CET BASIC extensions library (CBExt)
W32 App Builder Version

Introduction

CET BASIC object-oriented extension is small set of external functions to allow CET BASIC programs utilize objects defined outside of BASIC runtime system, usually as Dynamic Link Library.

CBExt supports:

- Dynamic object creation and deletion
- Data encapsulation
- Run-time type checking
- Garbage collection

Terms

Class file	file containing one or more class definitions. Class file must be loaded to BASIC runtime system to make classes available to BASIC program.
Class	data definitions and data manipulation methods. Class is known to BASIC program by its name.
Object	class instance, referenced from BASIC program using **Object ID.**
Object ID	integer (%) BASIC variable
Local object	object that cannot be transferred across CHAIN/LINK calls. Local objects are deleted automatically when program exit or link to another program
Common object	object that can be transferred through CHAIN/LING by declaring **Object ID** as COMMON variable. Common objects are deleted automatically when whole program exits.
Method	request to object to perform some actions with object's data, system resources attached to object, and so on.
Property	object property, can be read or written by BASIC program. Properties can have read access only or both read and write.
Static method	request to class to perform some actions with common class data.
Static property	class property, can be read or written by BASIC program. Properties can have read access only or both read and write.

Data types

CET BASIC programs uses following data types when call CBExt:

- Integer integer BASIC variable (%)
- Double float BASIC variable (without suffix). CBExt provides automatic conversion from internal CET BASIC representation to double-precision float format and back.
- String string BASIC variable ($)
- Object integer BASIC variable. CBExt checks types of object what this variable is referenced to.

CET BASIC program interface to CBExt

Functions

CBExt functions are called from CET BASIC program using CALL statement Most of functions accepts variable number of arguments. Number of arguments in variable part must be passed to function as last argument.

Load class file

cetObjectLoadClassFile(FileName$)

FileName$ - class file name.

In order to use classes from class file it must be loaded into BASIC runtime. Runtime keeps class file loaded until it exits. Runtime can have some class files loaded when it starts.

Create Common Object (can be passed across CHAIN/LINK)

cetObjectCreateCommon(ADDROF(Object%), ClassName$, ..., Count%)

Object% - integer variable where new object reference will be placed
ClassName$ - class name (class file containing class must be loaded)
... - parameters for class constructor
Count% - parameters counter (do not include Object and ClassName arguments)

Creates object that will not be destroyed automatically when CHAIN/LINK will be performed. To pass object reference through CHAIN/LINK declare variable Object% as COMMON.

Create Local Object (destroyed automatically when RUN/CHAIN/LINK)

cetObjectCreate(ADDROF(Object%), ClassName$, ..., Count%)

Object% - integer variable where new object reference will be placed
ClassName$ - class name (class file containing class must be loaded)
... - parameters for class constructor
Count% - parameters counter (do not include Object and ClassName arguments)

Destroy Object

cetObjectDestroy(Object%)

Object% - object reference

Explicit object destroying. All local objects are destroyed implicitly when BASIC program finishes executing. All common objects are destroyed implicitly when BASIC runtime exits.

Execute Method

cetObjectExecute(Object%, {Method$, Method%}, ..., Count%)

Object% - object reference
Method$ - method name
Method% - method ordinal number
... parameters

Count% - parameters counter (do not include Object and Method arguments)

Method to be executed can be defined by method name or by ordinal number. Ordinal number is defined inside of class file.

Get Property value

cetObjectGetProperty(Object%, {Property$, Property%}, ADDROF(Result[%|$]))

Object% - object reference
Property$ - property name
Property% - property ordinal number
Result, Result%, Result$ - return value

Property can be defined by its name or ordinal number. Result variable must be right type.

Set Property value

cetObjectSetProperty(Object%, {Property$, Property%}, Value[%|$])

Object% - object reference
Property$ - property name
Property% - property ordinal number
Value, Value%, Value$ - new value

Property can be defined by its name or ordinal number. Property must have write access.

Execute Static Method

cetObjectExecuteStatic(Class$, {Method$, Method%}, ..., Count%)
Class$ - class name
Method$ - method name
Method% - method ordinal number
... parameters
Count% - parameters counter (do not include Object and Method arguments)

Get Static Property value

cetObjectGetStatic(Class$, {Property$, Property%}, ADDROF(Result[%|$]))

Class$ - class name
Property$ - property name
Property% - property ordinal number
Result, Result%, Result$ - return value

Property can be defined by its name or ordinal number. Result variable must be right type.

Set Static Property value

cetObjectSetStatic(Class$, {Property$, Property%}, Value[%|$])

Class$ - class name
Property$ - property name
Property% - property ordinal number
Value, Value%, Value$ - new value

Property can be defined by its name or ordinal number. Property must have write access.

Run-time errors

All functions defined above can generate trappable BASIC errors. There are common errors for CBExt and errors specific for class.

Common errors are:

- 13001 Bad class file (class file not found or its format is invalid)
- 13002 Method not implemented
- 13003 Class not found
- 13004 Method not found
- 13005 Invalid type of argument (passed to cetObject… function)
- 13006 Invalid object reference (Object% parameter is not reference to valid object)

Macros

CBExt is supposed to be used with C-style preprocessor Bpp.

Class file can have set of macro defined in 'header' file that can be included into BASIC program using #include operator. Macros can simplify programming and improve BASIC program readability.
For example, instead of

```
CALL cetObjectCreate( ADDROF(printer%), "Printer", "LaserJet", 1)
CALL cetObjectGetProperty( printer%, "PageLength", ADDROF(pagelen%) )
CALL cetObjectExecute( printer%, "OutText", str$, 1 )
```

programmer can use

```
PrinterCreate( printer%, "LaserJet" )
PrinterGetPageLength( printer%, pagelen% )
PrinterOutText( printer%, str$ )
```

Preloaded class files

W/32 runtime preloads some class files when it starts. If class file not found, Message Box is displayed.