH)
cc_rtn_code strAssign(ccString *pString, ccULONG pOffset = 0,
    ccULONG pLength = ccS_REMAINING_LENGTH)
```

Input

`pString`	Contains the string value (or part thereof) to assign, or `NULL`.
`pOffset`	Zero-based *w-char* character offset into the string to assign.
`pLength`	Length of the substring (in *w-char* characters) to assign, or `ccS_REMAINING_LENGTH`.

Return
Return code indicating the success or otherwise of the function.

Action
Replaces the string value contained in this *resource interface* with a substring of a string (`pString`) beginning at an offset (`pOffset`) and of a specified length (`pLength`). If `pString` is `NULL`, or `pOffset` is out of range, then an empty string is assumed for `pString`. A value of `ccS_REMAINING_LENGTH` for `pLength` indicates the rest of the string from the offset.

Note that the substring specified by `pOffset` and `pLength` must be a well-formed Unicode substring, otherwise `ccTAC_RTN_BAD_OBJ_VAL` will be returned.

ccString::strDeleteStr

```
cc_rtn_code strDeleteStr(ccULONG pOffset = 0,
    ccULONG pLength = ccS_REMAINING_LENGTH)
```

Input

`pOffset`	Zero-based *w-char* character offset into this string to delete.
`pLength`	Length of the substring (in *w-char* characters) to delete, or `ccS_REMAINING_LENGTH`.

Return
Return code indicating the success or otherwise of the function.

Action
Deletes the substring of the string value contained in this *resource interface* beginning at an offset (`pOffset`) and of a specified length (`pLength`). A value of `ccS_REMAINING_LENGTH` for `pLength` indicates the rest of the string from the offset. If `pOffset` is out of range then no action occurs.

Note that the substring specified by `pOffset` and `pLength` must be a well-formed Unicode substring, otherwise `ccTAC_RTN_BAD_OBJ_VAL` will be returned.

ccString::strFindAndRepStr

`cc_rtn_code strFindAndRepStr(ccSTR pSearchStr, ccSTR pReplStr)`

Input

pSearchStr	Substring to search for, or NULL.
pReplStr	String to replace the found substring, or NULL.

Return

Return code indicating the success or otherwise of the function.

Action

Searches the string value contained in this *resource interface* for the existence of a specified substring (pSearchStr) and replaces each occurrence of the substring with a string (pReplStr). If either pSearchStr or pReplStr is NULL, or the substring is not found, then no action occurs.

Note that the strings specified by pSearchStr and pReplStr must be a well-formed Unicode strings, otherwise ccTAC_RTN_BAD_OBJ_VAL will be returned.

ccString::strGetChar

`ccCHAR strGetChar(ccINT pOffset)`

Input

pOffset	Zero-based *u-char* character offset into this string.

Return

The *u-char* character at the specified offset, or a null-terminator character (U+0000).

Action

Gets the character at an offset (pOffset) into the string value contained in this *resource interface*. If pOffset is a non-negative number then the offset is from the beginning of the string, otherwise it is from the end of the string (-1 is the last *u-char* character of the string). If pOffset is out of range then a null-terminator character (U+0000) is returned.

ccString::strGetStrBuff

`ccMSTR strGetStrBuff(ccULONG pMinLen = 0)`

Input

pMinLen	The minimum size (in *w-char* characters) of the returned buffer. This value does not include space for a null-terminator.

Return

A pointer to the requested string buffer containing the null-terminated string.

Action

Gets a pointer to the raw string buffer of the string value contained in this *resource interface*. The size of the buffer is large enough to hold the string value, or the size specified in pMinLen if that size is larger. The contents of the string buffer can be directly modified via the returned pointer (but not modified beyond the size of the buffer). A null-terminating character (0x00) can be inserted into the buffer to indicate the end of the new modified string.

When finished with the string buffer, or before using any other ccString member function for any string, strReleaseStrBuff() <u>must</u> be called.

Related Information
strReleaseStrBuff

ccString::strGetStrPtr

```
ccSTR strGetStrPtr()
```

Return
A temporary pointer to the raw (null-terminated) string value contained in this *resource interface*.

Action
Gets a pointer to the raw string value contained in this *resource interface*. The string value **MUST NOT BE MODIFIED IN ANY WAY**. The returned pointer is temporary, and will become invalid if any other `ccString` member function for any string is called. The returned pointer is typically used only for reading the string value for use with string functions that require a raw null-terminated string.

ccString::strInsertStr

```
cc_rtn_code strInsertStr(ccULONG pOffset, ccSTR pInsStr,
    ccULONG pLength = ccS_REMAINING_LENGTH, ccCHAR pPad = L' ')
```

Input

`pOffset`	Zero-based *w-char* character offset into this string.
`pInsStr`	The string to insert.
`pLength`	Length of the initial substring (in *w-char* characters) of the string to insert, or `ccS_REMAINING_LENGTH`.
`pPad`	A UCS-2 pad character.

Return
Return code indicating the success or otherwise of the function.

Action
Inserts the initial substring of the specified character length (`pLength`) of a string (`pInsStr`) into the string value contained in this *resource interface* beginning at an offset (`pOffset`). If the value of `pLength` is greater than the *w-char* length of `pInsStr` then the extra places at the end of `pInsStr` will effectively be filled with the pad character (`pPad`) before the insertion. The pad character must be a UCS-2 (BMP Unicode scalar value) character, otherwise `ccTAC_RTN_GENERAL_ERR` will be returned. `pInsStr` itself will not be altered. A value of `ccS_REMAINING_LENGTH` for `pLength` indicates the length of `pInsStr`. `pOffset` contains a zero-based character offset into this string value. The first character inserted into this string value will be at offset `pOffset`. If `pOffset` is greater than the length of this string value then the extra places between the end of this string value and the position at `pOffset` will be filled with the pad character.

Note that the substring specified by the first `pLength` characters of `pInsStr` must be a well-formed Unicode substring, and `pOffset` must address a valid *u-char* character, otherwise `ccTAC_RTN_BAD_OBJ_VAL` will be returned.

ccString::strLength

```
ccULONG strLength()
```

Return
The *w-char* character length of this string.

Action
Returns the *w-char* character length of the string value contained in this *resource interface*. An empty string will return a length of 0.

Other Information
strUCharCount ▪ strWCharCount

ccString::strPutChar

`cc_rtn_code strPutChar(ccCHAR pCharacter, ccINT pOffset)`

Input

pCharacter	*u-char* character, as a Unicode scalar value, to put.
pOffset	Zero-based *u-char* character offset into this string.

Return
Return code indicating the success or otherwise of the function.

Action
Replaces the *u-char* character at an offset (pOffset) of the string value contained in this *resource interface* with another character (pCharacter). If pOffset is a non-negative number then the offset is from the beginning of the string, otherwise it is from the end of the string (-1 is the last *u-char* character of the string). If pOffset is out of range then no action occurs.

Note that pCharacter must be a valid Unicode scalar value, otherwise ccTAC_RTN_BAD_OBJ_VAL will be returned.

ccString::strReleaseStrBuff

`void strReleaseStrBuff(ccULONG pNewLen = ccSTR_AUTOLEN)`

Input

pNewLen	The new *w-char* character length of the released buffer, or ccSTR_AUTOLEN.

Action
Releases the raw string buffer of the string value of this *resource interface* returned from a call to strGetStrBuff(). If a null-terminator character (U+0000) was not inserted into the buffer, but the desired character length of the string in the buffer is different from the original length, then pNewLen must contain the desired character length of the string. Otherwise, the default value (ccSTR_AUTOLEN) of pNewLen should be used. If the value of pNewLen is ccSTR_AUTOLEN then the string in the buffer is expected to be null-terminated (which indicates the length of that string).

Related Information
strGetStrBuff

ccString::strStrip

`void strStrip(ccCHAR pMode = L'B', ccSTR pList = L" ")`

Input

pMode	Determines the location of characters to strip from this string. Possible values are: 'L', 'R', 'B', 'A'.
pList	A list of *u-char* characters to strip from this string.

Action
Strips off *u-char* characters specified in a list (pList) from the string value of this *resource interface*. pList contains the characters to strip. Each character in pList is removed from the string value depending on the value of pMode. pMode specifies from which part of this string value to strip the characters. pMode is interpreted as follows:

pMode	Action
'L'	(**L**eft) Remove all leading characters as specified in pList.
'R'	(**R**ight) Remove all trailing characters as specified in pList.
'B'	(**B**oth) Remove all leading and trailing characters as specified in pList.
'A'	(**A**ll) Remove all characters as specified in pList.

If the value of pMode is not one of the above characters then no action occurs.

ccString::strUCharCount

```
ccULONG strUCharCount()
```

Return
The *u-char* character length of this string.

Action
Returns the *u-char* character length of the string value contained in this *resource interface*. An empty string will return a length of 0.

Other Information
strLength ▪ strWCharCount

ccString::strWCharCount

```
cc_rtn_code strWCharCount(ccULONG &pCount, ccINT pStartOff,
    ccINT pNumUChars = -1)
```

Input

pStartOff	Zero-based *w-char* or *u-char* character offset into this string.
pNumUChars	The number of *u-char* characters in the specified substring of this string.

Output

pCount	*w-char* character length of the specified substring.

Return
Return code indicating the success or otherwise of the function.

Action
Returns the *w-char* character length (pCount) of the string value contained in this *resource interface*, beginning at a *w-char* or *u-char* offset (pStartOff) and a subsequent number (pNumUChars) of *u-char* characters from that offset. If pStartOff is a non-negative number then it is a *w-char* offset into this string. If pStartOff is a negative number then the negation of pStartOff is a *u-char* offset into this string. pNumUChars is the number of *u-char* characters specifying the substring of this string beginning at the effective offset implied by pStartOff. A value of -1 for pNumUChars indicates the rest of the characters in this string from the effective offset.

Note that the substring specified by pStartOff and pNumUChars must be a well-formed Unicode substring, otherwise ccTAC_RTN_BAD_OBJ_VAL will be returned.

Other Information
strLength ▪ strUCharCount

3.8 ccMemoryBlock Class

The `ccMemoryBlock` class is the *resource interface* containing an internal *stack object* memory *resource value*. The contained *resource value* (sometimes called a 'memory block') is modified by the *resource interface* member functions. The *resource value* of a memory *resource interface* can be shared with other memory *resource interfaces* and memory *stack objects*. This class exposes only C++ virtual functions that point directly into the *ETAC interpreter*.

The `ccMemoryBlock` class is defined in the file ExternTACLib_***n***.h which must be included in C++ source code for creating *external TAC libraries*. The pre-processor definition `ccTAC_VRSN` (defined in ExternTACLib_***n***.h) defines the number ***n*** which indicates the version of the `ccMemoryBlock` class in use. That number is the same as the ***n*** in AppETAC_***n***.h which must be included in C++ source code for application programs. The inclusion of those two files ensures that the appropriate version of the `ccMemoryBlock` class is created by the *ETAC interpreter*.

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.

3.8.1 Function Summary

The following list is a summary of the member functions of the `ccMemoryBlock` class.

ccMemoryBlock Class Function Summary

Function	Description
mbAllocate	Allocates or reallocates the memory block size, retaining the data if possible.
mbAppendMem	Appends another memory block to the memory block.
mbApplyBOM	Inserts or removes the actual BOM signature of the memory block.
mbCopyMem	Copies the whole memory block to another.
mbCvtDataTo	Converts the data to the specified *data form*.
mbDuplicateMem	Duplicates the whole memory block to another.
mbExport	Exports the data in the memory block to pre-allocated external memory or to a string.
mbGetDataPtr	Gets a temporary pointer to the memory block.
mbGetDataSize	Gets the size of the data in the memory block.
mbGetErrCode	Gets the error code of certain functions.
mbGetMemSize	Gets the size of the memory block.
mbImport	Appends a string to the data in the memory block.

mbInsert	Inserts an amount of data at the start of a memory block into the memory block at an offset.
mbLoad	Loads pre-allocated external memory into the memory block, overwriting existing data.
mbReadWholeFile	Loads the data in a disk file into the memory block. Overwrites existing data.
mbRepDataForm	Marks the memory object with the specified *data form*.
mbRepDstPath	Replaces the destination file path of the memory block.
mbRepSrcPath	Replaces the source file path of the memory block.
mbReserveExtraMem	Extends the size of the reserved memory in the memory block. The data size is unchanged.
mbSet	Sets all the data in the memory block to a byte value.
mbSetDataSize	Sets the size of the data in the memory block.
mbWriteWholeFile	Writes all the data in the memory block to a disk file.

3.8.2 Member Functions

The following points apply to the member functions of the `ccMemoryBlock` class.

- A 'memory block' is the actual total memory allocated.
- A *memory block* is automatically expanded to accommodate additional data.
- The 'useable data' in a *memory block* is the leading amount of data in that *memory block* that is considered valid; data after the *usable data* is considered to be garbage.
- The actual size of a *memory block* is equal to or larger than the *usable data* of that *memory block*.

The definitions of the `ccMemoryBlock` class member functions are as follows. Note that when modifying a memory *resource value*, memory *stack objects* in *ETAC code* and in other memory *resource interfaces* can refer to that *resource value*.

ccMemoryBlock::mbAllocate

`void mbAllocate(ccULONG pSize = ccM_DEFAULT_SIZE)`

Input

`pSize` Size (in bytes) of the allocated or reallocated memory block, or `ccM_DEFAULT_SIZE`.

Action

Allocates or reallocates the memory block of the memory *resource value* of this *resource interface* to the specified size (`pSize`), retaining the usable data if possible. The default size (`ccM_DEFAULT_SIZE`) is 50,000 bytes. If the value of `pSize` is smaller than the size of the usable data size, then the size of the usable data will be truncated to the value of `pSize`.

ccMemoryBlock::mbAppendMem

`void mbAppendMem(ccMemoryBlock *pMemory)`

Input

`pMemory` Contains the data to append to this memory block, or `NULL`.

Action
Appends the usable data of a memory block (pMemory) to the usable data of the memory *resource value* of this *resource interface*. pMemory points to the *resource interface* containing the data to append. If pMemory is NULL then no action occurs.

ccMemoryBlock::mbApplyBOM

`ccBOOL mbApplyBom(ccBOOL pInsert = true)`

Input

pInsert Indicates whether to insert (true) or remove (false) the BOM signature.

Return
Returns true if pInsert is true and a BOM signature was inserted or if pInsert is false and a BOM signature was removed. Returns false in other cases.

Action
Inserts (if pInsert is true) or removes (if pInsert is false) the appropriate BOM (byte order mark) signature of the usable data of the memory *resource value* (memory block) of this *resource interface*. If pInsert is true, a BOM signature is inserted at the beginning of the memory block as appropriate for the memory block's *data form* after the removal of an existing BOM signature. If pInsert is false, an existing BOM signature is removed from the memory block. If the memory block's *data form* is ccMO_BIN or ccMO_TXT, no action occurs and false is returned.

The Unicode® BOM signatures are defined as follows for the stated encoding scheme. A BOM signature must be at the beginning of the usable data of the memory block for it to be recognised.

UTF-8: EFBBBF_{H}
UTF-16LE: FFFE_{H} **UTF-32LE**: $\mathrm{FFFE0000}_{H}$
UTF-16BE: FEFF_{H} **UTF-32BE**: $\mathrm{0000FEFF}_{H}$

ccMemoryBlock::mbCopyMem

`void mbCopyMem(ccMemoryBlock *&pDestMem)`

Output

pDestMem Receives a *copy* of this memory *resource object*.

Action
Copies the *resource object* of this *resource interface* to the *resource object* of another memory *resource interface* (pDestMem). If pDestMem contains NULL, a *resource interface* for pDestMem is allocated before the *copy*. If pDestMem and this *resource interface* contain the same memory *resource value* (memory block) then no action occurs. Both *resource interfaces* will share the same memory block after the *copy*.

ccMemoryBlock::mbCvtDataTo

`ccBOOL mbCvtDataTo(ccINT pDataForm)`

Input

pDataForm Indicates the *data form* to convert to. Possible values are: ccMO_BIN, ccMO_TXT, ccMO_U8, ccMO_U16_LE, ccMO_U16_BE, ccMO_U32_LE, ccMO_U32_BE, ccMO_NATIVE, ccMB_VERIFY, ccMB_DETERMINE.

Return
As describe under **Action**.

Action
Converts the usable data of the memory *resource value* (memory block) of this *resource interface* to the requested *data form* (pDataForm) as specified in the table Data Form Indicators. The consequence is undefined for invalid values of pDataForm. false is returned if an attempted conversion fails. Refer to the **cvt_data_to** *command* in "The Official ETAC Programming Language" (ETACProgLang(Official).pdf) for details of the conversion.

If the value of pDataForm is ccMB_VERIFY or the same as the internal *data form* indicator, then the usable data of the memory block is verified to conform to its internal *data form* indicator, and true is returned on a successful verification.

If the value of pDataForm is ccMB_DETERMINE, then the usable data of the memory block is (re)determined as is done for the mbReadWholeFile() function, and the usable data will not be modified except for the possible removal of the BOM signature. true is returned. The internal *data form* indicator and BOM status in the memory *resource value* may be modified.

Additional Information
mbReadWholeFile

ccMemoryBlock::mbDuplicateMem

`void mbDuplicateMem(ccMemoryBlock *&pDestMem)`

Output

pDestMem Receives a *duplicate* of this memory block.

Action
Duplicates the *resource object* of this *resource interface* to the *resource object* of another memory *resource interface* (pDestMem). If pDestMem contains NULL, a *resource interface* for pDestMem is allocated before the *duplication*. If pDestMem points to this *resource interface*, and the contained memory *resource value* (memory block) has no other *managed reference* pointing to it, then no action occurs. Otherwise, the *resource value* of pDestMem is released and a new one is allocated before the *duplication*, and the two *resource interfaces* (if they are not the same one) will not share the same memory block after the *duplication*. After this function completes, the memory block of pDestSeq will have only one *managed reference* pointing to it.

ccMemoryBlock::mbExport

`ccULONG mbExport(void *pDestMem, ccULONG pMemSize)`
`ccBOOL mbExport(ccString *&pString)`

Input

pMemSize The size (in bytes) of the external memory area to receive data.

Output

pDestMem A pointer to the external memory area to receive data (must not be NULL).

pString A string value to receive data.

Return
Syntax 1: The number of bytes copied to the external memory area.

Syntax 2: Returns true if the memory block data was converted and exported to pString successfully, otherwise returns false.

Action
Syntax 1
Copies the usable data of the memory block of this *resource interface* to pre-allocated external

memory (`pDestMem`), overwriting any existing data. It is important that `pDestMem` points to memory (allocated by the caller before this function is called) of at least the size contained in `pMemSize`, otherwise the consequence is unpredictable. The size of the data copied to the external memory will be the minimum of the size contained in `pMemSize` and the size of the usable data of this memory block. `pDestMem` must not be `NULL`, otherwise the consequence is unpredictable.

Syntax 2
Copies the usable data of the memory block of this *resource interface* to the string value of another *resource interface* (`pString`), overwriting the existing string value. The usable data of the memory block is assumed to contain a UTF or Windows-1252 string. The string in the memory block must occupy the whole of the usable data without containing a string terminator character (U+0000), otherwise the consequence is undefined. If necessary, an internal copy of the usable data is converted via the equivalent of `mbCvtDataTo(ccMO_NATIVE)`. If `pString` contains `NULL`, a *resource interface* for `pString` is allocated before the data is copied.

Additional Information
mbCvtDataTo

Other Information
mbGetDataSize

ccMemoryBlock::mbGetDataPtr

```
ccBYTE *mbGetDataPtr()
```

Return
A temporary pointer to the raw memory data contained in this *resource interface*.

Action
Gets a temporary pointer to the raw data in the memory block of this *resource interface*. The returned pointer is temporary, and will become invalid if any other `ccMemoryBlock` member function that could alter this memory block is called. The usable data of this memory block can be altered via the returned pointer, but the possibility of other *managed references* pointing to this memory block should be taken into account. Note that memory *stack objects* in *ETAC code* can refer to this memory block.

Other Information
mbGetDataSize

ccMemoryBlock::mbGetDataSize

```
ccULONG mbGetDataSize()
```

Return
The byte size of the usable data of the memory block contained in this *resource interface*.

Action
Gets the size of the usable data of the memory block contained in this *resource interface*. The size of the actual memory block can be larger than the size of the usable data of that memory block. The data after the usable data of the memory block is considered to be garbage.

Other Information
mbGetMemSize

ccMemoryBlock::**mbGetErrCode**

`ccINT mbGetErrCode()`

Return
Returns the current error code.

Action
Gets the error code after calling certain functions. The non-zero error code can be passed to `etacProcessTACError()`.

Related Information
etacProcessTACError

ccMemoryBlock::**mbGetMemSize**

`ccULONG mbGetMemSize()`

Return
The byte size of the memory block contained in this *resource interface*.

Action
Gets the size of the memory block contained in this *resource interface*. The size of the memory block can be greater than the size of the usable data of that memory block. The data after the usable data of the memory block is considered to be garbage.

Other Information
mbGetDataSize

ccMemoryBlock::**mbImport**

```
ccBOOL mbImport(ccSTR pString1, ccULONG pOffset = 0,
    ccULONG pStrLen = ccM_REMAINING_AMOUNT)
ccBOOL mbImport(ccString *pString2)
```

Input

`pString1`	Contains the string (or part thereof) to import, or `NULL`.
`pOffset`	Zero-based *w-char* character offset into the string to import.
`pStrLen`	Length of the substring (in *w-char* characters) to import, or `ccM_REMAINING_AMOUNT`.
`pString2`	Contains the string value to import, or `NULL`.

Return
Returns `true` if the specified string or substring was imported successfully, otherwise returns `false`.

Action
Appends a substring of a string (`pString1`) or whole string (`pString2`) to the end of the usable data of the memory block contained in this *resource interface*. The usable data size of this memory block is automatically expanded to accommodate the string.

Syntax 1
The string to append is a substring of `pString1` beginning at an offset (`pOffset`) and of a specified length (`pStrLen`). If `pOffset` is out of range then the consequence is unpredictable. If `pString1` is `NULL` then no action occurs. A value of `ccM_REMAINING_AMOUNT` for `pStrLen` indicates the rest of the string from the offset. An internal copy of the specified string or

substring may be automatically converted before being appended to the memory block data. Refer to the heading **Adding Data to Memory** under the **add** *operator* in "The Official ETAC Programming Language" (ETACProgLang(Official).pdf) for details of the conversion.

Note that the substring specified by `pOffset` and `pStrLen` must be a well-formed Unicode substring, otherwise `false` will be returned.

Syntax 2
The string to append is the string value contained in the *resource interface* pointed to by `pString2`. If `pString2` is `NULL` then no action occurs.

ccMemoryBlock::**mbInsert**

```
void mbInsert(ccMemoryBlock *pMemory, ccULONG pOffset = 0,
    ccULONG pStart = 0, ccULONG pAmount = ccM_REMAINING_AMOUNT)
```

Input

pMemory	Contains the memory block (or part thereof) to insert, or `NULL`.
pOffset	Offset (in bytes) of where the data is to be inserted into this memory block.
pStart	Offset (in bytes) into the source memory block (`pMemory`) of the data to insert.
pAmount	Size of the data (in bytes) to insert, or `ccM_REMAINING_AMOUNT`.

Action
Inserts an amount (`pAmount`) of usable data at an offset (`pStart`) of a memory block (`pMemory`) into the memory block contained in this *resource interface* at an offset (`pOffset`). The size of this memory block is automatically expanded to accommodate the inserted data if necessary. The value of `pOffset` must be less than the size of the usable data of this memory block, and the value of `pStart` must be less than the size of the usable data of the memory block of `pMemory`, otherwise the consequence is unpredictable. If `pMemory` is `NULL` then no action occurs.

Other Information
mbGetDataSize

ccMemoryBlock::**mbLoad**

```
void mbLoad(void *pMem, ccULONG pMemSize)
```

Input

pMem	Pointer to initialised raw external memory data to load, or `NULL`.
pMemSize	Size (in bytes) of the raw external memory data to load.

Action
Replaces the usable data of the memory block of this *resource interface* with the data in pre-allocated external memory (`pMem`) of the specified size (`pMemSize`). It is important that `pMem` points to memory (allocated by the caller before this function is called) of at least the size contained in `pMemSize`, otherwise the consequence is unpredictable. If `pMem` is `NULL` then the size of the usable data of this memory block becomes zero (the size of the actual memory block itself remains unaltered), and `pMemSize` is ignored.

ccMemoryBlock::**mbReadWholeFile**

```
ccULONG mbReadWholeFile(ccSTR pFilePath, ccULONG pFlags = 0x00000000)
ccULONG mbReadWholeFile(ccString *pFilePath, ccULONG pFlags = 0x00000000)
```

Input

`pFilePath`	Contains the file path of the file data to load, or `NULL`.
`pFlags`	A combination of the following binary flags: `ccRF_ERR_FATAL`, `ccRF_NO_DATA_CHK`, `ccRF_BOM`, `ccRF_NBOM`, `ccRF_RET_BOM_STATE`.

Return

The size of the loaded data in bytes, `-1`, or `0`.

Action

Loads all of the data in a disk file (`pFilePath`) into the usable data of the memory block of this *resource interface*, replacing the existing usable data. If `pFilePath` is `NULL` then no action occurs. The size of this memory block is automatically expanded to accommodate the loaded data if necessary. The internal input file path of the memory block will be set to the value of `pFilePath`. A fatal error (the application program or ETAC could terminate) will occur if the specified file cannot be read from or does not exist and `ccRF_ERR_FATAL` is set in `pFlags`.

`pFlags` contains a combination of the following values.

pFlags	**Meaning**
`ccRF_ERR_FATAL`	Causes a fatal error event that cannot be captured when a fatal error occurs. If this option is omitted, a fatal error event causes `-1` to be returned, and the error number can be obtained from the function `mbGetErrCode()`.
`ccRF_NO_DATA_CHK`	Reads the input file data without checking its *data form*. The memory block *data form* indicator will be `ccMO_BIN`, and the internal BOM status flag will be cleared. If this value is omitted, the *data form* of the input file data will be determined.
`ccRF_BOM`	Ignored for memory block *data form* indicators of `ccMO_BIN` and `ccMO_TXT`. Forces the internal BOM status flag of the memory block to be set. Mutually exclusive with `ccRF_NBOM`.
`ccRF_NBOM`	Ignored for memory block *data form* indicators of `ccMO_BIN` and `ccMO_TXT`. Forces the internal BOM status flag of the memory block to be cleared. Mutually exclusive with `ccRF_BOM`.
`ccRF_RET_BOM_STATE`	Returns the initial BOM status of the memory block. `-1` will be returned if the BOM status flag is set, otherwise `0` will be returned.

If `ccRF_RET_BOM_STATE` is set in `pFlags`, the return value could be `-1`, indicating either that the internal BOM status flag was set, or a fatal error event had occurred. To determine whether a fatal error event had occurred, call `mbGetErrCode()` which will return a non-zero error code (otherwise it returns zero if a fatal error event had not occurred). That non-zero error code can be passed to `etacProcessTACError()`. If a fatal error event had not occurred, the returned value indicates the BOM status before the `pFlags` option `ccRF_BOM` or `ccRF_NBOM` is applied, but after the file data is read.

If the specified file has no data (empty file), then the usable data in the memory block will be zero and the internal *data form* indicator of the memory block will be `ccMO_BIN`.

Refer to the **read_file** *command* in “The Official ETAC Programming Language” (ETACProgLang(Official).pdf) for details of the process of reading the file data via this function.

Additional Information
Unicode File Specification ▪ mbGetErrCode

Related Information
etacProcessTACError

ccMemoryBlock::mbRepDataForm

`ccINT mbRepDataForm(ccINT pDataForm, ccULONG pFlags = 0x00000000)`

Input

pDataForm	Value to set the internal *data form* indicator of the memory *resource value*. Possible values are: ccMO_BIN, ccMO_TXT, ccMO_U8, ccMO_U16_LE, ccMO_U16_BE, ccMO_U32_LE, ccMO_U32_BE, ccMO_NATIVE, ccMB_RETAIN, ccMB_RET_BOM_STATE.
pFlags	Binary flags indicating whether to set or clear the internal BOM status flag. Possible mutually exclusive values are: ccMO_BOM, ccMO_NBOM.

Return
The existing internal *data form* indicator or BOM status.

Action
Sets or returns the internal *data form* (pDataForm) and BOM status (pFlags) indicators of the memory *resource value* of this *resource interface*. The various *data forms* to set are specified in the table Data Form Indicators. The consequence is undefined for invalid values of pDataForm. The contents of the memory *resource value* is not modified, but the *data form* and\or BOM indicators internal to the memory *resource value* may be modified.

If the value of pDataForm is ccMB_RETAIN, the internal *data form* indicator of the memory *resource value* will remain unmodified.

If the value of pDataForm is ccMB_RET_BOM_STATE, the internal BOM status of the memory *resource value* is returned; the returned value will be -1 if the BOM status flag is set, otherwise it will be 0. The internal *data form* indicator of the memory *resource value* will remain unmodified.

The value of pFlags determines whether to set (ccMO_BOM) the internal BOM status flag of the memory *resource value*, or clear it (ccMO_NBOM). ccMO_BOM and ccO_NBOM are mutually exclusive. The value of pFlags will have no effect if the new or existing internal *data form* indicator is either ccMO_BIN or ccMO_TXT.

The internal *data form* indicator of a memory *stack object* is maintained automatically by other relevant functions and *commands*. This function should only be used in those cases when it is not possible for the internal *data form* indicator to be automatically maintained.

Warning

Changing the internal *data form* indicator of a memory *stack object* to a *data form* that is inconsistent with the actual *data form* of the memory data could result in errors at a later stage, or the consequence could be unpredictable. However, it is always safe to change the *data form* indicator of any memory *stack object* to ccMO_BIN.

ccMemoryBlock::**mbRepDstPath**

```
cc_rtn_code mbRepDstPath(ccString *&pOldPath, ccSTR pNewPath)
void mbRepDstPath(ccString *&pOldPath, ccString *pNewPath)
```

Input

pNewPath The new internal destination file path to set in the memory *resource value*, or NULL.

Output

pOldPath The existing internal destination file path in the memory *resource value* before it is altered.

Return

Return code indicating the success or otherwise of the function (for the first syntax only).

Action

Gets (pOldPath) or sets (pNewPath) the internal destination file path of the memory *resource value* of this *resource interface*. The internal destination file path remains unmodified if pNewPath is NULL. The internal destination file path is used by the mbWriteWholeFile() function.

Additional Information

Unicode File Specification

Related Information

mbWriteWholeFile

ccMemoryBlock::**mbRepSrcPath**

```
cc_rtn_code mbRepSrcPath(ccString *&pOldPath, ccSTR pNewPath)
void mbRepSrcPath(ccString *&pOldPath, ccString *pNewPath)
```

Input

pNewPath The new internal source file path to set in the memory *resource value*, or NULL.

Output

pOldPath The existing internal source file path in the memory *resource value* before it is altered.

Return

Return code indicating the success or otherwise of the function (for the first syntax only).

Action

Gets (pOldPath) or sets (pNewPath) the internal source file path of the memory *resource value* of this *resource interface*. The internal source file path remains unmodified if pNewPath is NULL. The internal source file path is used by the mbReadWholeFile() function.

Additional Information

Unicode File Specification

Related Information

mbReadWholeFile

ccMemoryBlock::**mbReserveExtraMem**

```
ccULONG mbReserveExtraMem(ccULONG pSize)
```

Input

pSize The amount (in bytes) of memory to extend this memory block.

Return

The new size of this memory block in bytes.

Action

Extends the size of the memory block of this *resource interface* by a specified amount (pSize). The size of the usable data of this memory block remains unchanged.

ccMemoryBlock::**mbSet**

```
void mbSet(ccBYTE pChar = '\0')
```

Input

pChar The byte value to set for this memory block.

Action

Sets all of the data of the memory block of this *resource interface* to the specified byte value (pChar). The size of the usable data of this memory block remains unchanged.

ccMemoryBlock::**mbSetDataSize**

```
ccULONG mbSetDataSize(ccULONG pSize)
```

Input

pSize The size (in bytes) to set for the usable data of this memory block.

Return

The new size of the usable data of the memory block contained in this *resource interface*.

Action

Sets the size (pSize) of the usable data of the memory block contained in this *resource interface*. If the value of pSize is greater than the size of the memory block, then the size of the usable data will be set to the size of the memory block. Note that if the value of pSize is greater than the size of the existing usable data, then the data beyond that existing usable data is considered to be garbage.

Other Information

mbGetDataSize ▪ mbGetMemSize ▪ mbReserveExtraMem

ccMemoryBlock::**mbWriteWholeFile**

```
ccBOOL mbWriteWholeFile(ccSTR pFilePath, ccBOOL pAppend = false,
    ccULONG pFlags = 0x00000000)
ccBOOL mbWriteWholeFile(ccString *pFilePath, ccBOOL pAppend = false,
    ccULONG pFlags = 0x00000000)
```

Input

pFilePath Contains the file path of the file to which data will be written, or NULL.

pAppend Determines whether to append (true) to or replace (false) the file data.

pFlags	A combination of the following binary flags: ccWF_ERR_FATAL, ccWF_WIN_1252, ccWF_BOM, ccWF_NBOM.

Return
Returns true if all the usable data of this *resource interface* was written to the specified file, otherwise returns false.

Action
Writes all the usable data of the memory block of this *resource interface* to a specified (pFilePath) disk file, appending to or replacing the existing file data as specified (pAppend). A fatal error (the application program or ETAC could terminate) will occur if the specified file cannot be written to. If the file does not exist, a new one will be created (this includes creating the relevant non-existing directories).

pFlags contains a combination of the following values.

pFlags	**Meaning**
ccWF_ERR_FATAL	Causes a fatal error event that cannot be captured when a fatal error occurs. If this option is omitted, a fatal error event causes false to be returned, and the error number can be obtained from the function mbGetErrCode().
ccWF_WIN_1252	Temporarily converts the memory data to Windows-1252 before writing if possible. No BOM signature is written if this option is specified and the conversion is successful.
ccWF_BOM	Ignored for memory block *data form* indicators of ccMO_BIN and ccMO_TXT. Forces a BOM signature to be written unless pAppend is true or the ccWF_WIN_1252 option is specified. Mutually exclusive with ccWF_NBOM. This option does not affect the internal BOM status of the memory block.
ccWF_NBOM	Ignored for memory block *data form* indicators of ccMO_BIN and ccMO_TXT. Prevents a BOM signature from being written. Mutually exclusive with ccWF_BOM. This option does not affect the internal BOM status of the memory block.

If the return value is false, either not all the memory block data was written to the file, or a fatal error event occurred. To determine whether a fatal error event had occurred, call mbGetErrCode() which will return a non-zero error code (otherwise it returns zero if a fatal error event had not occurred). That non-zero error code can be passed to etacProcessTACError().

pFilePath contains the file path of the file to be written to. If pFilePath is an empty string, then the internal input file path of the memory block is used as pFilePath. If pFilePath is NULL, then the internal output file path of the memory block is used instead. But if the internal output file path is empty, then the internal input file path of the memory block is used as pFilePath. The internal output file path of the memory block is set to the effective value of pFilePath. A fatal error event will occur if the effective value of pFilePath is an empty string.

Refer to the **write_file** *command* in “The Official ETAC Programming Language” (ETACProgLang(Official).pdf) for the effects of the ccWF_WIN_1252 option of pFlags and the internal BOM status of the memory block.

Additional Information
Unicode File Specification ▪ mbGetErrCode

Related Information
etacProcessTACError

3.9 ccSequence Class

The ccSequence class is the *resource interface* containing an internal *sequence resource value*. The contained *resource value* is modified by the *resource interface* member functions. The *resource value* of a *sequence resource interface* can be shared with other *sequence resource interfaces* and *sequences*. This class exposes only C++ virtual functions that point directly into the *ETAC interpreter*.

The ccSequence class is defined in the file ExternTACLib_***n***.h which must be included in C++ source code for creating *external TAC libraries*. The pre-processor definition ccTAC_VRSN (defined in ExternTACLib_***n***.h) defines the number ***n*** which indicates the version of the ccSequence class in use. That number is the same as the ***n*** in AppETAC_***n***.h which must be included in C++ source code for application programs. The inclusion of those two files ensures that the appropriate version of the ccSequence class is created by the *ETAC interpreter*.

3.9.1 Function Summary

The following list is a summary of the member functions of the ccSequence class.

ccSequence Class Function Summary

Function	Description
sAppendSeq	Appends another *sequence* to the *sequence*.
sCopySeq	Copies the whole *sequence* to another.
sDeleteAll	Deletes all the elements of the *sequence*.
sDeleteElms	Deletes a range of elements of the *sequence*.
sDuplicateSeq	Duplicates the whole *sequence* to another.
sGet	Gets the value of a *stack object* from the *sequence* into a variable.
sGetElmType	Gets the *stack object* type of an element in the *sequence*.
sInsert	Inserts a value in a variable into the *sequence*.
sPut	Replaces an element in the *sequence* with the value in a variable.
sSize	Gets the number of elements in the *sequence*.

3.9.2 Member Functions

The definitions of the ccSequence class member functions are as follows. Note that when modifying a *sequence resource value*, *sequences* in *ETAC code* and in other *sequence resource interfaces* can refer to that *resource value*.

ccSequence::**sAppendSeq**

```
void sAppendSeq(ccSequence *pSrcSeq, ccBOOL pCopyElms = true)
```

Input

pSrcSeq — Contains the elements of the *sequence resource value* to append to this *sequence resource value*, or NULL.

pCopyElms — Determines whether to *copy* (true) or *duplicate* (false) the source elements to this *sequence resource value*.

Action
Appends the elements of a *sequence resource value* (`pSrcSeq`) to the *sequence resource value* of this *resource interface*. If `pCopyElms` is `true`, then each element of `pSrcSeq` is *copied*, otherwise each element of `pSrcSeq` is *duplicated*, before being appended to the end of this *sequence*. If `pSrcSeq` is `NULL` then no action occurs.

ccSequence::sCopySeq

`void sCopySeq(ccSequence *&pDestSeq)`

Output

`pDestSeq` Receives a *copy* of this *sequence resource value*.

Action
Copies the *resource object* of this *resource interface* to the *resource object* of another *sequence resource interface* (`pDestSeq`). If `pDestSeq` contains `NULL`, a *resource interface* for `pDestSeq` is allocated before the *copy*. If `pDestSeq` and this *resource interface* contain the same *sequence resource value* then no action occurs. Both *resource interfaces* will share the same *sequence resource value* after the *copy*.

ccSequence::sDeleteAll

`void sDeleteAll()`

Action
Deletes all the elements of the *sequence resource value* of this *resource interface*.

ccSequence::sDeleteElms

`void sDeleteElms(ccULONG pIndex = ccLAST_IDX, ccULONG pAmount = 1)`

Input

`pIndex` The zero-based index of the first element to delete, or `ccLAST_IDX`.

`pAmount` The number of elements to delete from the specified index (`0` or greater).

Action
Deletes a number (`pAmount`) of elements of the *sequence resource value* of this *resource interface* beginning with the element at the specified index (`pIndex`). A value of `ccLAST_IDX` for `pIndex` indicates the last element in this *sequence resource value*. The value of `pAmount` can be greater than the number of elements after the first one to be deleted. If `pIndex` is out of range then no action occurs.

ccSequence::sDuplicateSeq

`void sDuplicateSeq(ccSequence *&pDestSeq)`

Output

`pDestSeq` Receives a *duplicate* of this *sequence resource value*.

Action
Duplicates the *resource object* of this *resource interface* to the *resource object* of another *sequence resource interface* (`pDestSeq`). If `pDestSeq` contains `NULL`, a *resource interface* for `pDestSeq` is allocated before the *duplication*. If `pDestSeq` points to this *resource interface*, and the contained *sequence resource value* has no other *managed reference* pointing to it, then no action occurs. Otherwise, the *resource value* of `pDestSeq` is released and a new one is allocated

before the *duplication*, and the two *resource interfaces* (if they are not the same one) will not share the same *sequence resource value* after the *duplication*. After this function completes, the *sequence resource value* of pDestSeq will have only one *managed reference* pointing to it.

ccSequence::sGet

```
cc_rtn_code sGet(ccStackObj *&pStackObj, ccULONG pIndex = ccLAST_IDX)
cc_rtn_code sGet(ccINT &pInteger, cc_tac_type pType1 = ccTAC_INT,
    ccULONG pIndex = ccLAST_IDX)
cc_rtn_code sGet(ccDEC &pDecimal, ccULONG pIndex = ccLAST_IDX)
cc_rtn_code sGet(ccString *&pString, cc_tac_type pType2 = ccTAC_STR,
    ccULONG pIndex = ccLAST_IDX)
cc_rtn_code sGet(ccSequence *&pSequence, cc_tac_type pType3 = ccTAC_SEQ,
    ccULONG pIndex = ccLAST_IDX)
cc_rtn_code sGet(ccMemoryBlock *&pMemory, ccULONG pIndex = ccLAST_IDX)
cc_rtn_code sGet(ccDictionary *&pDictionary, ccULONG pIndex = ccLAST_IDX)
```

Input

pType1	Type of *stack object* expected at the specified index. Possible values are: ccTAC_INT, ccTAC_CMDI, ccTAC_OPRI, ccTAC_MARK, ccTAC_NULL, ccTAC_EXE.
pType2	Type of *stack object* expected at the specified index. Possible values are: ccTAC_STR, ccTAC_CMD, ccTAC_OPR.
pType3	Type of *stack object* expected at the specified index. Possible values are: ccTAC_SEQ, ccTAC_PROC.
pIndex	Zero-based index of the element to extract, or ccLAST_IDX.

Output

pStackObj	Receives the *stack object* existing at the specified index.
pInteger	Receives the integer value existing at the specified index.
pDecimal	Receives the decimal value existing at the specified index.
pString	Receives the string value existing at the specified index.
pSequence	Receives the *sequence resource value* existing at the specified index.
pMemory	Receives the memory *resource value* existing at the specified index.
pDictionary	Receives the *dictionary resource value* existing at the specified index.

Return

Return code indicating the success or otherwise of the function.

Action

Each function obtains a *copy* of the value of the *stack object* at an index (pIndex) of the *sequence resource value* of this *resource interface* into the specified output variable. A value of ccLAST_IDX for pIndex indicates the last element in this *sequence resource value*. For output variables that point to a *resource interface*, that *resource interface* is automatically released prior to receiving the specified item in a new *resource interface*. If the element at pIndex is not of the same type as specified or implied by the function, or if pIndex is out of range, then the function returns a non-success return code.

Additional Information

TAC Object Types

ccSequence::sGetElmType

```
cc_rtn_code sGetElmType(cc_tac_type &pType, ccULONG pIndex = ccLAST_IDX)
```

Input

pIndex — The zero-based index of the *stack object* element from which to obtain the type, or ccLAST_IDX.

Output

pType — Receives the type of *stack object* existing at the specified index.

Return

Return code indicating the success or otherwise of the function.

Action

Gets the type (pType) of the *stack object* at an index (pIndex) of the *sequence resource value* of this *resource interface*. A value of ccLAST_IDX for pIndex indicates the last element in this *sequence resource value*. If pIndex is out of range then the function returns a non-success return code.

Additional Information

TAC Object Types

ccSequence::sInsert

```
ccBOOL sInsert(ccStackObj *pStackObj, ccULONG pIndex)
ccBOOL sInsert(ccINT pInteger, ccULONG pIndex,
    cc_tac_type pType1 = ccTAC_INT)
ccBOOL sInsert(ccDEC pDecimal, ccULONG pIndex)
ccBOOL sInsert(ccSTR pString, ccULONG pIndex,
    cc_tac_type pType2 = ccTAC_STR)
ccBOOL sInsert(ccString *pString, ccULONG pIndex,
    cc_tac_type pType2 = ccTAC_STR)
ccBOOL sInsert(ccSequence *pSequence, ccULONG pIndex,
    cc_tac_type pType3 = ccTAC_SEQ)
ccBOOL sInsert(ccMemoryBlock *pMemory, ccULONG pIndex)
ccBOOL sInsert(ccDictionary *pDictionary, ccULONG pIndex)
```

Input

pStackObj	Contains a *stack object* to insert at the specified index (should not be NULL).
pInteger	Contains an integer value to insert at the specified index.
pDecimal	Contains a decimal value to insert at the specified index.
pString	Contains a string value to insert at the specified index, or NULL (only if pType2 is ccTAC_STR).
pSequence	Contains a *sequence resource value* to insert at the specified index, or NULL.
pMemory	Contains a memory *resource value* to insert at the specified index, or NULL.
pDictionary	Contains a *dictionary resource value* to insert at the specified index, or NULL.
pType1	Type of *stack object* to be inserted at the specified index. Possible values are: ccTAC_INT, ccTAC_CMDI, ccTAC_OPRI, ccTAC_MARK, ccTAC_NULL, ccTAC_EXE.
pType2	Type of *stack object* to be inserted at the specified index. Possible values are: ccTAC_STR, ccTAC_CMD, ccTAC_OPR.

pType3	Type of *stack object* to be inserted at the specified index. Possible values are: ccTAC_SEQ, ccTAC_PROC.
pIndex	Zero-based index at which the element is to be inserted, or ccLAST_IDX.

Return

Returns true if the function succeeded, otherwise returns false.

Action

Each function inserts a *stack object* containing the specified item into the *sequence resource value* of this *resource interface* at an index (pIndex). Existing elements within this *sequence resource value* at and after the specified index (if any) will be moved to their next position before the insertion is made. No element within this *sequence resource value* is deleted. A value of ccLAST_IDX for pIndex indicates the last element in this *sequence resource value*. An element can be inserted after the last one in this *sequence resource value* by specifying an index value one greater than the last index value. For input *resource variables*, a *copy* of the *resource object* of that variable is inserted. For input *resource variables* that contain NULL, the inserted *stack object* will have an empty *resource value*, except for a *stack object resource variable* (syntax 1) and for string variables if pType2 is not ccTAC_STR (syntaxes 4 and 5). In those cases, the function returns false. If pIndex is out of range then the function returns false.

Additional Information

TAC Object Types

ccSequence::sPut

```
ccBOOL sPut(ccStackObj *pStackObj, ccULONG pIndex = ccNEXT_IDX)
ccBOOL sPut(ccINT pInteger, cc_tac_type pType1 = ccTAC_INT,
    ccULONG pIndex = ccNEXT_IDX)
ccBOOL sPut(ccDEC pDecimal, ccULONG pIndex = ccNEXT_IDX)
ccBOOL sPut(ccSTR pString, cc_tac_type pType2 = ccTAC_STR,
    ccULONG pIndex = ccNEXT_IDX)
ccBOOL sPut(ccString *pString, cc_tac_type pType2 = ccTAC_STR,
    ccULONG pIndex = ccNEXT_IDX)
ccBOOL sPut(ccSequence *pSequence, cc_tac_type pType3 = ccTAC_SEQ,
    ccULONG pIndex = ccNEXT_IDX)
ccBOOL sPut(ccMemoryBlock *pMemory, ccULONG pIndex = ccNEXT_IDX)
ccBOOL sPut(ccDictionary *pDictionary, ccULONG pIndex = ccNEXT_IDX)
```

Input

pStackObj	Contains a *stack object* to put at the specified index (should not be NULL).
pInteger	Contains an integer value to put at the specified index.
pDecimal	Contains a decimal value to put at the specified index.
pString	Contains a string value to put at the specified index, or NULL (only if pType2 is ccTAC_STR).
pSequence	Contains a *sequence resource value* put at the specified index, or NULL.
pMemory	Contains a memory *resource value* put at the specified index, or NULL.
pDictionary	Contains a *dictionary resource value* put at the specified index, or NULL.
pType1	Type of *stack object* to be put at the specified index. Possible values are: ccTAC_INT, ccTAC_CMDI, ccTAC_OPRI, ccTAC_MARK, ccTAC_NULL, ccTAC_EXE.
pType2	Type of *stack object* to be put at the specified index. Possible values are: ccTAC_STR, ccTAC_CMD, ccTAC_OPR.

pType3	Type of *stack object* to be put at the specified index. Possible values are: ccTAC_SEQ, ccTAC_PROC.
pIndex	Zero-based index at which the element is to be put, or can be ccNEXT_IDX or ccLAST_IDX.

Return

Returns true if the function succeeded, otherwise returns false.

Action

Each function replaces the element of the *sequence resource value* of this *resource interface* at an index (pIndex) with a *stack object* containing the specified item. A value of ccNEXT_IDX for pIndex indicates the next element after the last one in this *sequence resource value*; ccLAST_IDX for pIndex indicates the last element. For input *resource variables*, a *copy* of the *resource object* of that variable is put. For input *resource variables* that contain NULL, the inserted *stack object* will have an empty *resource value*, except for a *stack object resource variable* (syntax 1) and for string variables if pType2 is not ccTAC_STR (syntaxes 4 and 5). In those cases, the function returns false. If pIndex is out of range then the function returns false.

Additional Information

TAC Object Types

ccSequence::**sSize**

```
ccULONG sSize()
```

Return

The number of element in the *sequence resource value* contained in this *resource interface*.

Action

Gets the number of elements in the *sequence resource value* contained in this *resource interface*.

3.10 ccDictionary Class

The ccDictionary class is the *resource interface* containing an internal *dictionary resource value*. The contained *resource value* is modified by the *resource interface* member functions. The *resource value* of a *dictionary resource interface* can be shared with other *dictionary resource interfaces* and *dictionaries*. This class exposes only C++ virtual functions that point directly into the *ETAC interpreter*.

The ccDictionary class is defined in the file ExternTACLib_***n***.h which must be included in C++ source code for creating *external TAC libraries*. The pre-processor definition ccTAC_VRSN (defined in ExternTACLib_***n***.h) defines the number ***n*** which indicates the version of the ccDictionary class in use. That number is the same as the ***n*** in AppETAC_***n***.h which must be included in C++ source code for application programs. The inclusion of those two files ensures that the appropriate version of the ccDictionary class is created by the *ETAC interpreter*.

3.10.1 Function Summary

The following list is a summary of the member functions of the ccDictionary class.

ccDictionary Class Function Summary

Function	Description
dCopyDict	Copies the whole *dictionary* to another.
dDeleteAll	Deletes all the *dictionary items* in the *dictionary*.
dDeleteItem	Deletes the *dictionary item* at an index.
dDuplicateDict	Duplicates the whole *dictionary* to another.
dExecItemObj	Executes the *stack object* of a *dictionary item* at an index.
dFindItem	Gets the position of a *dictionary item* in the *dictionary*.
dGetDictFlags	Gets the flags associated with the *dictionary*.
dGetDictName	Gets the name of the *dictionary*.
dGetItemName	Gets the name of the *dictionary item* at an index.
dGetItemObj	Gets the value of a *stack object* of a *dictionary item* into a variable.
dGetItemType	Gets the *stack object* type of a *dictionary item* at an index.
dNewItem	Creates a new *dictionary item* from the value of a variable.
dNumSameItems	Determines the number of *dictionary items* having the same name (*dictionary keyword*) in the *dictionary*.
dPutDictFlags	Sets or clears selected flags associated with this *dictionary*.
dPutItemObj	Replaces the *stack object* of a *dictionary item* with the value of a variable.
dSetDictName	Sets the name of the *dictionary*.
dSetItemName	Sets the name of the *dictionary item* at an index.
dSize	Gets the total number of *dictionary items* in the *dictionary*.

3.10.2 Member Functions

The following points apply to the member functions of the `ccDictionary` class.

- *Dictionary items* are accessed via an index number. Index number zero is the bottommost *dictionary item*; the highest index (for example, `ccLAST_IDX`) is the topmost *dictionary item*.
- *Dictionary keywords* do not need to be unique within the same *dictionary* and among other *dictionaries*.

The definitions of the `ccDictionary` class member functions are as follows. Note that when modifying a *dictionary resource value*, *dictionaries* in *ETAC code* and in other *dictionary resource interfaces* can refer to that *resource value*.

ccDictionary::dCopyDict

`void dCopyDict(ccDictionary *&pDestDict)`

Output

`pDestDict` Receives a *copy* of this *dictionary resource value*.

Action

Copies the *resource object* of this *resource interface* to the *resource object* of another *dictionary resource interface* (`pDestDict`). If `pDestDict` contains `NULL`, a *resource interface* for `pDestDict` is allocated before the *copy*. If `pDestDict` and this *resource interface* contain the same *dictionary resource value* then no action occurs. Both *resource interfaces* will share the same *dictionary resource value* after the *copy*.

ccDictionary::dDeleteAll

`void dDeleteAll()`

Action
Deletes all the *dictionary items* of the *dictionary resource value* of this *resource interface*. Note that if this *dictionary resource value* has *stack objects linked* to it then no *dictionary item* will be deleted.

ccDictionary::dDeleteItem

`ccBOOL dDeleteItem(ccULONG pIndex)`

Input
`pIndex` The zero-based index of the *dictionary item* to delete, or `ccLAST_IDX`.

Return
Returns `true` if the function succeeded, otherwise returns `false`.

Action
Deletes the *dictionary item* of the *dictionary resource value* of this *resource interface* at the specified index (`pIndex`). A value of `ccLAST_IDX` for `pIndex` indicates the last (topmost) *dictionary item* in this *dictionary resource value*. If this *dictionary resource value* has *stack objects linked* to it, or `pIndex` is out of range, then the *dictionary item* will not be deleted and this function will return `false`.

ccDictionary::dDuplicateDict

`void dDuplicateDict(ccDictionary *&pDestDict)`

Output
`pDestDict` Receives a *duplicate* of this *dictionary resource value*.

Action
Duplicates the *resource object* of this *resource interface* to the *resource object* of another *dictionary resource interface* (`pDestDict`). If `pDestDict` contains `NULL`, a *resource interface* for `pDestDict` is allocated before the *duplication*. If `pDestDict` points to this *resource interface*, and the contained *dictionary resource value* has no other *managed reference* pointing to it, then no action occurs. Otherwise, the *resource value* of `pDestDict` is released and a new one is allocated before the *duplication*, and the two *resource interfaces* (if they are not the same one) will not share the same *dictionary resource value* after the *duplication*. After this function completes, the *dictionary resource value* of `pDestDict` will have only one *managed reference* pointing to it.

ccDictionary::dExecItemObj

`cc_rtn_code dExecItemObj(ccULONG pIndex, ccTAC *pTAC)`

Input
`pIndex` The zero-based index of the *dictionary item* to execute, or `ccLAST_IDX`.
`pTAC` A pointer to the *ETAC interface* in use (must not be `NULL`).

Return
Return code indicating the success or otherwise of the function or executed *stack object*.

Action
Executes (*activates*) the *stack object* of the *dictionary item* at an index (`pIndex`) of the *dictionary resource value* of this *resource interface*. A value of `ccLAST_IDX` for `pIndex` indicates the last (topmost) *dictionary item* in this *dictionary resource value*. The return code returned by the executed *stack object* is returned by this function.

If `pIndex` is out of range then the function returns a non-success return code. If `pTAC` is `NULL`, the consequence is unpredictable.

ccDictionary::dFindItem

```
ccULONG dFindItem(ccSTR pItemName, int pInstance = 1)
```

Input

`pItemName`	The name of the *dictionary item* to get (should not be `NULL`).
`pInstance`	A positive or negative integer indicating which instance of the named *dictionary item* to find (must not be `0`).

Return
Returns the position of the found *dictionary item* from the top of the contained *dictionary resource value* (topmost *dictionary item* is at position `1`). Otherwise returns `ccNOT_FOUND` if the *dictionary item* was not found.

Action
Gets the position of the *dictionary item* having the specified (`pItemName`) *dictionary keyword* existing in the *dictionary resource value* of this *resource interface*. Note that the position of the *dictionary item* is not the same as the item's index. If `pInstance` is positive (*n*), the function will search for the n^{th} instance of the *dictionary item* with name `pItemName` from the top of the *dictionary resource value*. If `pInstance` is negative (–*n*), the function will search for the n^{th} instance of the *dictionary item* with name `pItemName` from the bottom of the *dictionary resource value*.

If `pItemName` is `NULL`, the function will return `ccNOT_FOUND`. If `pInstance` is `0`, the consequence is undefined.

ccDictionary::dGetDictFlags

```
ccULONG dGetDictFlags()
```

Return
Returns the binary flags of this *dictionary resource value*.

Action
Gets the binary flags associated with the *dictionary resource value* of this *resource interface*. The returned flags include the internal flags for this *dictionary resource value* as well as the programmer-defined flags set via `dPutDictFlags()`.

Additional Information
Dictionary Binary Flags ▪ dPutDictFlags

ccDictionary::**dGetDictName**

```
void dGetDictName(ccString *&pName)
```

Output

pName Receives the name of this *dictionary resource value*.

Action

Retrieves the name of the *dictionary resource value* of this *resource interface*. If pName contains NULL, a *resource interface* for pName is allocated to receive the requested name.

ccDictionary::**dGetItemName**

```
ccBOOL dGetItemName(ccString *&pName, ccULONG pIndex)
```

Input

pIndex The zero-based index of the *dictionary item* from which to obtain the *dictionary keyword*, or ccLAST_IDX.

Output

pName Receives the *dictionary keyword* (name) of the specified *dictionary item*.

Return

Returns true if the function succeeded, otherwise returns false.

Action

Gets the *dictionary keyword* (pName) of the *dictionary item* at an index (pIndex) of the *dictionary resource value* of this *resource interface*. A value of ccLAST_IDX for pIndex indicates the last (topmost) *dictionary item* in this *dictionary resource value*. If pName contains NULL, a *resource interface* for pName is allocated to receive the requested name. If pIndex is out of range then the function returns false.

ccDictionary::**dGetItemObj**

```
cc_rtn_code dGetItemObj(ccULONG pIndex, ccStackObj *&pStackObj)
cc_rtn_code dGetItemObj(ccULONG pIndex, ccINT &pInteger,
    cc_tac_type pType1 = ccTAC_INT)
cc_rtn_code dGetItemObj(ccULONG pIndex, ccDEC &pDecimal)
cc_rtn_code dGetItemObj(ccULONG pIndex, ccString *&pString,
    cc_tac_type pType2 = ccTAC_STR)
cc_rtn_code dGetItemObj(ccULONG pIndex, ccSequence *&pSequence,
    cc_tac_type pType3 = ccTAC_SEQ)
cc_rtn_code dGetItemObj(ccULONG pIndex, ccMemoryBlock *&pMemory)
cc_rtn_code dGetItemObj(ccULONG pIndex, ccDictionary *&pDictionary)
```

Input

pType1 Type of *stack object* expected at the specified index. Possible values are: ccTAC_INT, ccTAC_CMDI, ccTAC_OPRI, ccTAC_MARK, ccTAC_NULL, ccTAC_EXE.

pType2 Type of *stack object* expected at the specified index. Possible values are: ccTAC_STR, ccTAC_CMD, ccTAC_OPR.

pType3 Type of *stack object* expected at the specified index. Possible values are: ccTAC_SEQ, ccTAC_PROC.

`pIndex`	Zero-based index of the *dictionary item* to access, or `ccLAST_IDX`.

Output

`pStackObj`	Receives the *stack object* existing at the specified index.
`pInteger`	Receives the integer value existing at the specified index.
`pDecimal`	Receives the decimal value existing at the specified index.
`pString`	Receives the string value existing at the specified index.
`pSequence`	Receives the *sequence resource value* existing at the specified index.
`pMemory`	Receives the memory *resource value* existing at the specified index.
`pDictionary`	Receives the *dictionary resource value* existing at the specified index.

Return

Return code indicating the success or otherwise of the function.

Action

Each function obtains a *copy* of the value of the *stack object* of the *dictionary item* at an index (`pIndex`) of the *dictionary resource value* of this *resource interface* into the specified output variable. A value of `ccLAST_IDX` for `pIndex` indicates the last (topmost) *dictionary item* in this *dictionary resource value*. For output variables that point to a *resource interface*, that *resource interface* is automatically released prior to receiving the specified item in a new *resource interface*. If the element at `pIndex` is not of the same type as specified or implied by the function, or if `pIndex` is out of range, then the function returns a non-success return code.

Additional Information

TAC Object Types

ccDictionary::dGetItemType

```
cc_rtn_code dGetItemType(cc_tac_type &pType, ccULONG pIndex)
```

Input

`pIndex`	The zero-based index of the *dictionary item* from which to obtain the associated *stack object* type, or `ccLAST_IDX`.

Output

`pType`	Receives the type of *stack object* existing at the specified index.

Return

Return code indicating the success or otherwise of the function.

Action

Gets the type (`pType`) of the *stack object* of the *dictionary item* at an index (`pIndex`) of the *dictionary resource value* of this *resource interface*. A value of `ccLAST_IDX` for `pIndex` indicates the last (topmost) *dictionary item* in this *dictionary resource value*. If `pIndex` is out of range then the function returns a non-success return code.

Additional Information

TAC Object Types

ccDictionary::**dNewItem**

```
ccBOOL dNewItem(ccSTR pItemName, ccStackObj *pStackObj,
    ccULONG pIndex = ccNEXT_IDX)
ccBOOL dNewItem(ccSTR pItemName, ccINT pInteger,
    cc_tac_type pType1 = ccTAC_INT, ccULONG pIndex = ccNEXT_IDX)
ccBOOL dNewItem(ccSTR pItemName, ccDEC pDecimal,
    ccULONG pIndex = ccNEXT_IDX)
ccBOOL dNewItem(ccSTR pItemName, ccSTR pString,
    cc_tac_type pType2 = ccTAC_STR, ccULONG pIndex = ccNEXT_IDX)
ccBOOL dNewItem(ccSTR pItemName, ccString *pString,
    cc_tac_type pType2 = ccTAC_STR, ccULONG pIndex = ccNEXT_IDX)
ccBOOL dNewItem(ccSTR pItemName, ccSequence *pSequence,
    cc_tac_type pType3 = ccTAC_SEQ, ccULONG pIndex = ccNEXT_IDX)
ccBOOL dNewItem(ccSTR pItemName, ccMemoryBlock *pMemory,
    ccULONG pIndex = ccNEXT_IDX)
ccBOOL dNewItem(ccSTR pItemName, ccDictionary *pDictionary,
    ccULONG pIndex = ccNEXT_IDX)
```

Input

pItemName	The *dictionary keyword* (name) of a new *dictionary item* to insert at the specified index (should not be NULL).
pStackObj	Contains a *stack object* of a new *dictionary item* to insert at the specified index (should not be NULL).
pInteger	Contains an integer value of a new *dictionary item* to insert at the specified index.
pDecimal	Contains a decimal value of a new *dictionary item* to insert at the specified index.
pString	Contains a string value of a new *dictionary item* to insert at the specified index, or NULL (only if pType2 is ccTAC_STR).
pSequence	Contains a *sequence resource value* of a new *dictionary item* to insert at the specified index, or NULL.
pMemory	Contains a memory *resource value* of a new *dictionary item* to insert at the specified index, or NULL.
pDictionary	Contains a *dictionary resource value* of a new *dictionary item* to insert at the specified index, or NULL.
pType1	Type of *stack object* of the *dictionary item* to be inserted at the specified index. Possible values are: ccTAC_INT, ccTAC_CMDI, ccTAC_OPRI, ccTAC_MARK, ccTAC_NULL, ccTAC_EXE.
pType2	Type of *stack object* of the *dictionary item* to be inserted at the specified index. Possible values are: ccTAC_STR, ccTAC_CMD, ccTAC_OPR.
pType3	Type of *stack object* of the *dictionary item* to be inserted at the specified index. Possible values are: ccTAC_SEQ, ccTAC_PROC.
pIndex	Zero-based index at which the specified item is to be inserted, or ccNEXT_IDX.

Return

Returns true if the function succeeded, otherwise returns false.

Action

Each function inserts a *dictionary item* having a *stack object* containing the specified item into the *dictionary resource value* of this *resource interface* at an index (pIndex). Existing *dictionary items* within this *dictionary resource value* at and after the specified index (if any) will be moved to the next position (towards the top of the *dictionary*) before the insertion is made. No *dictionary item* within this *dictionary resource value* is deleted. A value of `ccNEXT_IDX` for `pIndex` indicates a *dictionary item* created after the last one in this *dictionary resource value*; that *dictionary item* becomes the topmost *dictionary item*. For input *resource variables*, a *copy* of the *resource object* of that variable is inserted. For input *resource variables* that contain `NULL`, the inserted *stack object* will have an empty *resource value*, except for a *stack object resource variable* (syntax 1) and for string variables if `pType2` is not `ccTAC_STR` (syntaxes 4 and 5). In those cases, the function returns `false`. If this *dictionary resource value* has *stack objects linked* to it or `pIndex` is out of range or `pItemName` is `NULL` then no *dictionary item* will be inserted and the function will return `false`.

Additional Information
TAC Object Types

ccDictionary::dNumSameItems

`ccULONG` **`dNumSameItems`**`(ccSTR` **`pItemName`**`)`

Input

`pItemName`	The *dictionary keyword* (name) to search for (should not be `NULL`).

Return
Returns the number of *dictionary items* having the same specified *dictionary keyword* in this *dictionary resource value*.

Action
Gets the number of *dictionary items* having the same specified (`pItemName`) *dictionary keyword* in the *dictionary resource value* of this *resource interface*.

If `pItemName` is `NULL` then the function will return `0`.

ccDictionary::dPutDictFlags

`void` **`dPutDictFlags`**`(ccULONG` **`pFlags`**`, ccULONG` **`pMask`** `= 0x0000FFFF)`

Input

`pFlags`	The binary flags to set or clear.
`pMask`	Indicates which flags in `pFlags` will take effect.

Action
Sets or clears the programmer-defined binary flags (`pFlags`) associated with the *dictionary resource value* of this *resource interface* based on the corresponding set flags in a mask (`pMask`). The programmer can set flags only in the least significant 16 bits of `pFlags`; flags set in the most significant 16 bits of `pFlags` are ignored (they are used internally by ETAC). Those programmer-defined flags are for programmer use for any purpose; the *ETAC interpreter* ignores those flags. `pMask` determines which flags in `pFlags` will take effect. A set flag (`1`) in `pMask` indicates that the corresponding flag (whether it is set or clear) in `pFlags` will take effect. A clear flag (`0`) in `pMask` indicates that the corresponding flag setting in the *dictionary resource value* will remain unchanged (the corresponding flag in `pFlags` is ignored). For example, if `pMask` is `0x00000000` then this function will have no effect; if `pMask` is `0x0000FFFF` (the default) then all 16 programmer-defined flags in `pFlags` will take effect (the programmer-defined flags in the *dictionary resource value* will be modified to those in `pFlags`).

Additional Information
Dictionary Binary Flags ▪ dGetDictFlags

ccDictionary::dPutItemObj

```
ccBOOL dPutItemObj(ccULONG pIndex, ccStackObj *pStackObj)
ccBOOL dPutItemObj(ccULONG pIndex, ccINT pInteger,
    cc_tac_type pType1 = ccTAC_INT)
ccBOOL dPutItemObj(ccULONG pIndex, ccDEC pDecimal)
ccBOOL dPutItemObj(ccULONG pIndex, ccSTR pString,
    cc_tac_type pType2 = ccTAC_STR)
ccBOOL dPutItemObj(ccULONG pIndex, ccString *pString,
    cc_tac_type pType2 = ccTAC_STR)
ccBOOL dPutItemObj(ccULONG pIndex, ccSequence *pSequence,
    cc_tac_type pType3 = ccTAC_SEQ)
ccBOOL dPutItemObj(ccULONG pIndex, ccMemoryBlock *pMemory)
ccBOOL dPutItemObj(ccULONG pIndex, ccDictionary *pDictionary)
```

Input

pStackObj	Contains a *stack object* of a new *dictionary item* to put at the specified index (should not be NULL).
pInteger	Contains an integer value to put into the *dictionary item* at the specified index.
pDecimal	Contains a decimal value to put into the *dictionary item* at the specified index.
pString	Contains a string value to put into the *dictionary item* at the specified index, or NULL (only if pType2 is ccTAC_STR).
pSequence	Contains a *sequence resource value* to put into the *dictionary item* at the specified index, or NULL.
pMemory	Contains a memory *resource value* to put into the *dictionary item* at the specified index, or NULL.
pDictionary	Contains a *dictionary resource value* to put into the *dictionary item* at the specified index, or NULL.
pType1	Type of *stack object* of the *dictionary item* to be put at the specified index. Possible values are: ccTAC_INT, ccTAC_CMDI, ccTAC_OPRI, ccTAC_MARK, ccTAC_NULL, ccTAC_EXE.
pType2	Type of *stack object* of the *dictionary item* to be put at the specified index. Possible values are: ccTAC_STR, ccTAC_CMD, ccTAC_OPR.
pType3	Type of *stack object* of the *dictionary item* to be put at the specified index. Possible values are: ccTAC_SEQ, ccTAC_PROC.
pIndex	Zero-based index at which the specified item is to be put, or ccLAST_IDX.

Return

Returns true if the function succeeded, otherwise returns false.

Action

Each function replaces the *stack object* of a *dictionary item* of the *dictionary resource value* of this *resource interface* at an index (pIndex) with a *stack object* containing the specified item. A value of ccLAST_IDX for pIndex indicates the last (topmost) *dictionary item*. For input *resource variables*, a *copy* of the *resource object* of that variable is put. For input *resource variables* that contain NULL, the inserted *stack object* will have an empty *resource value*, except for a *stack object resource variable* (syntax 1) and for string variables if pType2 is not ccTAC_STR (syntaxes 4 and 5). In those cases, the function returns false. If pIndex is out of range then the function returns false.

Additional Information
TAC Object Types

ccDictionary::dSetDictName

```
void dSetDictName(ccSTR pName)
```

Input

pName	Contains the name to set for this *dictionary resource value*, or NULL.

Action
Sets the name (pName) of the *dictionary resource value* of this *resource interface*. If pName is NULL, an empty name is set.

ccDictionary::dSetItemName

```
ccBOOL dSetItemName(ccSTR pName, ccULONG pIndex)
```

Input

pName	Contains the name to set for the *dictionary keyword* of the specified *dictionary item* (should not be NULL).
pIndex	The zero-based index of the *dictionary item* to modify, or ccLAST_IDX.

Return
Returns true if the function succeeded, otherwise returns false.

Action
Replaces the name (pName) of the *dictionary keyword* of the *dictionary item* at an index (pIndex) of the *dictionary resource value* of this *resource interface*. A value of ccLAST_IDX for pIndex indicates the last (topmost) *dictionary item* in this *dictionary resource value*. If pName is NULL, or pIndex is out of range, then the function returns false.

ccDictionary::dSize

```
ccULONG dSize()
```

Return
The number of *dictionary items* in the *dictionary resource value* contained in this *resource interface*.

Action
Gets the number of *dictionary items* in the *dictionary resource value* contained in this *resource interface*.

3.11 ccDataObject Class

The *data object resource interface* is an emulation based on the *sequence resource interface*, and has no member functions as such. It is important to note that the details of the emulation can be different in future versions of ETAC without notice. The C++ programmer must not in any way directly use the *sequence resource interface* member functions for a *data object resource interface*.

A *data object* can be constructed via helper functions designed for that purpose. A *data object* contains a *data dictionary*, which can be assessed as a regular *dictionary* via a *dictionary resource interface*. The *data dictionary* of a *data object resource interface* can be accessed via a helper function.

The helper functions for a *data object resource interface* are described in 3.12.1 Data Object Helpers.

The ccDataObject class is defined as a typedef in the file ExternTACLib_***n***.h which must be included in C++ source code for creating *external TAC libraries*. The pre-processor definition ccTAC_VRSN (defined in ExternTACLib_***n***.h) defines the number ***n*** which indicates the version of the ccDataObject class in use. That number is the same as the ***n*** in AppETAC_***n***.h which must be included in C++ source code for application programs. The inclusion of those two files ensures that the appropriate version of the ccDataObject class is created by the *ETAC interpreter*.

3.12 Helper Functions

Helper functions are implemented as C++ source code in the file ETIExtraFnts.cpp. The ETIExtraFnts.cpp file is included (via the pre-processor #include directive) at a place in the C++ source code where functions are defined. To include only the required function definitions existing in the file, the C++ programmer defines the appropriate pre-processor definition for the required functions before the file #include directive. If no such pre-processor definitions are made, all the functions in the file are included. The said pre-processor definition for each helper function is specified under the “Inclusion Definition” heading of the function description block later in this document.

If no helper functions are used in the C++ source file, ETIExtraFnts.cpp need not be included.

3.12.1 Data Object Helpers

The following list is a summary of the helper functions of the ccDataObject class.

ccDataObject Class Function Summary

Function	**Description**
ccMakeDataObj	Associates a *data object* with a *data dictionary*.
ccGetDataDict	Obtains the *data dictionary* of a *data object*.

Data Object Examples

The following example illustrates how to create a *data object*. Error checking code and other irrelevant items are not shown in the illustration.

```
#define ccF_MAKE_DATA_OBJ
#include "ETIExtraFnts.cpp"

ccRTNCODE; /* Declare the return code variable. */
ccNEW_DATAOBJECT    (DataObj); /* The data object to create. */
ccNEW_DICTIONARY    (DataDict); /* The data dictionary of the created data object. */
bool                 Rtn;

   /* Allocate members for the data object via the data dictionary. */
   Rtn = DataDict->dNewItem(L"doInt", 35L); assert(Rtn);
   Rtn = DataDict->dNewItem(L"doStr", L"String item"); assert(Rtn);

   /* Create the data object from the data dictionary. */
   Rtn = ccMakeDataObj(DataObj, DataDict); assert(Rtn);

   /* Allocate more members for the data object if desired. */
   Rtn = DataDict->dNewItem(L"doDec", 42.6); assert(Rtn);

   ccPUSH((DataObj)); /* Push the data object onto the object stack. */
```

```
ccEXITLBL:
   /* Release resource interfaces. */
   ccFREE(DataDict);
   ccFREE(DataObj); ♦
```

The following example illustrates how to obtain a *data object* from the *object stack*. Error checking code and other irrelevant items are not shown in the illustration.

```
#define ccF_GET_DATA_DICT
#include "ETIExtraFnts.cpp"

ccRTNCODE; /* Declare the return code variable. */
ccNEW_DATAOBJECT    (DataObj); /* The data object from the object stack. */
ccNEW_DICTIONARY    (DataDict); /* The data dictionary of the obtained data object. */
bool                 Rtn;

   /* Pull the data object from the object stack. */
   ccPULL((DataObj));

   /* Obtain the data dictionary from the data object. */
   ccCALL(ccGetDataDict(DataDict, DataObj));

   /* Allocate more members for the data object if desired. */
   Rtn = DataDict->dNewItem(L"doDec", 42.6); assert(Rtn);

ccEXITLBL:
   /* Release resource interfaces. */
   ccFREE(DataDict);
   ccFREE(DataObj); ♦
```

The definitions of the `ccDataObject` class helper functions are as follows.

ccMakeDataObj

`ccBOOL` **`ccMakeDataObj`** `(ccDataObject *`**`pDataObj`**`, ccDictionary *`**`pDataDict`**`)`

Input

`pDataObj`	Contains an allocated empty *data object resource interface* (must not be `NULL`).
`pDataDict`	Contains an initialised *dictionary resource interface* as the *data dictionary* for `pDataObj` (must not be `NULL`).

Output

`pDataObj`	Receives a *copy* of the *dictionary resource value* contained in `pDataDict`.
`pDataDict`	Receives an internal name identifying the *dictionary resource value* of `pDataDict` as a *data dictionary*.

Return

Returns `true` if the function succeeded, otherwise returns `false`.

Inclusion Definition

`#define ccF_MAKE_DATA_OBJ`

Inclusions Required

ETIExtraFnts.cpp

Action

Initialises a *data object resource interface* (`pDataObj`) with a *copy* of the *dictionary resource value* contained in `pDataDict` as a *data dictionary*. `pDataObj` and `pDataDict` must both point to a pre-allocated *resource interface*, otherwise the consequence is unpredictable. The *resource interface* of `pDataObj` must be empty; the *resource interface* of `pDataDict` can contain *dictionary items* before this function is called. Further *dictionary items* can be allocated to `pDataDict` after this function returns.

Note that both the *resource interfaces* of `pDataObj` and `pDataDict` will have a *managed reference* to the same *data dictionary resource value* after this function returns to the caller.

ccGetDataDict

```
cc_rtn_code ccGetDataDict(ccDictionary *&pDataDict, ccDataObject
    *pDataObj)
```

Input

`pDataObj` Contains an initialised *data object resource interface* (must not be `NULL`).

Output

`pDataDict` Receives a *copy* of the *data dictionary* of `pDataObj`.

Return

Return code indicating the success or otherwise of the function.

Inclusion Definition

`#define ccF_GET_DATA_DICT`

Inclusions Required

ETIExtraFnts.cpp

Action

A *copy* of the *data dictionary* contained in `pDataObj` is put into the *resource interface* of `pDataDict`. `pDataObj` must contain a properly initialised *data object* (via `ccMakeDataObj()`) before this function is called. Further *dictionary items* can be allocated to `pDataDict` after this function returns.

Note that both the *resource interfaces* of `pDataObj` and `pDataDict` will have a *managed reference* to the same *data dictionary resource value* after this function returns to the caller.

Other Information

ccMakeDataObj

3.13 External TAC Library Functions

An *external TAC library* is designed by a C++ programmer, therefore all exportable functions within it are defined by the programmer. Those exportable functions, however, need to satisfy certain requirements so that the *ETAC interpreter* can call them. The first function that the *ETAC interpreter* calls after loading an *external TAC library* (typically via **`@ImportLib`**) is the mapping function, `tacGetCCMapping()`. This is a required function that supplies important information to the *ETAC interpreter*. The other functions that the C++ programmer defines are the *ETL functions*. These functions are optional, but it would be pointless not to have at least one *ETL function*.

ExternTACLib_***n***.h, where ***n*** is the version number of the *ETAC interface* and *resource interfaces* in use, must be included in C++ source code for creating *external TAC libraries*.

3.13.1 Mapping Function

The mapping function, `tacGetCCMapping()`, is called by the *ETAC interpreter* via the name "`tacGetCCMapping`" exported by the *external TAC library* (the *ETAC interpreter* uses the **Windows**® `GetProcAddress()` function to call the mapping function by the said name).

The designer of an *external TAC library* must create a unique *comop* name for each *ETL function* in that library. Each *comop* name must satisfy the syntax of a *variable identifier*. A *comop* name for use as an *operator* must be prefixed with an ampersand character (`&`). An *ETL function* is identified by its *comop* name in *ETAC code*. In addition, the designer can optionally classify the *comop* names into arbitrary (possibly overlapping) groups for the convenience of the ETAC programmer. The name of each group must also satisfy the syntax of a *variable identifier*. Also, each *ETL function* must correspond to a unique ordinal number (as defined for an exported DLL function).

An ETAC programmer can specify one or more groups (or "classes") to load from the *external TAC library*. The mapping function satisfies that request by supplying the caller with a list of *comop* names matching the union of the requested classes, and also supplies a corresponding list of ordinal numbers for those *comop* names. A name based on the *comop* name is used by an ETAC programmer to identify the corresponding *ETL function*, and the corresponding ordinal number is used by the *ETAC interpreter* to execute the *ETL function*.

tacGetCCMapping

```
extern "C" long ccTACAPI tacGetCCMapping(ccSTR *&pComopNameList,
    ccULONG *&pComopOrdList, ccSTR *pClasses, void *pReserved1,
    void *pReserved2, void *pReserved3)
```

Input

`pClasses`	A pointer to a `NULL` terminated string list containing classification names of the *ETL functions* requested by the caller, or `NULL`.

Output

`pComopNameList`	A `NULL` terminated string list of *comop* names for the *ETL functions* indicated by `pClasses` requested by the caller.
`pComopOrdList`	A zero-terminated integer list of ordinal numbers of the *ETL functions* corresponding the *comop* names of `pComopNameList`.

Return

Returns the version of the *ETAC interface* and *resource interfaces* used by the *external TAC library* (should be the value of `ccTAC_VRSN`).

Inclusions Required

ExternTACLib_***n***.h

Action

This is a PROGRAMMER-DEFINED function that is called by the *ETAC interpreter* after loading an *external TAC library*. The *ETAC interpreter* calls this function from the **`load_lib`** *command*. One of the arguments of **`load_lib`** is a class list which is passed to this function in `pClasses`. The class list allows the **`load_lib`** caller to use only the specified subsets of the *ETL functions* by listing the class names of those *ETL functions*. If a class list is not supplied, all the *ETL functions* are loaded into the *ETAC interpreter*.

`pComopNameList` is a `NULL` terminated string (`ccSTR`) array defined by the designer of the *external TAC library* to contain *comop* names matching the union of the classes specified in `pClasses`. If `pClasses` is `NULL`, `pComopNameList` must contain all the *comop* names corresponding to the *ETL functions*.

`pComopOrdList` is a zero-terminated unsigned long (`ccULONG`) array defined by the designer of the *external TAC library* to contain the ordinal numbers corresponding to the names in `pComopNameList`.

`pReserved1`, `pReserved2`, and `pReserved3` are for future use and should be ignored.

3.13.2 ETL Functions

The main purpose of creating an *external TAC library* is to provide *ETL functions* for use in *ETAC code*. *ETL functions* are implemented in C++ code, and called by the *ETAC interpreter* via its DLL ordinal number (the *ETAC interpreter* uses the **Windows**® `GetProcAddress()` function to call the *ETL function* by the said ordinal number).

In the following boxed description, "*`ETL_Function`*" represents the programmer-defined name of an *ETL function*.

ETL Function

`extern "C" cc_rtn_code ccTACAPI` ***`ETL_Function`***`(ccTAC *`**`pTAC`**`)`

Input

`pTAC` A pointer to the *ETAC interface* in use.

Return

Return code indicating the success or otherwise of the function.

Inclusions Required

ExternTACLib_***n***.h

Action

This is a PROGRAMMER-DEFINED function that is called by the *ETAC interpreter* or C++ code. The function can be designed for use as a *command* (typical) or *operator*. If designed for use as an *operator*, then the function is responsible for processing all the *object stack* arguments and replacing the final mark 0 *stack object* with the result. All *resource interfaces* created within the function must be released before the function ends. Return codes returned by member functions of the *ETAC interface* (`pTAC`) or *resource interfaces* should be returned by this function to be processed by the *ETAC interpreter*.

This function can be called directly from C++ code (perhaps from an application program or another *external TAC library*). In that case, the C++ programmer uses the methods documented by Microsoft® to call the DLL function, passing the *ETAC interface* to it (returned from a call to `ccGetTACIF()`), as the only argument. If this function is called from another *ETL function* in the same *external TAC library*, then `ccGetTACIF()` need not be called to obtain the *ETAC interface*; `pTAC` can be used instead.

Other Information

ccGetTACIF

3.14 AppETAC Functions

AppETAC.dll contains a few predefined functions for use by a C++ application program. Only the initialisation function, `etacSetAppETAC()`, and release function, `etacRelease()`, need to be called mandatorily; the other functions can be called optionally. These functions are exported by AppETAC.dll with the pre-processor definition names as specified under the heading "Export Name Definition" of each description box below. Those pre-processor names can be used with the **Windows**® `GetProcAddress()` function if desired.

AppETAC_***n***.h, where ***n*** is the version number of the *ETAC interface* and *resource interfaces* in use, must be included in C++ source code when calling the **AppETAC** functions.

3.14.1 Initialisation Function

This function (`etacSetAppETAC()`) must be called from an application program before any interaction with the *ETAC interpreter* can be made. The function sets up the *ETAC interpreter* as specified in the start-up parameters passed to the function when called. The start-up parameters are defined in aeAppETACPars_***v***.h, where ***v*** is the version of the start-up parameter structure, and is the value defined for aeAEP_VRSN within the file. aeAppETACPars_***v***.h is automatically included by AppETAC_***n***.h.

The table below describes the details of the start-up parameters. Some of these start-up parameters are analogous to the ones for RunETAC.exe.

AppETAC Start-up Parameters

Parameter and Type	Meaning
aeVrsn (aeULONG)	The version of the current start-up parameter structure. This value should not be modified (default: aeAEP_VRSN).
aeFlags (aeULONG)	Desired start-up flags as described in the table below. The flags can be any combination of aeNO_LOADER, aeDEBUG, aeSOLO, and aeSILENT combined using the C++ 'BITWISE OR' (|) operator (default: 0x00000000).
aeInclDirs (aeSTR)	The file path of a text file containing a list of inclusion directories (default: NULL).
aePPDefs (aeSTR)	A string containing ETAC pre-processor definition names separated by spaces (default: NULL).
aeLoaderArg (aeSTR)	The string argument for the ETAC *loader script* (default: NULL).
aeScripts (aeSTR)	NULL terminated array of *ETAC code* files to execute after the *loader script* has finished running (default: NULL).
aeLogFilePath (aeSTR)	Path and file name of the log file (default: NULL).
aeRtnCode (aeRTNCDE)	Return code returned from execution of etacSetAppETAC() (default: 0).

The table below describes the details of the start-up flags specified at aeFlags in the start-up parameters. The default is that no flags are set.

AppETAC Start-up Flags

Flag	Meaning
aeNO_LOADER	Specifies not to search for the *loader script*. The default is to search for and prompt the user for the *loader script* if not found.
aeDEBUG	Runs the specified *ETAC text script* files (specified at: aeScripts) and other *ETAC text script* in interactive debugging mode (see etacSetDebug). Debugging is off by default.
aeSOLO	Specifies that only one instance of AppETAC.dll is allowed to run at a time on the same computer. The default is to allow many instances of AppETAC.dll to run simultaneously.
aeSILENT	Does not display a message box when etacProcessTACError() is called (see etacProcessTACError). The default is to display a message box when the said function is called.

etacSetAppETAC

```
extern "C" ccTAC * ccTACAPI etacSetAppETAC(aeULONG pTACIFVrsn,
    app_fnt pAppFnt, aeAppETACPars *pPars = NULL, void *pReserved1 = NULL,
    void *pReserved2 = NULL)
```

Input

pTACIFVersn	The version of the *ETAC interface* to be returned (should be ccTAC_VRSN).
pAppFnt	A pointer to the *call-back function*, or NULL.
pPars	Start-up parameters, or NULL.

Return
A pointer to the *ETAC interface* of version pTACIFVrsn.

Inclusions Required
AppETAC_***n***.h

Export Name Definition
aeETAC_SET_APPETAC

Action
etacSetAppETAC() must only be called once from an application program to set up the *ETAC interpreter*. If the application program in any way causes the *command* **run_app_fnt** to be *activated*, then a *call-back function* must be defined by the C++ programmer and passed to this function in the pAppFnt parameter. To use the start-up parameters, a variable of type aeAppETACPars needs to be defined, and the address of that variable passed to this function in the pPars parameter. The function returns a pointer to the *ETAC interface* for interacting with the *ETAC interpreter*.

The function etacRelease() must be called before the main window of the application program is destroyed.

pReserved1 and pReserved2 are for future use and should be ignored.

Related Information
etacRelease

3.14.2 Auxiliary Functions

AppETAC.dll contains some auxiliary functions that can optionally be called from the application program.

etacProcessTACError

```
extern "C" void ccTACAPI etacProcessTACError(cc_rtn_code pErrCode)
```

Input

pErrCode	The return code from a function with a return type of cc_rtn_code.

Inclusions Required
AppETAC_***n***.h

Export Name Definition
aeETAC_PROC_TAC_ERR

Action

Processes a TAC error code (`pErrCode`) by converting it to a text message, logging the message to a file, and (if `aeSILENT` was not specified in the start-up flags) displaying it on the screen. `pErrCode` contains an error code as returned from functions with a return type of `cc_rtn_code`. If no such processing is desired, then this function should not be called. The consequence is undefined if the value of `pErrCode` is `ccTAC_RTN_SUCCESS` (or zero).

Other Information

ccERROR

etacRelease

```
extern "C" void ccTACAPI etacRelease()
```

Inclusions Required

AppETAC_*n*.h

Export Name Definition

`aeETAC_RELEASE`

Action

Releases the AppETAC.dll resources. This function must be called before the main window of the application program is destroyed.

etacSetDebug

```
extern "C" void ccTACAPI etacSetDebug(bool pOn = true)
```

Input

`pOn`	Determines whether to set the debug state on (`true`) or off (`false`).

Inclusions Required

AppETAC_*n*.h

Export Name Definition

`aeETAC_SET_DEBUG`

Action

Sets the debug state to on (`pOn` is `true`) or off (`pOn` is `false`). When the debug state is on, the debug window and the trace window appear so that the programmer can trace and debug *ETAC text script*. When the debug state is off, the debug window and the trace window do not appear. The default setting of the debug state is determined by the start-up flag `aeDEBUG`.

This function must not be called while an *ETAC session* is active (the consequence is undefined otherwise). Calls to this function are not nested.

Other Information

AppETAC Start-up Flags

etacShowAbout

```
extern "C" void ccTACAPI etacShowAbout()
```

Inclusions Required

AppETAC_*n*.h

Export Name Definition

`aeETAC_SHOW_ABOUT`

Action
Shows the AppETAC about box.

3.15 Application Program Call-back Function

An application program can optionally include a single *call-back function*, which is called via the **run_app_fnt** *command* from *ETAC code*. The *call-back function* is designed by the C++ programmer to perform various operations as desired.

AppETAC_***n***.h, where ***n*** is the version number of the *ETAC interface* and *resource interfaces* in use, must be included in C++ source code when defining the *call-back function*.

3.15.1 Call-back Function

In the following boxed description, "*Call-back_Function*" represents the programmer-defined name of the *call-back function*.

Call-back Function

`extern "C" cc_rtn_code ccTACAPI` ***`Call-back_Function`***`(ccTAC *`**`pTAC`**`)`

Input

pTAC	A pointer to the *ETAC interface* in use.

Return
Return code indicating the success or otherwise of the function.

Inclusions Required
AppETAC_***n***.h

Action
This is a PROGRAMMER-DEFINED function that is called by the *ETAC interpreter* via the **run_app_fnt** *command* in *ETAC code*. The proper arguments for this function are expected to exist on the *object stack*. This function can return *stack objects* on the *object stack* for the **run_app_fnt** caller. Return codes returned by member functions of the *ETAC interface* (pTAC) or *resource interfaces* should be returned as the return value of this function to be processed by the *ETAC interpreter*.

The first *stack object* argument for this function is typically a command number (an integer *stack object*) defined by the designer of this function. The command number is only a convention used by the programmer to distinguish among different operations of this function. The remaining *stack object* arguments (if any) are designed by the programmer to be appropriate for the value of that command number.

Appendix A

Compatibility Issues

A.1 Introduction

The *ETAC interface* and *resource interfaces* use the virtual table produced by the C++ compiler to communicate with the *ETAC interpreter*. A virtual table is hidden from the C++ programmer, and typically consists of an array of pointers to the virtual function definitions of a C++ class. An instance of a C++ class has internal access to the virtual table, by which means it calls the appropriate virtual functions. Since virtual tables are hidden, they do not form part of the specification of the C++ language, thus the order of the pointers within a virtual table is determined by the C++ compiler implementer. In addition, a virtual table can be implemented either at the beginning of a class instance or at the end. As a result, the object code produced by C++ compilers designed by different vendors may not be inter-compatible with respect to calling virtual functions. For example, if a virtual function of a class instance is called from code compiled by a C++ compiler designed by a different vendor, the order of the pointers within the virtual table of that class instance may be different than the order produced by the compiler of that other vendor. The result is that the wrong function may be called by that other code.

The *ETAC interpreter* was compiled with MSVC 7.1 (Microsoft® Visual C++® compiler version 7.1). Any C++ code that needs to communicate with the *ETAC interpreter* must be compatible with MSVC 7.1 with respect to the virtual tables and function calling conventions. The *ETAC interpreter* uses the native C calling convention known as ‹`__cdecl`› within MSVC when communicating with C++ code. All MSVC compilers after version 7.1 are expected to be compatible with version 7.1 with respect to the ordering of the pointers within the virtual table of C++ classes. C++ compilers designed by other vendors may produce a different ordering of the pointers within the virtual table. The easiest solution to the problem is to obtain a C++ compiler compatible with MSVC 7.1, such as the one provided by Microsoft® for free (subject to terms and conditions). If that is not a viable solution, then the workaround described in the following section can be used.

A.2 Workaround for Non-MSVC Compilers

The first requirement for the workaround is that the non-MSVC compiler must produce the virtual table at the beginning of a class instance. If the non-MSVC compiler produces the virtual table in any other position, then no workaround is possible. The workaround consists of attempting to coerce the non-MSVC compiler to produce the function pointers in the correct order within the virtual table of the *ETAC interface* and each *resource interface*. This may be achieved by simply making a copy of ExternTACLib_***n***.h, and reordering the virtual function declarations such that the order of the virtual function pointers in the produced virtual table is identical to the order of the virtual function pointers expected by the said interfaces (this reordering will need to be done whenever a new ExternTACLib_***n***.h is released). The C++ programmer using a non-MSVC compiler needs to know the order of the virtual function pointers produced by that compiler. Typically, the order of the produced function pointers are the same as the order of the virtual function declarations presented in the class definition. The compatibility issue arises because the MSVC compiler does not produce the function pointers in the order presented in the class definition for overloaded virtual functions, but produces those pointers in reverse order — non-overloaded function pointers are produced in the given order.

The following tables show the order of the function pointers in the virtual table of the *ETAC interface* and of each *resource interface*. Element indexes for each virtual table are indicated within square brackets. Version 1 of the *ETAC interface* has been redefined from previous.

Virtual Tables for ETAC Interfaces

ccTAC (version 1)

```
[0]  ccCountToMark(unsigned long)
[1]  ccDeleteDict(unsigned long, unsigned long)
[2]  ccExecCmd(unsigned long)
[3]  ccExecCmd(unsigned short const *)
[4]  ccExecETAC(class ccMemoryBlock *)
[5]  ccExecETAC(unsigned short const *)
[6]  ccGetDict(class ccDictionary * &, unsigned short const *, int)
[7]  ccGetDictOfItem(class ccDictionary * &, unsigned short const *, int)
[8]  ccGetObjType(unsigned char &)
[9]  ccGetTACIF(struct HINSTANCE__ *, long)
[10] ccNew(class ccDictionary * &)
[11] ccNew(class ccMemoryBlock * &)
[12] ccNew(class ccSequence * &)
[13] ccNew(class ccStackObj * &)
[14] ccNew(class ccString * &)
[15] ccPop(unsigned long)
[16] ccPull(class ccDictionary * &, bool)
[17] ccPull(class ccMemoryBlock * &)
[18] ccPull(class ccSequence * &, unsigned char)
[19] ccPull(class ccStackObj * &)
[20] ccPull(class ccString * &, unsigned char)
[21] ccPull(double &)
[22] ccPull(long &, unsigned char)
[23] ccPush(unsigned short const *, unsigned char)
[24] ccPush(class ccDictionary *, bool)
[25] ccPush(class ccMemoryBlock *)
[26] ccPush(class ccSequence *, unsigned char)
[27] ccPush(class ccStackObj *)
[28] ccPush(class ccString *, unsigned char)
[29] ccPush(double)
[30] ccPush(long, unsigned char)
[31] ccRelease(class ccDictionary * &)
[32] ccRelease(class ccMemoryBlock * &)
[33] ccRelease(class ccSequence * &)
[34] ccRelease(class ccStackObj * &)
[35] ccRelease(class ccString * &)
```

ccDictionary

```
[0]  dCopyDict(class ccDictionary * &)
[1]  dDeleteAll(void)
[2]  dDeleteItem(unsigned long)
[3]  dDuplicateDict(class ccDictionary * &)
[4]  dExecItemObj(unsigned long, class ccTAC *)
```

```
[5]  dFindItem(unsigned short const *, int)
[6]  dGetDictFlags(void)
[7]  dGetDictName(class ccString * &)
[8]  dGetItemName(class ccString * &, unsigned long)
[9]  dGetItemObj(unsigned long, class ccDictionary * &)
[10] dGetItemObj(unsigned long, class ccMemoryBlock * &)
[11] dGetItemObj(unsigned long, class ccSequence * &, unsigned char)
[12] dGetItemObj(unsigned long, class ccStackObj * &)
[13] dGetItemObj(unsigned long, class ccString * &, unsigned char)
[14] dGetItemObj(unsigned long, double &)
[15] dGetItemObj(unsigned long, long &, unsigned char)
[16] dGetItemType(unsigned char &, unsigned long)
[17] dNewItem(unsigned short const *, unsigned short const *, unsigned
       char, unsigned long)
[18] dNewItem(unsigned short const *, class ccDictionary *, unsigned long)
[19] dNewItem(unsigned short const *, class ccMemoryBlock *, unsigned
       long)
[20] dNewItem(unsigned short const *, class ccSequence *, unsigned char,
       unsigned long)
[21] dNewItem(unsigned short const *, class ccStackObj *, unsigned long)
[22] dNewItem(unsigned short const *, class ccString *, unsigned char,
       unsigned long)
[23] dNewItem(unsigned short const *, double, unsigned long)
[24] dNewItem(unsigned short const *, long, unsigned char, unsigned long)
[25] dNumSameItems(unsigned short const *)
[26] dPutDictFlags(unsigned long, unsigned long)
[27] dPutItemObj(unsigned long, unsigned short const *, unsigned char)
[28] dPutItemObj(unsigned long, class ccDictionary *)
[29] dPutItemObj(unsigned long, class ccMemoryBlock *)
[30] dPutItemObj(unsigned long, class ccSequence *, unsigned char)
[31] dPutItemObj(unsigned long, class ccStackObj *)
[32] dPutItemObj(unsigned long, class ccString *, unsigned char)
[33] dPutItemObj(unsigned long, double)
[34] dPutItemObj(unsigned long, long, unsigned char)
[35] dSetDictName(unsigned short const *)
[36] dSetItemName(unsigned short const *, unsigned long)
[37] dSize(void)
```

ccSequence and **ccDataObject**

```
[0]  sAppendSeq(class ccSequence *, bool)
[1]  sCopySeq(class ccSequence * &)
[2]  sDeleteAll(void)
[3]  sDeleteElms(unsigned long, unsigned long)
[4]  sDuplicateSeq(class ccSequence * &)
[5]  sGet(class ccDictionary * &, unsigned long)
[6]  sGet(class ccMemoryBlock * &, unsigned long)
[7]  sGet(class ccSequence * &, unsigned char, unsigned long)
```

```
[8]  sGet(class ccStackObj * &, unsigned long)
[9]  sGet(class ccString * &, unsigned char, unsigned long)
[10] sGet(double &, unsigned long)
[11] sGet(long &, unsigned char, unsigned long)
[12] sGetElmType(unsigned char &, unsigned long)
[13] sInsert(unsigned short const *, unsigned long, unsigned char)
[14] sInsert(class ccDictionary *, unsigned long)
[15] sInsert(class ccMemoryBlock *, unsigned long)
[16] sInsert(class ccSequence *, unsigned long, unsigned char)
[17] sInsert(class ccStackObj *, unsigned long)
[18] sInsert(class ccString *, unsigned long, unsigned char)
[19] sInsert(double, unsigned long)
[20] sInsert(long, unsigned long, unsigned char)
[21] sPut(unsigned short const *, unsigned char, unsigned long)
[22] sPut(class ccDictionary *, unsigned long)
[23] sPut(class ccMemoryBlock *, unsigned long)
[24] sPut(class ccSequence *, unsigned char, unsigned long)
[25] sPut(class ccStackObj *, unsigned long)
[26] sPut(class ccString *, unsigned char, unsigned long)
[27] sPut(double, unsigned long)
[28] sPut(long, unsigned char, unsigned long)
[29] sSize(void)
```

ccMemoryBlock

```
[0]  mbAllocate(unsigned long)
[1]  mbAppendMem(class ccMemoryBlock *)
[2]  mbApplyBOM(bool)
[3]  mbCopyMem(class ccMemoryBlock * &)
[4]  mbCvtDataTo(long)
[5]  mbDuplicateMem(class ccMemoryBlock * &)
[6]  mbExport(class ccString * &)
[7]  mbExport(void *, unsigned long)
[8]  mbGetDataPtr(void)
[9]  mbGetDataSize(void)
[10] mbGetErrCode(void)
[11] mbGetMemSize(void)
[12] mbImport(unsigned short const *, unsigned long, unsigned long)
[13] mbImport(class ccString *)
[14] mbInsert(class ccMemoryBlock *, unsigned long, unsigned long,
       unsigned long)
[15] mbLoad(void *, unsigned long)
[16] mbReadWholeFile(unsigned short const *, unsigned long)
[17] mbReadWholeFile(class ccString *, unsigned long)
[18] mbRepDataForm(long, unsigned long)
[19] mbRepDstPath(class ccString * &, unsigned short const *)
[20] mbRepDstPath(class ccString * &, class ccString *)
[21] mbRepSrcPath(class ccString * &, unsigned short const *)
```

```
[22] mbRepSrcPath(class ccString * &, class ccString *)
[23] mbReserveExtraMem(unsigned long)
[24] mbSet(unsigned char)
[25] mbSetDataSize(unsigned long)
[26] mbWriteWholeFile(unsigned short const *, bool, unsigned long)
[27] mbWriteWholeFile(class ccString *, bool, unsigned long)
```

ccString

```
[0]  strAppend(unsigned short const *, unsigned long, unsigned long)
[1]  strAppend(unsigned long)
[2]  strAppend(class ccString *, unsigned long, unsigned long)
[3]  strAssign(unsigned short const *, unsigned long, unsigned long)
[4]  strAssign(class ccString *, unsigned long, unsigned long)
[5]  strDeleteStr(unsigned long, unsigned long)
[6]  strFindAndRepStr(unsigned short const *, unsigned short const *)
[7]  strGetChar(long)
[8]  strGetStrBuff(unsigned long)
[9]  strGetStrPtr(void)
[10] strInsertStr(unsigned long, unsigned short const *, unsigned long,
       unsigned long)
[11] strLength(void)
[12] strPutChar(unsigned long, long)
[13] strReleaseStrBuff(unsigned long)
[14] strStrip(unsigned long, unsigned short const *)
[15] strUCharCount(void)
[16] strWCharCount(unsigned long &, long, long)
```

ccStackObj

```
[0]  soCopyObj(class ccStackObj * &)
[1]  soDuplicateObj(class ccStackObj * &)
```

A.3 Possible Future Compatibility Resolution

A future version of AppETAC.dll may be released with a linkable static library (for example, ETACIFace_*n*.lib) for linking with a main application program or an *external TAC library*. The static library would be in COFF (Common Object File Format) format, and would contain pseudo *ETAC interface* and *resource interface* C++ class definitions to automatically implement access to the real *ETAC interface* and *resource interfaces*. Such a system would alleviate the requirement of a workaround for non-MSVC compilers. Appropriate C++ header files would also be included with the static library. However, there are currently no plans to release such a static library.

Bibliography

The Official ETAC Programming Language Copyright © Victor Vella (2020).

Glossary

A

activate

a) 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.

b) 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*.

c) 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*.

boolean value

An integer interpreted as consisting of 32 binary flags, or a 'true' (`-1`) or 'false' (`0`) value. The *true* value is represented by having all the 32 binary flags set (achieved by the value `-1` based on a two's complement representation of integers). The *false* value is represented by having all the 32 binary flags unset. A *boolean value* is typically assigned by a hexadecimal number if used as binary flags, or by the `true` or `false` *intrinsic commands* if used as a logical condition.

C

call-back function (applies to an application program)

The C++ function (defined in an application program) that is executed when the **`run_app_fnt`** *command* is *activated*. There can be only one *call-back function* for each *main ETAC session* evoked by an application program.

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* cannot contain characters above U+00FF. *Comop identifiers* are case-sensitive.

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

compound stack object
A *stack object* that has a *resource value*. Sequence, procedure, dictionary, and memory *stack objects* are *compound stack objects*. *ETAC functions* and *data objects* are also effectively *compound stack objects*.

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*.

custom comop number
A positive integer identifying a particular custom module to execute for the *comop*. The module exists in the *standard TAC library* or an *external TAC library*, and is implemented in machine code not *ETAC code*.

D

data dictionary
The *dictionary* contained in a *data object*. That *data dictionary* is identified by the name defined by the private pre-processor definition ‹_DATA_DICT›.

data form
The form of data contained in a memory *stack object*. The different forms of data are specified in the table Data Form Indicators. A memory *stack object* contains an internal member indicating the *data form* of the memory's data.

data object
The container of *dictionary* used as a programmer-defined data structure consisting of *stack objects* identified by name (see *dictionary keyword*). The *dictionary* itself is identified by the name defined by the private pre-processor definition ‹_DATA_DICT›.

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 keyword
The label of a *dictionary item*. A *dictionary keyword* typically has the syntax of a *comop identifier*.

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

duplicate (of a *stack object*)
To entirely reproduce a *stack object* and its value into another *stack object* (replacing that other *stack object*) such that the reproduced value and the original value share no resources. *Duplication* is recursive. If the *stack object* does not have a *resource value*, then the *embedded value* of that *stack object* is reproduced.

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 function
The container of a special ETAC created *procedure* that creates a *local dictionary* then assigns the *object stack* arguments to that *dictionary* before calling the programmer-defined *procedure*. An *ETAC function* is typically accessed via a *function command*.

ETAC interface
An implementation of the definition of a particular C++ class by means of which C++ code can initially interact with the *ETAC interpreter*. The term "*ETAC interface*" may also be used to mean the said class itself. The C++ class name defining the *ETAC interface* is `ccTAC`.

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 then 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 session

The period devoted to the processing of *ETAC code* by the *TAC processor* after having been processed by the *script interpreter* or *binary interpreter* (whichever is appropriate). New *ETAC sessions* can exist among a given *ETAC session* for different *ETAC code*. Therefore, a given *ETAC session* can produce a new *ETAC session* (relating to different *ETAC code* from the given *ETAC session*) so that when the new *ETAC session* ends, the given *ETAC session* resumes.

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".

ETAC variable

A *dictionary item* whose *dictionary keyword* is in the form of a *variable identifier*, and whose *stack object* is intended to be different at various times during an *ETAC session*. The 'value' of an *ETAC variable* is the value of the said *stack object*. The '*variable object*' is the said *stack object* itself.

ETL function

A C++ language function defined in an *external TAC library* to be executed from *ETAC code*.

external TAC library

A library of functions implemented by a programmer in the C++ programming language to extend the functionality of the ETAC programming language. The functions exist in a **Windows**® DLL (dynamic linked library), but are used as *comops* or *ETAC functions* by the ETAC programmer.

F

function command

A *command* associated with a *dictionary item* whose *stack object* is an *ETAC function*. When a *function command* is 'called', then its corresponding *ETAC function* is executed. When a *function command* is '*activated*', then its corresponding *ETAC function* is pushed onto the *object stack*.

function member

A *member* whose *stack object* is an *ETAC function*.

I

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

intrinsic command
A *command* that is associated with a function defined internally to the *ETAC interpreter*, or a *stack object* created by such a *command*. The *activation* of an *intrinsic command* does not involve the *dictionary stack* (an *intrinsic command* is *activated* directly).

intrinsic operator
An *operator* that is associated with a function defined internally to the *ETAC interpreter*, or a *stack object* created by such an *operator*. The *activation* of the *stack object* created when an *intrinsic operator* is *activated* does not involve the *dictionary stack* (an *intrinsic operator* is *activated* directly).

L

link (of a *comop stack object*)
A *comop stack object* that has an internal reference directly to its *dictionary item*, thus avoiding a *dictionary* search for that *comop*.

loader script
This is an automatically loaded *ETAC code* file typically used to set up the ETAC environment before the specified *ETAC code* files are executed. The file name for the default *loader script* is RunETAC.btac or RunETAC.etac. The default location of the *loader script* is in the System directory under the installed RunETAC directory.

local dictionary
A *dictionary*, typically existing temporarily, that is identified by the name defined by the pre-processor definition ‹_LOCAL_DICT›. A *local dictionary* is typically used to contain the local variables of an *ETAC function*.

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.

M

main ETAC session
An *ETAC session* and all other new *ETAC sessions* produced directly or indirectly from that *ETAC session*, but not itself produced from any other *ETAC session*. A *main ETAC session* is typically begun via the RunETAC.exe and the AppETAC.dll computer programs.

managed reference
An internal reference to a *resource value* using a reference counting system. Circular references are not supported.

member (of a *data object*)
A *dictionary item* of the *dictionary* contained in a *data object*.

member variable (of a *data object*)
A *member* of a *data object* that is an *ETAC variable* (or rarely a *TAC variable*).

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*. One, where the operands are delimited by parentheses, and two, 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*.

P

procedure
A special *sequence*, which, when *activated*, the elements of that *sequence* get *activated*. The elements of a *procedure* are typically command *stack objects*.

procedure expression
A group of *script tokens* that creates a *procedure* when *activated*.

R

resource interface

An implementation of the definition of certain C++ classes by means of which C++ code can interact with non-numerical *TAC objects*. The term "*resource interface*" may also be used to mean one of the said classes themselves. The C++ class names defining *resource interfaces* are: `ccDictionary`, `ccSequence`, `ccDataObject`, `ccMemoryBlock`, `ccString`, and `ccStackObj`.

resource object

The *managed reference* part of a *resource interface* understood as if it were a *stack object*.

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).

resource variable

A variable that effectively points to an instance of a *resource interface*.

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 *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.

sequence

A *stack object* having a *resource value* consisting of any number (including zero) of indexed *stack objects* understood as a unit. The indexed *stack objects* are the 'elements' of the *sequence*. The first *element* begins at index one, the second *element* is at index two, and so on. The number of *elements* in a given *sequence* is variable but limited by available memory. The *elements* of a *sequence* can be any type of *stack objects*, including *sequences*.

sequence expression

A group of *script tokens* that creates a *sequence* when *activated*.

stack object

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

standard TAC library

This is a library of custom *comops* that are implemented internally to the *ETAC interpreter*. Each *comop* has a unique *custom comop number*. Custom *comops* must be loaded via the execute custom *TAC text instruction* or the custom *command* before they can be *activated* (this is done automatically by the *loader script*).

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”.

TAC variable

A *dictionary item* whose *dictionary keyword* has the syntax of a *comop identifier*, and whose *stack object* is intended to be different at various times during an *ETAC session*. The ‘value’ of a *TAC variable* is the value of the said *stack object*. The ‘*variable object*’ is the said *stack object* itself.

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 a 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 <u>not</u> a *u-char* (it is a *w-char*).

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.

variable object

The *stack object* identified by a *TAC variable*, *ETAC variable*, or *member variable*.

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 a 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 one *w-char*.