CA-Clipper®

For DOS

Version 5.3

Technical Reference Guide

June 1995

COMPUTER ASSOCIATES®
Software superior by design.

PRINTED ON
RECYCLED PAPER

Printed in the United States of America
Computer Associates International, Inc.
Publisher

Contents

Chapter 1: Introduction

Chapter 2: Internal Architecture

Chapter 3: Using the Extend System API

Chapter 4: Extend System API Reference

Chapter 5: Item API Reference

Chapter 6: Using the Fixed Memory API

Chapter 7: Fixed Memory API Reference

Chapter 8: Using the Virtual Memory API

Chapter 9: Virtual Memory API Reference

Chapter 10: Error System API Reference

Chapter 11: File System API Reference

Chapter 12: General Terminal API Reference

Chapter 13: Replaceable Database Driver API Reference

Chapter 14: Light Lib Graphics API Reference

Chapter 15: CA-Clipper/Exospace API Reference

Index

Chapter 1 Introduction

This is the *Technical Reference Guide* for CA-Clipper 5.3. It contains specification-level documentation for each of the CA-Clipper APIs, as well as information on how to use many of them. API is an abbreviation for Application Programming Interface, which describes systems that give you the means to interface your CA-Clipper applications with other systems.

For a DOS online version of the reference chapters in this guide, accessible while operating your program editor or any other development utility, use *The Guide To CA-Clipper*. This is an online documentation system that uses the Norton Instant Access Engine. See your *Programming and Utilities Guide* for more information on how to use this system.

This release of CA-Clipper includes a Workbench that provides a Windows online Help system. In addition to providing Workbench help, it also allows you to access the reference chapters in this guide. See your *User Guide* for information on how to use this system.

Important! *Some of the topics in these online documentation systems are intended for advanced CA-Clipper developers. Much of this information is presented at a fairly high level and requires programming knowledge beyond the CA-Clipper language. Other parts are useful to users of all levels.*

Based on your own experience level with the CA-Clipper language, you should decide whether you wish to take advantage of these new and advanced features. The Technical Reference Guide addresses extensions to CA-Clipper. Understanding this information should enable you to increase the power and effectiveness of your applications.

Internal Architecture

This chapter provides you with an overview of the internal architecture of the CA-Clipper runtime environment. The information presented here will be helpful when you are designing C or Assembly language programs that you want to interface with a CA-Clipper application.

Extend System API

Two chapters concerning the Extend System API are provided: one that explains how to use the system and another that serves as an alphabetical reference to the functions that make up this API.

The Extend System API is a set of functions designed for use in your C, C++, or Assembly language programs. They allow you to retrieve parameters passed from a CA-Clipper program, store their values for manipulation, and return values back to the CA-Clipper program. Thus, using the Extend System API you can call functions written in C, C++, or Assembly directly from a CA-Clipper program.

Item API

A single chapter serves as an alphabetical reference to the Item API. This API gives your Extend routines the ability to manipulate data items using CA-Clipper data types, including the ability to evaluate code blocks.

Fixed Memory (FM) API

Two chapters concerning the FM API are provided: one that explains how to use the system and another that serves as an alphabetical reference to the functions that make up this API.

The functions in this API are designed for use within the functions you write using the Extend System API. They let you allocate and free fixed memory segments when you need only a small amount of memory.

Virtual Memory (VM) API

Two chapters concerning the VM API are provided: one that explains how to use the system and another that serves as an alphabetical reference to the functions that make up this API.

The functions in this API are designed to give the functions you write using the Extend System API access to the powerful CA-Clipper Virtual Memory Manager (VMM). The VMM allows you to work with strings and arrays that are much larger than the conventional memory available on your computer by swapping data into and out of conventional memory, giving the effect of a much larger virtual memory space.

Error System API

A single chapter serves as an alphabetical reference to the Error System API. This API lets your Extend routines produce error "objects" that are compatible with the CA-Clipper Error system and call the current CA-Clipper Error Handler routine.

File System API

A single chapter serves as an alphabetical reference to the File System API. This API gives your Extend routines access to the CA-Clipper low-level file routines, allowing you to create, access, and delete files without the added overhead of the C library I/O functions or custom Assembler routines.

General Terminal API

A single chapter serves as an alphabetical reference to the General Terminal API. This API is a set of functions that give you access to the CA-Clipper General Terminal system, allowing you to control the terminal from within your Extend routines.

Replaceable Database Driver (RDD) API

CA-Clipper provides a default database driver, DBFNTX, and several other RDDs to give you access to the database, memo, and index file formats of many popular database software products. This capability allows CA-Clipper applications to access and manipulate files created by other database engines.

The RDD API is a system that lets you create your own RDDs. It is documented in a single chapter that serves as a categorized, alphabetical reference.

Light Lib Graphics API

A single chapter serves as an alphabetical reference to the LightLib Graphics API. This API gives your Extend routines the ability to manipulate your output in different video modes (both text and graphic).

Organization of Reference Items

Each of the API reference chapters is arranged in alphabetical order by item name. Each item begins on a page by itself so you can easily find what you need. The entries are divided into the subsections listed below. If a subsection is inappropriate for a particular item, it is omitted.

Note: The "Replaceable Database Driver API Reference" chapter does not strictly adhere to this standard organization because the structures and methods that it documents are slightly different than the functions defined for the other APIs.

- ***Item name*** is the name of the item.
- ***Short descriptor*** describes the purpose or action of the item.
- ***C Prototype*** defines a prototypical usage of the specified item. See the C Prototype Symbols and Conventions section below for more information on variable names and other special symbols used in the prototype representations.
- ***Arguments*** defines the meaning of each item argument. Arguments are variables and/or keywords used at the invocation of an item.
- ***Returns*** defines the return value of a function, including its data type.

- ***Description*** describes the basic operation and usage of the item.
- ***Notes*** describes operations that demand special note, including eccentric or exceptional behavior. This section also describes synergistic or conflicting behaviors that relate to other items.
- ***Examples*** are code fragments with resulting output, if appropriate. Examples generally serve three purposes: usage demonstration, validation of operational rules and/or return values, and illumination of a programming concept.
- ***Files*** is the list of files on which the current item depends. These files fall into the following categories:
  - Libraries—usually located in \CLIP53\LIB
  - Header files—usually located in \CLIP53\INCLUDE
- ***See Also*** is an alphabetical list of items related to the usage of the current item.

Note: The structure of items is designed to facilitate use based on the level of expertise you may have with an item. If you are experienced, a quick glance at the item name, short descriptor, and prototype sections should give enough information. If you are less experienced, read the more detailed information thoroughly.

C Prototype Symbols and Conventions

The C Prototype representations used in the alphabetical reference chapters show how to use the item. The general format is the return data type followed by the function name (presented using a combination of uppercase and lowercase letters) with arguments listed in parentheses. Arguments in square brackets ([ ]) are optional.

Arguments are always preceded by their data type. If a data type is in uppercase letters, it is a typedef defined in the header file Clipdefs.h or one of the .api header files. Otherwise, it is a C data type.

Argument names consist of a data type designator followed by a logical descriptor. The data type designation is a prefix chosen from the table that follows. Prefixes are always lowercase and logical descriptors are always capitalized. Logical descriptors describe the meaning of the argument. For example, *uiLength* represents the length of a string as an unsigned integer.

C Prototype Argument Prefixes

Type Prefix	Data Type	Typedef in Clipdefs.h
c	unsigned char	BYTE
fp	unsigned char far * char far *	BYTEP
fzp	unsigned char far *	BYTEP (NULL-terminated)
i	short, int	SHORT
ip	short far * int far *	SHORTP
ui	unsigned short, unsigned int	USHORT WORD
uip	unsigned short far * unsigned int far *	USHORTP WORDP
l	long	LONG
lp	long far *	LONGP
ul	unsigned long	ULONG, DWORD
ulp	unsigned long far *	ULONGP, DWORDP
b	unsigned short	BOOL
bp	unsigned short far *	BOOLP
d	double	XDOUBLE
p	void *	ERRORP
h	unsigned short	HANDLE FHANDLE
itm	void _near *	ITEM
eval	struct	EVALINFO
eval	EVALINFO far *	EVALINFOP
vnp	void near *	NEARP
vlp	void far *	FARP

Tip: A glossary of CA-Clipper 5.3 terms used throughout the documentation is provided in your *Reference Guide, Volume 2*. It is a comprehensive dictionary with entries consisting of the item name, the identity of one or more categories to which the item belongs, and a short definition.

Chapter 2
Internal Architecture

The CA-Clipper internal architecture supports the Replaceable Database Driver architecture as well as the alternate Terminal Drivers and other services. This architecture consists of several interconnected language and memory features in which specific elements may be combined and used. Some of these features and code specifications are important to advanced programming in C and Assembly.

Note: References to C programming language also pertain to C++.

In This Chapter

This chapter provides C and Assembly language programmers an overview of the internal architecture of the CA-Clipper runtime environment. Although the information presented in this chapter may not be essential to your understanding of the basics of interfacing CA-Clipper to Assembly or C, it is certainly helpful for large-scale system design in general.

Note: The information provided in this chapter is specific to CA-Clipper version 5.3 and is not guaranteed to be accurate for past or future versions of CA-Clipper.

The following topics are covered:

- Overview of the runtime environment
- The eval stack
- VALUE
- Variable management
- Understanding references

Overview of the Runtime Environment

CA-Clipper applications are based on the C large memory model convention. A CA-Clipper application uses all available conventional memory while it runs. Both segment and offset (far pointers) address code and data in CA-Clipper programs.

CA-Clipper object modules contain a combination of *pcode* (CA-Clipper code) and Assembly language. Once object modules are linked with the CA-Clipper runtime system, they become data. The CA-Clipper runtime system, found mostly in CLIPPER.LIB, translates the proprietary pcode format of object files into a running application.

Upon startup, the CA-Clipper runtime system initializes several areas, or *arenas*, for executing pcode. These arenas include: the code arena, DGROUP, and fixed memory heap, as well as the dynamic memory arena.

Code Arena

The code arena is where CA-Clipper's runtime system operates. All C and Assembly code executed during the course of the application exists in this fixed-size chunk of memory.

The size of the code arena is determined at link time; it is always the same for a given (.EXE). This is also known as an application's *load size*.

Load size is affected only slightly by the amount of compiled CA-Clipper code in the application (because, by default, all compiled CA-Clipper code is dynamically overlaid). However, the code arena is significantly affected by the amount of non-CA-Clipper (C and Assembler) code being used. This includes CA-Clipper support code and third-party library code written in C or Assembly.

DGROUP

DGROUP is the location of the evaluation stack (see The Eval Stack, later in this chapter), static and local references, and other items.

CA-Clipper, at runtime, uses DGROUP sparingly when creating Extend routines. As you execute any CA-Clipper application, you may use the //INFO switch to discover the amount of DGROUP available. This information is reported in the "DS Avail" field.

Fixed Heap

The memory allocated for system tables and other non-virtualized data is stored in the fixed heap. Memory allocated by C or Assembler functions (for example, Summer '87 versions of third-party libraries or overlay managers) is usually fixed memory. An application's fixed-memory requirements depend on which features of the system are being used.

For most CA-Clipper applications it ranges from 16K to 64K. Third-party libraries or overlay managers, which allocate fixed memory, may increase this requirement, as may certain CA-Clipper programming practices, (e.g., building a large number of runtime macros).

Note that, once expanded, the fixed heap does not shrink. This is a design necessity of the CA-Clipper runtime system to accommodate the needs of the dynamic arena.

Use the CA-Clipper //INFO switch from the DOS command line to discover how much fixed heap is available. This information is reported in the "Fixed Heap" field.

Dynamic Arena

The dynamic arena is the real-memory storage area of CA-Clipper's Virtual Memory Manager (VMM). The entire dynamic arena is managed by VMM. It is in VM that public and private symbols, strings, code blocks, objects, arrays, and database buffers are stored. Access to this area is restricted to the VM API (detailed in the chapter "Using the Virtual Memory API" found later in this guide).

After allocation of the other arenas is complete, CA-Clipper uses all remaining real memory for its dynamic arena. If this arena is too small, programs may be sluggish and may generate "memory low" errors at random intervals. If a program shrinks the dynamic arena by expanding the fixed heap too much, or makes an allocation that VM cannot fulfill, the user will receive an internal VM error.

The Eval Stack

Internally, CA-Clipper is organized as a stack-based machine. It uses an area of memory called the eval stack (evaluation stack) to contain operands, function parameters, and intermediate results. The eval stack is simply a contiguous group of VALUEs that are accessed as a stack. Its use is analogous to use of the processor stack by C programs.

For example, in a CA-Clipper function call, parameters are placed (pushed) onto the eval stack before the function is executed. The function operates on the topmost items in the eval stack and produces a result. After the function completes, the parameter values are removed (popped) from the eval stack and the function result is posted.

VALUE

Internally, CA-Clipper represents data values using a small data structure called a VALUE. The content and format of a VALUE change, depending on the type of data they represent. Simple data, such as integers, are coded directly into the VALUE. Larger data, such as strings or arrays, cannot be directly contained in a VALUE. Instead, the VALUE contains a "reference" to the string or array (references are discussed in the following sections).

Variable Management

The internal representation of a CA-Clipper variable reflects the variable's storage class (these are discussed briefly in the following sections). For all storage classes, retrieving a variable's value consists of pushing its VALUE onto the eval stack. Assigning to a variable consists of copying a VALUE from the eval stack into the variable.

LOCAL Storage Class

LOCAL variables are the simplest variables. They are, in effect, locations within the eval stack. To push a LOCAL variable, the system copies a VALUE from one position in the eval stack to another.

STATIC Storage Class

STATIC variables are similar to LOCAL variables. However, they cannot be interspersed with the other contents of the eval stack since they do not come and go with function activations. Instead, they are assigned fixed locations at one end of the eval stack. Like LOCAL variables, pushing a STATIC variable consists of copying its VALUE.

MEMVAR Storage Class

MEMVAR is the storage class of PRIVATE and PUBLIC variables. These variables are more complex than LOCAL or STATIC variables because they consist of more than just a VALUE: they also have an associated symbolic name that lets you refer to them by name during execution (e.g., in macros). MEMVAR variable references are stored in a dedicated virtual memory segment. Pushing a MEMVAR involves locating the VALUE that is currently associated with the variable's name and copying that VALUE onto the eval stack.

FIELD Storage Class

Variables of the FIELD storage class differ from other variables because they have no memory location at all. To push a FIELD, the system generates a request to a database driver. The database driver creates an appropriate VALUE (in the form of an ITEM, which is discussed in the "Item API Reference" chapter) which is then copied onto the eval stack.

Understanding References

It is important to distinguish between the different meanings of the word "reference." In CA-Clipper, there are two distinct kinds of references:

1. ***An object reference (an OREF):*** This is a special internal quantity that identifies the location of a character, array, or code block value in virtual memory. Because these kinds of values can be very large, a VALUE cannot directly contain them. Instead, the VALUE contains an OREF that allows the data to be located when needed. Whenever necessary, the system performs a de-referencing operation to convert an OREF into a memory address.

2. ***A variable reference (a VREF):*** Like an OREF, a VREF is a special internal quantity. Instead of referring to a piece of data, however, a VREF refers to a CA-Clipper variable. A VREF is created whenever a variable is passed by reference in a function call or DO...WITH statement. VREFs are special in that they exist only on the eval stack and only as parameters during a function call. A VREF cannot be assigned to a variable or stored in an array.

When dealing with C, the terminology of references can become ambiguous. This is because C programmers often use the words reference and pointer interchangeably. Where CA-Clipper is involved, however, the two words have very distinct meanings. A reference is one of the special quantities discussed previously. A pointer is an actual memory address or a C variable containing such an address.

Array Values

As noted, a VALUE cannot directly contain an array. Instead, it contains an OREF for the array. When an array is assigned to a variable, the system overwrites the variable's VALUE with a new VALUE containing an OREF to the array. The array itself is a group of VALUEs stored in virtual memory. Each element of the array is a VALUE and, since any VALUE can contain an OREF, each element can refer to another array.

Character Values

As noted, a VALUE cannot directly contain character data. Instead, it contains an OREF that allows the data to be located when it is needed. As with arrays, assigning a character value to a variable overwrites the variable's VALUE with a new VALUE containing an OREF to the character data. Note that, as with arrays, assigning a character value from one variable to another duplicates the VALUE (i.e., the OREF); the character data itself is not duplicated.

Note that this reference-based memory management technique is the same for strings, arrays, and code blocks. CA-Clipper's garbage collector monitors OREFs. When there are no longer any references to a particular piece of data, the space occupied by that data is automatically reclaimed.

You may easily observe the array reference technique at the CA-Clipper level: if the same array reference is assigned to two variables, you can use either variable to modify the array.

With character data, however, the reference handling is not noticed at the CA-Clipper level. CA-Clipper operators and functions never act directly on character data—they always create a new character value as their result.

Although unnoticed at the CA-Clipper level, the reference technique does have an effect at the Extend System API level.

Parameters

At the CA-Clipper level, passing a parameter by reference means that CA-Clipper will create a VREF (previously discussed) and place it onto the eval stack before calling the function. This lets the function (whether written in CA-Clipper or C) gain access to the variable.

Passing by value, on the other hand, means that CA-Clipper places a copy of the specified VALUE onto the eval stack. Note, in particular, that passing a string by value does not mean that CA-Clipper will make a duplicate copy of the string. Only the VALUE (i.e., the OREF) is copied, not the string itself.

The Extend System API's _parc() function accesses character type parameters passed from CA-Clipper. It returns a pointer to a series of bytes representing the parameter value.

To obtain this pointer, _parc() first locates the parameter's VALUE. If the parameter was a reference to a variable (a VREF), _parc() finds the variable first and retrieves the VALUE from there. Otherwise, the VALUE is present on the eval stack and _parc() obtains it directly. Once _parc() has obtained the VALUE, it extracts the OREF from it and performs a de-referencing operation. It then places a lock on the virtual memory segment containing the character data (preventing it from moving in conventional memory) and returns a pointer to the first byte of the data.

As noted previously, the fact that OREFs are often duplicated when character values are assigned, although transparent at the CA-Clipper level, can become an issue when you use _parc(). If a string's OREF has been duplicated, the string may be referred to by several variables or array elements at the same time. This is why the documentation for _parc() warns against using the pointer obtained from _parc() to modify the string directly—doing so may have the effect (at the CA-Clipper level) of modifying several variables at once.

Summer '87 also allowed multiple references to the same string in certain cases. Generally, though, when a string was assigned to a variable, it was duplicated, giving each variable its own copy of the string.

Summary

This chapter has described the memory model within which CA-Clipper functions and the conceptual partitioning of the model into code, DGROUP, fixed heap, and dynamic arenas.

The CA-Clipper runtime operates in the code arena, running all C and Assembly code in this fixed-size portion of memory.

DGROUP contains the evaluation stack which contains operands, function parameters, and intermediate results. It also contains static VALUEs, ITEMs, and near data used by C and ASM programs.

The fixed heap is an expandable, unshrinkable area of memory used for system tables and other non-virtualized data, as well as for memory allocated by C and Assembly programs.

The dynamic memory arena is managed by the VMM and may contain public and private symbols, strings, code blocks, objects, arrays, database buffers, and user VM API data.

The implications of the variable types, LOCAL, STATIC, MEMVAR, and FIELD, was briefly discussed, as well as the proper addressing of data through object (OREF) and variable (VREF) referencing. The internal handling of parameters was also detailed as part of this discussion.

Chapter 3
Using the Extend System API

The Extend Application Programming Interface (API) is a key element in CA-Clipper's open architecture. It allows you to write routines in other languages that can be invoked directly from a CA-Clipper program, just as functions or procedures written in CA-Clipper.

In This Chapter

This chapter introduces you to the Extend System API, explains its basic concepts, and provides specific details about using it with C, C++, and Assembly language programs. The following topics are covered:

- The Extend System Protocol
- Protected-mode addressing
- Parameter and return values
- Using the Extend System API with C and C++
- Using the Extend System API with Assembly language
- Operating on CA-Clipper values
- Compatibility issues

Note: References to the C programming language also pertain to C++.

The Extend Protocol

The Extend System API consists of several interrelated elements:

- ***Extend routines:*** Routines written in a language other than CA-Clipper that can be called from CA-Clipper with or without return values.
- ***Runtime data structures:*** A set of internal data structures within a compiled CA-Clipper application that allow parameter and return values to be manipulated.
- ***Interface functions:*** Functions supplied in CLIPPER.LIB which allow Extend routines to access and manipulate the above runtime data structures.
- ***Header files and definitions:*** A series of header files are provided with CA-Clipper in the INCLUDE subdirectory that help the C and Assembly language programmer use the Extend System API effectively and efficiently.

Perhaps the most important element of the Extend System API is the set of guidelines that specify how it is used. These guidelines, known collectively as the Extend Protocol, address the following issues:

- Function naming rules
- Memory model
- Calling conventions
- Data types and type conversions
- Allowable operations

Function Naming

Calling Extend routines from CA-Clipper necessarily involves linking two or more object files containing compiled code. One or more of the object files will be compiled CA-Clipper programs; others will be compiled Extend routines written in another language (usually C or Assembly language).

During compilation, CA-Clipper makes a note of any procedures or functions that the source program refers to, but does not define. These references are known formally as external references. External references can be created by calling a function or procedure that is not defined in the object, by specifically requesting (using REQUEST) a symbol, or by declaring the symbol as EXTERNAL. The compiler places the names of all external references in the object file along with the compiled program code. During linking, the names in the object file are used by the linker to *resolve* the external references between modules. For each external reference, the linker searches other object modules and libraries for a public symbol with the same name. If such a symbol is found, the linker includes the defining module in the executable file. Otherwise, a link error is generated.

When you link Extend routines with a CA-Clipper application, care must be taken to ensure that the names of the Extend routines will be properly resolved by the linker. The following rules apply to Extend routine names (public symbols) in object modules:

- Symbols must begin with a letter or an underscore character and may contain only letters, numbers, and underscore characters. Assemblers typically have less restrictive naming conventions, but functions that are to be called from CA-Clipper programs must conform to the CA-Clipper naming convention.

- CA-Clipper recognizes only the first ten characters of procedure and function names. Names longer than ten characters are truncated during compilation. Many languages allow longer names, but you should give names of ten characters or less to functions that are to be called from CA-Clipper.

- CA-Clipper programs are not case-sensitive and all procedure and function names are converted to uppercase during compilation. Extend routines which are to be called from CA-Clipper should be given names containing only uppercase characters.

- Many compilers form the name of a public symbol by adding an underscore character to the name of the associated routine. Microsoft refers to this as *symbol decoration*. For functions that are to be called from CA-Clipper, symbols should not be decorated in any fashion.

Note: Most linkers operate in case-insensitive mode unless a special linking option is used. It is still good practice, however, to ensure that function names are written to the object file as all uppercase.

Memory Model

The Intel 8086 family of microprocessors uses a *segmented addressing* scheme. Programs are organized into memory *segments*, each of which may be up to 64 K in size. Memory addresses consist of two parts: a segment number and an offset. When a program needs to access a particular memory location, it loads the appropriate segment number into a special hardware register called a segment register. Various offset values are then used to access the data in that segment.

The term *memory model* refers to the organization of memory segments in a program. *Small model* programs are programs that can fit entirely in one or two segments. For these programs, the segment registers are loaded at startup, and they retain the same values for the duration of execution. Under this scheme, a memory location can be specified using only an offset, since the segment numbers are always the same. These simple offset addresses are often referred to as *near* addresses.

Large model programs consist of many segments. In these programs, all addresses are specified using both a segment part and an offset part, and the segment registers are loaded with appropriate values whenever necessary. Memory addresses which include both segment and offset parts are called *far* addresses.

Compiled CA-Clipper applications are large model programs, and the addresses used within them are far addresses.

Calling Convention

When one compiled function calls another, they must both use the same *calling convention*. This means that they must both use the same standardized method for calling, returning, and handling parameter and return values.

Compiled CA-Clipper applications use the C large model calling convention for calling and returning. Function calls are implemented using far call instructions and called functions must return via a far return instruction (these are call and return instructions that use far addresses; see the Memory Model section).

The C convention places other requirements on called functions including the preservation of certain hardware registers. These requirements are of concern primarily to Assembly language programmers and are discussed in the Using the Extend System API with Assembly Language section later in this chapter.

Although the C convention is used for calling and returning, it is not used for parameters passed from a compiled CA-Clipper application to a C or Assembly language function nor for values returned from a C or Assembly language function to a compiled CA-Clipper application. Handling of parameter and return values is discussed in the following section.

Protected-Mode Addressing

This section provides basic information about protected-mode addressing. It covers the following topics:

- Real-mode addressing vs. protected-mode addressing
- 16-bit, segmented, protected-mode addressing
- Segment descriptors on 80286 and 80386 (or greater) machines
- Protected-mode programming restrictions

Protected-Mode Addressing

Utilizing the processor's protected mode allows you to address considerably more memory than you can in its real mode. This section describes how 16-bit protected-mode addressing differs from real-mode addressing.

What Is Protected Mode?

Protected mode, or protected virtual address mode, is the native addressing mode of the Intel 80286, 80386, and 80486 (or greater) microprocessors. It supports multi-tasking, virtual-memory operating systems.

Protected mode differs from real mode, which is the native addressing mode of the Intel 8086/88 microprocessors, in that protected mode allows access to physical memory addresses above 1 MB. (This memory is also called extended memory.) While an 8086 or 8088 microprocessor can only address 1 MB, an 80286/386/486 CPU running in 16-bit protected mode can address up to 16 MB of memory.

DOS and Protected Mode

DOS was developed for the 8086/88 microprocessors, so it runs only in real mode. Because the top 384 KB of the first megabyte of memory (the High Memory Area) is reserved for the ROM BIOS and memory-mapped hardware, there is no more than 640 KB of usable memory that DOS is able to address directly.

CA-Clipper/Exospace helps you to overcome this restriction on 80286/386/486 machines. CA-Clipper/Exospace allows the development of protected-mode CA-Clipper programs that run under DOS and can directly address much more than 1 MB of memory.

Real-Mode Addressing

The standard way to designate a memory location in real mode is with segment:offset notation, usually written with hexadecimal digits. The segment portion of the address refers to a base location in memory. The offset designates the number of bytes from the base.

In real mode, a logical address in segment:offset notation refers to a specific location in physical memory. For example, if you use the C code below to assign a value to the variable MyPointer, you can dereference MyPointer and get the contents stored at physical address 3B004 (logical address 3B00:0004):

```
char far *MyPointer;
MyPointer = (char far *)(((long)0x3B00 << 16) + 0x4);
```

From the logical address 3B00:0004, the CPU takes the segment address and multiplies it by 0x10 to get the base location 0x3B000. Then it adds the offset component, to arrive at the physical address 3B004. Using 20 bits, a CPU running in real mode can address up to 1 MB (00000-FFFFF) of physical memory.

From each base address, the CPU can utilize offset values from 0000 to FFFF, allowing it to address up to 64 KB for each segment of memory.

You can utilize as many as 4096 different segment:offset addresses that refer to the same physical address in real mode. For example, the address 3AFF:0014 refers to the same address as 3B00:0004:

```
0x3AFF * 16 = 0x3AFF0
0x3AFF0 + 0x0014 = 0x3B004
```

In real mode, you can utilize offsets of up to 0xFFFF within any segment, whether it is logically correct or not. If you use an address beyond the boundaries of your allocated segment of memory, it is easy to overwrite code or data in another segment or program, or even in the operating system. Also, in real mode, you can use any address as code or as data, so your program can store data in a code segment or attempt to execute data.

16-Bit Protected-Mode Addressing

Protected mode adds a level of indirection to the addressing process. Instead of pointing to a base location in physical memory, the segment portion of a logical address becomes a selector, which is an index into a descriptor table. The descriptor then refers to a physical memory segment.

Each descriptor contains information about the type (code or data), the limit (its size), and the base address of the physical segment to which it refers. If a program attempts to write beyond the segment limit, to write into a code segment, or to execute data, a General Protection Fault (Interrupt 0x0D) occurs. (For more information about descriptors, see the Descriptors section in this chapter.)

Each protected-mode descriptor on an 80286 can have a limit from 0h to FFFFh, allowing you to access from 1 byte to 64 KB of physical memory through that descriptor. On a 286, the portion of the descriptor that points to the base physical address contains 24 bits, so an 80286 CPU running in protected mode can address up to 16 MB of physical memory.

This indirect addressing system, which protects you from inadvertently overwriting code or data and allows you to address more memory, may require a few adjustments in your real-mode programs. The section, Protected-Mode Programming Restrictions, in this chapter, lists techniques that will not work in protected mode.

Descriptors

A segment descriptor is an 8-byte data structure that contains information about a segment. Your application should not have to generate descriptors, and, in general, you do not have access to them. You may find it helpful, however, to understand the information within a descriptor. The CPU uses this information to translate logical addresses to linear addresses, and, in 16-bit, segmented addressing, to keep you from accessing or overwriting memory that belongs to another process.

On 80286 machines, a segment descriptor contains the following information about a segment:

- Base physical address (24 bits)

  The base address is a 24-bit pointer to the physical location for the beginning of the segment. The CPU adds the offset in a protected-mode address to this address to determine the complete physical address.

- Limit (16 bits)

  The limit is the size of the segment (the size of a segment can as large as 64 KB). The CPU uses the limit to determine the last legal offset you can address in a given segment.

- Type (4 bits)

  The type describes whether the segment is a code segment (and therefore executable) or a data segment (and therefore modifiable). It also describes whether a code segment can be read as data as well as executed, whether a data segment is read-only or read/write, and which direction a data segment expands.

- Descriptor privilege level (2 bits)

  There are 4 possible privilege levels. Zero is the highest level, 3 the lowest. The privilege level allows a control program to protect the system from user programs. Except under DPMI, CA-Clipper/Exospace programs run at level 0.

- Its presence or absence in physical memory

  The present bit specifies whether the described segment is actually stored in physical memory or not. When a program attempts to access a segment that has been swapped out of memory, the CPU generates a fault, which a virtual memory manager can trap.

In addition to the information about the segment it describes, a descriptor has one bit that describes the descriptor itself. The S bit specifies whether the descriptor itself is a segment descriptor or a system descriptor, such as a Task State Segment or a system gate.

On the 80286 chip, the remaining 16 bits are reserved.

Protected-Mode Programming Restrictions

This section lists the restrictions imposed on your programs when they run in protected mode. You will find a description of each restriction and instructions for correcting code that violates the restriction.

Some programs are more likely to violate protected-mode programming restrictions than others:

- Programs written entirely in higher-level languages like C do not violate most of the restrictions.
- Library functions may violate some of these limitations. The CA-Clipper/Exospace distribution diskettes contain replacements for C compiler runtime library functions that have protected-mode violations.
- Assembly language programs may violate some of the restrictions. These programs can always be corrected.

If your program violates any of these protected-mode programming restrictions, you can modify it to work properly in both real and protected modes, so that you do not have to maintain two separate versions. Often the resulting code runs as fast or faster than code that does not run in protected mode.

Finding Protection Faults

The protected-mode hardware traps protection faults, so you do not have to find them yourself. If a program attempts to perform an operation that is illegal in protected mode, the hardware traps it and generates a General Protection Fault (Interrupt 0Dh). Then, CA-Clipper/Exospace issues the message "General Protection Fault" along with the program address and other information. Protection faults are fatal; you will not be able to continue program execution.

No Dereference of Null Far Pointers

The protected-mode selector 0 points not to a physical address, but to the 0 descriptor in the descriptor table, which is a reserved selector. While you may have null far pointers, you cannot read or write through the pointer. If you do, a protection fault occurs.

Most of the time, dereferencing a null pointer is unintentional. However, sometimes code may try to access real-mode interrupt vectors, for example, by using a 0 segment to construct a pointer.

Note: Some compilers evaluate the following expression from right to left. This causes a protection fault in protected mode if x is NULL:

```
if ( (x != NULL) && (*x == 4) )
```

Compilers that have this behavior may also have an option to ensure that expressions like the one above are evaluated from left to right.

Check for null pointers in your code before you dereference them. To access video memory or the BIOS data area, use the transparent selectors listed in the Absolute Addresses section in this chapter.

No Arithmetic on Segment Addresses

Because a protected-mode selector points to a descriptor, and not directly to a physical segment, the arithmetic that you can do with the segment portion of a pointer in real-mode applications does not work in protected mode. Segment arithmetic occurs most often in pointer manipulation.

In real mode, adding 1 to a segment value describes a physical location 16 bytes (or one paragraph) higher in memory. In protected mode, adding 1 to a selector results in an error.

You can freely do arithmetic on the offset part of a pointer. For example, you can subtract the offset of a starting pointer from the offset of an ending pointer, as long as the selectors are the same in both pointers. Two pointers that have the same selector refer to the same segment in memory.

Normal C pointer arithmetic works in protected mode. For example, the following code is valid:

```
int a[30];
int *p;
p = &a[0];
p += 3;
                    /* == &a[3] */
```

You can also split pointers. For example, to copy from the middle of a table to the same offset in another table, you can change just the selector address.

It is relatively cumbersome to look inside pointers in C, so segment arithmetic problems occur most often in Assembly language. Memory managers are the most likely kind of software to violate this restriction. Segment arithmetic also shows up in library functions like spawn or system that call the DOS function 4Bh. Uninitialized far pointers may cause a protection exception, as well.

The 80286 instruction set does not have any 32-bit instructions suitable for doing pointer arithmetic. Therefore, each compiler generates its own 32-bit pointer arithmetic from the processor's 16-bit arithmetic instructions.

If your program does segment arithmetic to overcome the 64 KB segment size limit of the 80x86 machines, use your compiler's far or huge pointers, or use the CA-Clipper/Exospace huge allocation functions. If your program converts integers or long integers to pointers, make sure that the resulting values are valid protected-mode selector values, and that they are the addresses that you intended. (See Absolute Addresses in this chapter.)

No Execution of Data

The protected-mode addressing system maintains a strong separation between code and data. You cannot execute code within a data segment. If you branch to or execute a call into a data segment, a protection fault occurs.

Most of the time, executing data is unintentional. However, some assembly-language programs intentionally branch to data. For example, some programs construct instructions in a data area on the stack to invoke an interrupt handler and then jump to the dynamically generated code.

No Writing to Code

The protected-mode separation of code and data also prevents you from writing to code segments. An attempt to write to a code segment will result in a protection fault.

In real mode, writing to code often results in a abnormal termination, perhaps long after the actual write occurred. In protected mode, overwriting code immediately generates a protection fault.

Assembly-language programs sometimes intentionally write to code. Such self-modifying code does not work in protected mode.

Some compiler library functions construct an interrupt instruction further on in the function and then execute it. Other programs store data in the code segment as a coding optimization in order to save a few CPU cycles. Interrupt handlers, such as those in commercial libraries that support interrupt-driven communications, sometimes store data in code segments to save CPU cycles in the handler. These functions can be modified so that they will work equally well in either real mode or protected mode.

Uninitialized Register Structures

General purpose interrupt signaling C functions written for DOS, such as int86x(), intdos(), and sysint(), may be passed a structure with values for the segment registers. If these values are not assigned, a protection exception will occur when the library function tries to load the uninitialized data from the structure into the segment registers.

Replacement modules for these functions are provided in the CA-Clipper/Exospace library (EXOSPACE.LIB).

Absolute Addresses

In protected mode, selectors do not refer directly to physical addresses, so if you attempt to use a selector:offset address to obtain an absolute physical (or linear) address, you are not likely to receive what you expect.

Except when DPMI is active, CA-Clipper/Exospace offers transparent access to the most frequently used absolute hex segment addresses—for the BIOS data area, the system ROM BIOS, and various video memory and optional ROM areas. The default transparent selectors are:

```
0x40
0x9F00
0x9F40
0x9F80
0xA000
0xA200
0xA400
. . .          (every 0x200)
0xFE00
```

Except for selector 0x40, there is no transparent addressing under DPMI.

To manage the real-mode interrupt vectors, use DOS function 25h or 35h, or the CA-Clipper/Exospace library functions instead of absolute addresses.

Pointers Passed to DOS and the BIOS

Software interrupts, such as DOS and BIOS interrupts, that pass data pointers to DOS or the BIOS, require special handling to ensure that the data pointers will be accessible while DOS or the BIOS is executing. Most DOS interrupts are handled transparently by CA-Clipper/Exospace, and support is provided, in the form of packages, for common interrupts such as INT 10 and INT 25. See the EXOSPACE PACKAGE section in the "CA-Clipper Protected Mode Linker—EXOSPACE.EXE" chapter in the *Programming and Utilities Guide*.

Too Many Mode Switches Per Second

Because DOS runs in real mode and your CA-Clipper application runs in protected mode, each call to DOS requires a switch to real mode, then back to protected mode when the call returns. Frequent switches between protected mode and real mode will degrade your program's performance. There may be an additional delay of 50 to 600 microseconds before an external interrupt can be serviced. The length of the delay depends on the quality of the machine's BIOS code. Typically, delays greater than 150 microseconds only occur on 80286 machines. To see how much time a mode switch takes on your machine, run PMINFO. Most systems are able to handle at least 1000 switches per second.

Parameter and Return Values

An important feature of the Extend System API is the ability of C and Assembly language functions to access parameters passed from CA-Clipper and supply return values to CA-Clipper. This section discusses this capability.

CA-Clipper Parameters Versus C Parameters

Values passed from a compiled CA-Clipper application to an Extend routine do not use the standard C parameter passing convention. Instead, a set of special interface functions provides access to parameters passed from CA-Clipper and also allows values to be returned.

The standard C convention *is* used for calls from Extend routines to all API functions. The functions use the C convention for calling, returning, and parameter and return value handling.

Note: In this chapter, parameters passed from a compiled CA-Clipper application are referred to as CA-Clipper *parameters*. Values returned to a compiled CA-Clipper application are called CA-Clipper *return values*. Parameter and return values passed to and from the functions are referred to as *C parameters* and *C return values*.

Getting Information About CA-Clipper Parameters

CA-Clipper maintains an internal stack-oriented data structure used to manage CA-Clipper parameters. This parameter stack contains information on the number of parameters passed and their data types. For simple data types (numeric values, for example), the parameter stack also contains the actual parameter value. For more complex types (strings or arrays), the parameter stack does not contain the value but contains information that allows the value to be located. The Item API (described later in this guide) provides the C or Assembly programmer with more flexible access to this stack, while the Extend System API functions are more limited, but easier to use.

This internal CA-Clipper parameter area should not be confused with the processor stack used to pass C parameters. When a C or Assembly language function initially gains control from the CA-Clipper application, it will have been passed no C parameters at all; the processor stack will contain only the return address used to return to the CA-Clipper application.

The Extend System API provides an interface function called _parinfo() that allows you to determine the number of CA-Clipper parameters passed and the type of each parameter.

A separate function, _parinfa(), is provided for CA-Clipper parameters containing array values. _parinfa() allows you to determine the number of elements in such an array, as well as the type of each element.

Accessing CA-Clipper Parameters

The Extend System API provides a set of interface functions that allow you to access CA-Clipper parameter values. These functions have names beginning with the characters _par and are referred to as the _par functions.

The _par functions retrieve information from CA-Clipper's parameter stack, translate it to an appropriate form, and return it to your Extend routine as a C return value.

When calling the _par functions, you must supply a C parameter specifying which of the supplied CA-Clipper parameters you want to access. If the CA-Clipper parameter is an array, an additional C parameter is required to indicate which element of the array is to be accessed.

Posting a CA-Clipper Return Value

CA-Clipper maintains an internal data structure used to record the return value from a C or Assembly language function. This area contains information on the type of the value being returned as well as the actual value (or, for complex data types, information that allows the value to be located).

The Extend System API provides a set of interface functions that allow you to place a value into CA-Clipper's return value area. This is referred to as *posting* a return value. These functions have names beginning with _ret and are referred to as the _ret functions.

The _ret functions take a C parameter of a particular type, make a copy of the value (allocating memory for the copy if necessary), translate it to the appropriate internal form, and place it in the return value area. Later, when the Extend routine returns to the CA-Clipper function that called it, the posted value appears in the CA-Clipper application as a normal CA-Clipper return value.

_ret functions are available to post various types of return values. You must call the _ret function that is appropriate for the type of value you wish to post. You must also supply a correctly typed C parameter specifying the value to be posted.

Note: The return value area only accommodates a single value. If your C or Assembly language function posts more than one return value during a single call, only the last value posted takes effect; values posted earlier are ignored.

CA-Clipper Parameters Passed by Reference

CA-Clipper allows a form of parameter passing called pass-by-reference. In CA-Clipper source code, a reference parameter is specified by placing the special reference operator (@) in front of a variable name in a list of function arguments. Variables passed in this fashion can be modified by the called function. CA-Clipper arrays are always passed by reference without need for the special reference operator.

Note: CA-Clipper's pass-by-reference facility should not be confused with pass-by-reference in C. In C, any value operated on through a pointer variable is said to be handled by reference. In this respect, several of the interface functions (i.e., those that return pointers) involve references. CA-Clipper reference parameters, however, are a higher-level type of reference involving special information on the CA-Clipper parameter stack that allow the interface functions to modify the values in the associated CA-Clipper variables.

For CA-Clipper parameters that are passed by reference, the CA-Clipper parameter stack contains different information than for normal CA-Clipper parameters. Instead of value-related information, the parameter stack contains information that allows the specified CA-Clipper variable to be modified.

This distinction is normally unimportant when using the _par functions; they operate transparently on reference parameters by locating the variable and processing its value as the parameter value. However, a special set of interface functions allow you to store values directly to CA-Clipper variables that are passed by reference. These functions have names beginning with _stor and are referred to as the _stor functions.

The _stor functions take a C parameter, copy the value (allocating memory for the copy if necessary), translate it to the appropriate internal form, and place it in a CA-Clipper variable that was passed by reference.

_stor functions are available for various data types. You must call the _stor function that is appropriate for the value you wish to store. You must supply C parameters specifying the value to be stored and the parameter to be affected. If the variable being assigned contains an array value, an additional C parameter is required to specify which array element is to be affected.

Note: The _stor functions have no effect on CA-Clipper parameters not passed by reference. Note, however, that array elements can always be assigned by the _stor functions, even if the array value is not contained in a CA-Clipper variable passed by reference. This is because array values are themselves a type of reference. The _parinfo() function can be used to determine whether a CA-Clipper parameter is a variable passed by reference.

Using the Extend System API with C and C++

There are several necessary steps in establishing a CA-Clipper program interface. This section describes the issues involved and provides information on the Extend System API components that you will find useful.

Note: A general discussion of C and C++ programming is beyond the scope of this guide. For general information on C and C programming, refer to your C documentation.

Note: The material in this section pertains to Microsoft C, version 8.0. Other C compilers that use a compatible calling convention can also be used with CA-Clipper. If your C function performs floating point calculations, however, you must use Microsoft C. Refer to the Using Floating Point Values section in this chapter.

The Extend.api Header File

The C header file, Extend.api, is supplied with CA-Clipper. This file contains function prototypes for the Extend System API functions. Extend.api should be included (#include) at the top of any C source file containing functions which are to be called from CA-Clipper applications. Including this unmodified header file at the start of every C program written to interface with CA-Clipper will minimize version shock associated with future updates to the Extend System API.

In addition to the function prototypes, Extend.api sets up parameter checking macros and type constants that make using the Extend System API interface easier. The following tables summarize these.

Extend.api Manifest Constants

Constant	Value	CA-Clipper Type
UNDEF	0	NIL
CHARACTER	1	Character
NUMERIC	2	Numeric
LOGICAL	4	Logical
DATE	8	Date
MPTR	32	Passed by reference
MEMO	65	Memo
ARRAY	512	Array
BLOCK	1024	Code Block

Extend.api C Interface Macros

Macro	Description
ALENGTH(int)	Number of elements in array parameter
ISARRAY(int)	Parameter is an array
ISBYREF(int)	Parameter was passed by reference
ISCHAR(int)	Parameter is a character
ISDATE(int)	Parameter is a date
ISLOG(int)	Parameter is a logical
ISMEMO(int)	Parameter is a memo
ISNUM(int)	Parameter is a numeric
PCOUNT	Number of parameters passed

Within a C Extend routine, you can use PCOUNT to determine how many parameters are passed from CA-Clipper. The IS macros provide a convenient way to determine whether a particular parameter is of a particular CA-Clipper data type. The macros yield a non-zero value if the specified parameter is of the requested type.

Note: The Extend.h header file used in previous releases of CA-Clipper is still valid; however, it is recommended that you use the Extend.api header file instead.

Declaring the C Function

C functions to be called from CA-Clipper applications should be declared using the CLIPPER macro defined in Extend.api. The CLIPPER macro declares the function as a *void PASCAL* type. This prevents the C compiler from adding an underscore to the function name and forces the function name to all uppercase. This allows your C function to match CA-Clipper's naming convention.

As noted above, CA-Clipper parameters are not passed to your C function using the C parameter conventions. For this reason, your C function should always declare a void parameter list. For details, refer to the Parameter and Return Values section in this chapter.

Accessing the Extend System API

Calling the Extend System API functions from C is straightforward. For examples describing operations on various data types, refer to the Operating on CA-Clipper Values section.

Using Floating Point Values

In CA-Clipper 5.3, floating point operations are implemented using an augmented subset of the Microsoft C 8.0 floating point emulation. This system, present in CLIPPER.LIB, is used to provide maximum execution speed when a numeric coprocessor is not present.

If your C function performs floating point calculations, you must compile it with Microsoft C version 8.0 (or higher) and you must use the C compiler's /FPi option when compiling. This instructs the compiler to use the floating point emulation for your function.

If you compile your C functions with a different C compiler (or fail to use the /FPi option with Microsoft C), the presence of floating point operations in your C function will cause the resulting object file to contain link references to a floating point support package that is incompatible with CLIPPER.LIB. This will normally cause the linker to produce many "duplicate symbol" warnings. Running the resulting executable program will generally cause a system crash requiring a reboot.

Warning! *If you are using a C compiler other than Microsoft C, you may experience linking problems even if your code performs no floating point calculations. With some C compilers, the mere presence of the word "double" in a source program is sufficient to cause a link request for the floating point support package. If you experience this problem, make sure you have no double variables in your C functions. You may also need to modify a copy of the Extend.api file supplied with CA-Clipper to remove references to double values (Extend.api uses a C typedef called XDOUBLE that may help with this modification).*

Using C Runtime Library Functions

In general, your C functions can safely call C library functions. The following guidelines should be noted, however:

- As a general rule, you must use library functions that conform to the large model calling convention.
- Library functions that perform floating point calculations cannot be used unless they have been compiled using the Microsoft C floating point emulation (see Using Floating Point Values).
- Most graphics functions use floating point values, making them unusable with CA-Clipper for the same reason.
- Library functions that perform memory allocation will not function under CA-Clipper. To allocate memory from a C function, use only the Virtual Memory (VM) or Fixed Memory (FM) API service functions provided for this purpose.
- Spawn functions (i.e., functions that run executable files) will not work under CA-Clipper.
- CLIPPER.LIB defines certain standard C string-handling functions. These are functionally compatible with the normal C versions.

Example

In the following example, the C functions Sinrec() and Sinfunc() work together to compute the sine of an angle. The functions also demonstrate a useful technique: isolating the interface code from the operational code. This makes your functions easier to read and debug. These functions can be called from a C main function, allowing debugging without the overhead of the CA-Clipper runtime support. Additionally, it makes your code less sensitive to the details of the Extend System API.

```
/***
*    Sinrec( <expN> ) -> <expN>
*
*    CA-Clipper-callable interface module
*
*    Uses the Extend System _parnd() function to
*    retrieve a numeric value from CA-Clipper, as a
*    double, which is assigned to the variable degrees.
*
*    Passes degrees to the discrete C callable function
*    sinfunc().
*
*    The return value of sinfunc(), sine, is returned
*    back to CA-Clipper using the Extend System function
*    _retnd().
*/

#include "extend.api"
#include "math.h"

double sinfunc(double degrees);

CLIPPER SINREC()
{
    double degrees = 0.0;
    double answer = 0.0;

    if (PCOUNT == 1 && ISNUM(1))
    {
       degrees = _parnd(1);
       answer = sinfunc(degrees);
       _retnd(answer);
    }
    else
    _retnd( (double)-1 );
}
```

```
/***
*    Discrete C Callable module.
*
*    Convert degrees to radians and compute sine.
*/

double sinfunc(double degrees)
{
    double sine = 0.0;
    double pi = 3.1415926535;
    double radians = 0.0;

    radians = degrees * (pi/180);
    sine = sin(radians);

    return(sine);
}
```

The following lines of code could then be used to call this function from a CA-Clipper program. The function argument is divided by one in this example to make sure it is a double:

```
// CA-Clipper function call
? SinRec( 90/1 )        // Result: 1.00
```

Compiling with Microsoft C

To compile a Microsoft C 8.0 routine, use the following syntax:

```
CL /c /AL /FPi /Gs /G2 /Oalt /Zl <filename>.c
```

Its arguments are defined below:

- CL invokes the Microsoft C compiler
- /c means to compile without linking
- /AL sets the program configuration for the large memory model
- /FPi uses the floating point emulation math library
- /Gs removes calls to the stack-checking routine
- /G2 generates 286 instructions (this option is recommended to improve performance)
- /Oalt invokes certain optimization features (i.e., relaxed alias checking, loop optimization, and favored execution speed)
- /Zl removes default library name from the object file

Note: Microsoft C 8.0 advanced optimizations may produce rather serious problems. For example, using the /Ox option (maximum optimization) sometimes compiles the typical code fragment:

```
for ( . . . )

   if (pointer != NULL && pointer->member . . . )

as if it were written:

for ( . . . )

      {
      temp = pointer->member;
      if (pointer != NULL && temp . . . )
      ...
      }
```

This will fail in any protected-mode environment, including CA-Clipper/Exospace, Windows 3.x in enhanced mode, and OS/2.

Compiling with Borland C++

CA-Clipper/Exospace is compatible with the Borland C++ compiler, Versions 2.0, 3.0, and 3.1 large model. CA-Clipper/Exospace supports the Microsoft subset of the C++ extensions. You should use the Borland C++ options that specify Microsoft C 6.0-compatible objects, such as the -Vs option.

CA-Clipper/Exospace does not support the Borland Graphics Library (BGI), or Borland floating point math routines. If you use Borland runtime library functions, you do so at your own risk, as described in the previous section.

To compile a Borland C++ routine, use the following syntax:

```
BCC -c -ml -2 -f- -X <filename>.c
```

Its arguments are defined below:

- BCC invokes the Borland C++ compiler
- -c means to compile without linking
- -ml sets the program configuration for the large memory model
- -2 generates 286 instructions. This option is recommended to improve performance

- -f specifies that the program contains no floating point calculations. Borland's floating point support is incompatible with the CA-Clipper floating point, so use of this option is required.
- -X suppresses auto dependency output. This information is only useful to other Borland utilities and can be suppressed, resulting in smaller .OBJ files.

Linking with C Object Files

To link CA-Clipper and C object files to produce an executable file (.EXE), use the following syntax:

```
EXOSPACE FILE <clipperObject>, <cObject>
```

If a linker other than CA-Clipper/Exospace is being used, you must add the following libraries (in order): LLIBG, CLIPPER, EXTEND, TERMINAL, DBFNTX, and LLIBCE.

There are several issues to be aware of when linking C and CA-Clipper:

- ***Object module order:*** Compiled CA-Clipper modules should be listed before C modules. This ensures that the linker processes segment information in the correct order.
- ***C runtime libraries:*** If your C function uses functions from the C runtime support library, you must include the appropriate library in the link. Be sure to use the large model library. Some library functions may violate protected-mode rules, and thus be incompatible with CA-Clipper/Exospace. CA-Clipper/Exospace provides replacement routines for library functions that violate these rules. These routines are contained in EXOSPACE.LIB.

  If your C function performs floating point operations, you must use the Microsoft C compiler, specify floating point emulation, and link with the library that contains floating point emulation (normally LLIBCE.LIB, shown in the previous linking example). For more information, refer to Using Floating Point Values.
- ***Library order:*** The order in which you specify LIB files on the linker's command line is significant because there are some symbols that are defined in both CLIPPER.LIB and other libraries. The CA-Clipper libraries are listed first so that their definitions take precedence over the ones in the C library.

- ***Graphics:*** To use CA-Clipper/Exospace with Microsoft C and graphics, specify the CA-Clipper/Exospace INT10 package in your link script. You cannot use graphics on some computers without the INT10 package.

  For more information on packages, see the EXOSPACE PACKAGE section in the "CA-Clipper Protected Mode Linker—EXOSPACE.EXE" chapter of the *Programming and Utilities Guide.*

Linking with C++ Object Files

Use CA-Clipper/Exospace to link your program as usual, and include a reference to the compiled Borland C++ modules in the link script.

Using the Extend System API with Assembly Language

The Extend System API supports CA-Clipper-callable functions written in Assembly language using the same functions used with C functions. These functions must be called using the C large model calling convention.

This section gives you information on using the Extend System API to write Assembly language functions to be called from CA-Clipper programs.

Note: A general discussion of Assembly language programming is beyond the scope of this guide. For general information on Assembly language and Assembly language programming, refer to your Assembler documentation.

The C Large Model Calling Convention

The Extend System API functions use the C large model calling convention. What follows is a list of key points to observe when writing Assembly language code that complies with this convention. For detailed information on calling C routines from Assembly language, refer to your Assembler documentation.

General Requirements

- If using Microsoft MASM 5.0 or higher, use the .MODEL LARGE directive to establish segments. Otherwise, be sure to give your code segment a class of CODE.
- If your function requires large amounts of statically allocated data, use a separate segment for this data. Don't place the segment in DGROUP.
- Declare your CA-Clipper-callable entry point FAR to make sure that it returns to CA-Clipper with a far return instruction.
- Declare your CA-Clipper-callable entry point PUBLIC.
- Preserve the following registers on entry to your function and restore them on exit: DS, SS, SI, DI, BP. Additionally, maintain the hardware direction flag in the *cleared* state.

Function Calling

- Include EXTRN declarations for any Extend System API functions that your function uses or INCLUDE the Extasm.inc header file (discussed in the following section). Remember to add an extra underscore to the names of the Extend System API functions and declare them FAR.
- Remember to preserve any registers in which you have active data values. Any C function that you call will only preserve DS, SS, SI, DI, and BP.
- Before calling any of the Extend System API functions (or any other C function), ensure that both DS and SS are equal to DGROUP. If your function could modify DS or SS, make sure to restore the initial values before calling Extend System API functions.
- Push C parameter values onto the processor stack in the *reverse* order in which they are listed in the function description. This is because the stack grows downward. Pushing the parameters in reverse results in their being placed in ascending memory locations.
- For long (doubleword) values, push the high-order part first, then the low-order part.

- For pointer values, push the segment part first, then the offset part.
- After the Extend System API function has returned, reset the processor stack pointer by adding a number equal to the number of bytes of parameters that were pushed for the function call.
- Remember that WORD (C int) return values are contained in the AX register on return. DWORD (C long or pointer) return values are contained in the DX:AX register pair.

Warning! *Since Assembly language gives you access to the lowest level of software and hardware interaction, you must exercise appropriate caution. Failure to comply with the calling convention will usually cause the system to crash.*

The Extasm.inc Header File

An Assembly language header file called Extasm.inc is supplied with CA-Clipper. It is recommended that you include this file at the top of any Assembly language source file containing functions which are to be called from CA-Clipper applications. Including this unmodified header file will minimize version shock associated with future updates to the Extend System API. This file contains EXTRN declarations for the Extend System API functions.

In addition to the EXTRN declarations, Extasm.inc defines manifest constants for the Extend System API type codes. These are summarized in the table below:

Extasm.inc Manifest Constants

Constant	Value	CA-Clipper Type
UNDEF	0	NIL
CHARACTER	1	Character
NUMERIC	2	Numeric
LOGICAL	4	Logical
DATE	8	Date
MPTR	32	Passed by reference
MEMO	65	Memo
ARRAY	512	Array
BLOCK	1024	Code block

Code Template

The following is a shell of a typical CA-Clipper user-defined function written in Assembly language. This example uses manual segment directives. See the sections later in this chapter for examples that use the Microsoft MASM simplified directives.

Note: The best way to avoid use of DGROUP in Assembly language is to use the stack for values. However, when values are simply too large for the stack, you should consider creating a new segment and reference data from there. Of course, you must assume that DS points to DGROUP at the start of your code, and therefore; push DS to the stack and load DS with your new data segment. You will also have to ensure that DS points to DGROUP when calling any API function, as well as restoring DS at the end of your routine.

```
PUBLIC    <func>
EXTRN     <extend_func>:FAR
DGROUP    GROUP <dseg>

<dseg>    SEGMENT PUBLIC 'DATA'
;
;         <your data declarations>
;
<dseg>    ENDS

<cseg>    SEGMENT 'CODE'
          ASSUME cs:<cseg>, ds:DGROUP

<func>    PROC  FAR

          push  bp          ; save registers
          mov   bp, sp
          push  ds
          push  si
          push  di

          <your code goes here>

          pop   di          ; restore registers
          pop   si
          pop   ds
          pop   bp

<func>    ENDP              ; end of routine

<cseg>    ENDS              ; end of code segment
          END
```

Example

The following is an operational Assembly language routine that clears a region of the screen by calling the hardware BIOS screen routines.

```
;
; CLEARIT( <top>, <left>, <bottom>, <right> ) -> NIL
;
; Assembly example to clear a region of the
; screen using PCBIOS
;

PUBLIC    CLEARIT              ; declare as public

EXTRN     __parni:FAR          ; declare functions as external
EXTRN     __ret:FAR

DGROUP    GROUP _DATA          ; combine this data segment
                               ; with CA-Clipper

_DATA     SEGMENT  'DATA'      ; start of data segment

          TOP      DB   0
          LEFT     DB   0
          BOTTOM   DB   0
          RIGHT    DB   0

_DATA     ENDS                 ; end of data segment

_TEXT     SEGMENT  'CODE'      ; start of code segment

          ASSUME   cs:_TEXT, ds:DGROUP

CLEARIT   PROC  FAR            ; declare far procedure

          push  bp             ; preserve return address
          mov   bp, sp
          push  ds             ; save registers
          push  si
          push  di

          mov   ax, 1          ; point to parameter
          push  ax             ; place on stack
          call  __parni        ; call CA-Clipper Extend System
                               ; function
          add   sp, 2          ; restore stack
          mov   TOP, al        ; assign parameter to top

          mov   ax, 2          ; repeat process for next parameter
          push  ax
          call  __parni
          add   sp, 2
          mov   LEFT, al       ; assign to left
```

```
            mov   ax, 3         ; repeat process for next parameter
            push  ax
            call  __parni
            add   sp, 2
            mov   BOTTOM, al    ; assign to bottom

            mov   ax, 4         ; repeat process for next parameter
            push  ax
            call  __parni
            add   sp, 2
            mov   RIGHT, al     ; assign to right

            mov   ch, TOP       ; place coordinates in CX:DX
            mov   cl, LEFT
            mov   dh, BOTTOM
            mov   dl, RIGHT

            mov   ax, 0600h     ; request roll up service
            mov   bh, 07        ; normal attribute
            int   10h           ; issue video interrupt

            pop   di
            pop   si
            pop   ds            ; restore registers
            pop   bp
            cld                 ; ensure direction flag is clear

            call  __ret         ; post NIL return value to cA-Clipper

            RET                 ; return to CA-Clipper
CLEARIT     ENDP                ; end of proc
_TEXT       ENDS                ; end of code segment
            END
```

To call this function from a CA-Clipper program:

```
ClearIt(10, 10, 20, 60)
```

Assembling with Microsoft MASM 5.1 and 6.1

To assemble an Assembly language program with Microsoft MASM 5.1 and 6.1, use the following syntax:

```
MASM <filename>;
```

The file name is assumed to have an (.ASM) extension. The semicolon tells MASM to use default file names for object and listing files. This statement creates an object file with the same name as the source file and an extension of .OBJ.

Linking with Assembly Language Object Files

To link the CA-Clipper and Assembly object files to produce a single executable (.EXE) file, use the following syntax:

```
EXOSPACE FILE <clipperObject>, <assemblyObject>
```

By default, compiled CA-Clipper objects contain linker directives that cause CA-Clipper/Exospace to automatically search the CA-Clipper support libraries. If your Assembly language code uses functions from other libraries such as C runtime support libraries, you must include those libraries in the link.

Operating on CA-Clipper Values

This section discusses the different CA-Clipper data types and describes how each is handled under the Extend System API. For each type, examples are given in both C and Assembly language.

The Item API provides another means of manipulating CA-Clipper values. This API was designed to provide a higher degree of control and flexibility over CA-Clipper values, as required by the RDD API. When should you use the Item API instead of the Extend System API? If your Extend routine only needs to accept simple parameters from a CA-Clipper routine and/or return simple values back to the CA-Clipper routine, the Extend System API will provide a simple, straight-forward means of accomplishing this. If, however, you need to manipulate CA-Clipper values that consist of code blocks, or need access to data in multidimensional arrays, you should use the Item API.

CA-Clipper Data Types

CA-Clipper data types are *high-level* types; they have only an indirect correspondence with the way data is represented at the machine level. In CA-Clipper, nine basic data types are available:

- Array
- Character
- Code block
- Date
- Logical
- Memo
- NIL
- Numeric
- Predefined object

C provides a more machine-oriented type set. The C types relevant to use of the Extend System API are:

- Integer
- Long integer
- Double (double-precision floating point)
- Character pointer (a far memory address)

The type set provided in Assembly language is more fundamental still:

- Word
- Doubleword (used for both long integers and pointer types)
- Quadword (used for double-precision floating point values)

Note: Some CA-Clipper types are not accessible by Extend routines. Others have no directly corresponding type in C and, therefore, cannot be easily stored and retrieved by the Extend System API functions. The data types that are applicable to the Extend System API are discussed in the following sections.

Array Values

Internally, CA-Clipper stores an array value as a series of data structures, each containing one value of any type (including another array reference). A CA-Clipper variable cannot directly contain an array. Instead, it contains an indirect reference to the series of array values.

Accessing Arrays

CA-Clipper array values cannot be directly manipulated through the Extend System API. However, individual array elements may be inspected or assigned using the interface functions. Only a single array dimension is accessible; if an array element contains a reference to another array, the subarray is not accessible using the Extend System API. The Item API must be used to manipulate multidimensional arrays.

Array elements are accessed by passing an extra parameter to the Extend System API functions for other data types.

Operations on Array Values

Since CA-Clipper always passes array values by reference, operations on arrays are performed through the _stor functions. It is not necessary to pass arrays by reference to use the _stor functions. Only array *elements* may be modified and arrays themselves cannot be resized or reassigned.

Posting an Array Return Value

Array values cannot be posted as return values in CA-Clipper.

Assigning an Array Value to a Reference Parameter

An array value cannot be stored to a CA-Clipper variable passed by reference. However, the _stor functions can be used to modify the individual elements of any array value passed as a CA-Clipper parameter.

Note: Since array values are themselves references, array elements can always be modified using the _stor functions. It is not necessary to assign the array value to a CA-Clipper variable and pass the variable by reference.

Examples

For examples of accessing arrays, refer to the examples in the Numeric Values and Date Values sections.

Character and Memo Values

CA-Clipper treats memo values the same as character values. This discussion applies to both.

Internally, CA-Clipper represents character values as a series of bytes in memory, along with length and allocation information. The bytes of a character value may contain any value, including unprintable characters and null bytes.

There is no particular correspondence between a character value and a program variable. Variables that contain the same character value may share a single copy of the value or each one may have its own copy. For example, the following CA-Clipper code:

```
cExp1 := cExp2 := "Shared text value"
```

will declare two variables that share the same text value. This is a runtime optimization and not a behavioral characteristic of the CA-Clipper language. This behavior should not be relied upon.

Accessing Character Values

The _parc() interface function returns a far pointer to a series of bytes representing a character value passed from CA-Clipper.

By convention, C character strings are terminated with a null byte. This allows the length of a character string to be determined by searching for the null byte. The bytes pointed to by the _parc() return value obey this convention and are always terminated by a null byte. However, since CA-Clipper character values are allowed to contain embedded null bytes, scanning for a null byte is not always an accurate method to determine the length of a passed character value.

The _parclen() interface function allows an exact determination of the actual length of a character value without searching for a null byte. _parclen() determines the length by accessing the internal data structure for the character value.

Operations on Character Values

The pointer returned by _parc() can be used to inspect the character value. However, it may or may not point to the actual bytes of the character value passed from CA-Clipper. In some cases, the pointer will point to a copy of the actual value. In other cases, it will point to an internal value that is shared by more than one CA-Clipper variable. The following code demonstrates this behavior:

```
cExp1 := cExp2 := "string."
```

Both *cExp1* and *cExp2* refer to the same string in memory, but they represent two distinct values. This is an optimization of CA-Clipper memory usage and not a intrinsic characteristic of the language. When one of these two variables is changed, the shared reference to one string ends and both variables will contain references to unique strings:

```
cExp2 := "No longer a shared " + cExp1

? cExp2             // Result: No longer a shared string.
? cExp1             // Result: string.
```

For these reasons, the pointer returned from _parc() should not be used to directly modify the character value. Doing so may have the effect (when viewed from the CA-Clipper level) of modifying several variables at once. Also, it is a violation of the CA-Clipper level scoping rules to modify a value that was not passed by reference. Other programmers using the function (including yourself as soon as you forget the details of the function) will not expect the function to modify a parameter unless it is passed by reference.

Warning! *The pointer returned by _parc() is only guaranteed for the duration of the C or Assembly function that initially retrieves it. The pointer value should never be saved and used during any subsequent activations of the function.*

If the purpose of the function is to effect a change to a CA-Clipper value, you should either use _retc() to return a modified *copy* of the value, or pass the associated CA-Clipper parameter by reference with the special reference operator (@) and use the _storc() interface function to modify it.

Posting a Character Return Value

The _retc() interface function allows you to post a character value as a CA-Clipper return value.

When you call _retc(), you pass a C parameter: a far pointer to a string. _retc() scans the string for a null byte to determine its length. It then creates a character value by making a copy of the string and posts the new value into the CA-Clipper return value area.

If you wish to return a string that contains embedded null bytes, use the _retclen() function. _retclen() takes an additional C parameter which specifies the logical length of the string. The length of the string is not determined by scanning it for a null byte.

Assigning a Character Value to a Reference Parameter

The _storc() interface function allows you to store a character value into a CA-Clipper variable that was passed by reference.

When you call _storc(), you pass two C parameters: a far pointer to a null-terminated string and an integer which specifies the CA-Clipper parameter to be affected. You must pass a third parameter when storing into an array. _storc() scans the string for a null byte to determine its length. It then creates a character value by copying the string and assigns the new value to the specified CA-Clipper variable.

If you wish to store a string containing embedded null bytes, use the _storclen() function. _storclen() takes an additional C parameter that specifies the logical length of the string. The length of the string is not determined by scanning it for a null byte.

C Example

This C example retrieves a CA-Clipper parameter containing a character value. The function encrypts the character value by performing a bitwise exclusive OR (XOR) on each of its characters using a supplied numeric key value. The modified string is then posted as the CA-Clipper return value.

The following example, Crypto.c, performs a simple XOR encryption on a character value. To decrypt, call the function again with the same key value. Crypto.c uses the FM API functions _xgrab() and _xfree() to perform fixed memory allocations. These functions use protocols identical to the C malloc() and free() functions. See the "Fixed Memory API Reference" chapter for more details about the use of these fixed memory allocation functions.

Notice how the C code is divided into an interface function and a normal C function that does the actual encryption. The C function is isolated from the details of dealing with CA-Clipper by the interface function. Also notice that the two functions have the same name but different capitalization. Remember, that your linker must be in case-insensitive mode for this technique to be successful.

From CA-Clipper:

```
newString := Crypto(cString, nCryptoKey)
```

In C:

```
#include "extend.api"
void crypto( char *source, char *dest,
                unsigned int length, int cryptval );

/***
*
*   CRYPTO( <cString>, <nCrypt> ) -> <cCryptStr>
*
*   Encrypts <cString> with the encryption value <nCrypt>.
*/
```

```
CLIPPER CRYPTO()
{
   char *clipstr;
   char *retstr;
   unsigned int length;

   // Check parms first
   //
   if ( PCOUNT == 2 && ISCHAR(1) && ISNUM(2) )
   {
      clipstr = _parc(1);
      length = _parclen(1);

      retstr = _xgrab(length + 1);
      retstr[length] = '\0';           // Etiquette

      crypto( clipstr, retstr, length, _parni(2) )
      _retclen( retstr, length );      // CA-Clipper has the
                                       // value,
      _xfree( retstr );                // so free the memory

   }
   else
      _ret;                            // Error return NIL

}


void crypto( char *source, char *dest,
                unsigned int length, int cryptval )
{
   int count;
   char cryptchar;

   // Truncate encrypting value to a character
   //
   cryptchar = (char)cryptval;

   for ( count = 0; count < length; count++ )
   {
      dest[count] = source[count] ^ cryptchar;
   }

}
```

Assembly Language Example

This Assembly example scans a character value passed from CA-Clipper for digit characters. If found, the digit characters are converted to a binary number, which is then used to post a numeric value as the CA-Clipper return value. Note that this example uses the Microsoft MASM simplified segment directives.

From CA-Clipper:

```
nNumber := FindNum("Mr. Jones is 33 years old")
```

In Assembly language:

```
;
; FINDNUM( <expC> ) -> <expN>
;
; Finds the first numeric in <expC> and
; returns that as a number.
;


.MODEL    LARGE                 ; simplified directives
          INCLUDE  EXTASM.INC
          PUBLIC   FINDNUM
.CODE

FINDNUM      PROC  FAR
          push  si              ; preserve si

          ; check parameter type
          mov   ax, 1
          push  ax
          call  __parinfo       ; _parinfo(1)
          add   sp, 2           ; reset stack
          test  ax, CHARACTER
          jnz   goodParam

          ; bad param, return NIL
          call  __ret
          jmp   gBye

goodParam:
          ; get pointer to param value
          mov   ax, 1
          push  ax
          call  __parc
          add   sp, 2           ; reset stack

          ; scan value for digit characters...stop at null byte
          ;
          mov   es, dx          ; place pointer in ES:SI
          mov   si, ax
```

```
tryAgain:
        ; get a character in AL
        ;
        lods  byte ptr es:[si]
        cmp   al, '9'           ; above '9' ?
        ja tryAgain             ; yes, skip it

        cmp   al, '0'           ; below '0' ?
        jnb   foundDigit        ; no, it's a digit (0-9)
        or al, al               ; hit null byte?
        jnz   tryAgain          ; no, keep going

        ; no digits in string, post a zero return value
        ;
        xor   dx, dx
        jmp   noMoreDigits

foundDigit:
        ; convert printable digits to a binary integer
        ; (value will accumulate in DX register)
        xor   bx, bx            ; BX := 0
        xor   dx, dx            ; DX := 0

nextDigit:
        mov   bl, al            ; BX := digit character
        sub   bl, '0'           ; subtract ASCII bias value
        mov   ax, 10
        mul   dx                ; multiply accumulated value by 10
        mov   dx, ax            ; put result back in DX
        add   dx, bx            ; add new digit

        ; get next character
        ;
        lods  byte ptr es:[si]
        cmp   al, '9'           ; above '9'?
        ja noMoreDigits         ; yes, stop converting
        cmp   al, '0'           ; below '0'?
        jnb   nextDigit         ; no, keep going

noMoreDigits:
        ; post accumulated value as CA-Clipper return value
        ;
        push  dx
        call  __retni
        add   sp, 2

gBye:
        ; return to caller
        ;
        pop   si                ; restore
        RET

FINDNUM  ENDP
        END
```

Date Values

Internally, CA-Clipper represents date values as numeric quantities. Under the Extend System API, however, they are manipulated as character strings. If you need access to date information as numerics, you should investigate the Item API (see the "Item API Reference" chapter).

Accessing Date Values

The _pards() function returns a far pointer to a null-terminated string containing a printable representation of a date value. The string has the format *yyyymmdd*, where *yyyy* specifies the year, *mm* specifies the month, and *dd* specifies the day.

Operations on Date Values

The pointer returned by _pards() can be used to inspect the date value. It does not point to the actual CA-Clipper date value. Instead, it points to a statically allocated buffer that contains a copy of the value in printable form. Subsequent calls to _pards() will overwrite this buffer, so you should make a copy of the value as needed.

Posting a Date Return Value

The _retds() function allows you to post a date value as a CA-Clipper return value. When you call _retds(), you pass a far pointer to a null-terminated string. The string must have the format shown for _pards() in the Accessing Date Values section. _retds() converts the string into a CA-Clipper date value and posts the value into the CA-Clipper return value area.

Assigning a Date Value to a Reference Parameter

The _stords() interface function allows you to store a date value into a CA-Clipper variable that is passed by reference. _stords() takes two C parameters: a far pointer to a null-terminated string representing the date value and an integer specifying which CA-Clipper parameter is to be affected. You must pass a third parameter, which represents the array element to be accessed, when storing into an array. The string must have the format shown for _pards() in the Accessing Date Values section. _stords() converts the string into a CA-Clipper date value and stores the value into the specified CA-Clipper variable.

C Example

This C example uses two arrays passed from CA-Clipper. Dates in the first array are modified using the _stords() function so that they have the same century digits as the corresponding dates in the second array.

From CA-Clipper:

```
WhatFor( aDates, aOtherDates )
```

In C:

```
#include "Extend.api"

/* useful C macro for below */
#define ISADATE(param, index)  (_parinfa(param, index) & DATE)

CLIPPER WHATFOR(void)
{
    int length;
    int index;
    char *datePtr;
    char dateBuff1[9];

    if ( ISARRAY(1) && ISARRAY(2) )
    {
       /* get count of elements */
       length = ALENGTH(1);

    /* reduce the count if the other array has fewer elements */
       if ( length > ALENGTH(2) )
       {
          length = ALENGTH(2);
       }

       /* scan */
       for (index = 1; index <= length; index++)
       {
          if ( ISADATE(1, index) && ISADATE(2, index) )
          {
             /* get pointer to date string */
             datePtr = _pards(1, index);

             /* must make a copy of the string! */
             strcpy(dateBuff, datePtr);

             /* get pointer to date string in other array */
             datePtr = _pards(2, index);

             /* change the century digits of our saved copy */
             dateBuff[0] = datePtr[0];
             dateBuff[1] = datePtr[1];

             /* store modified date value into the first array */
             _stords(dateBuff, 1, index);
          }
       }
    }

    /* always returns NIL */
    _ret();
}
```

Assembly Language Example

This Assembly example retrieves a date value passed from CA-Clipper and uses it to set the operating system date. The function returns a logical value: true (.T.) if the date was set successfully and false (.F.) otherwise. Note that this example uses the Microsoft MASM simplified segment directives.

From CA-Clipper:

```
DosDate( CTOD("08/06/95") )
```

In Assembly language:

```
.MODEL    LARGE                ; simplified directives
          INCLUDE  EXTASM.INC
          PUBLIC   DOSDATE
.CODE

DOSDATE      PROC  FAR

          pop   si             ; preserve caller's SI

          ; check param type
          mov   ax, 1
          call  __parinfo
          add   sp, 2          ; reset stack
          and   ax, DATE       ; is date value?
          jz gBye              ; no, forget it
                               ; note: AND instruction leaves
                               ; 0 in AX for _retl()

          ; get pointer to date string in DX:AX
          mov   ax, 1
          call  __pards
          add   sp, 2          ; reset stack

          ; put pointer into ES:SI
          mov   es, dx
          mov   si, ax

          ; convert year from printable to binary value in CX
          mov   ax, 4          ; (# of digits for subroutine)
          call  ToBin          ; (see below)
          mov   cx, ax         ; result into CX

          ; convert month from printable to binary value in DH
          mov   ax, 2          ; (# of digits)
          call  ToBin
          mov   dh, al         ; result in DH

          ; convert month from printable to binary value in DH
          mov   ax, 2          ; (# of digits)
          call  ToBin
          mov   dl, al         ; result in DL
```

```
; call DOS to set date
        mov   al, 2Bh
        int   21h

        ; use DOS return value as CA-Clipper return value
        sub   ah, ah          ; clear AH

gBye:
        push  ax
        call  __retl          ; post value to CA-Clipper
        add   sp, 2           ; reset stack

        push  si              ; restore
        ret

DOSDATE  ENDP

;
;  Convert printable digits at ES:SI into a binary value in AX.
;  Call with AX set to the number of digits to convert; leaves
;  ES:SI pointing to the next digit after the converted digits.
;

ToBin    PROC  NEAR

        push  bx              ; preserve registers
        push  cx
        push  dx

        ; use CX as the loop counter
        mov   cx, ax          ; count passed from above

        ; value will accumulate in DX register
        sub   bx, bx          ; BX := 0
        sub   dx, dx          ; DX := 0

nextDigit:
        ; get digit
        lods  byte ptr es:[si]
        mov   bl, al          ; BX := digit character
        sub   bl, '0'         ; subtract ASCII bias value
        mov   ax, 10
        mul   dx              ; multiply accumulated value by 10
        mov   dx, ax          ; put result back in DX
        add   dx, bx          ; add new digit
        loop  nextDigit       ; do it CX times

        mov   ax, dx          ; put final result in AX

        pop   dx              ; restore registers
        pop   cx
        pop   bx

        RET

ToBin    ENDP
        END
```

Logical Values

Internally, CA-Clipper represents logical values as 16-bit words containing the number 1 for true (.T.) or 0 for false (.F.).

Accessing Logical Values

The _parl() function returns a 16-bit (word) integer value. If the value of the CA-Clipper parameter is true (.T.), the value returned from _parl() is 1. If the value of the CA-Clipper parameter is false (.F.), the value returned from _parl() is 0.

Operations on Logical Values

The value returned by _parl() can be operated on in any way supported by the language used to write the Extend routine. It is an actual value extracted from CA-Clipper's internal structures and passed to the Extend routine as this value.

Posting a Logical Return Value

The _retl() function allows you to post a logical value as a CA-Clipper return value. _retl() takes a C integer (word) value. If the value is zero, a false (.F.) value is posted; otherwise, a value of true (.T.) is posted.

Assigning a Logical Value to a Reference Parameter

The _storl() interface function allows you to store a logical value into a CA-Clipper variable that is passed by reference. When you call _storl(), you pass two C parameters: the first determines which logical value will be stored and the second specifies which CA-Clipper parameter is to be affected. You must pass a third parameter, representing the array element, when storing into an array. If the first C parameter is zero, a false (.F.) value is stored; otherwise, a true (.T.) value is stored. In either case, _storl() converts the supplied value to a CA-Clipper logical value and stores the new value to the specified CA-Clipper variable.

C Example

This C example retrieves two logical values and returns the logical XOR of the values.

From CA-Clipper:

```
lResult := LXor( lValue1, lValue2 )
```

In C:

```
#include "extend.api"

CLIPPER LXOR(void)
{
    int log1;
    int log2;

    if ( ISLOG(1) && ISLOG(2) )
    {
       /* get logical values */
       log1 = _parl(1);
       log2 = _parl(2);

       /* post XOR of them as CA-Clipper return value */
       _retl(log1 ^ log2);
    }
    else
    {
       /* bad params, post NIL */
       _ret();
    }
}
```

Assembly Language Example

This Assembly example returns the current setting of the DOS disk write verify setting. Note that the example uses the Microsoft MASM simplified segment directives.

From CA-Clipper:

```
lVerify := VerifyFlag()
```

In Assembly language:

```
.MODEL      LARGE                ; simplified directives
            INCLUDE  EXTASM.INC
            PUBLIC   VERIFYFLAG
.CODE

VERIFYFLAG  PROC  FAR

            mov   ah, 54h        ; DOS function number
            int   21h            ; call DOS, flag (0/1) in AL

            ; post logical return value
            sub   ah, ah         ; clear AH
            push  ax
            call  __retl
            add   sp, 2          ; reset stack

            RET

VERIFYFLAG  ENDP
            END
```

Numeric Values

Internally, CA-Clipper represents numeric values in any of several forms, depending on the magnitude of the value and the operations performed on it.

The Extend System API also provides several forms of numeric representation depending on the needs of the Extend routine. The interface functions automatically convert the value to the requested form.

Accessing Numeric Values

Three different _par functions are provided for retrieving numeric values from CA-Clipper parameters. The functions differ only in the type of C value they return.

- The _parni() function returns a 16-bit (word) signed integer value. If the value of the requested CA-Clipper parameter is greater than the maximum value representable by a 16-bit integer, the value returned from _parni() is the low-order (least significant) 16 bits of the passed value.
- The _parnl() function returns a 32-bit (doubleword) signed integer value. If the value of the requested CA-Clipper parameter is greater than the maximum value representable by a 32-bit integer, the value returned from _parnl() is the low-order (least significant) 32 bits of the passed value.
- The _parnd() function returns an eight-byte (IEEE) double-precision floating point value. This format is sufficient to represent any possible CA-Clipper numeric value.

Note: From Assembly language, the value returned from _parnd() appears as a far pointer returned in the DX:AX register pair. The pointer points to a statically allocated buffer that contains a copy of the floating point value. Subsequent calls to _parnd() will overwrite this buffer, so you should make a copy of the value as needed.

Operations on Numeric Values

The numeric values returned by the _par functions can be operated on in the normal way in either C or Assembly language.

Note: If your C function manipulates floating point values, you must use the Microsoft C compiler version 8.0 and specify floating point emulation. For more information, refer to the section titled Using Floating Point Values.

Posting a Numeric Return Value

Three different _ret functions allow you to post a numeric value as a CA-Clipper return value. The functions differ only in the type of C parameter that you pass. All of them convert the passed C value into a CA-Clipper numeric value and post the new value into the CA-Clipper return value area.

- The _retni() function takes a C parameter of type int (a 16-bit signed integer).
- The _retnl() function takes a C parameter of type long (a 32-bit signed integer).
- The _retnd() function takes a C parameter of type double (an 8-byte IEEE floating point value).

Assigning a Numeric Value to a Reference Parameter

Three different _stor functions allow you to store a numeric value into a CA-Clipper variable that is passed by reference. All of the functions take two C parameters: the first specifies the value and the second specifies the CA-Clipper parameter to be affected. You must pass a third parameter, the array element, when storing into an array. The functions differ only in the type of C value that you pass. All of them convert the passed C value into a CA-Clipper numeric value and assign the new value to the specified CA-Clipper variable.

- The _storni() function takes a C parameter of type int (a 16-bit signed integer).
- The _stornl() function takes a C parameter of type long (a 32-bit signed integer).
- The _stornd() function takes a C parameter of type double (an 8-byte IEEE floating point value).

C Example

This C example retrieves numeric values from an array passed from CA-Clipper. It returns the array index of the element containing the greatest value. The function uses C doubles to contain the array values.

From CA-Clipper:

```
nIndex := AMax( {1, 5, 45, 22, 5, 9} )
```

In C:

```
#include "extend.api"

CLIPPER AMAX(void)
{
    int length;
    double val;
    int index;
    double highestVal;
    int highestIndex;

    if ( ISARRAY(1) )
    {
       length = ALENGTH(1);

       highestVal = 0;
       highestIndex = 0;

       /* scan array for highest value */
       for (index = 1; index <= length; index++)
       {
          val = _parnd(1, index);

          if (val > highestVal)
          {
             highestVal = val;
             highestIndex = index;
          }
       }

       /* post return value to CA-Clipper */
       _retni(highestIndex);
    }
    else
    {
       /* bad params, return NIL */
       _ret();
    }
}
```

Assembly Language Example

This Assembly example sends a specified 8-bit value to a specified hardware I/O port. The _parni() function is used to retrieve the two numeric values. Note that this example uses the Microsoft MASM simplified segment directives:

From CA-Clipper:

```
OutPort(nPort, nValue)
```

In Assembly Language:

```
.MODEL    LARGE                ; simplified directives
          INCLUDE  EXTASM.INC
          PUBLIC   OUTPORT
.CODE

OUTPORT   PROC  FAR

          push  si             ; preserve caller's SI

          ; check parameter types
          mov   ax, 1
          push  ax
          call  __parinfo      ; _parinfo(1)
          add   sp, 2          ; reset stack
          test  ax, NUMERIC
          jz gBye              ; forget it

          mov   ax, 2
          push  ax
          call  __parinfo      ; _parinfo(2)
          add   sp, 2          ; reset stack
          test  ax, NUMERIC
          jz gBye              ; forget it

          ; get port number
          mov   ax, 1
          push  ax
          call  __parni        ; _parni(1)
          add   sp, 2          ; reset stack
          mov   si, ax         ; save value in SI
                               ; (_par functions will preserve)

          ; get value to write
          mov   ax, 2
          push  ax
          call  __parni        ; _parni(2)
          add   sp, 2          ; reset stack

          ; output
          out   dx, al
```

```
gBye:
        call  __ret             ; post NIL ret value
                                ; back to CA-Clipper
        pop   si                ; restore
        RET

OUTPORT ENDP
        END
```

The NIL Value

NIL is a special data type in CA-Clipper used to represent the absence of a value. The NIL type, by definition, has only a single value: the NIL value. Thus, there is no interface function for accessing the value of a NIL parameter.

Nonetheless, detecting the presence of NIL values is useful since NIL is used to indicate omitted CA-Clipper parameters. C and Assembly language functions can test for the presence of NIL CA-Clipper parameters using _parinfo().

Compatibility Issues

This section discusses the compatibility issues concerned with using the Extend System API in previous and future versions of CA-Clipper, as well as compatibility with C compilers other than Microsoft C 8.0. You will want to read it if you have used the Extend System API in previous versions of CA-Clipper or if you are attempting to convert libraries designed for use with C to CA-Clipper.

Autumn '86

EXTENDA.MAC is provided in CA-Clipper as a means of retaining compatibility with Assembly language code written for Autumn '86. It is mentioned here only for the reference of those developers to which it applies. For information on EXTENDA.MAC refer to your Summer '87 documentation.

Warning! *EXTENDA.MAC macros do not save the registers they use automatically. You must, therefore, save registers of importance to you before calling any of these macros.*

Summer '87

EXTENDA.INC is provided in CA-Clipper as a means of retaining compatibility with Assembly language code written for Summer '87. For information on EXTENDA.INC refer to your Summer '87 documentation.

Nandef.h and Extend.h existed as separate header files in Summer '87. In CA-Clipper version 5.2, they were incorporated into Extend.api. If you included (#include) Nandef.h in your C programs, you can safely remove the #include under CA-Clipper versions 5.2 and higher. Extend.h has been supplied for compatibility reasons, but it only includes a single definition: #include "Extend.api".

C Compiler Caveats

One of the biggest problems that arises when attempting to interface C routines written for use with other languages is the C libraries themselves. C library routines are notorious for assuming that the runtime environment is in a certain condition or that memory allocation is performed through the use of standard C allocation techniques.

CA-Clipper does not use these memory allocation techniques nor does it leave the runtime environment in a C-like condition. Because of this, Extend routines will often attempt to pull in a large portion of the C runtime libraries or fail with meaningless results. The best way to ensure that your C code will be completely compatible with CA-Clipper is to avoid the use of built-in library routines altogether.

Obviously, it is not always possible to avoid the C runtime libraries, but care should be taken to avoid those functions that use certain types of runtime support.

Memory Allocation

Memory allocation is probably the number one pitfall when writing C language Extend routines for use with CA-Clipper. Beginning with version 5.0, CA-Clipper used the VM and FM APIs for memory allocation. Using these APIs for memory allocation is a requirement for writing compatible Extend routines. It is equally important that you properly free the memory you allocate.

Since all memory must be allocated through the VM or FM API, it should be no surprise that C memory allocation routines must be avoided at all costs. This includes alloca(), calloc(), malloc(), vmalloc(), and their companion freeing, locking, and unlocking functions. You can find which runtime C library functions require C memory allocation by either compiling a small program and checking the external references created in the object module or by viewing the library source code or documentation.

Note: malloc() and free() are available in the CA-Clipper runtime system and are simply mapped to _xalloc() and _xfree(). Since the CA-Clipper runtime system does not use malloc() and free(), you should avoid having this extra code overhead pulled in by replacing malloc() and free() calls with the appropriate FM or VM API calls. Furthermore, support for malloc() and free() is not guaranteed in future releases of CA-Clipper and is simply provided in versions 5.2 and higher for the convenience of the C programmer. Note also, that calls to these functions must adhere to the rules set down in the FM API.

String Manipulation

Most string manipulation functions are passed pointers to source and target strings, which means they are very safe and can be easily used with CA-Clipper. You need only allocate the source and target buffers using CA-Clipper memory management functions and pass the addresses to the string functions as usual.

When developing a C library for use by CA-Clipper and C programmers, this address-only passing method used by string manipulation functions is an excellent way to isolate memory allocation and deallocation.

Stream I/O

You should not use stream (buffered) I/O in Extend routines because they allocate memory. If your Extend routines require low-level file I/O, you should consider using the Filesys API file manipulation functions. These functions are always present in CLIPPER.LIB and used by all internal routines.

Furthermore, if your Extend routine needs to perform diagnostic text input and output, you should use cprintf(), cscanf(), sprintf(), and sscanf() rather than printf() or scanf(). You can also display values directly using the GT (General Terminal) API.

Math Libraries

CA-Clipper versions Summer '87 through 5.2 use the MSC Alternate Math routine for floating point calculations. CA-Clipper 5.3 uses the MSC floating point emulation for floating point calculations. If your code calls for any floating point code, you *must* link in MSC 8.0 LLIBCE.LIB. This library supports the large memory model and includes floating point emulation support routines.

Because of the differences in the design and implementation of the floating point emulation routines between vendors, only Microsoft C supports the use of C floating point routines that are compatible with CA-Clipper. Other vendor's implementations of the floating point emulation routines cause incompatibilities to arise between the floating point routines intrinsic in CA-Clipper and those that would be brought in by the non-Microsoft library.

As much as possible, you should avoid using runtime C library code that attempts to call floating point routines. This is mandatory when using a C compiler other that Microsoft C. Obviously, the math library routines fall into this category as does any use of the float or double variable types. Less obvious are the functions that can accept these variable types as parameters (such as printf()). Most graphics library routines also use floating point routines and should be avoided as well.

Linker Problems

Obviously, the best situation is one where you do not have to link in the C runtime library at all; however, if you must link with a C library, be sure to list the CA-Clipper libraries on the link line first. This is important because the automatic search request that CA-Clipper/Exospace performs to find the CA-Clipper libraries is performed *after* libraries on the link line are searched. This means you may accidentally replace some of the CA-Clipper low-level functions with versions from the C library which may not be compatible.

For example, the following linker command line causes the linker to search LLIBCE.LIB *before* CLIPPER.LIB:

```
EXOSPACE FI <filename> LIB LLIBCE
```

In the next example, however, the linker searches CLIPPER.LIB first:

```
EXOSPACE FI <filename> LIB LLIBG, CLIPPER, EXTEND, TERMINAL,
DBFNTX, LLIBCE NODEFLIB
```

C++ Extensions

The way in which C++ implements method operation overloading is by using a mechanism referred to as *symbol decoration* (or "name mangling"). This is how the polymorphic behaviors of methods are achieved under most C++ implementations. Symbol decoration is not compatible with CA-Clipper calling conventions in general and, therefore, cannot be tolerated.

Borland C

While we strongly recommend that you use Microsoft C products to link with CA-Clipper, we recognize that some users will want to create quick-and-dirty C functions with other products in hopes of avoiding the purchase of additional products. If you decide to attempt to use Borland C, remember that Borland's floating point and startup routines differ significantly from Microsoft's. Floating point operations are prohibited. The equivalent compile line syntax for Borland C is:

```
bcc -c -ml -f- -N- -G -O -Z <cfile>
```

Debuggers

Neither C source code nor C variables are visible from within the CA-Clipper debugger. This is because the C compiler does not include CA-Clipper-style debugging information.

Both CodeView and Turbo Debugger can be used to debug user-compiled C functions within a CA-Clipper application. CA-Clipper source code and CA-Clipper runtime library code will be displayed as machine code without symbolic information. Your C code (if properly compiled for the target debugger) will display properly as C source. You can set a breakpoint on your C function name, run to that breakpoint, and begin debugging from there.

Your C code must be compiled with the debugger information switch and the application must be linked with the proper linker and options for that product. For example, to debug with CodeView, compile with the /Zi option (instead of /Zl), replace /Oalt with /Od and then link using MS LINK adding the /CO option.

Important! *Some applications may be too large to debug using conventional debugging tools. It is best to test your Extend routines using a very small application before including them in a production application.*

Summary

This chapter has introduced you to the CA-Clipper Extend System API for interfacing with C, C++, and Assembly language. In it, you have learned the basic architecture of the system and most of the functions and macros available. The header files Extend.api and Extasm.inc are provided as source code in the \CLIP53\INCLUDE directory for your reference and use.

The next chapter provides an alphabetical reference of all Extend API interface and service functions.

Chapter 4
Extend System Reference Listing

Chapter 4
Extend System API Reference

The Extend System API is a set of functions designed for use in your C, C++, or Assembly language programs. They allow you to retrieve parameters passed from a CA-Clipper program, store their values for manipulation, and return values back to the CA-Clipper program. Thus, using the Extend System API you can call functions written in C, C++, or Assembly directly from a CA-Clipper program. This chapter is an alphabetical reference to all of the functions in the CA-Clipper Extend System API.

The prototypes for these functions are defined in the header file Extend.api, located in the \CLIP53\INCLUDE directory. The functions themselves are defined in CLIPPER.LIB, located in the \CLIP53\LIB directory.

Note: CA-Clipper uses the Microsoft C large model calling conventions. Also, any references to the C programming language also pertain to C++.

_parc()

Retrieve a character parameter

C Prototype

```
#include "extend.api"
char * _parc(
                int iParamNum
                [, int iArrayIndex]
              )
```

Arguments

iParamNum is the one-based ordinal position of the parameter in the parameter list.

iArrayIndex is an array index that specifies a particular element if the *iParamNum* parameter is an array.

Returns

_parc() returns a far pointer to a series of bytes that represent the character value.

Description

_parc() retrieves a character value passed as a parameter from CA-Clipper.

The pointer returned by _parc() may or may not point to the actual character value passed from CA-Clipper. In some cases, the pointer will point to a copy of the actual value. In other cases, it will point to an internal value that is shared by more than one CA-Clipper variable.

The pointer returned from _parc() should not directly modify the character value, since this can produce unpredictable results. To modify a character value, use _retc() returning a modified copy of the value or passing the associated CA-Clipper parameter by reference; then use _storc() to modify it.

Examples

- From C:

```
char *str;
str = _parc(1);
```

- From Assembly language:

```
EXTRN  __parc:FAR
       mov   ax, 1
       push  ax
       call  __parc            ; pointer returned in DX:AX
       add   sp, 2             ; reset stack pointer
```

Files Library is CLIPPER.LIB, header file is Extend.api.

See Also _parclen(), _retc(), _ret(), _storc()

_parclen()

Retrieve the length of a character parameter

C Prototype

```
#include "extend.api"
unsigned int _parclen(
                        int iParamNum
                        [, int iArrayIndex]
                      )
```

Arguments

iParamNum is the one-based ordinal position of the parameter in the parameter list.

iArrayIndex is an array index that specifies a particular element if the *iParamNum* parameter is an array.

Returns

_parclen() returns the length of the character value as an unsigned integer type.

Description

_parclen() returns the length of a character value passed as a parameter from CA-Clipper. The byte containing the null terminator is not included in the logical length. Embedded null bytes, however, are included if present.

Examples

- From C:

```
unsigned int len;
len = _parclen(1);
```

- From Assembly language:

```
EXTRN __parclen:FAR
      mov   ax, 1
      push  ax
      call  __parclen          ; length returned in AX
      add   sp, 2              ; reset stack pointer
```

Files

Library is CLIPPER.LIB, header file is Extend.api.

See Also

_parc(), _retc(), _storclen()

_parcsiz()

Retrieve the memory allocated for character parameters passed by reference

C Prototype

```
#include "extend.api"
unsigned int _parcsiz(
                        int iParamNum
                        [, int iArrayIndex]
                      )
```

Arguments

iParamNum is the one-based ordinal position of the parameter in the parameter list.

iArrayIndex is an array index that specifies a particular element if the *iParamNum* parameter is an array.

Returns

_parcsiz() returns the number of bytes of memory allocated for the specified parameter, including its null terminator.

Description

_parcsiz() is included for compatibility with previous releases of CA-Clipper. It is not designed to work with STATIC and LOCAL variables, and its use is not recommended.

Warning! *Obsolete items are not in keeping with the current CA-Clipper programming philosophy, and we strongly discourage their use as they may not be supported in future releases of CA-Clipper.*

Examples

- From C:

```
unsigned int len;
len = _parcsiz(1);
```

- From Assembly language:

```
EXTRN __parclen:FAR
      mov   ax, 1
      push  ax
      call  __parcsiz          ; length returned in AX
      add   sp, 2              ; reset stack pointer
```

Files Library is CLIPPER.LIB, header file is Extend.api.

See Also _parclen()

_pards()

Retrieve a date parameter as a string

C Prototype

```
#include "extend.api"
char * _pards(
                int iParamNum
                [, int iArrayIndex]
              )
```

Arguments

iParamNum is the one-based ordinal position of the parameter in the parameter list.

iArrayIndex is an array index that specifies a particular element if the *iParamNum* parameter is an array.

Returns

_pards() returns a far pointer to a series of bytes representing the date value.

Description

_pards() retrieves a date value passed as a parameter from CA-Clipper, converts it to a null-terminated character string of the form *yyyymmdd*, and returns a far pointer to that string.

Note: _pards() uses a single statically allocated buffer to contain the string representation of the date value. Each call to _pards() overwrites this buffer. If your Extend function must access more than one date, you should preserve the string value before calling _pards() again.

Examples

- From C:

```
char *str;
str = _pards(1);
```

- From Assembly language:

```
EXTRN __pards:FAR
      mov   ax, 1
      push  ax
      call  __pards          ; pointer returned in DX:AX
      add   sp, 2            ; reset stack pointer
```

Files Library is CLIPPER.LIB, header file is Extend.api.

See Also _retds(), _stords()

_parinfa()

Determine the length or element type of an array parameter

C Prototype

```
#include "extend.api"
int _parinfa(
                int iParamNum,
                unsigned int uiArrayIndex
            )
```

Arguments

iParamNum is the one-based ordinal position of the parameter in the parameter list.

uiArrayIndex is the array element index.

Returns

If *uiArrayIndex* is zero, _parinfa() returns the number of elements in the specified array. Otherwise, _parinfa() returns an integer value that indicates the data type of the specified array element. The type codes are summarized in the table below:

_parinfa() Return Values

Value	CA-Clipper Type	Extend.api Manifest Constant
0	NIL	UNDEF
1	Character	CHARACTER
2	Numeric	NUMERIC
4	Logical	LOGICAL
8	Date	DATE
32	Passed by reference	MPTR
65	Memo	MEMO
512	Array	ARRAY

Description

_parinfa() determines the length of an array parameter or the type of one of its elements. You can also determine the length of an array using the ALENGTH() macro defined in Extend.api.

Warning! *Type checking is important since CA-Clipper arrays may contain mixed data types. To assure that your Extend routine receives the correct data type, check the data type of each element before you use it.*

Examples

- From C:

```
int len;
int type;
len = _parinfa(1, 0);
type = _parinfa(1, 1);
```

- From Assembly language:

```
EXTRN __parinfa:FAR
      mov   ax, 0             ; params pushed in reverse
      push  ax
      mov   ax, 1
      push  ax
      call  __parinfa         ; length returned in AX
      add   sp, 4             ; reset stack pointer
      mov   ax, 1
      push  ax
      push  ax
      call  __parinfa         ; type code returned in AX
      add   sp, 4             ; reset stack pointer
```

Files

Library is CLIPPER.LIB, header file is Extend.api.

See Also

_parinfo()

_parinfo()

Determine the parameter count or the data type of a parameter

C Prototype

```
#include "extend.api"
int _parinfo(
                int iParamNum
            )
```

Arguments

iParamNum is the one-based ordinal position of the parameter in the parameter list.

Returns

If *iParamNum* is zero, _parinfo() returns the number of parameters passed from CA-Clipper. Otherwise, _parinfo() returns an integer value representing the data type of the indicated parameter. The following table summarizes the type codes:

_parinfo() Return Values

Value	CA-Clipper Type	Extend.api Manifest Constant
0	NIL	UNDEF
1	Character	CHARACTER
2	Numeric	NUMERIC
4	Logical	LOGICAL
8	Date	DATE
32	Passed by reference	MPTR
65	Memo	MEMO
512	Array	ARRAY

Description

_parinfo() determines either the CA-Clipper data type of a parameter or the number of parameters passed. You can also determine the number of parameters passed by using the PCOUNT macro defined in Extend.api.

To use _parinfo() to determine if a parameter is of a particular type and was also passed by reference, perform a logical OR of the desired type code and the MPTR code; then test for the resulting value. For example, when a CA-Clipper variable containing a character value is passed by reference, _parinfo() returns 33: MPTR (32) OR'ed with CHARACTER (1).

The ISBYREF() macro defined in Extend.api also determines if a parameter is passed by reference. This pseudofunction accepts a parameter number as its argument and returns one, (true (.T.)), if the parameter was passed by reference.

Examples

- From C:

```
int count;
int type;
count = _parinfo(0);
type = _parinfo(1);
```

- From Assembly language:

```
EXTRN __parinfo:FAR
      mov   ax, 0
      push  ax
      call  __parinfo          ; count returned in AX
      add   sp, 2              ; reset stack pointer
      mov   ax, 1
      push  ax
      call  __parinfo          ; type code returned in AX
      add   sp, 2              ; reset stack pointer
```

Files

Library is CLIPPER.LIB, header file is Extend.api.

See Also

_parinfa()

_parl()

Retrieve a logical parameter as an integer

C Prototype

```
#include "extend.api"
int _parl(
              int iParamNum
              [, int iArrayIndex]
          )
```

Arguments

iParamNum is the one-based ordinal position of the parameter in the parameter list.

iArrayIndex is an array index that specifies a particular element if the *iParamNum* parameter is an array.

Returns

_parl() returns an int type value where one represents true (.T.) and zero represents false (.F.).

Description

_parl() retrieves a logical value passed as a parameter from CA-Clipper. If the specified CA-Clipper parameter was true (.T.), _parl() returns one; otherwise, it returns zero.

Examples

- From C:

```
int log;
log = _parl(1);
```

- From Assembly language:

```
EXTRN __parl:FAR
      mov   ax, 1
      push  ax
      call  __parl            ; value returned in AX
      add   sp, 2             ; reset stack pointer
```

Files

Library is CLIPPER.LIB, header file is Extend.api.

See Also

_retl(), _storl()

_parnd()

Retrieve a numeric parameter as a double

C Prototype

```
#include "extend.api"
double _parnd(
                int iParamNum
                [, int iArrayIndex]
              )
```

Arguments

iParamNum is the one-based ordinal position of the parameter in the parameter list.

iArrayIndex is an array index that specifies a particular element if the *iParamNum* parameter is an array.

Returns

_parnd() returns the value of the specified parameter as a double.

Description

_parnd() retrieves a numeric value passed as a parameter from CA-Clipper and converts it to a double-precision floating point value.

_parnd() returns a far pointer to a statically allocated buffer that contains the double numeric value. When you call _parnd() from C, the double value is automatically copied from the buffer into your C variable. If you call from Assembly language, however, you should take care to preserve the double value since subsequent calls to _parnd() will overwrite the buffer.

Examples

- From C:

```
double num;
num = _parnd(1);
```

- From Assembly language:

```
EXTRN  __parnd:FAR
       mov   ax, 1
       push  ax
       call  __parnd          ; value pointed to by DX:AX
       add   sp, 2            ; reset stack pointer
       mov   es, dx           ; load pointer in ES:BX
       mov   bx, ax           ; copy double into save area
                              ; (creation of save area not shown)
                              ;
       mov   word ptr MySaveArea, es:[bx]
       mov   word ptr MySaveArea+2, es:[bx+2]
       mov   word ptr MySaveArea+4, es:[bx+4]
       mov   word PTR MySaveArea+6, es:[bx+6]
```

Files Library is CLIPPER.LIB, header file is Extend.api.

See Also _parni(), _parnl(), _retnd(), _stornd()

_parni()

Retrieve a numeric parameter as an integer

C Prototype

```
#include "extend.api"
int _parni(
              int iParamNum
              [, int iArrayIndex]
           )
```

Arguments

iParamNum is the one-based ordinal position of the parameter in the parameter list.

iArrayIndex is an array index that specifies a particular element if the *iParamNum* parameter is an array.

Returns

_parni() returns the value of the specified parameter as an integer.

Description

_parni() retrieves a numeric value passed as a parameter from CA-Clipper and converts it to an integer (type).

Examples

- From C:

```
int num;
num = _parni(1);
```

- From Assembly language:

```
EXTRN __parni:FAR
      mov   ax, 1
      push  ax
      call  __parni          ; value returned in AX
      add   sp, 2            ; reset stack pointer
```

Files

Library is CLIPPER.LIB, header file is Extend.api.

See Also

_parnd(), _parnl(), _retni(), _storni()

_parnl()

Retrieve a numeric parameter as a long

C Prototype

```
#include "extend.api"
long _parnl(
              int iParamNum
              [, int iArrayIndex]
            )
```

Arguments

iParamNum is the one-based ordinal position of the parameter in the parameter list.

iArrayIndex is an array index that specifies a particular element if the *iParamNum* parameter is an array.

Returns

_parnl() returns the value of the specified parameter as a long.

Description

_parnl() retrieves a numeric value passed as a parameter from CA-Clipper and converts it to a long.

Examples

- From C:

```
long num;
long = _parnl(1);
```

- From Assembly language:

```
EXTRN __parnl:FAR
      mov   ax, 1
      push  ax
      call  __parnl        ; value returned in DX:AX
      add   sp, 2          ; reset stack pointer
```

Files

Library is CLIPPER.LIB, header file is Extend.api.

See Also

_parnd(), _parni(), _retnl(), _stornl()

_ret()

Post a NIL return value

C Prototype

```
#include "extend.api"
void _ret(void)
```

Returns

_ret() has no return value.

Description

_ret() posts a NIL value into CA-Clipper's return value area. When your Extend routine returns control to the calling CA-Clipper program, the posted value becomes the CA-Clipper return value of your Extend routine.

Note: None of the following _ret functions return control to CA-Clipper. The _ret functions only post a return value to be used by CA-Clipper as the Extend routines return value. This does not happen until the Extend routine actually returns control in whatever manner is normal for that language (e.g., the *return* statement in C).

Examples

- From C:

```
_ret();
```

- From Assembly language:

```
EXTRN __ret:FAR
      call __ret
```

Files

Library is CLIPPER.LIB, header file is Extend.api.

See Also

_retc(), _retclen(), _retds(), _retl(), _retnd(), _retni(), _retnl()

_retc()

Post a character return value using a null-terminated string

C Prototype

```
#include "extend.api"
void _retc(
                char far * fpString
          )
```

Arguments

fpString is a far pointer to the null-terminated string whose contents are to be returned as a character value.

Returns

_retc() has no return value.

Description

_retc() posts a character value into CA-Clipper's return value area. When your Extend routine returns control to the calling CA-Clipper program, the posted value becomes the CA-Clipper return value of your Extend routine.

_retc() determines the logical length of the character value by scanning the supplied string for a null terminator byte. If you are returning binary data that may contain embedded null bytes, use _retclen() instead.

Note: _retc() automatically allocates memory in the CA-Clipper heap and makes a copy of the supplied string. The string need not be preserved after the call to _retc().

Examples

- From C:

```
_retc("hello world");
```

- From Assembly language:

```
EXTRN __retc:FAR
      mov   dx, seg MyString      ; pass address of string
                                  ; (creation of string not shown)
      mov   ax, offset MyString
      push  dx
      push  ax
      call  __retc
      add   sp, 4                 ; reset stack pointer
```

Files Library is CLIPPER.LIB, header file is Extend.api.

See Also _parc(), _parclen(), _retclen(), _storc()

_retclen()

Post a character return value with explicit length

C Prototype

```
#include "extend.api"
void _retclen(
                char far * fpString,
                unsigned int uiLength
              )
```

Arguments

fpString is a far pointer to the data representing the character value to be returned.

uiLength is the logical length of the string.

Returns

_retclen() has no return value.

Description

_retclen() posts a character value into CA-Clipper's return value area. When your Extend routine returns control to the calling CA-Clipper program, the posted value becomes the CA-Clipper return value of your Extend routine.

Since _retclen() allows you to specify an explicit logical length for the character value, the string you supply does not need to be null-terminated and may safely contain embedded null bytes.

Note: _retclen() automatically allocates memory in the CA-Clipper heap and makes a copy of the supplied string. The string need not be preserved after the call to _retclen().

Examples

- From C:

```
char data[5] = {1, 0, 2, 0, 3, 3};  // two embedded null bytes
_retclen(data, 5);
```

- From Assembly language:

```
EXTRN __retclen:FAR
      mov   ax, 5                ; params pushed in reverse
      push  ax
      mov   dx, seg MyData       ; pass address of data
                                 ; (creation of MyData not shown)
      mov   ax, offset MyData
      push  dx
      push  ax
      call  __retclen
      add   sp, 6                ; reset stack pointer
```

Files Library is CLIPPER.LIB, header file is Extend.api.

See Also _parc(), _parclen(), _retc(), _storclen()

_retds()

Post a date return value using a date string

C Prototype

```
#include "extend.api"
void _retds(
              char far * fpString
            )
```

Arguments

fpString is a far pointer to a null-terminated string representing the date value to be returned. The date string must have the form *yyyymmdd*.

Returns

_retds() has no return value.

Description

_retds() posts a date value into CA-Clipper's return value area. When your Extend routine returns control to the calling CA-Clipper program, the posted value becomes the CA-Clipper return value of your Extend routine.

_retds() converts the supplied string into a CA-Clipper date value. If you supply a string that does not represent a valid date, a blank date value will be stored.

Note: _retds() converts the supplied string to a date value. The string need not be preserved after the call to _retds().

Examples

- From C:

```
_retds("19991231");
```

- From Assembly language:

```
EXTRN __retds:FAR
      mov   dx, seg MyString     ; pass address of string
                                 ; (creation of string not shown)
      push  dx
      mov   ax, offset MyString
      push  ax
      call  __retds
      add   sp, 4
```

Files Library is CLIPPER.LIB, header file is Extend.api.

See Also _pards(), _stords()

_retl()

Post a logical return value

C Prototype

```
#include "extend.api"
void _retl(
             int iLogical
          )
```

Arguments

iLogical is an integer type that represents the logical value to be returned. A value of zero is interpreted as false (.F.). Any other value is interpreted as true (.T.).

Returns

_retl() has no return value.

Description

_retl() posts a logical value into CA-Clipper's return value area. When your Extend routine returns control to the calling CA-Clipper program, the posted value becomes the CA-Clipper return value of your Extend routine.

Examples

- From C:

```
_retl(1);
```

- From Assembly language:

```
EXTRN __retl:FAR
      mov   ax, 1
      push  ax
      call  __retl
      add   sp, 2              ; reset stack pointer
```

Files

Library is CLIPPER.LIB, header file is Extend.api.

See Also

_parl(), _storl()

_retnd()

Post a numeric return value using a double-precision numeric value

C Prototype

```
#include "extend.api"
void _retnd(
              double dNumber
            )
```

Arguments

dNumber is a numeric expression of type double.

Returns

_retnd() has no return value.

Description

_retnd() posts a numeric value into CA-Clipper's return value area. When your Extend routine returns control to the calling CA-Clipper program, the posted value becomes the CA-Clipper return value of your Extend routine.

Examples

- From C:

```
_retnd( (double)3.14 );
```

- From Assembly language:

```
EXTRN __retnd:FAR
      push  word ptr (MyPi+6)     ; push double number
                                  ; (creation of MyPi not shown)
      push  word ptr (MyPi+2)
      push  word ptr (MyPi+4)
      push  word ptr MyPi
      call  __retnd
      add   sp, 8                 ; reset stack pointer
```

Files

Library is CLIPPER.LIB, header file is Extend.api.

See Also

_parnd(), _parni(), _parnl(), _retni(), _retnl(), _stornd()

_retni()

Post a numeric return value using an integer

C Prototype

```
#include "extend.api"
void _retni(
              int iNumber
            )
```

Arguments

iNumber is a numeric expression of integer (type).

Returns

_retni() has no return value.

Description

_retni() posts a numeric value into CA-Clipper's return value area. When your Extend routine returns control to the calling CA-Clipper program, the posted value becomes the CA-Clipper return value of your Extend routine.

Examples

- From C:

```
_retni(99);
```

- From Assembly language:

```
EXTRN __retni:FAR
      mov   ax, 99
      push  ax
      call  __retni
      add   sp, 2              ; reset stack pointer
```

Files

Library is CLIPPER.LIB, header file is Extend.api.

See Also

_parnd(), _parni(), _parnl(), _retnd(), _retnl(), _storni()

_retnl()

Post a numeric return value using a long

C Prototype

```
#include "extend.api"
void _retnl(
              long lNumber
            )
```

Arguments

lNumber is a numeric expression of type long.

Returns

_retnl() has no return value.

Description

_retnl() posts a numeric value into CA-Clipper's return value area. When your Extend routine returns control to the calling CA-Clipper program, the posted value becomes the CA-Clipper return value of your Extend routine.

Examples

- From C:

```
_retnl(1234567689L);
```

- From Assembly language:

```
EXTRN __retni:FAR
      mov   dx, 1883
      mov   ax, 52501
      push  dx                ; push hi-order first
      push  ax
      call  __retnl
      add   sp, 4             ; reset stack pointer
```

Files

Library is CLIPPER.LIB, header file is Extend.api.

See Also

_parnd(), _parni(), _parnl(), _retnd(), _retni(), _stornl()

_storc()

Assign a character value to a referenced variable using a null-terminated string

C Prototype

```
#include "extend.api"
int _storc(
              char far * fpString,
              int iParamNum
              [, int iArrayIndex]
          )
```

Arguments

fpString is a far pointer to a null-terminated string representing the character value to be assigned.

iParamNum is the one-based ordinal position in the parameter list of the parameter to be assigned.

iArrayIndex is an array index that specifies a particular element if the *iParamNum* parameter is an array.

Returns

_storc() returns one if the function is successful; otherwise, it returns zero.

Description

_storc() stores a character value to a variable passed by reference as a parameter from CA-Clipper. If the parameter specified by *iParamNum* is not passed by reference, _storc() ignores the call and returns a value of zero.

_storc() determines the logical length of the character value by scanning the supplied string for a null terminator byte. If you are attempting to assign binary data that may contain embedded null bytes, use _storclen() instead.

Note: _storc() automatically allocates memory in the CA-Clipper heap and makes a copy of the supplied string. The string need not be preserved after the call to _storc().

Examples

- From C:

```
_storc("hello world", 1);
```

- From Assembly language:

```
EXTRN __storc:FAR
      mov   ax, 1                ; iParamNum
      push  ax
      mov   dx, seg MyString     ; pass address of string
                                 ; (creation of string not shown)
      mov   ax, offset MyString
      push  dx
      push  ax
      call  __storc
      add   sp, 6                ; reset stack pointer
```

Files Library is CLIPPER.LIB, header file is Extend.api.

See Also _parc(), _parclen(), _retc(), _retclen(), _storclen()

_storclen()

Assign a character value to a referenced variable using a string with explicit length

C Prototype

```
#include "extend.api"
int _storclen(
                char far * fpString,
                unsigned int uiLength,
                int iParamNum
                [, int iArrayIndex]
              )
```

Arguments

fpString is a far pointer to data representing the character value to be assigned.

uiLength specifies the logical length of the character value to be assigned.

iParamNum is the one-based ordinal position in the parameter list of the parameter to be assigned.

iArrayIndex is an array index that specifies a particular element if the *iParamNum* parameter is an array.

Returns

_storclen() returns one if the function is successful; otherwise, it returns zero.

Description

_storclen() stores a character value to a variable passed by reference as a parameter from CA-Clipper. If the parameter specified by *iParamNum* is not passed by reference, _storclen() ignores the call and returns a value of zero.

Since _storclen() allows you to specify an explicit logical length for the character value, the string you supply does not need to be null-terminated and may safely contain embedded null bytes.

Note: _storclen() automatically allocates memory in the CA-Clipper heap and makes a copy of the supplied string. The string need not be preserved after the call to _storclen().

Examples

- From C:

```
char data[5] = {1, 0, 2, 0, 3, 3};
_storclen(data, 5, 1);
```

- From Assembly language:

```
EXTRN __storclen:FAR
      mov   ax, 1             ; iParamNum
      push  ax
      mov   ax, 5             ; logical length
      push  ax
      mov   dx, seg MyData    ; pass address of data
                              ; (creation of MyData not shown)
      mov   ax, offset MyData
      push  dx
      push  ax
      call  __storclen
      add   sp, 8             ; reset stack pointer
```

Files Library is CLIPPER.LIB, header file is Extend.api.

See Also _parc(), _parclen(), _retc(), _retclen(), _storc()

_stords()

Assign a date value to a referenced variable using a date string

C Prototype

```
#include "extend.api"
int _stords(
              char far * fpString,
              int iParamNum
              [, int iArrayIndex]
            )
```

Arguments

fpString is a far pointer to a null-terminated string representing the date value to be assigned. The string must have the form *yyyymmdd*.

iParamNum is the one-based ordinal position in the parameter list of the parameter to be assigned.

iArrayIndex is an array index that specifies a particular element if the *iParamNum* parameter is an array.

Returns

_stords() returns one if the function is successful; otherwise, it returns zero.

Description

_stords() stores a date value to a variable passed by reference as a parameter from CA-Clipper. If the parameter specified by *iParamNum* is not passed by reference, _stords() ignores the call and returns a value of zero.

_stords() converts the supplied string into a date value. If you supply a string that does not represent a valid date, a blank date value will be stored.

Note: _stords() converts the supplied string into a date value. The string need not be preserved after the call to _stords().

Examples

- From C:

```
_stords("19991231", 1);
```

- From Assembly language:

```
EXTRN __stords:FAR
      mov   ax, 1                ; iParamNum
      push  ax
      mov   dx, seg MyString     ; pass address of string
                                 ; (creation of string not shown)
      mov   ax, offset MyString
      push  dx
      push  ax
      call  __stords
      add   sp, 6                ; reset stack pointer
```

Files

Library is CLIPPER.LIB, header file is Extend.api.

See Also

_pards(), _retds()

_storl()

Assign a logical value to a referenced variable

C Prototype

```
#include "extend.api"
int _storl(
              int iLogical,
              int iParamNum
              [, int iArrayIndex]
           )
```

Arguments

iLogical is an integer representing the logical value to be assigned. A value of zero represents false (.F.). Any other value represents true (.T.).

iParamNum is the one-based ordinal position in the parameter list of the parameter to be assigned.

iArrayIndex is an array index that specifies a particular element if the *iParamNum* parameter is an array.

Returns

_storl() returns one if the function is successful; otherwise, it returns zero.

Description

_storl() stores a logical value to a variable passed by reference as a parameter from CA-Clipper. If the parameter specified by *iParamNum* is not passed by reference, _storl() ignores the call and returns a value of zero.

Examples

- From C:

```
_storl(1, 1);
```

- From Assembly language:

```
EXTRN  __storl:FAR
       mov   ax, 1              ; iParamNum
       push  ax
       push  ax
       call  __storl
       add   sp, 4              ; reset stack pointer
```

Files Library is CLIPPER.LIB, header file is Extend.api.

See Also _parl(), _retl()

_stornd()

Assign a numeric value to a referenced variable using a double-precision type value

C Prototype

```
#include "extend.api"
int _stornd(
              double dNumber,
              int iParamNum
              [, int iArrayIndex]
            )
```

Arguments

dNumber is a numeric expression of double-precision type.

iParamNum is the one-based ordinal position in the parameter list of the parameter to be assigned.

iArrayIndex is an array index that specifies a particular element if the *iParamNum* parameter is an array.

Returns

_stornd() returns one if the function is successful; otherwise, it returns zero.

Description

_stornd() stores a numeric value to a variable passed by reference as a parameter from CA-Clipper. If the parameter specified by *iParamNum* is not passed by reference, _stornd() ignores the call and returns a value of zero.

Examples

- From C:

```
_stornd( (double)3.14, 1 );
```

- From Assembly language:

```
EXTRN __stornd:FAR
      mov   ax, 1
      push  ax                      ; iParamNum
      push  word ptr (MyPi+6)       ; push double number
                                    ; (creation of MyPi not shown)
      push  word ptr (MyPi+2)
      push  word ptr (MyPi+4)
      push  word ptr MyPi
      call  __stornd
      add   sp, 10                  ; reset stack pointer
```

Files Library is CLIPPER.LIB, header file is Extend.api.

See Also _parnd(), _parni(), _parnl(), _retnd(), _retni(), _retnl(), _storni(), _stornl()

_storni()

Assign a numeric value to a referenced variable using an integer

C Prototype

```
#include "extend.api"
int _storni(
              int iNumber,
              int iParamNum
              [, int iArrayIndex]
            )
```

Arguments

iNumber is a numeric expression of type integer.

iParamNum is the one-based ordinal position in the parameter list of the parameter to be assigned.

iArrayIndex is an array index that specifies a particular element if the *iParamNum* parameter is an array.

Returns

_storni() returns one if the function is successful; otherwise, it returns zero.

Description

_storni() stores a numeric value to a variable passed by reference as a parameter from CA-Clipper. If the parameter specified by *iParamNum* is not passed by reference, _storni() ignores the call and returns a value of zero.

Examples

- From C:

```
_storni( 123, 1 );
```

- From Assembly language:

```
EXTRN __storni:FAR
      mov   ax, 1
      push  ax                ; iParamNum
      mov   ax, 123
      push  ax                ; iNumber
      call  __storni
      add   sp, 4             ; reset stack pointer
```

Files Library is CLIPPER.LIB, header file is Extend.api.

See Also _parnd(), _parni(), _parnl(), _retnd(), _retni(), _retnl(), _stornd(), _stornl()

_stornl()

Assign a numeric value to a referenced variable using a long

C Prototype

```
#include "extend.api"
int _stornl(
              long lNumber,
              int iParamNum
              [, int iArrayIndex]
            )
```

Arguments

lNumber is a numeric expression of type long.

iParamNum is the one-based ordinal position in the parameter list of the parameter to be assigned.

iArrayIndex is an array index that specifies a particular element if the *iParamNum* parameter is an array.

Returns

_stornl() returns one if the function is successful; otherwise, it returns zero.

Description

_stornl() stores a numeric value to a variable passed by reference as a parameter from CA-Clipper. If the parameter specified by *iParamNum* is not passed by reference, _stornl() ignores the call and returns a value of zero.

Examples

- From C:

```
_stornl( 123456789L, 1 );
```

- From Assembly language:

```
EXTRN __stornl:FAR
      mov   ax, 1
      push  ax                    ; iParamNum
      mov   dx, 1883
      mov   ax, 52501
      push  dx                    ; push hi-order first
      push  ax
      call  __stornl
      add   sp, 6                 ; reset stack pointer
```

Files Library is CLIPPER.LIB, header file is Extend.api.

See Also _parnd(), _parni(), _parnl(), _retnd(), _retni(), _retnl(), _stornd(), _storni()

Chapter 5
Item API Reference Listing

Chapter 5
Item API Reference

The Item API gives your Extend routines the ability to manipulate data items using CA-Clipper data types, including the ability to evaluate code blocks. This chapter is an alphabetical reference to all of the functions in the CA-Clipper Item API.

The prototypes for these functions are defined in the header file Item.api, located in the \CLIP53\INCLUDE directory. Data types used in the prototypes are defined in the header file Clipdefs.h or one of the .api header files, also located in \CLIP53\INCLUDE. The functions themselves are defined in CLIPPER.LIB, located in the \CLIP53\LIB directory.

_evalLaunch()

Call CA-Clipper code blocks from C

C Prototype

```
#include "item.api"
ITEM _evalLaunch(
                    EVALINFOP evalInfoP
                  )
```

Arguments

evalInfoP is a properly set EVALINFO structure (as set by _evalNew() and _evalPutParam()).

Returns

The item that is the result of the evaluation.

Description

The _evalLaunch() function allows you to call any code that can be defined in a code block from C. Note also that you may call a particular symbol by name if you pass a character string item to _evalNew() when creating the EVALINFO structure.

_evalLaunch() requires a EVALINFO structure properly initialized via _evalNew() with parameters placed in the structure via _evalPutParam(). Failure to follow this protocol exactly will result in catastrophe.

During the process of launching an evaluation, the Item API creates new item references for each item placed in the parameter list. After an _evalLaunch() call is made, you must call _evalRelease() to individually release all of the references, or the items used as parameters will never be released from object memory for the garbage collector.

Examples

```
/*
* USERDO()
* --------
*/

CLIPPER userDO( void )
{
   EVALINFO info;
   USHORT   uiParam;
   ITEM     retP;

   /* Get evaluation expression */

   if ( PCOUNT < 1 )
   {
      _ret();
      return;
   }
   else
   {
      _evalNew( &info, _itemParam( 1 ) );
   }

   /* Get parameters */

   for ( uiParam=2; uiParam <= PCOUNT; uiParam++ )
   {
      _evalPutParam( &info, _itemParam(uiParam) );
   }

   /* Launch evaluation information */

   retP = _evalLaunch( &info );

   /* Release ITEMs associated w/eval info */

   _evalRelease( &info );

   _itemReturn ( retP );
   _itemRelease( retP );

   return;
}
```

Files Library is CLIPPER.LIB, header file is Item.api.

See Also _evalNew(), _evalPutParam(), _evalRelease()

_evalNew()

Initialize an EVALINFO structure for use

C Prototype

```
#include "item.api"
BOOL _evalNew(
                  EVALINFOP evalInfoP,
                  ITEM itmEval
              )
```

Arguments

evalInfoP is the structure to initialize.

itmEval is a character or block item that is to be evaluated.

Returns

True (.T.) if the operation was successful.

Description

The _evalNew() function resets an EVALINFO structure for evaluation of a code block or routine name. If *itmEval* is a character item, it is assumed to be the symbol name of a routine to execute. Otherwise, *itmEval* is assumed to be a valid code block.

Examples

```
/*
* USERDO()
* --------
*/

CLIPPER userDO( void )
{
   EVALINFO info;
   USHORT   uiParam;
   ITEM     retP;

   /* Get evaluation expression */

   if ( PCOUNT < 1 )
   {
      _ret();
      return;
   }
   else
   {
      _evalNew( &info, _itemParam( 1 ) );
   }

   /* Get parameters */

   for ( uiParam=2; uiParam <= PCOUNT; uiParam++ )
   {
      _evalPutParam( &info, _itemParam(uiParam) );
   }

   /* Launch evaluation information */

   retP = _evalLaunch( &info );

   /* Release ITEMs associated w/eval info */

   _evalRelease( &info );

   _itemReturn ( retP );
   _itemRelease( retP );

   return;
}
```

Files Library is CLIPPER.LIB, header file is Item.api.

See Also _evalLaunch(), _evalPutParam()

_evalPutParam()

Place a parameter in an EVALINFO structure

C Prototype

```
#include "item.api"
BOOL _evalPutParam(
                        EVALINFOP evalInfoP,
                        ITEM itmParam
                    )
```

Arguments

evalInfoP is the structure in which to add the parameter.

itmParam is the parameter to add to the structure.

Returns

True (.T.) if the operation was successful.

Description

The _evalPutParam() function allows you to place parameters in an EVALINFO structure one at a time. Parameters are placed in the order that they are passed (i.e., the first call to _evalPutParam() places the first parameter, the second call places the second parameter, and so on).

Note: If a particular evaluation does not require parameters, there is no need to call this function at all, simply call _evalNew() and then _evalLaunch().

Warning! *Parameters in an EVALINFO structure should not be released (via _itemRelease() or _evalRelease()) until the evaluation is performed or the EVALINFO structure is no longer needed.*

Examples

```
/*
* USERDO()
* --------
*/

CLIPPER userDO( void )
{
    EVALINFO info;
    USHORT   uiParam;
    ITEM     retP;

    /* Get evaluation expression */

    if ( PCOUNT < 1 )
    {
       _ret();
    return;
    }
    else
    {
       _evalNew( &info, _itemParam( 1 ) );
    }

    /* Get parameters */

    for ( uiParam=2; uiParam <= PCOUNT; uiParam++ )
    {
       _evalPutParam( &info, _itemParam(uiParam) );
    }

    /* Launch evaluation information */

    retP = _evalLaunch( &info );

    /* Release ITEMs associated w/eval info */

    _evalRelease( &info );

    _itemReturn ( retP );
    _itemRelease( retP );

    return;
}
```

Files Library is CLIPPER.LIB, header file is Item.api.

See Also _evalLaunch(), _evalNew(), _evalRelease(), _itemRelease()

_evalRelease()

Release all parameter references in an EVALINFO structure

C Prototype

```
#include "item.api"
BOOL _evalRelease(
                  EVALINFOP evalInfoP
                 )
```

Arguments

evalInfoP is the structure *already used to launch an evaluation.*

Returns

True (.T.) if the operation was successful.

Description

During the process of launching an evaluation, the Item API creates new item references for each item placed in the parameter list. After an _evalLaunch() call is made, you must call _evalRelease() to individually release all of the references.

Warning! *It is vitally important that items be released once you no longer have need for them. For* ***each launch*** *of a particular EVALINFO structure, you* ***must*** *call _evalRelease(). Failure to do so may cause a CA-Clipper stack fault or memory errors.*

Warning! *Do* ***NOT*** *pass this function an EVALINFO structure that has* ***NOT*** *been subject to an _evalLaunch().*

Examples

```
/*
* USERDO()
* --------
*/

CLIPPER userDO( void )
{
   EVALINFO info;
   USHORT   uiParam;
   ITEM     retP;

   /* Get evaluation expression */

   if ( PCOUNT < 1 )
   {
      _ret();
      return;
   }
   else
   {
      _evalNew( &info, _itemParam( 1 ) );
   }

   /* Get parameters */

   for ( uiParam=2; uiParam <= PCOUNT; uiParam++ )
   {
      _evalPutParam( &info, _itemParam(uiParam) );
   }

   /* Launch evaluation information */

   retP = _evalLaunch( &info );

   /* Release ITEMs associated w/eval info */

   _evalRelease( &info );

   _itemReturn ( retP );
   _itemRelease( retP );

   return;
}
```

Files Library is CLIPPER.LIB, header file is Item.api.

See Also _evalLaunch(), _evalPutParam()

_itemArrayGet()

Retrieve an element of an item of type array

C Prototype

```
#include "item.api"
ITEM _itemArrayGet(
                        ITEM itmArr,
                        USHORT uiElem
                    )
```

Arguments

itmArr is the item of type array from which to retrieve an element.

uiElem is the element number to retrieve.

Returns

A new item containing a reference to the element number requested from the array.

Description

You can use _itemArrayGet() to retrieve a particular element of an array. The element retrieved may be any type supported by the Item API, and may even be another array.

Since _itemArrayGet() creates a new item reference, do not forget to release the new reference with _itemRelease() after use. Failure to do so could result in a CA-Clipper stack fault.

Note: Since CA-Clipper arrays are one-based, not zero-based, the *uiElem* parameter must not be zero.

Examples

```
/*
* MYACLONE()
* ----------
* myAClone( <aArray1> ) --> aNewArray
*/

HIDE ITEM near _xAClone( ITEM aSrc );

CLIPPER ARRAYCLONE( void )
{
   ITEM aSource;          // Source array
   ITEM aCloned;          // New array

   /* Do parameters check */

   if ( PCOUNT != 1 )
   {
      return;
   }

   /* Get source array as ITEM */

   aSource = _itemParam( 1 );

   /* Clone source array  */

   aCloned = _xAClone( aSource );

   /* Return clone, release clone and source */

   _itemReturn ( aCloned );

   _itemRelease( aSource );
   _itemRelease( aCloned );

   return;
}
```

```
HIDE ITEM near _xAClone( ITEM aSrc )
{
   ITEM   temp;
   ITEM   aTemp;
   ITEM   aNew;

   USHORT i;
   USHORT nLen;

   // If item not an array, return NIL
   if ( !( _itemType( aSrc ) == ARRAY ) )
      return ( _itemNew( NULL ) );

   // Get number of elements, create new array
   nLen = _itemSize( aSrc );
   aNew = _itemArrayNew( nLen );

   // For each element... get source & copy
   for ( i = 1; i <= nLen; i++ )
   {
      temp = _itemArrayGet( aSrc, i );

      // If it's an array, copy via recursion
      if ( _itemType( temp ) == ARRAY )
      {
         aTemp = _xAClone( temp );
         _itemArrayPut( aNew, i, aTemp );
         _itemRelease( aTemp );
      }
      else
      {
         _itemArrayPut( aNew, i, temp );
      }

      _itemRelease( temp );

   }

   return ( aNew );

}
```

Files Library is CLIPPER.LIB, header file is Item.api.

See Also _itemArrayNew(), _itemArrayPut(), _itemRelease()

_itemArrayNew()

Create a new item as an array

C Prototype

```
#include "item.api"
ITEM _itemArrayNew(
                        USHORT uiSize
                    )
```

Arguments

uiSize is the number of elements with which to initialize the array. _itemArrayNew() will accept a zero-element array. This means that arrays created will have no elements ({}).

Returns

A new item of type array with a _itemSize() of *uiSize*.

Description

You can use the _itemArrayNew() function to create a new CA-Clipper-level array. *uiSize* will accept zero and will create a zero-length array.

Note: When creating a new item reference, remember to release the item reference with _itemRelease() after returning it to CA-Clipper or when it is no longer needed.

Examples

```
/*
* MYACLONE()
* ----------
* myAClone( <aArray1> ) --> aNewArray
*/

HIDE ITEM near _xAClone( ITEM aSrc );

CLIPPER ARRAYCLONE( void )
{
   ITEM aSource;          // Source array
   ITEM aCloned;          // New array

   /* Do parameters check */

   if ( PCOUNT != 1 )
   {
      return;
   }

   /* Get source array as ITEM */

   aSource = _itemParam( 1 );

   /* Clone source array  */

   aCloned = _xAClone( aSource );

   /* Return clone, release clone and source */

   _itemReturn ( aCloned );

   _itemRelease( aSource );
   _itemRelease( aCloned );

   return;
}
```

```
HIDE ITEM near _xAClone( ITEM aSrc )
{
    ITEM   temp;
    ITEM   aTemp;
    ITEM   aNew;

    USHORT i;
    USHORT nLen;

    // If item not an array, return NIL
    if ( !( _itemType( aSrc ) == ARRAY ) )
       return ( _itemNew( NULL ) );

    // Get number of elements, create new array
    nLen = _itemSize( aSrc );
    aNew = _itemArrayNew( nLen );

    // For each element... get source & copy
    for ( i = 1; i <= nLen; i++ )
    {
       temp = _itemArrayGet( aSrc, i );

       // If it's an array, copy via recursion
       if ( _itemType( temp ) == ARRAY )
       {
          aTemp = _xAClone( temp );
          _itemArrayPut( aNew, i, aTemp );
          _itemRelease( aTemp );
       }
       else
       {
          _itemArrayPut( aNew, i, temp );
       }

       _itemRelease( temp );

    }

    return ( aNew );

}
```

Files Library is CLIPPER.LIB, header file is Item.api.

See Also _itemArrayGet(), _itemArrayPut(), _itemRelease(), _itemSize(), _itemType()

_itemArrayPut()

Place an item into a CA-Clipper-level array element

C Prototype

```
#include "item.api"
ITEM _itemArrayPut(
                        ITEM itmArr,
                        USHORT uiElem,
                        ITEM itmAdd
                    )
```

Arguments

itmArr is the array into which you want to place a new item.

uiElem is the element number of the array *itmArr* for the new item. Remember that CA-Clipper arrays element numbers begin at one, not zero-like C arrays. A zero value in this parameter will not be accepted.

itmAdd is the new item to place into *itmArr* at element *uiElem.*

Returns

_itemArrayPut() always returns *itmArr*.

Description

The function _itemArrayPut() allows you to place values (as items) into CA-Clipper-level arrays. Note that if the root array was passed to your Extend routine as a parameter, changes made through _itemArrayPut() *will be reflected* at the CA-Clipper level. This is because all arrays in CA-Clipper are passed by reference rather than by value.

Examples

```
/*
* MYACLONE()
* ----------
* myAClone( <aArray1> ) --> aNewArray
*/

HIDE ITEM near _xAClone( ITEM aSrc );

CLIPPER ARRAYCLONE( void )
{
   ITEM aSource;          // Source array
   ITEM aCloned;          // New array

   /* Do parameters check */

   if ( PCOUNT != 1 )
   {
      return;
   }

   /* Get source array as ITEM */

   aSource = _itemParam( 1 );

   /* Clone source array  */

   aCloned = _xAClone( aSource );

   /* Return clone, release clone and source */

   _itemReturn ( aCloned );

   _itemRelease( aSource );
   _itemRelease( aCloned );

   return;
}
```

```
HIDE ITEM near _xAClone( ITEM aSrc )
{
   ITEM   temp;
   ITEM   aTemp;
   ITEM   aNew;

   USHORT i;
   USHORT nLen;

   // If item not an array, return NIL
   if ( !( _itemType( aSrc ) == ARRAY ) )
      return ( _itemNew( NULL ) );

   // Get number of elements, create new array
   nLen = _itemSize( aSrc );
   aNew = _itemArrayNew( nLen );

   // For each element... get source & copy
   for ( i = 1; i <= nLen; i++ )
   {
      temp = _itemArrayGet( aSrc, i );

      // If it's an array, copy via recursion
      if ( _itemType( temp ) == ARRAY )
      {
         aTemp = _xAClone( temp );
         _itemArrayPut( aNew, i, aTemp );
         _itemRelease( aTemp );
      }
      else
      {
         _itemArrayPut( aNew, i, temp );
      }

      _itemRelease( temp );

   }

   return ( aNew );

}
```

Files Library is CLIPPER.LIB, header file is Item.api.

See Also _itemArrayGet(), _itemArrayNew()

_itemCopyC()

Copy a character item's value into a C buffer

C Prototype

```
#include "item.api"
USHORT _itemCopyC(
                        ITEM itmChar,
                        BYTEP fpBuffer,
                        USHORT uiSize
                    )
```

Arguments

itmChar is the item from which to get a character value.

fpBuffer is the preallocated buffer to copy the item's value into.

uiSize is *fpBuffer*'s size (one-based). If this parameter is zero, _itemCopyC() will assume that the buffer is of adequate size to hold the string.

Returns

The number of characters copied into the character buffer.

Description

The _itemCopyC() function copies an item's character value into a buffer of *uiSize* length. This is useful when you need to be in control of allocation and deallocation of the buffer used for the item's value.

Warning! *You are responsible for allocating and freeing a buffer of adequate size for the item. There is* ***no need*** *to call _itemFreeC() when using _itemCopyC() as with _itemGetC().*

Examples

```
/*
*
* CharCount( cString, cChar )
*
* Count occurrences of a single character
* in a CA-Clipper string.
*
*/

CLIPPER CharCount( void )
{
   USHORT uiChars = 0;
   USHORT uiLen;
   USHORT i;
   HANDLE vmhString;
```

```
BYTEP  cStringP;
BYTE   cFindMe;

ITEM   itemString, itemFindMe, itemRet;


if (PCOUNT != 2)
{
   _ret();              // NOTE: Withhold service
   return;              // Early return!
}

itemRet    = _itemPutNL( NULL, 0 );
itemString = _itemParam( 1 );
itemFindMe = _itemParam( 2 );

if ( (_itemType( itemString ) == CHARACTER) &&
      (_itemType( itemFindMe ) == CHARACTER) )
{
   _itemCopyC( itemFindMe, &cFindMe, 1 );

   vmhString = _xvalloc( _itemSize( itemString ), NULL );
   cStringP = _xvlock( vmhString );

   uiLen = _itemCopyC( itemString, cStringP, NULL );

   for( i = 0; i < uiLen; i++ )
   {
      if ( cStringP[i] == cFindMe )
         uiChars++;
   }

   _xvunlock( vmhString );
   _xvfree( vmhString );

   itemRet = _itemPutNL( itemRet, (long)uiChars );

}

_itemReturn( itemRet );

_itemRelease( itemRet );
_itemRelease( itemString );
_itemRelease( itemFindMe );

return;
}
```

Files Library is CLIPPER.LIB, header file is Item.api.

See Also _itemGetC()

_itemFreeC()

Frees a character string allocated by the Item API

C Prototype

```
#include "item.api"
BOOL _itemFreeC(
                    BYTEP fpItemVal
                )
```

Arguments

fpItemVal is a pointer returned by the Item API in response to an _itemGetC() call.

Returns

True if the requested pointer was released.

Description

When _itemGetC() is asked to provide a character string from a character item, it allocates memory for the string from one of several places. You need to call the _itemFreeC() function to release the memory once the string is no longer necessary.

Warning! *Do* ***not*** *call _itemFreeC() unless the character string was retrieved through _itemGetC(). Calling _itemFreeC() with a NULL pointer, a pointer already freed, or a string not retrieved by _itemGetC() will result in a VM integrity fault or other internal error.*

Examples

```
/*
*
* CharCount( cString, cChar )
*
* Count occurrences of a single character
* in a CA-Clipper string.
*
*/

CLIPPER CharCount( void )
{
   USHORT uiChars = 0;
   USHORT uiLen;
   USHORT i;

   BYTEP  cStringP;
   BYTEP  cFindMeP;

   ITEM   itemString, itemFindMe, itemRet;
```

```
if (PCOUNT != 2)
{
   _ret();                    // NOTE: Withhold service
   return;                    // Early return!
}

itemRet    = _itemPutNL( NULL, 0 );
itemString = _itemParam( 1 );
itemFindMe = _itemParam( 2 );

if ( (_itemType( itemString ) == CHARACTER) &&
     (_itemType( itemFindMe ) == CHARACTER) )
{
   if (_itemSize( itemFindMe ) == 1)
   {
      cFindMeP = _itemGetC( itemFindMe );
      cStringP = _itemGetC( itemString );

      uiLen = _itemSize( itemString );

      for( i = 0; i < uiLen; i++ )
      {
         if ( cStringP[i] == *cFindMeP )
            uiChars++;
      }

      _itemFreeC( cFindMeP );
      _itemFreeC( cStringP );


      itemRet = _itemPutNL( itemRet, (long)uiChars );
   }
}

_itemReturn( itemRet );

_itemRelease( itemRet );
_itemRelease( itemString );
_itemRelease( itemFindMe );

return;
}
```

Files Library is CLIPPER.LIB, header file is Item.api.

See Also _itemGetC()

_itemGetC()

Retrieve a character value from an item

C Prototype

```
#include "item.api"
BYTEP _itemGetC(
                    ITEM itmChar
                  )
```

Arguments

itmChar is the item from which to retrieve a character value.

Returns

A pointer to a copy of the string contained within *itmChar*.

Description

_itemGetC() returns a pointer to a copy of the character string or memo contained within *itmChar*. If *itmChar* is not a character value (type memo or character), the results will be unpredictable.

Warning! *The pointer returned must be freed through an _itemFreeC() call.*

Examples

```
/*
*
* CharCount( cString, cChar )
*
* Count occurrences of a single character
* in a CA-Clipper string.
*
*/

CLIPPER CharCount( void )
{
   USHORT uiChars = 0;
   USHORT uiLen;
   USHORT i;

   BYTEP  cStringP;
   BYTEP  cFindMeP;

   ITEM   itemString, itemFindMe, itemRet;
```

```
   if (PCOUNT != 2)
   {
      _ret();                 // NOTE: Withhold service
      return;                 // Early return!
   }

   itemRet    = _itemPutNL( NULL, 0 );
   itemString = _itemParam( 1 );
   itemFindMe = _itemParam( 2 );

   if ( (_itemType( itemString ) == CHARACTER) &&
        (_itemType( itemFindMe ) == CHARACTER) )
   {
      if (_itemSize( itemFindMe ) == 1)
      {
         cFindMeP = _itemGetC( itemFindMe );
         cStringP = _itemGetC( itemString );

         uiLen = _itemSize( itemString );

         for( i = 0; i < uiLen; i++ )
         {
               if ( cStringP[i] == *cFindMeP )
                  uiChars++;
         }

         _itemFreeC( cFindMeP );
         _itemFreeC( cStringP );


         itemRet = _itemPutNL( itemRet, (long)uiChars );
      }
   }

   _itemReturn( itemRet );

   _itemRelease( itemRet );
   _itemRelease( itemString );
   _itemRelease( itemFindMe );

   return;
}
```

Files Library is CLIPPER.LIB, header file is Item.api.

See Also _itemFreeC(), _itemPutC(), _itemPutCL()

_itemGetDS()

Retrieve a date as a character string

C Prototype

```
#include "item.api"
BYTEP _itemGetDS(
                    ITEM itmDate,
                    BYTEP fpBuffer
                  )
```

Arguments

itmDate is the item from which to retrieve a date string.

fpBuffer is the 8 character buffer to place the date string within.

Returns

A zero terminated string of the format *yyyymmdd* representing the date value contained within the item passed.

Description

_itemGetDS() allows you to retrieve the value of a CA-Clipper-level date as a character string. The returned string is a copy of the date, so any changes to the string cannot be reflected to the CA-Clipper level if the item is a parameter.

Note: The date string returned is zero-terminated.

Warning! *You are responsible for allocating and freeing the buffer where the date string is placed. Note that the recommended destination memory address is the stack, since values returned from _itemGetDS() will always be 8 characters long.*

Examples

```
/*
*
* Quarter( dDate ) -> nQuarter
*
*
* Given a date, determines what quarter it lies in
*
*/
```

```
CLIPPER Quarter( void )
{
   ITEM itemDate, itemQuarter;
   BYTE sDate[8];              // Date buffer: YYYYMMDD\0

   if ( PCOUNT == 0 )
   {
      _ret();                  // Withhold service!
      return;
   }


   itemQuarter = _itemPutNL( NULL, 0 );
   itemDate = _itemParam( 1 );

   if ( _itemType( itemDate ) == DATE )
   {
      _itemGetDS( itemDate, sDate );

      sDate[6] = NULL;

      if (sDate[4] == '1')
      {
         itemQuarter = _itemPutNL( itemQuarter, 4 );
      }
      else
      {
         switch ( sDate[5] )
         {
            case '1':
            case '2':
            case '3':
               itemQuarter = _itemPutNL( itemQuarter, 1 );
               break;

            case '4':
            case '5':
            case '6':
               itemQuarter = _itemPutNL( itemQuarter, 2 );
               break;

            default:
               itemQuarter = _itemPutNL( itemQuarter, 3 );

         }
      }
   }

   _itemReturn( itemQuarter );
   _itemRelease( itemDate );
   _itemRelease( itemQuarter );

   return;
}
```

Files Library is CLIPPER.LIB, header file is Item.api.

See Also _itemPutDS()

_itemGetL()

Retrieve a logical value from an item

C Prototype

```
#include "item.api"
BOOL _itemGetL(
                  ITEM itmLogic
               )
```

Arguments

itmLogic is the item from which to retrieve a logical value.

Returns

The boolean value true (.T.) or false (.F.).

Description

The _itemGetL() function is used to retrieve a logical value from the item passed, and return it as a boolean value. If *itmLogic* is not a CA-Clipper-level logical type (LOGICAL), the results will be unpredictable.

Examples

```
/*
*  YesNo( lExpr ) -> cValue
*
*  Given a logical expression, returns the
*  string "Yes" if true, and "No " if false.
*
*/

CLIPPER YesNo( void )
{
   ITEM itemParam, itemCRet;

   // Warning: DGROUP!

   itemCRet = _itemPutC( NULL, "No " );

   if (PCOUNT > 0)
   {
      itemParam = _itemParam( 1 );
      if (_itemGetL( itemParam ))
         itemCRet = _itemPutC( itemCRet, "Yes");

      _itemRelease( itemParam );
   }

   _itemRelease( _itemReturn( itemCRet ) );
   return;
}
```

Files Library is CLIPPER.LIB, header file is Item.api.

See Also _itemPutL()

_itemGetND()

Retrieve a numeric value as double-precision type

C Prototype

```
#include "item.api"
XDOUBLE _itemGetND(
                        ITEM itmNumber
                    )
```

Arguments

itmNumber is the item from which to retrieve a double-precision type value.

Returns

The numeric value in *itmNumber* as double-precision type.

Description

The _itemGetND() function returns the CA-Clipper-level numeric stored in *itmNumber* as a floating point C double-precision value. Note that the use of XDOUBLE ensures floating point routines will not be linked in unless they become necessary.

Files

Library is CLIPPER.LIB, header file is Item.api.

See Also

_itemGetNL(), _itemPutND()

_itemGetNL()

Retrieve a numeric long value from an item

C Prototype

```
#include "item.api"
long _itemGetNL(
                    ITEM itmNumber
                )
```

Arguments

itmNumber is the item from which to retrieve a long value.

Returns

The numeric contained within the item passed as a C-style long.

Description

The _itemGetNL() function is used to retrieve a numeric long from an item. If the value contained within *itmNumber* is not a numeric, or is a numeric with the DOUBLE flag set (see _itemType()), then the results of calling this function will be unpredictable.

Examples

```
#include "clipdefs.h"
#include "extend.api"
#include "item.api"
#include "fm.api"

HIDE ITEM near _xMakeSub( USHORTP apSize, USHORT dims );

/*
*
* MYARRAY()
*
* Just like ARRAY()!
*/
CLIPPER MYARRAY(void)

{
   ITEM       arrayP, elementsP;
   SHORT      i;
   BOOL       check;
   USHORT     pcount = PCOUNT;
   USHORTP    apSize;
```

```
    /* Check parameters - must be at least one and all */
    /* NUMERIC */

    if ( pcount )
    {
/* CAUTION: _xgrab() will cause VM IE if there isn't  */
/* enough  memory */
       apSize = (USHORTP)_xgrab( pcount * sizeof( USHORT ) );
       /* store the parameters into apSize[] in */
       /* reverse order. */

       for (i = 0; i < pcount; i++)
       {
          elementsP = _itemParam( pcount-i );
          check = _itemType( elementsP ) == NUMERIC &&
                ( apSize[i] =
                (USHORT)_itemGetNL( elementsP ) ) <= 4096;
          _itemRelease( elementsP );
          if (!check)
             break;
       }
       /* If all of the parameters check out ...*/

       if( check )
       {
          arrayP = _xMakeSub( apSize, pcount );
          _itemReturn ( arrayP );
          _itemRelease( arrayP );
       }
       else
       {
    _itemRelease( _itemReturn( _itemNew( NULL ) ) ); }
       _xfree( apSize );
    }
}

HIDE ITEM near _xMakeSub( USHORTP apSize, USHORT dims )
{
    ITEM   arrayP;
    ITEM   tempP;
    USHORT i;
    arrayP = _itemArrayNew( apSize[ --dims ] );
    if ( dims )
    {
       for (i = 1; i <= apSize[ dims ]; i++)
       {
          tempP = _xMakeSub( apSize, dims );
          _itemArrayPut( arrayP, i, tempP );
          _itemRelease( tempP );
       }
    }
    return (arrayP);
}
```

Files Library is CLIPPER.LIB, header file is Item.api.

See Also _itemGetND(), _itemPutNL(), _itemType()

_itemNew()

Creates a NIL or empty item

C Prototype

```
#include "item.api"
ITEM _itemNew(
                    ITEM itmNull
                )
```

Arguments

itmNull is an item that should always be NULL.

Returns

A new item of undefined (NIL) type.

Description

The _itemNew() function allows you to create a NIL item to use as a return value.

Note: When creating a new item reference, remember to release the item reference with _itemRelease() after returning it to CA-Clipper or when it is no longer needed.

Examples

```
#include "clipdefs.h"
#include "extend.api"
#include "item.api"
#include "fm.api"

HIDE ITEM near _xMakeSub( USHORTP apSize, USHORT dims );

/*
*
* MYARRAY()
*
* Just like ARRAY()!
*/
CLIPPER MYARRAY(void)

{
   ITEM       arrayP, elementsP;
   SHORT      i;
   BOOL       check;
   USHORT     pcount = PCOUNT;
   USHORTP    apSize;
```

```
   /* Check parameters - must be at least one and all */
   /* NUMERIC */

   if ( pcount )
   {
/* CAUTION: _xgrab() will cause VM IE if there isn't */
/* enough  memory */
      apSize = (USHORTP)_xgrab( pcount * sizeof( USHORT ) );
      /* store the parameters into apSize[] in */
      /* reverse order. */

      for (i = 0; i < pcount; i++)
      {
         elementsP = _itemParam( pcount-i );
         check = _itemType( elementsP ) == NUMERIC &&
               ( apSize[i] =
               (USHORT)_itemGetNL( elementsP ) ) <= 4096;
         _itemRelease( elementsP );
         if (!check)
            break;
      }
      /* If all of the parameters check out ...*/

      if( check )
      {
         arrayP = _xMakeSub( apSize, pcount );
         _itemReturn ( arrayP );
         _itemRelease( arrayP );
      }
      else
      {
   _itemRelease( _itemReturn( _itemNew( NULL ) ) ); }
      _xfree( apSize );
   }
}

HIDE ITEM near _xMakeSub( USHORTP apSize, USHORT dims )
{
   ITEM   arrayP;
   ITEM   tempP;
   USHORT i;
   arrayP = _itemArrayNew( apSize[ --dims ] );
   if ( dims )
   {
      for (i = 1; i <= apSize[ dims ]; i++)
      {
         tempP = _xMakeSub( apSize, dims );
         _itemArrayPut( arrayP, i, tempP );
         _itemRelease( tempP );
      }
   }
   return (arrayP);
}
```

Files Library is CLIPPER.LIB, header file is Item.api.

See Also _itemArrayNew(), _itemRelease()

_itemParam()

Retrieve a parameter passed to an Extend routine from CA-Clipper

C Prototype

```
#include "item.api"
ITEM _itemParam(
                    USHORT uiParamNo
                )
```

Arguments

uiParamNo is the parameter number to retrieve.

Returns

The requested parameter number or an empty (NIL) item if the parameter number does not exist.

Description

The _itemParam() function is used to retrieve parameters passed from CA-Clipper to your Extend routine. _itemParam() creates a new item and returns that item with the parameter number requested.

Note: CA-Clipper parameter numbers begin at one, not zero.

Note: _itemParam() creates a new item reference. Remember to release the item reference with _itemRelease() after returning it to CA-Clipper or when it is no longer needed.

Examples

```
#include "clipdefs.h"
#include "extend.api"
#include "item.api"
#include "fm.api"

HIDE ITEM near _xMakeSub( USHORTP apSize, USHORT dims );

/*
*
* MYARRAY()
*
* Just like ARRAY()!
*/
CLIPPER MYARRAY(void)

{
   ITEM       arrayP, elementsP;
   SHORT      i;
   BOOL       check;
   USHORT     pcount = PCOUNT;
   USHORTP    apSize;
```

```
   /* Check parameters - must be at least one and all */
   /* NUMERIC */
   if ( pcount )
   {
/* CAUTION: _xgrab() will cause VM IE if there isn't */
/* enough  memory */
      apSize = (USHORTP)_xgrab( pcount * sizeof( USHORT ) );
      /* store the parameters into apSize[] in */
      /* reverse order. */

      for (i = 0; i < pcount; i++)
      {
         elementsP = _itemParam( pcount-i );
         check = _itemType( elementsP ) == NUMERIC &&
               ( apSize[i] =
               (USHORT)_itemGetNL( elementsP ) ) <= 4096;
         _itemRelease( elementsP );
         if (!check)
            break;
      }
      /* If all of the parameters check out ...*/

      if( check )
      {
         arrayP = _xMakeSub( apSize, pcount );
         _itemReturn ( arrayP );
         _itemRelease( arrayP );
      }
      else
      {
   _itemRelease( _itemReturn( _itemNew( NULL ) ) ); }
      _xfree( apSize );
   }
}

HIDE ITEM near _xMakeSub( USHORTP apSize, USHORT dims )
{
   ITEM   arrayP;
   ITEM   tempP;
   USHORT i;
   arrayP = _itemArrayNew( apSize[ --dims ] );
   if ( dims )
   {
      for (i = 1; i <= apSize[ dims ]; i++)
      {
         tempP = _xMakeSub( apSize, dims );
         _itemArrayPut( arrayP, i, tempP );
         _itemRelease( tempP );
      }
   }
   return (arrayP);
}
```

Files Library is CLIPPER.LIB, header file is Item.api.

See Also _itemRelease(), _itemReturn()

_itemPutC()

Place a zero-terminated character value into an item

C Prototype

```
#include "item.api"
ITEM _itemPutC(
                  ITEM itmChar,
                  BYTEP fpChars
               )
```

Arguments

itmChar is the item into which you want to place the string. If *itmChar* is NULL, _itemPutC() will create a new item.

fpChars is the zero-terminated string to place into the item.

Returns

A new item reference containing the zero-terminated string passed.

Description

The _itemPutC() function is used to associate a zero-terminated string with an item. Any previous value in *itmChar* will be destroyed.

The data located at *fpChars* is copied into a new space associated with the item that is returned. It is the programmer's responsibility to free the allocation of *fpChars* in whatever manner it was allocated.

Note: _itemPutC() must copy the intended string from your buffer at *fpChars*. If *fpChars* is unusually large, _itemPutC() may cause a VM failure due to insufficient real memory to accomplish a large copy. Typically, this error would manifest itself as an internal error 5302.

Caution! *Do not assume that the address in* ***itmChar*** *will be the same address returned by _itemPutC().*

Examples

```
/*
*  YesNo( lExpr ) -> cValue
*
*  Given a logical expression, returns the
*  string "Yes" if true, and "No " if false.
*
*/

CLIPPER YesNo( void )
{
   ITEM itemParam, itemCRet;

   // Warning: DGROUP!

   itemCRet = _itemPutC( NULL, "No " );

   if (PCOUNT > 0)
   {
      itemParam = _itemParam( 1 );
      if (_itemGetL( itemParam ))
         itemCRet = _itemPutC( itemCRet, "Yes");

      _itemRelease( itemParam );
   }

   _itemRelease( _itemReturn( itemCRet ) );
   return;
}
```

Files Library is CLIPPER.LIB, header file is Item.api.

See Also _itemGetC(), _itemPutCL()

_itemPutCL()

Place a character value into an item, ignoring null terminators

C Prototype

```
#include "item.api"
ITEM _itemPutCL(
                    ITEM itmChar,
                    BYTEP fpChars,
                    USHORT uiLen
                  )
```

Arguments

itmChar is the item into which you want to place the string. If *itmChar* is NULL, _itemPutCL() will create a new item.

fpChars is the zero-terminated string to place into the item.

uiLen is the number of bytes to place into the item.

Returns

A new item reference containing the string passed.

Description

The _itemPutCL() function is used to associate any string with an item, ignoring embedded nulls. Any previous value in *itmChar* will be destroyed.

The data located at *fpChars* is copied into a new space associated with the item that is returned. It is the programmer's responsibility to free the allocation of *fpChars* in whatever manner it was allocated.

Note: _itemPutCL() must copy the intended string from your buffer at *fpChars*. If *fpChars* is unusually large, _itemPutCL() may cause a VM failure due to insufficient real memory to accomplish a large copy. Typically, this error would manifest itself as an internal error 5302.

Caution! *Do not assume that the address in* ***itmChar*** *will be the same address returned by _itemPutCL().*

Examples

```
/*
*
* BootSector( [nDrive] ) -> cBootSecBuff
*
* Read the boot sector from drive nDrive,
* if there is no drive specified, BootSector()
* reads drive A.
*
* (0=A, 1=B, 2=C, etc...)
*
* Warning: Does not check for errors.
*          Needs dos.h & LLIBCA
*/

CLIPPER BootSector( void )
{
    ITEM   itemDriveN, itemBuff;
    BYTEP  bufferP;
    HANDLE vmBuffer;
    USHORT uiDriveN = 0;          // Default: A
    USHORT uiLen = 1024;

    union  REGS  preCallRx, postCallRx;
    struct SREGS theSegs;

    if ( PCOUNT > 0 )
    {
       itemDriveN = _itemParam( 1 );

       if ( _itemType( itemDriveN ) | NUMERIC )
          uiDriveN = (USHORT) _itemGetNL( itemDriveN );

       _itemRelease( itemDriveN );
    }

    vmBuffer = _xvalloc( uiLen );
    bufferP  = _xvlock( vmBuffer );

    segread( &theSegs );

    theSegs.es     = FP_SEG( bufferP );
    preCallRx.x.bx = FP_OFF( bufferP );
    preCallRx.x.ax = 0x0201;            // BIOS 02 / 1 sector
    preCallRx.x.cx = 1;                 // Track 0, Sector 1
    preCallRx.x.dx = uiDriveN;          // Side 0, Drive uiDriveN

    int86x( 0x13, &preCallRx, &postCallRx, &theSegs );

    if ( postCallRx.x.cflag )
       ERRMSG( "\r\nERROR on read!" );

    itemBuff = _itemPutCL( NULL, bufferP, uiLen );

    _xvunlock( vmBuffer );
    _xvfree( vmBuffer );

    _itemRelease( _itemReturn( itemBuff ) );

    return;
}
```

Files Library is CLIPPER.LIB, header file is Item.api.

See Also _itemGetC(), _itemPutC()

_itemPutDS()

Place a date string into an item

C Prototype

```
#include "item.api"
ITEM _itemPutDS(
                    ITEM itmDate,
                    BYTEP fpDate
                  )
```

Arguments

itmDate is the item into which you want to place the date string. If this item is NULL, a new item will be created.

fpDate is a zero-terminated string representing the date to assign to the new item. The format of *fpDate* must be *yyyymmdd*.

Returns

A new date item reference containing the date passed.

Description

The _itemPutDS() function is used to place a zero-terminated string date into an item. If the date string is malformed, the new date item will be set to a NULL date.

Caution! *Do not assume that the address in* ***itmDate*** *will be the same address returned by _itemPutDS().*

Files

Library is CLIPPER.LIB, header file is Item.api.

See Also

_itemGetDS()

_itemPutL()

Place a logical value into an item

C Prototype

```
#include "item.api"
ITEM _itemPutL(
                    ITEM itmLogic,
                    BOOL bValue
                )
```

Arguments

itmLogic is the item in which you want to place the logical value. If *itmLogic* is NULL, _itemPutL() will create a new item.

bValue is the logical (boolean) value to place into the item. As with C, zero represents false (.F.), and nonzeros represent true (.T.).

Returns

A new item reference containing the logical value passed.

Description

The _itemPutL() function allows you to place a logical value into an item. Note that if the first parameter passed to _itemPutL() is NULL, _itemPutL() will create a new item and place the logical value *bValue* in that item. The data contained in *itmLogic* is released to SVOS for garbage collection.

Caution! *Do not assume that the address in* ***itmLogic*** *will be the same address returned by _itemPutL().*

Examples

```
/*
*
* IsDouble( nNumber ) -> lIsIt
*
*
* Returns true if the number passed is
* internally represented as a double.
*/

CLIPPER IsDouble( void )
{
    ITEM itemRet, itemNum;

    itemRet = _itemPutL( NULL, FALSE );

    if (PCOUNT > 0)
    {
       itemNum = _itemParam( 1 );

       if (_itemType( itemNum ) & DOUBLE)
          itemRet = _itemPutL( itemRet, TRUE );

       _itemRelease( itemNum );
    }

    _itemRelease( _itemReturn( itemRet ) );

    return;
}
```

Files Library is CLIPPER.LIB, header file is Item.api.

See Also _itemGetL()

_itemPutND()

Places a double-precision number into an item

C Prototype

```
#include "item.api"
ITEM _itemPutND(
                    ITEM itmNumber,
                    XDOUBLE dNum
                )
```

Arguments

itmNumber is the item into which you want to place the number. If this item is NULL, _itemPutND() will create a new item.

dNum is the double-precision value to assign to the item.

Returns

A new double-precision value item reference containing the double-precision value passed.

Description

The _itemPutND() function is used to place a numeric double-precision value into an item. You should use a double-precision value to represent a number when you know that the number falls within the range of 1.7E+308 to 1.7E-308.

Caution! *Do not assume that the address in* ***itmNumber*** *will be the same address returned by _itemPutND().*

Files

Library is CLIPPER.LIB, header file is Item.api.

See Also

_itemGetND(), _itemPutNL()

_itemPutNL()

Place a long number into an item

C Prototype

```
#include "item.api"
ITEM _itemPutNL(
                    ITEM itmNumber,
                    long lNum
                  )
```

Arguments

itmNumber is the item into which you want to place the number. If this item is NULL, _itemPutNL() will create a new item.

lNum is the signed long integer number to assign to the item.

Returns

A new signed long integer item reference containing the numeric passed.

Description

The _itemPutNL() function is used to place a numeric long integer into an item. You should use a long integer value to represent a number when you know that the number will fall within the range of -2,147,483,648 to 2,147,483,647.

Caution! *Do not assume that the address in* ***itmNumber*** *will be the same address returned by _itemPutNL().*

Examples

```
/*
*
* CharCount( cString, cChar )
*
* Count occurrences of a single character
* in a CA-Clipper string.
*
*/

CLIPPER CharCount( void )
{
   USHORT uiChars = 0;
   USHORT uiLen;
   USHORT i;

   BYTEP  cStringP;
   BYTEP  cFindMeP;
```

```
ITEM   itemString, itemFindMe, itemRet;

if (PCOUNT != 2)
{
   _ret();           // NOTE: Withhold service
   return;           // Early return!
}

itemRet    = _itemPutNL( NULL, 0 );
itemString = _itemParam( 1 );
itemFindMe = _itemParam( 2 );

if ( (_itemType( itemString ) == CHARACTER) &&
      (_itemType( itemFindMe ) == CHARACTER) )
{
   if (_itemSize( itemFindMe ) == 1)
   {
      cFindMeP = _itemGetC( itemFindMe );
      cStringP = _itemGetC( itemString );

      uiLen = _itemSize( itemString );

      for( i = 0; i < uiLen; i++ )
      {
         if ( cStringP[i] == *cFindMeP )
            uiChars++;
      }

      _itemFreeC( cFindMeP );
      _itemFreeC( cStringP );


      itemRet = _itemPutNL( itemRet, (long)uiChars );
   }
}

_itemReturn( itemRet );

_itemRelease( itemRet );
_itemRelease( itemString );
_itemRelease( itemFindMe );

return;
}
```

Files Library is CLIPPER.LIB, header file is Item.api.

See Also _itemGetNL(), _itemPutND()

_itemRelease()

Make an item available for garbage collection

C Prototype

```
#include "item.api"
BOOL _itemRelease(
                  ITEM itmRef
                 )
```

Arguments

itmRef is the item to release.

Returns

True (.T.) if the item passed was successfully released.

Description

The _itemRelease() function drops your Extend routine's reference to a CA-Clipper-level item. If your Extend routine was the only reference to that item, then the item becomes garbage and a candidate for collection. However, if the item is referred to elsewhere, then it will not be collected until its last reference is released.

Warning! *It is vitally important that items be released once you no longer have need for them. Failure to do so may cause a CA-Clipper stack fault or memory errors.*

Examples

```
/*
*
* CharCount( cString, cChar )
*
* Count occurrences of a single character
* in a CA-Clipper string.
*
*/

CLIPPER CharCount( void )
{
   USHORT uiChars = 0;
   USHORT uiLen;
   USHORT i;

   BYTEP  cStringP;
   BYTEP  cFindMeP;
```

```
ITEM   itemString, itemFindMe, itemRet;

if (PCOUNT != 2)
{
   _ret();           // NOTE: Withhold service
   return;           // Early return!
}

itemRet    = _itemPutNL( NULL, 0 );
itemString = _itemParam( 1 );
itemFindMe = _itemParam( 2 );

if ( (_itemType( itemString ) == CHARACTER) &&
     (_itemType( itemFindMe ) == CHARACTER) )
{
   if (_itemSize( itemFindMe ) == 1)
   {
      cFindMeP = _itemGetC( itemFindMe );
      cStringP = _itemGetC( itemString );

      uiLen = _itemSize( itemString );

      for( i = 0; i < uiLen; i++ )
      {
         if ( cStringP[i] == *cFindMeP )
            uiChars++;
      }

      _itemFreeC( cFindMeP );
      _itemFreeC( cStringP );


      itemRet = _itemPutNL( itemRet, (long)uiChars );
   }
}

_itemReturn( itemRet );

_itemRelease( itemRet );
_itemRelease( itemString );
_itemRelease( itemFindMe );

return;
}
```

Files Library is CLIPPER.LIB, header file is Item.api.

See Also _evalPutParam(), _itemArrayGet(), _itemArrayNew(), _itemNew(), _itemParam(), _itemReturn()

_itemReturn()

Return an item to CA-Clipper

C Prototype

```
#include "item.api"
ITEM _itemReturn(
                    ITEM itmRet
                  )
```

Arguments

itmRet is the item you wish to return to CA-Clipper.

Returns

The same ITEM that was posted as the return value.

Description

The _itemReturn() function is used to send an item back to CA-Clipper in the form of a return value.

Note: Once an item is returned to CA-Clipper via _itemReturn(), it is considered as "referenced" by the CA-Clipper runtime system, and therefore, is not a candidate for garbage collection. However, your C function *must still call _itemRelease()* after posting an item for return so that once the CA-Clipper variable which received the return value goes out of scope, it **does** get properly collected.

Examples

```
/*
*
* CharCount( cString, cChar )
*
* Count occurrences of a single character
* in a CA-Clipper string.
*
*/

CLIPPER CharCount( void )
{
   USHORT uiChars = 0;
   USHORT uiLen;
   USHORT i;

   BYTEP  cStringP;
   BYTEP  cFindMeP;
```

```
ITEM   itemString, itemFindMe, itemRet;

if (PCOUNT != 2)
{
   _ret();           // NOTE: Withhold service
   return;           // Early return!
}

itemRet    = _itemPutNL( NULL, 0 );
itemString = _itemParam( 1 );
itemFindMe = _itemParam( 2 );

if ( (_itemType( itemString ) == CHARACTER) &&
      (_itemType( itemFindMe ) == CHARACTER) )
{
   if (_itemSize( itemFindMe ) == 1)

      cFindMeP = _itemGetC( itemFindMe );
      cStringP = _itemGetC( itemString );

      uiLen = _itemSize( itemString );

      for( i = 0; i < uiLen; i++ )
      {
         if ( cStringP[i] == *cFindMeP )
            uiChars++;
      }

      _itemFreeC( cFindMeP );
      _itemFreeC( cStringP );


      itemRet = _itemPutNL( itemRet, (long)uiChars );
   }
}

_itemReturn( itemRet );

_itemRelease( itemRet );
_itemRelease( itemString );
_itemRelease( itemFindMe );

return;
}
```

Files Library is CLIPPER.LIB, header file is Item.api.

See Also _itemParam(), _itemRelease()

_itemSize()

Determine an item's size

C Prototype

```
#include "item.api"
USHORT _itemSize(
                    ITEM itmRef
                  )
```

Arguments

itmRef is the item whose size you want to determine.

Returns

_itemSize() returns an unsigned short integer value indicating the size of the item referenced by *itmRef*.

Description

The _itemSize() function returns the storage size required for a particular item. For each CA-Clipper data type, length is determined according to the table below:

Return Size for Each Item Type

Item Type	Size Returned
ARRAY	Number of elements in array.
CHARACTER or MEMO	Storage length of the string.
UNDEF	Always returns zero.

Note: Size is determined from a one base, not zero, as are most programming structures in C. Thus, an array that has an _itemSize() of 42 actually contains 42 elements, and not 43 as one might expect. An array of length zero, however, may exist.

Examples

```
/*
*
* CharCount( cString, cChar )
*
* Count occurrences of a single character
* in a CA-Clipper string.
*
*/

CLIPPER CharCount( void )
{
   USHORT uiChars = 0;
   USHORT uiLen;
   USHORT i;
   HANDLE vmhString;

   BYTEP  cStringP;
   BYTE   cFindMe;

   ITEM   itemString, itemFindMe, itemRet;


   if (PCOUNT != 2)
   {
      _ret();           // NOTE: Withhold service
      return;           // Early return!
   }

   itemRet    = _itemPutNL( NULL, 0 );
   itemString = _itemParam( 1 );
   itemFindMe = _itemParam( 2 );

   if ( (_itemType( itemString ) == CHARACTER) &&
        (_itemType( itemFindMe ) == CHARACTER) )
   {
      _itemCopyC( itemFindMe, &cFindMe, 1 );

      vmhString = _xvalloc( _itemSize( itemString ), NULL );
      cStringP = _xvlock( vmhString );

      uiLen = _itemCopyC( itemString, cStringP, NULL );

      for( i = 0; i < uiLen; i++ )
      {
         if ( cStringP[i] == cFindMe )
            uiChars++;
      }

      _xvunlock( vmhString );
      _xvfree( vmhString );


      itemRet = _itemPutNL( itemRet, (long)uiChars );
      }
   _itemReturn( itemRet );

   _itemRelease( itemRet );
   _itemRelease( itemString );
   _itemRelease( itemFindMe );

   return;
}
```

Files Library is CLIPPER.LIB, header file is Item.api.

See Also _itemType()

_itemType()

Determine an item's type

C Prototype

```
#include "item.api"
USHORT _itemType(
                    ITEM itmRef
                  )
```

Arguments

itmRef is the item whose type you want to determine.

Returns

One of a number of manifest constants that indicate the item's CA-Clipper type, as shown in the following section.

Description

The _itemType() function is used to determine an item's CA-Clipper type. The returned type corresponds to the table below:

Item Type Manifest Constants from Extend.api

Manifest Constant	CA-Clipper Type
UNDEF	NIL or simply undefined
CHARACTER	Character string
NUMERIC	Numeric (long or double)
LOGICAL	Boolean value
DATE	Date value
MPTR	Item is passed by reference
MEMO	Memo field
ARRAY	CA-Clipper-level array
BLOCK	Code block
DOUBLE	Double numeric

These constants do not distinctly represent CA-Clipper types. Numeric values are divided into word and double, and system-defined objects as well as object type (class name) are not determinable. CA-Clipper-level objects are identified as arrays by the Item API.

Note: Computer Associates does not recommend use of the Item API to modify object values, as such modifications would violate an object's interface and encapsulation.

An item's _itemType() may be a combination of manifest constants from the previous table. For example, an item may be a character string passed by reference. In this case, the flags CHARACTER and MPTR would be set. A call to _itemType() for a numeric such as 42.102 would set both NUMERIC and DOUBLE flags. To test for multiple flags, simply OR them together (e.g., CHARACTER | MPTR).

Files Library is CLIPPER.LIB, header file is Item.api.

See Also _itemSize()

Chapter 6
Using the Fixed Memory API

The Fixed Memory (FM) Application Programming Interface (API) is a key element in CA-Clipper's open architecture. It allows you to allocate and free small, fixed memory segments and is designed for use within your Extend routines to aid in memory management.

In This Chapter

This chapter describes fixed memory and when you should use it and then demonstrates common usage of the FM API. The major topics covered are:

- What is fixed memory?
- What is the FM API?
- When should it be used?
- How CA-Clipper uses fixed memory
- The basic FM protocol

What Is Fixed Memory?

As its name implies, fixed memory is CA-Clipper runtime memory that is statically allocated. Fixed memory is not part of the dynamic memory pool. It is, instead, taken from a small area just above DGROUP. This area, called the *fixed memory heap*, can grow larger, but never smaller. Released segments within the fixed memory heap can be reused, but if a segment of a particular requested size is not available, the fixed memory heap will be expanded to meet the request. When the fixed memory heap grows, it takes away from the total amount of running space available to dynamic or virtual memory (VM).

What Is the FM API?

The FM API is a header file, Fm.api, which is a set of functions and rules that allow you to allocate and free small, fixed memory segments. It completely replaces C's malloc() and free() functions and works similarly to those functions. In fact, you should not use C functions that call malloc() or free(). The C functions malloc() and free() are available to the CA-Clipper Extend routine programmer, but both are simply wrappers for the _xAlloc() and _xFree() functions in this API. Because of the unnecessary overhead that these wrappers create, use of malloc() and free() is not recommended. For more information, see the "Using the Extend System API" chapter of this guide.

When Should You Use the FM API?

You should use the FM API when you need only a small amount of memory for a given Extend routine. If you need to allocate over 512 bytes of total memory for an Extend routine or subsystem written in C or Assembly, use the Virtual Memory (VM) API for optimal performance (see the "Using the Virtual Memory API" chapter of this guide). Even if you have many small allocations that total over 256 bytes, the VM API is a better choice because of its ability to declare heaps and suballocate them. If your Extend routine allocates under 128 bytes of total memory, you should consider using the C stack to store the information instead.

Memory Allocation Decision Matrix

Bytes	C Stack	Fixed Memory	Virtual Memory
0 - 128	Yes	No	No
129-512	No	Yes	Yes w/Heaps
513-65530	No	No	Yes

The primary disadvantage to using fixed memory over virtual memory for allocations of 129 to 512 bytes is the necessity of suballocating the memory yourself.

How CA-Clipper Uses Fixed Memory

Access to fixed memory is handled almost identically to C's malloc() and free() functions. CA-Clipper allocates fixed memory from its fixed memory pool, just below the base of dynamic (virtual) memory. Static symbol references and macro-compiled strings, as well as Extend routines, reside in fixed memory.

Take care when allocating fixed memory. If too much fixed memory is allocated and never freed, the boundary between fixed and dynamic memory moves into dynamic memory, expanding fixed memory but reducing dynamic memory. Once memory is taken from the dynamic memory area, it cannot be reallocated.

The Basic FM Protocol

You should use the FM API as a replacement for malloc() and free(). Always free allocated memory when it is no longer needed. The following pseudocode illustrates the standard FM API protocol:

```
Extend Routine:
   Allocate 500 bytes of memory
   Copy bytes into allocated memory
   .
   . <statements>
   .
   Free 500 bytes of memory
   return
```

Fixed memory can remain allocated across Extend routine activations. One Extend routine may allocate a fixed memory segment, while another might free it. Note that in order to maintain proper modular programming principles, the address of the fixed memory segment should be maintained as a *static* variable visible to the subsystem containing the functions which allocate and deallocate the fixed memory segment. Since C static variables are stored in DGROUP in large model programs and DGROUP is a very precious commodity in a CA-Clipper application, algorithms of this sort are often frowned upon. Furthermore, the FM segments allocated will remain in conventional memory even when not being used. A better solution is to virtualize the segments (by using the VM API) so that the segments may be swapped out to disk or EMS when they are not in use.

The Item API offers programmers the ability to evaluate code blocks from C or Assembly. Using this feature while fixed memory is allocated may also pose some difficulties. Since the code block evaluation will return the user to execution of CA-Clipper code, you will want to be careful not to leave segments allocated while evaluating code blocks as this practice may reduce the amount of virtual memory available to your application. The CA-Clipper runtime library uses fixed memory, and because of this the fixed memory *high-water mark* (i.e., the boundary between fixed memory and virtual memory) may needlessly be pushed into the virtual arena.

Summary

C or Assembly language programmers should be aware of the system-wide implications of using fixed memory. The FM API is a replacement for C malloc() and free() functions. Excess allocation of fixed memory affects the available dynamic memory. If your allocations force the movement of the fixed memory boundary into the dynamic area, you can never retrieve that memory for dynamic use. Use fixed memory only when allocating less than 512 bytes of memory. Free fixed memory when you no longer need it.

Chapter 7
Fixed Memory Reference Listing

_xalloc()
_xfree()
_xgrab()

Chapter 7
Fixed Memory API Reference

The Fixed Memory (FM) API gives your Extend routines the ability to allocate and free fixed memory segments when you need only a small amount of memory. This chapter is an alphabetical reference to all of the functions in the CA-Clipper FM API.

The prototypes for these functions are defined in the header file Fm.api, located in the \CLIP53\INCLUDE directory. The functions themselves are defined in CLIPPER.LIB, located in the \CLIP53\LIB directory.

_xalloc()

Allocate memory and return NULL if unsuccessful

C Prototype

```
#include "fm.api"
void far * _xalloc(
                        unsigned int uiSize
                    )
```

Arguments

uiSize is the number of bytes to allocate.

Returns

_xalloc() returns a far pointer to the allocated memory or NULL if the requested memory could not be allocated.

Description

_xalloc() lets a C or Assembly language function allocate memory from CA-Clipper's fixed heap. If the allocation request is unsuccessful, CA-Clipper returns a NULL pointer.

Use _xfree() to free memory allocated with _xalloc() after use.

Warning! *Fixed memory returned from _xalloc() is not cleared by the system and is, therefore, in an uninitialized state.*

Examples

- From C:

```
char *mem;
mem = (char)_xalloc(320);
```

- From Assembly language:

```
EXTRN __xalloc:FAR
      mov   ax, 320
      push  ax
      call  __xalloc          ; pointer in DX:AX
      add   sp, 2             ; reset stack pointer
```

Files

Library is CLIPPER.LIB, header file is Fm.api.

See Also

_xfree(), _xgrab()

_xfree()

Free allocated memory

C Prototype

```
#include "fm.api"
void _xfree(
              void far * vlpMem
            )
```

Arguments

vlpMem is a far pointer to memory allocated with _xalloc() or _xgrab().

Returns

_xfree() has no return value.

Description

_xfree() releases memory allocated by _xalloc() or _xgrab(). Note that the pointer returned by _xalloc() or _xgrab() must be passed as the argument to _xfree().

Note: In Fm.api, _exmback() is defined (using #define) to _xfree() to maintain compatibility with Summer '87.

Examples

- From C:

```
char *mem;
mem = _xgrab(320);
_xfree(mem);
```

- From Assembly language:

```
EXTRN __xgrab:FAR
EXTRN __xfree:FAR
      mov   ax, 320
      push  ax
      call  __xgrab          ; pointer in DX:AX
      add   sp, 2            ; reset stack pointer
      push  dx
      push  ax
      call  __xfree          ; give it right back
      add   sp, 4
```

Files

Library is CLIPPER.LIB, header file is Fm.api.

See Also

_xalloc(), _xgrab()

_xgrab()

Allocate memory, generating an error if unsuccessful

C Prototype

```
#include "fm.api"
void far * _xgrab(
                  unsigned int uiSize
                 )
```

Arguments

uiSize is the number of bytes to allocate.

Returns

_xgrab() returns a pointer to the allocated memory.

Description

_xgrab() lets a C or Assembly language function allocate memory from CA-Clipper's fixed heap. If the allocation request is unsuccessful, CA-Clipper raises an unrecoverable internal error. Thus, _xgrab() always returns a valid pointer (never NULL).

Use _xfree() to free memory allocated with _xgrab() after use.

Warning! *Fixed memory returned from _xgrab() is not cleared by the system and is, therefore, in an uninitialized state.*

Note: In Fm.api, _exmgrab() is defined (using #define) to _xgrab() to maintain compatibility with Summer '87.

Examples

- From C:

```
char *mem;
mem = (char)_xgrab(320);
```

- From Assembly language:

```
EXTRN __xgrab:FAR
      mov   ax, 320
      push  ax
      call  __xgrab          ; pointer in DX:AX
      add   sp, 2            ; reset stack pointer
```

Files

Library is CLIPPER.LIB, header file is Fm.api.

See Also

_xalloc(), _xfree()

Chapter 8
Using the Virtual Memory API

The Virtual Memory (VM) Application Programming Interface (API) is a key element in CA-Clipper's open architecture. It allows you to allocate and free virtual memory segments and is designed for use within your Extend routines to aid in memory management.

In This Chapter

This chapter describes virtual memory and when you should use it, then demonstrates common usage of the VM API. The major topics covered are:

- What is virtual memory?
- What is the VM API?
- When should it be used?
- How CA-Clipper uses virtual memory
- The basic VM protocol
- Handling multiple segments
- Using long-term locks
- Using heap functions

What Is Virtual Memory?

Virtual memory is a generic term for hardware or software techniques that allow a limited amount of real memory to emulate a much larger virtual memory space. This powerful technique allows your CA-Clipper programs to create large numbers of character strings and arrays without running out of memory. A CA-Clipper programmer need not be aware of virtual memory because the Virtual Memory Management (VMM) system handles everything transparently.

What Is the VM API?

The CA-Clipper Virtual Memory Management system does not extend to the C and Assembly language level; however, the VM API is provided to facilitate usage of virtual memory from Extend routines.

The VM API is a header file, Vm.api, and a set of functions that can be called from Extend routines, allowing direct communication with CA-Clipper's VMM system. The VM API lets you safely allocate large amounts of memory in your Extend routines so that they cooperate with, rather than cripple, the CA-Clipper programs running them.

Important! *Using a virtual memory system is considerably more difficult than using a fixed memory allocator (like C's malloc() and free()) and requires more care. It is quite easy to bring a working CA-Clipper application to a complete shutdown by calling a function that improperly uses the VM API. For this reason, Computer Associates recommends that only programmers with extensive knowledge of C or Assembly language use this API.*

When Should You Use the VM API?

When you allocate memory from an Extend routine, you have two choices. You can either use the FM API (see the "Using the Fixed Memory API" chapter of this guide) or the VM API. You should use the VM API whenever you allocate anything other than a very small amount of memory.

This is because the FM API functions, _xalloc() and _xgrab(), allocate fixed segments that permanently reside in conventional memory. Conventional memory is extremely limited and allocating a large block of it may not leave enough memory for your CA-Clipper application to continue.

With this in mind, a good way of determining if you need to use the VM API is to ask yourself two simple questions:

- How much memory do I need to allocate?

  If you need to allocate more than 256 bytes, you should probably use the VM API, especially if you are allocating the memory in multiple chunks that you do not need to access simultaneously. For example, if you allocate memory for several large structures to store information about a particular activity, you could use the VM API to retrieve data from these structures one at a time.

- Do I need to retain the allocated memory after returning to CA-Clipper?

  Many Extend routines need to retain data after they finish executing (e.g., a function that allocates memory for use as a data buffer by other routines). If you use the FM API, the data takes up conventional memory even when it isn't being used. A better solution is to use the VM API so that the data moves in and out of conventional memory as needed.

How CA-Clipper Uses Virtual Memory

Virtual memory consists of *segments* (also called *VM segments*) that move in and out of conventional memory, giving the effect of a much larger virtual memory space. The C and Assembly language functions that are part of CA-Clipper's runtime system explicitly tell the Virtual Memory Manager when to swap a VM segment out of conventional memory, when to bring one into conventional memory, and when to prevent one from moving out of conventional memory.

Note: The term VM segment refers to a section of virtual memory. Unlike in 80*n*86 terminology where a segment refers to a 64K section of memory, a VM segment is not a fixed-size block of memory—it can be one or many bytes long.

All of this work occurs at the C and Assembly language level and is completely transparent to the CA-Clipper programmer. As an Extend programmer, however, you must manage the complexity of virtual memory using the VM API.

The Basic VM Protocol

The Virtual Memory Manager does not refer to VM segments by a physical hardware address (after all, a VM segment may not even be in memory) but uses *segment handles* to identify them. To use a VM segment, you allocate it in order to obtain a segment handle. Then, you use the segment handle to lock, access, and unlock the segment frequently. Finally, you free the VM segment from memory.

The following pseudocode summarizes this protocol:

```
First Extend routine:
   Allocate the segment
   Lock it
   Fill it with initial values
   Unlock it

Second Extend routine:
   Lock segment
   Change the contents
   Unlock it

Third Extend routine:
   Lock it
   Get the contents
   Unlock it
   Free the segment
```

Note that this protocol never leaves the segment locked upon returning to CA-Clipper. This is to make available as much conventional memory as possible to the VMM system. There are times when you may wish to lock a segment for a long period of time (see the Using Long-Term Locks section later in this chapter) but, in general, you should leave a segment locked only as long as it must be locked.

The VM API functions _xvalloc(), _xvlock(), _xvunlock(), and _xvfree() perform the VM segment allocate, lock, unlock, and free operations, respectively.

Allocating a VM segment (_xvalloc()) allocates memory and returns a segment handle that you use in subsequent operations.

Locking a VM segment (_xvlock()) brings it into conventional memory, prevents it from being swapped out, and returns a pointer to the segment.

Unlocking a VM segment (_xvunlock()) invalidates its pointer and allows the segment to be swapped out of conventional memory.

Freeing a VM segment (_xvfree()) invalidates its segment handle and releases the segment from memory.

Handling Multiple Segments

The protocol is more complicated when you are dealing with more than one VM segment. You should try to avoid locking two segments at the same time, especially if either of the segments is potentially large (i.e., bigger than 4K bytes), because you run the risk of not having enough conventional memory available to hold them both. The basic rule is not to lock a segment unless it absolutely has to be locked.

In the following pseudocode fragment, two segments are allocated and alternately locked, accessed, and unlocked:

```
First Extend routine:
    Allocate segment1
    Allocate segment2

    Lock segment1

    Fill segment1 with initial values
    Unlock segment1

    Lock segment2
    Fill segment2 with initial values
    Unlock segment2

Second Extend routine:
    Lock segment1
    Retrieve value from segment1
    Unlock segment1

    Manipulate value retrieved

    Lock segment2
    Store value in segment2
    Retrieve value2 from segment2
    Unlock segment2

    Lock segment1
    Store value2 in segment1
    Unlock segment1

Third Extend routine:
    Lock segment1
    Retrieve value from segment1
    Unlock segment1

    Lock segment2
    Retrieve value2 from segment2
    Unlock segment2

    Free segment1
    Free segment2
    Return (value1 + value2)
```

Manipulating multiple VM segments can lead to some programming challenges. For example, how can you concatenate two strings into a third string without locking more than one string at a time? The basic approach is to create a *transfer* area, or buffer, that is small enough (less than 256 bytes) to always keep in memory and use this as the bridge between segments. The pseudocode for this operation is shown below:

```
Function to concatenate segment1 and segment2
   position = 0

   Allocate result segment

   // First copy segment1 into result segment
   While bytes left in segment1
      Lock segment1
      Move to position
      Copy bytes to transfer area
      Unlock segment1

      Lock result segment
      Move to position
      Copy bytes from transfer area
      Unlock result segment

      position = position + bytes copied
   End while

   // Then add second segment
   pos2 = 0
   While bytes left in segment2
      Lock segment2
      Move to pos2
      Copy bytes to transfer area
      Unlock segment2

      Lock result segment
      Move to position
      Copy bytes from transfer area
      Unlock result segment

      position = position + bytes copied
      pos2 = pos2 + bytes copied
   End while

   Return (result segment)
```

This method allows segments to be concatenated together into a third string as long as there is enough memory to hold the result string, making the function extremely safe.

While copying using this method is definitely more work than using fixed memory, it isn't much slower when the segments will all fit into conventional memory. Locking a segment that is currently in conventional memory is simply a matter of changing a flag, and therefore, is extremely fast.

Using Long-Term Locks

In some circumstances, you may need to leave a part of virtual memory locked for an extended period of time (e.g., in a library where you continually access a locked segment). A series of _xvlock() and _xvunlock() calls allows access to the segment in a section of conventional memory, but does so at the cost of the time it takes to continuously lock and unlock the segment. Since other segment accesses may cause the segment to move out of conventional memory, you may not be able to access the segment as fast as is necessary.

The solution is to use the _xvwire() function to hold a long-term lock on this persistent segment. The _xvwire() function takes longer than _xvlock() to perform its task but locks the requested segment in such a way as to minimally impact the allocation of other segments.

Segments that you lock with _xvlock() may be placed in the middle of the swap space. Thus, if you hold the lock for a long period of time, the swap space can become fragmented, reducing the size of the largest contiguous memory block available.

On the other hand, _xvwire() shuffles the swap space so that enough space for the wired segment is available at one end and moves the wired segment to that position. Thus, if you use _xvwire() to lock a segment for a long period of time, you avoid fragmentation and retain the maximum possible swap space for future allocations.

When a simple lock will do the job, you should not wire a segment as a way of avoiding the algorithmic constraints of using virtual memory (such as the concatenation of two segments discussed previously). On a system-wide scale, _xvwire() takes a lot of time to perform its task. Therefore, you should only wire segments that must remain locked for an extended period of time.

Using Heap Functions

CA-Clipper's VMM is most efficient when managing large blocks of memory and can become bogged down when managing a large number of small segments. As a rule of thumb, you want to use virtual memory to handle values larger than 256 bytes, but this is an arbitrary limit based on the memory profile of typical CA-Clipper applications.

There are some cases when you will want to use the VM API heap (_xvheap) functions to simulate a C-style heap and increase the efficiency of the CA-Clipper VMM system. For example, what if your Extend routine needs to allocate an unknown quantity of 200-byte areas? This is a good example of a situation where you would use a heap.

To decide whether or not to use a heap, weigh the number and size of your overall memory allocations. If your Extend routines have many allocations that collectively take less than 64K but more than 256 bytes, try using a heap. It may increase the efficiency of your application significantly.

Note: If you have to manipulate several potentially large strings, it may be best to store each of the strings in separate VM segments instead of using a heap.

Here is a simplified pseudocode example that uses a heap:

```
Extend routine 1:
   Make new heap
   Allocate segment 1 in heap
   Lock segment 1
   Fill with initial values
   Unlock segment 1
   Allocate segment 2 in heap
   Lock segment 2
   Fill with initial values
   Unlock segment 2

Extend routine 2:
   Lock segment 2 in heap
   Read values
   Unlock segment 2 in heap
   Allocate transitory segment 3 in heap
   Lock segment 3 in heap
   Fill it with values
   Post contents of segment 3 as a return value
   Unlock segment 3
   Free segment 3

Extend routine 3:
   Free segment 2 in heap
   Free segment 1 in heap
   Destroy the heap
```

A VM Example

The following example is a CA-Clipper-callable function that shows the VM API in action. The function opens a source file, reads the entire file (which could be several megabytes) into VM, then writes all the data back from VM to a new file. Of course, this is not the most efficient way to copy a file, but it does illustrate how to use the VM API:

```
/***
* VMCOPY.C
*
* Call from CA-Clipper:
*   VMCopy( cInFile, cOutFile ) --> lSuccess
*
*
* HANDLE ReadFile( char * pInFile )    Reads an entire file into VM
* HANDLE WriteFile( char * pOutFile )  Writes to new file from VM
*
* Compile: CL /c /AL /FPa /Gs /Oalt /Zl Vmcopy.c
*/


#include "extend.api"
#include "vm.api"
#include "io.h"
#include "stdio.h"
#include "fcntl.h"
#include "sys\types.h"
#include "sys\stat.h"

#define SEGMENT_SIZE (16 * 1024) // 16k buffer, cannot be too big
                                 // because it must be loaded
                                 // into available swap space

/* file info structure will be held in its own VM segment */
typedef struct FILE_INFO
{
   unsigned int uHandleCount;    // number of segment handles needed
   unsigned long ulFileSize;     // size of file to be copied
   HANDLE hOffset;               // start of segment handles array

} FILE_INFO;

/* Function prototypes */
static HANDLE  AllocFileInfo( char * pInFile );
static Boolean FreeFileInfo( HANDLE hFileInfo );
static Boolean ReadFile( char * pInFile, HANDLE hFileInfo );
static Boolean WriteFile( char * pOutFile, HANDLE hFileInfo );
```

```
CLIPPER VMCOPY()
{

   HANDLE hFileInfo = 0;          // handle of VM segment
                                  // that holds file info
   Boolean fSuccess = FALSE;      // flag to determine if
                                  // the function succeeded

   if (ISCHAR(1) & ISCHAR(2))
   {
      hFileInfo = AllocFileInfo( _parc(1) );

      if (hFileInfo != 0)
      {
         fSuccess = ReadFile( _parc(1), hFileInfo );

         if (fSuccess)
            fSuccess = WriteFile( _parc(2), hFileInfo );

         fSuccess = fSuccess & FreeFileInfo( hFileInfo );
      }
   }
   _retl( fSuccess );
}


/***
* ReadFile()
*
* Reads the input file and stores the data in the VM
* segments created by AllocFileInfo().  The read continues
* until all the segments (the number of which is stored in
* the file info structure) are processed.  If the input file
* does not exist, the function terminates and returns a
* FALSE.
*/

static Boolean ReadFile( char * pInFile, HANDLE hFileInfo )
{

   int hFile;                 // source file handle
   unsigned int uBytesRead;   // number of bytes read()
   unsigned int uIndex;       // index into handle array
   HANDLE * pHandles;         // pointer to segment handles array
   HANDLE hBuffer = 0;        // current segment handle temp value
   char * pBuffer;            // current buffer segment pointer
   FILE_INFO * pFileInfo;     // pointer to handle array segment
   Boolean fSuccess = FALSE;  // flag to determine success
```

```
    hFile = open( pInFile, O_RDONLY | O_BINARY );

    if (hFile != -1)
    {
       /* lock file info segment */
       pFileInfo = _xvlock( hFileInfo );

       if (pFileInfo != NULL)
       {

          pHandles = &(pFileInfo->hOffset);  // handle array pointer
          fSuccess = TRUE;                   // errors change flag

          for ( uIndex = 0; uIndex < pFileInfo->uHandleCount;
                uIndex++ )
          {
             /* get the handle to the next segment */
             hBuffer = pHandles[ uIndex * sizeof( HANDLE ) ];

             /* get a pointer to the buffer by locking segment */
             pBuffer = _xvlock( hBuffer );

             if (pBuffer == NULL)
             {
                fSuccess = FALSE;          // read error, exit
                break;
             }

             /* read the file into the buffer */
             uBytesRead = read( hFile, pBuffer, SEGMENT_SIZE );

             _xvunlock( hBuffer );         // unlock buffer segment

             if (uBytesRead <= 0 )
             {
                fSuccess = FALSE;          // read error, exit
                break;
             }
          }
          /* unlock the file info segment */
          _xvunlock( hFileInfo );
       }
       close( hFile );
    }
    return (fSuccess);
}
```

```
/***
* WriteFile()
*
* Writes the data stored in the VM segments to the output
* file.  The write continues until all the segments have
* been written.  If the output file does not exist it
* creates it.  If it does exist it opens it, truncates it to
* zero bytes, and writes the data to it.
*/

static Boolean WriteFile( char * pOutFile, HANDLE hFileInfo )
{
   int hFile;                    // source file handle
   unsigned int uBytesToWrite;   // number of bytes to be written
   unsigned int uBytesWritten;   // number of bytes on written
   unsigned long ulBytesLeft;    // total bytes left to read
   unsigned int uIndex = 0;      // index into handle array
   HANDLE hBuffer;               // current segment handle tmp value
   FILE_INFO * pFileInfo;        // pointer to handle array segment
   char * pBuffer;               // current buffer segment pointer
   HANDLE * pHandles;            // pointer segment handles array
   Boolean fSuccess;             // flag to determine success


   hFile = open( pOutFile, O_RDWR | O_CREAT | O_BINARY,
                 S_IREAD | S_IWRITE );

   if (hFile != -1)                          // valid file handle?
   {
      pFileInfo = _xvlock( hFileInfo );      // lock file info sgmnt
      ulBytesLeft  = pFileInfo->ulFileSize; // get bytes to write

      pHandles = &(pFileInfo->hOffset);

      for ( uIndex = 0; uIndex < pFileInfo->uHandleCount;
            uIndex++ )
      {
         /* get the handle to the next buffer segment */
         hBuffer = pHandles[ uIndex * sizeof( HANDLE ) ];

         /* get buffer pointer by locking the segment */
         pBuffer = _xvlock( hBuffer );

         if (pBuffer == NULL)
         {
            fSuccess = FALSE;       // _xvlock() failed, exit
             break;
         }

         /* if bytes left are less that the segment size,
            only write the number of bytes left on the
            counter.  Otherwise, write the full segment */

         if (ulBytesLeft < SEGMENT_SIZE)
            uBytesToWrite = (unsigned int) ulBytesLeft;

         else
            uBytesToWrite = SEGMENT_SIZE;

         uBytesWritten = write( hFile, pBuffer, uBytesToWrite );

         _xvunlock( hBuffer );      // unlock the buffer segment
```

```
            if (uBytesWritten != uBytesToWrite)
            {
               fSuccess = FALSE;       // write failed, exit
               break;
            }

            /* decrease bytes left counter by the number of
               bytes written */
            ulBytesLeft -= (unsigned long) uBytesWritten;
         }
         /* unlock the file info segment */
         _xvunlock( hFileInfo );

         close( hFile );
      }
      return (fSuccess);
}


/***
* AllocFileInfo()
*
* Allocates a VM segment that contains a file info structure
* and an array of buffer segment handles.  The number of
* buffer segments required is calculated from the file
* length and the buffer size.  The buffer segments are then
* allocated and their handles stored in the buffer handle
* array.  If successful a valid handle to the file info
* segment is returned.  Otherwise, a null handle is
* returned.
*/

static HANDLE AllocFileInfo( char * pInFile )
{
   int hFile;                    // source file handle
   unsigned long ulFileSize;     // temp variable for file size
   unsigned int uIndex;          // index into handle array
   unsigned int uHandleCount;    // temp value for handle count
   HANDLE hBuffer = 0;           // temp value for current
                                 // segment handle
   HANDLE hFileInfo = 0;         // file info segment handle
   FILE_INFO * pFileInfo;        // pointer to handle array segment
   HANDLE * pHandles;            // pointer segment handles array
   Boolean fSuccess = FALSE;     // flag to determine success

   hFile = open( pInFile, O_RDONLY | O_BINARY );

   if (hFile != -1)
   {
      /* find the size of the file */
      ulFileSize = lseek( hFile, 0L, SEEK_END );
      lseek( hFile, 0L, SEEK_SET );      // rewind to bof

      /* determine how many full segments are required */
      uHandleCount = (unsigned int) (ulFileSize /
                     (long) SEGMENT_SIZE);

      /* add one segment for the remaining data, if any */
      uHandleCount += ((ulFileSize % (long) SEGMENT_SIZE) ? 1 : 0);

      /* allocate a segment large enough to hold file info
         structure as well as the array of segment handles */
```

```
      hFileInfo = _xvalloc( sizeof( FILE_INFO ) +
                  (uHandleCount * sizeof( HANDLE )), 0 );

      if (hFileInfo != 0)
      {
         /* get a pointer to the file info segment by locking it */
         pFileInfo = _xvlock( hFileInfo );

         if (pFileInfo != NULL)
         {
            fSuccess = TRUE;              // errors will change flag

            /* store values to the file info structure */
            pFileInfo->uHandleCount = uHandleCount;
            pFileInfo->ulFileSize = ulFileSize;

            pHandles = &(pFileInfo->hOffset);

            for ( uIndex = 0; uIndex < pFileInfo->uHandleCount;
                  uIndex++ )
            {
               /* allocate a new segment */
               hBuffer = _xvalloc( SEGMENT_SIZE, 0 );

               if (hBuffer == 0)          // _xvalloc() failed,
                                          // exit
               {
                  fSuccess = FALSE;
                  break;
               }

               /* store the buffer handle into the file info */
               pHandles[ uIndex * sizeof( HANDLE ) ] = hBuffer;
            }
         }
         _xvunlock( hFileInfo );

         if (! fSuccess)
         {
             if ( hFileInfo )
                FreeFileInfo( hFileInfo );

             hFileInfo = 0;            // null the handle
                                       // (failure flag)
         }
      }
      close( hFile );
   }
   return (hFileInfo);
}
```

```
/***
* FreeFileInfo()
*
* Frees each buffer segment using the buffer handle array,
* and then frees the file info segment.  Returns TRUE
* successful, FALSE if an error occurred.
*/

static Boolean FreeFileInfo( HANDLE hFileInfo )
{
    unsigned int uIndex;          // index into handle array
    HANDLE hBuffer = 0;           // temp value for current segment
handle
    FILE_INFO * pFileInfo;        // pointer to handle array segment
    HANDLE * pHandles;            // pointer segment handles array
    Boolean fSuccess = FALSE;     // flag to determine success


    /* get a pointer to the file info segment by locking it */
    pFileInfo = _xvlock( hFileInfo );

    if (pFileInfo != NULL)
    {
        fSuccess = TRUE;                    // errors will change flag

        pHandles = &(pFileInfo->hOffset);   // pointer to handle array

        for ( uIndex = 0; uIndex < pFileInfo->uHandleCount; uIndex++
)
        {
            /* get the handle to the next segment */
            hBuffer = pHandles[ uIndex * sizeof( HANDLE ) ];

            _xvfree( hBuffer );             // free current buffer handle
        }

        _xvunlock( hFileInfo );             // unlock file info segment

        _xvfree( hFileInfo );               // free file info segment

    }
    return (fSuccess);
}
```

Summary

This chapter has given you an introduction to the architecture and strategies of the VM API. You have seen how and when to use the simple VM API allocation and deallocation functions, as well as the heap functions. You have also seen algorithms and techniques to optimize VM performance. The next chapter is an alphabetical reference guide to the VM API functions.

Chapter 9
Virtual Memory Reference Listing

_xvalloc()
_xvfree()
_xvheapalloc()
_xvheapdestroy()
_xvheapfree()
_xvheaplock()
_xvheapnew()
_xvheapresize()
_xvheapunlock()
_xvlock()
_xvlockcount()
_xvrealloc()
_xvsize()
_xvunlock()
_xvunwire()
_xvwire()

Chapter 9
Virtual Memory API Reference

The Virtual Memory (VM) API gives your Extend routines access to the CA-Clipper Virtual Memory system. This chapter is an alphabetical reference to all of the functions in the CA-Clipper VM API.

The prototypes for these functions are defined in the header file Vm.api, located in the \CLIP53\INCLUDE directory. The data types used in the prototypes are defined in the header file Clipdefs.h (automatically included by Vm.api), also located in \CLIP53\INCLUDE. The functions themselves are defined in CLIPPER.LIB located in the \CLIP53\LIB directory.

_xvalloc()

Allocate a VM segment

C Prototype

```
#include "vm.api"
HANDLE _xvalloc(
                    USHORT uiSize,
                    USHORT uiFlags
                  )
```

Arguments

uiSize is the size of the segment to allocate in bytes.

uiFlags is currently unused and must be set to zero.

Returns

If successful, _xvalloc() returns a 16-bit segment handle; otherwise, it returns zero.

Description

_xvalloc() allocates a VM segment, returning a handle that you can use in all subsequent VM operations involving that segment.

Note: To use the memory contained in the segment, your function must obtain a far pointer to physical memory by locking the segment with _xvlock() or _xvwire().

Warning! *You must eventually use _xvfree() to free VM segments allocated by _xvalloc().*

Notes

- ***Maximum size:*** The maximum number of bytes that can be allocated in a VM segment is 65,520, enough to hold the largest CA-Clipper string and a null terminator.

Examples

- This example allocates a segment with _xvalloc() and locks it with _xvlock(). After a string is copied into the locked segment, the segment is unlocked and freed (with _xvunlock() and _xvfree()):

```
// CA-Clipper include files
#include "extend.api"
#include "vm.api"

// Microsoft C include files
#include "string.h"

// Prototype
Boolean VMExample(char * spSrc);


Boolean VMExample(char * spSrc)

{
    HANDLE hSegment;
    char * spString;
    Boolean bResult = FALSE;

    if (hSegment = _xvalloc(strlen(spSrc) + 1, 0))
       {
       spString = _xvlock(hSegment);
       if (spString != NULL)
          {
          strcpy(spString, spSrc);

          .
          . <statements>
          .

          bResult = TRUE;

          _xvunlock(hSegment);
          }
       _xvfree(hSegment);
       }

    return (bResult);
}
```

Files Library is CLIPPER.LIB, header file is Vm.api.

See Also _xvheapnew(), _xvfree(), _xvlock(), _xvrealloc(), _xvwire()

_xvfree()

Free an allocated VM segment

C Prototype

```
#include "vm.api"
void _xvfree(
                HANDLE hSegment
             )
```

Arguments

hSegment is the VM segment handle returned by _xvalloc().

Returns

_xvfree() has no return value.

Description

_xvfree() releases a VM segment previously allocated by _xvalloc() and invalidates the handle of that segment.

Warning! *Do not use _xvfree() to free a locked segment. Use _xvlockcount() to determine the number of locks on a segment and _xvunlock() to release the locks.*

Examples

- This example allocates a segment with _xvalloc() and locks it with _xvlock(). After a string is copied into the locked segment, the segment is unlocked and freed (with _xvunlock() and _xvfree()):

```
// CA-Clipper include files
#include "extend.api"
#include "vm.api"

// Microsoft C include files
#include "string.h"

// Prototype
Boolean VMExample(char * spSrc);


Boolean VMExample(char * spSrc)
{
    HANDLE hSegment;
    char * spString;
    Boolean bResult = FALSE;

    if (hSegment = _xvalloc(strlen(spSrc) + 1, 0))
       {
       spString = _xvlock(hSegment);
       if (spString != NULL)
          {
          strcpy(spString, spSrc);

          .
          . <statements>
          .

          bResult = TRUE;

          _xvunlock(hSegment);
          }
       _xvfree(hSegment);
       }

    return (bResult);
}
```

Files Library is CLIPPER.LIB, header file is Vm.api.

See Also _xvalloc(), _xvlockcount(), _xvunlock()

_xvheapalloc()

Allocate a memory block from a segment heap

C Prototype

```
#include "vm.api"
USHORT _xvheapalloc(
                      HANDLE hSegment,
                      USHORT uiSize
                    )
```

Arguments

hSegment is the VM segment handle returned by _xvheapnew().

uiSize is the number of bytes to allocate from the segment heap.

Returns

If successful, _xvheapalloc() returns the offset of the allocated memory block; otherwise, it returns zero.

Description

_xvheapalloc() allocates a memory block from within a segment heap. If you request a size larger than the largest contiguous memory block within the segment heap (or larger than the segment heap itself), the function returns zero.

If the memory is successfully allocated, _xvheapalloc() returns an offset into the segment heap. You can use this offset, with the VM segment handle returned by _xvheapnew(), in all subsequent operations involving the allocated block.

Note: To use the allocated memory block, your function must obtain a far pointer to physical memory by locking the allocated memory with _xvheaplock().

Warning! *You must eventually use _xvheapfree() to free memory blocks allocated by _xvheapalloc().*

Examples

- This example creates a segment heap with _xvheapnew() and allocates a memory block in the segment heap. The block is then locked and the string is copied into it. Later, the memory block is unlocked, the memory freed, and the heap destroyed:

```
// CA-Clipper include files
#include "extend.api"
#include "vm.api"

// Microsoft C include files
#include "string.h"

// Prototype
Boolean VMHeapExample(char * spSrc);

#define HEAP_SIZE 4096

Boolean VMHeapExample(char * spSrc)
{
    HANDLE hSegment;
    unsigned uiStringOffset;
    unsigned uiBufflen;
    char * spString;
    Boolean bResult = FALSE;

    if (hSegment = _xvheapnew(HEAP_SIZE))
       {
       uiBufflen = strlen(spSrc) + 1;
       uiStringOffset = _xvheapalloc(hSegment, uiBufflen);
       if (uiStringOffset)
          {
          spString = _xvheaplock(hSegment, uiStringOffset);
          if (spString != NULL)
             {
             strcpy(spString, spSrc);

             .
             . <statements>
             .

             bResult = TRUE;

             _xvheapunlock(hSegment, uiStringOffset);
             }
          _xvheapfree(hSegment, uiStringOffset);
          }
       _xvheapdestroy(hSegment);
       }

    return (bResult);
}
```

Files Library is CLIPPER.LIB, header file is Vm.api.

See Also _xvalloc(), _xvheapdestroy(), _xvheapfree(), _xvheaplock(), _xvheapnew()

_xvheapdestroy()

Free the segment allocated for a segment heap

C Prototype

```
#include "vm.api"
void _xvheapdestroy(
                      HANDLE hSegment
                    )
```

Arguments

hSegment is the VM segment handle returned by _xvheapnew().

Returns

_xvheapdestroy() has no return value.

Description

_xvheapdestroy() frees a VM segment previously allocated as a segment heap with _xvheapnew() and invalidates the handle of that segment.

Warning! *_xvheapdestroy() frees the entire segment heap. Before using this function, you must use_xvheapfree() to free all memory blocks allocated within the segment heap.*

Examples

- This example creates a segment heap with _xvheapnew() and allocates a memory block in the segment heap. The block is then locked and the string is copied into it. Later, the memory block is unlocked, the memory freed, and the heap destroyed:

```
// CA-Clipper include files
#include "extend.api"
#include "vm.api"

// Microsoft C include files
#include "string.h"

// Prototype
Boolean VMHeapExample(char * spSrc);

#define HEAP_SIZE 4096

Boolean VMHeapExample(char * spSrc)
{
    HANDLE hSegment;
    unsigned uiStringOffset;
    unsigned uiBufflen;
    char * spString;
    Boolean bResult = FALSE;

    if (hSegment = _xvheapnew(HEAP_SIZE))
       {
       uiBufflen = strlen(spSrc) + 1;
       uiStringOffset = _xvheapalloc(hSegment, uiBufflen);
       if (uiStringOffset)
          {
          spString = _xvheaplock(hSegment, uiStringOffset);
          if (spString != NULL)
             {
             strcpy(spString, spSrc);

             .
             . <statements>
             .

             bResult = TRUE;

             _xvheapunlock(hSegment, uiStringOffset);
             }
          _xvheapfree(hSegment, uiStringOffset);
          }
       _xvheapdestroy(hSegment);
       }

    return (bResult);
}
```

Files Library is CLIPPER.LIB, header file is Vm.api.

See Also _xvheapfree(), _xvheapnew()

_xvheapfree()

Free an allocated block of segment heap memory

C Prototype

```
#include "vm.api"
void _xvheapfree(
                    HANDLE hSegment,
                    USHORT uiOffset
                  )
```

Arguments

hSegment is the VM segment handle returned by _xvheapnew().

uiOffset is the offset of the allocated memory block returned by _xvheapalloc().

Returns

_xvheapfree() has no return value.

Description

_xvheapfree() frees a memory block previously allocated in a segment heap by _xvheapalloc() and then invalidates the offset for that memory block.

Note: _xvheapfree() only frees blocks of memory allocated within the segment heap. After all allocated blocks of memory within the segment heap have been freed, use _xvheapdestroy() to free the entire segment heap.

Warning! *Do not use _xvheapfree() to free a locked memory block. Unlock the memory block first using _xvheapunlock().*

Examples

- This example creates a segment heap with _xvheapnew() and allocates a memory block in the segment heap. The block is then locked and the string is copied into it. Later, the memory block is unlocked, the memory freed, and the heap destroyed:

```
// CA-Clipper include files
#include "extend.api"
#include "vm.api"

// Microsoft C include files
#include "string.h"

// Prototype
Boolean VMHeapExample(char * spSrc);

#define HEAP_SIZE 4096

Boolean VMHeapExample(char * spSrc)
{
    HANDLE hSegment;
    unsigned uiStringOffset;
    unsigned uiBufflen;
    char * spString;
    Boolean bResult = FALSE;

    if (hSegment = _xvheapnew(HEAP_SIZE))
       {
       uiBufflen = strlen(spSrc) + 1;
       uiStringOffset = _xvheapalloc(hSegment, uiBufflen);
       if (uiStringOffset)
          {
          spString = _xvheaplock(hSegment, uiStringOffset);
          if (spString != NULL)
             {
             strcpy(spString, spSrc);

             .
             . <statements>
             .

             bResult = TRUE;

             _xvheapunlock(hSegment, uiStringOffset);
             }
          _xvheapfree(hSegment, uiStringOffset);
          }
       _xvheapdestroy(hSegment);
       }

    return (bResult);
}
```

Files Library is CLIPPER.LIB, header file is Vm.api.

See Also _xvheapalloc(), _xvheapdestroy(), _xvheapnew(), _xvheapunlock()

_xvheaplock()

Lock an allocated block of segment heap memory

C Prototype

```
#include "vm.api"
FARP _xvheaplock(
                   HANDLE hSegment,
                   USHORT uiOffset
                 )
```

Arguments

hSegment is the VM segment handle returned by _xvheapnew().

uiOffset is the offset of the allocated memory block returned by _xvheapalloc().

Returns

_xvheaplock() returns a far pointer to the allocated memory block, or a NULL pointer, if either argument is invalid.

Description

_xvheaplock() locks the entire segment heap, guaranteeing that the allocated memory block is located in conventional memory and will not be moved or swapped out by the VMM system. The pointer returned is valid until all blocks of memory within the segment are unlocked using _xvheapunlock().

Warning! *Do not leave a memory block locked unnecessarily. Because all locked segments reside in conventional memory, the more of them you lock, the greater the chance of exhausting conventional memory. Therefore, always unlock a memory block with _xvheapunlock() when you are not actively accessing it. Typically, this means locking and unlocking the same memory block several times within one function call.*

You must eventually unlock memory blocks locked by _xvheaplock() using _xvheapunlock().

Notes

- ***Error conditions:*** If there is not enough VM swap space available for the segment to be loaded into conventional memory, the system raises an internal error and halts.

Examples

- This example creates a segment heap with _xvheapnew() and allocates a memory block in the segment heap. The block is then locked and the string is copied into it. Later, the memory block is unlocked, the memory freed, and the heap destroyed:

```
// CA-Clipper include files
#include "extend.api"
#include "vm.api"

// Microsoft C include files
#include "string.h"

// Prototype
Boolean VMHeapExample(char * spSrc);

#define HEAP_SIZE 4096

Boolean VMHeapExample(char * spSrc)
{
    HANDLE hSegment;
    unsigned uiStringOffset;
    unsigned uiBufflen;
    char * spString;
    Boolean bResult = FALSE;

    if (hSegment = _xvheapnew(HEAP_SIZE))
       {
       uiBufflen = strlen(spSrc) + 1;
       uiStringOffset = _xvheapalloc(hSegment, uiBufflen);
       if (uiStringOffset)
          {
          spString = _xvheaplock(hSegment, uiStringOffset);
          if (spString != NULL)
             {
             strcpy(spString, spSrc);

             .
             . <statements>
             .

             bResult = TRUE;

             _xvheapunlock(hSegment, uiStringOffset);
             }
          _xvheapfree(hSegment, uiStringOffset);
          }
       _xvheapdestroy(hSegment);
       }

    return (bResult);
}
```

Files Library is CLIPPER.LIB, header file is Vm.api.

See Also _xvheapalloc(), _xvheapnew(), _xvheapunlock(), _xvlockcount()

_xvheapnew()

Allocate a VM segment for use as a segment heap

C Prototype

```
#include "vm.api"
HANDLE _xvheapnew(
                  USHORT uiSize
                 )
```

Arguments

uiSize is the size of the VM segment to allocate as a segment heap.

Returns

If successful, _xvheapnew() returns a 16-bit segment handle; otherwise, it returns zero.

Description

_xvheapnew() allocates a VM segment for use as a segment heap.

Warning! *You must eventually free a segment allocated by _xvheapnew() with _xvheapdestroy().*

Notes

- ***Actual heap size:*** In CA-Clipper, there are two bytes of overhead for every memory block allocated within the segment heap. Therefore, you cannot use a segment heap to hold a string that approaches the CA-Clipper maximum string length (65,519 bytes). In such cases, you should dedicate an entire VM segment to the string.
- ***Efficiency:*** The VMM system is most efficient when managing a small number of relatively large segments. The segment heap functions allow a single segment to be treated as a C-style heap (using _xvheapalloc() and _xvheapfree()), making it possible to store multiple data items in one segment. This can greatly increase the overall efficiency of the VMM system.

Examples

- This example creates a segment heap with _xvheapnew() and allocates a memory block in the segment heap. The block is then locked and the string is copied into it. Later, the memory block is unlocked, the memory freed, and the heap destroyed:

```
// CA-Clipper include files
#include "extend.api"
#include "vm.api"

// Microsoft C include files
#include "string.h"

// Prototype
Boolean VMHeapExample(char * spSrc);

#define HEAP_SIZE 4096

Boolean VMHeapExample(char * spSrc)
{
    HANDLE hSegment;
    unsigned uiStringOffset;
    unsigned uiBufflen;
    char * spString;
    Boolean bResult = FALSE;

    if (hSegment = _xvheapnew(HEAP_SIZE))
       {
       uiBufflen = strlen(spSrc) + 1;
       uiStringOffset = _xvheapalloc(hSegment, uiBufflen);
       if (uiStringOffset)
          {
          spString = _xvheaplock(hSegment, uiStringOffset);
          if (spString != NULL)
             {
             strcpy(spString, spSrc);

             .
             . <statements>
             .

             bResult = TRUE;

             _xvheapunlock(hSegment, uiStringOffset);
             }
          _xvheapfree(hSegment, uiStringOffset);
          }
       _xvheapdestroy(hSegment);
       }

    return (bResult);
}
```

Files Library is CLIPPER.LIB, header file is Vm.api.

See Also _xvheapalloc(), _xvheapdestroy(), _xvheapfree(), _xvheapresize()

_xvheapresize()

Resize a segment heap

C Prototype

```
#include "vm.api"
HANDLE _xvheapresize(
                        HANDLE hSegment,
                        USHORT uiSize
                     )
```

Arguments

hSegment is the VM segment handle returned by _xvheapnew().

uiSize is the new size in bytes.

Returns

If successful, _xvheapresize() returns the original VM segment handle; otherwise, it returns zero.

Description

_xvheapresize() resizes a previously allocated segment heap, shortening or lengthening the segment heap to match the specified size. If you shorten the segment heap, bytes at the end of the segment are lost.

If the resizing is unsuccessful (indicated by a return value of zero), the segment heap retains its original size.

Warning! *Do not use _xvheapresize() to resize a segment heap that has any allocated memory block locked. Unlock the memory block first using _xvheapunlock().*

Examples

- This example resizes a segment heap:

```
// CA-Clipper include files
#include "extend.api"
#include "vm.api"

// Prototype
Boolean HeapResize(HANDLE hSegment, unsigned uiNewSize);


Boolean HeapResize(HANDLE hSegment, unsigned uiNewSize)
{
    Boolean bResult = FALSE;

    // Don't attempt to resize locked segment
    //
    if (_xvlockcount(hSegment) == 0)
       {
       if (_xvheapresize(hSegment, uiNewSize))
          bResult = TRUE;
       }

    return (bResult);
}
```

Files Library is CLIPPER.LIB, header file is Vm.api.

See Also _xvheapnew()

_xvheapunlock()

Unlock an allocated block of segment heap memory

C Prototype

```
#include "vm.api"
void _xvheapunlock(
                        HANDLE hSegment,
                        USHORT uiOffset
                    )
```

Arguments

hSegment is the VM segment handle returned by _xvheapnew().

uiOffset is the offset of the allocated memory block returned by _xvheapalloc().

Returns

_xvheapunlock() has no return value.

Description

_xvheapunlock() unlocks a block of allocated memory previously locked by _xvheaplock(), and invalidates any pointers to it.

Note: _xvheaplock() locks the entire segment heap so that it cannot be moved or swapped out until all blocks of memory within it are unlocked with _xvheapunlock().

Warning! *After you unlock a memory block, you should not attempt to access it with the far pointer returned by _xvheaplock() since there is no guarantee that the memory block resides in the segment at that physical memory address. If you need further access to the memory block, lock it again and use the new pointer returned by _xvheaplock().*

Examples

- This example creates a segment heap with _xvheapnew() and allocates a memory block in the segment heap. The block is then locked and the string is copied into it. Later, the memory block is unlocked, the memory freed, and the heap destroyed:

```
// CA-Clipper include files
#include "extend.api"
#include "vm.api"

// Microsoft C include files
#include "string.h"

// Prototype
Boolean VMHeapExample(char * spSrc);

#define HEAP_SIZE 4096

Boolean VMHeapExample(char * spSrc)
{
    HANDLE hSegment;
    unsigned uiStringOffset;
    unsigned uiBufflen;
    char * spString;
    Boolean bResult = FALSE;

    if (hSegment = _xvheapnew(HEAP_SIZE))
       {
       uiBufflen = strlen(spSrc) + 1;
       uiStringOffset = _xvheapalloc(hSegment, uiBufflen);
       if (uiStringOffset)
          {
          spString = _xvheaplock(hSegment, uiStringOffset);
          if (spString != NULL)
             {
             strcpy(spString, spSrc);

             .
             . <statements>
             .

             bResult = TRUE;

             _xvheapunlock(hSegment, uiStringOffset);
             }
          _xvheapfree(hSegment, uiStringOffset);
          }
       _xvheapdestroy(hSegment);
       }

    return (bResult);
}
```

Files Library is CLIPPER.LIB, header file is Vm.api.

See Also _xvheapalloc(), _xvheaplock(), _xvheapnew()

_xvlock()

Lock a VM segment

C Prototype

```
#include "vm.api"
FARP _xvlock(
                HANDLE hSegment
            )
```

Arguments

hSegment is the VM segment handle returned by _xvalloc().

Returns

_xvlock() returns a far pointer to the base of the locked VM segment, or a NULL pointer if the VM segment handle is not valid (i.e., does not represent an allocated VM segment).

Description

_xvlock() guarantees that the VM segment is located in conventional memory and cannot be moved or swapped. The pointer returned is valid until the segment is unlocked using _xvunlock().

Note: A VM segment may be locked more than once. The VMM system maintains, for each segment, a lock count (_xvlockcount()) which is incremented each time you lock the segment and decremented each time you unlock the segment. A segment is not physically unlocked until its lock count is zero. You must eventually unlock all VM segments with _xvunlock().

Warning! *Do not leave a VM segment locked unnecessarily. Because all locked segments reside in conventional memory, the more of them you lock, the greater the chance of exhausting conventional memory. Therefore, always unlock a VM segment with _xvunlock() when you are not actively accessing it. Typically, this means locking and unlocking the same VM segment several times within one function call.*

If you have to lock a VM segment for an extended period of time, use _xvwire(). This function places the VM segment in an area of the swap space that least inhibits the VMM system.

Notes

- ***Error conditions:*** If there is not enough VM swap space available for the segment to be loaded into conventional memory, the system raises an internal error and halts.
- ***Garbage collection:*** The VMM system may take up to several seconds to lock a VM segment if locking it triggers garbage collection. Use _xvwire() for routines that require fast access to a buffer.

Examples

- This example allocates a segment with _xvalloc() and locks it using _xvlock(). After a string is copied into the locked segment, the segment is unlocked and freed (with _xvunlock() and _xvfree()):

```
// CA-Clipper include files
#include "extend.api"
#include "vm.api"

// Microsoft C include files
#include "string.h"

// Prototype
Boolean VMExample(char * spSrc);


Boolean VMExample(char * spSrc)
{
    HANDLE hSegment;
    char * spString;
    Boolean bResult = FALSE;

    if (hSegment = _xvalloc(strlen(spSrc) + 1, 0))
       {
       spString = _xvlock(hSegment);
       if (spString != NULL)
          {
          strcpy(spString, spSrc);

          .
          . <statements>
          .

          bResult = TRUE;

          _xvunlock(hSegment);
          }
       _xvfree(hSegment);
       }

    return (bResult);
}
```

Files Library is CLIPPER.LIB, header file is Vm.api.

See Also _xvalloc(), _xvlockcount(), _xvunlock(), _xvwire()

_xvlockcount()

Determine the number of locks on a VM segment

C Prototype

```
#include "vm.api"
USHORT _xvlockcount(
                        HANDLE hSegment
                    )
```

Arguments

hSegment is the VM segment handle returned by _xvalloc().

Returns

_xvlockcount() returns the number of locks on the VM segment.

Description

_xvlockcount() returns the number of locks active on the VM segment. If the lock count is zero, the segment is unlocked.

You can also use _xvlockcount() on segment heaps. Every locked block of allocated memory within the segment heap registers as one lock.

Examples

- This example resizes a segment heap if it is not currently locked:

```
// CA-Clipper include files
#include "extend.api"
#include "vm.api"

// Prototype
Boolean HeapResize(HANDLE hSegment, unsigned uiNewSize);


Boolean HeapResize(HANDLE hSegment, unsigned uiNewSize)
{
    Boolean bResult = FALSE;

    // Don't attempt to resize locked segment
    //
    if (_xvlockcount(hSegment) == 0)
       {
       if (_xvheapresize(hSegment, uiNewSize))
          bResult = TRUE;
       }

    return (bResult);
}
```

Files

Library is CLIPPER.LIB, header file is Vm.api.

See Also

_xvheaplock(), _xvlock(), _xvunlock(), _xvunwire(), _xvwire()

_xvrealloc()

Change the size of a VM segment

C Prototype

```
#include "vm.api"
HANDLE _xvrealloc(
                        HANDLE hSegment,
                        USHORT uiSize,
                        USHORT uiFlags
                    )
```

Arguments

hSegment is the VM segment handle returned by _xvalloc().

uiSize is the new VM segment size in bytes.

uiFlags is currently unused and must be set to zero.

Returns

If successful, _xvrealloc() returns the original VM segment handle; otherwise, it returns zero.

Description

_xvrealloc() resizes a previously allocated VM segment, shortening or lengthening the VM segment to match the specified size. If you shorten the VM segment, bytes at the end of the segment are lost.

If the resizing is unsuccessful (indicated by a return value of zero), the segment retains its original size.

Note: _xvrealloc() can resize locked segments, but the chance of success is lower because the function will be constrained by other locked segments that are currently in memory.

Warning! *Use _xvheapresize(), not _xvrealloc(), to resize a segment heap.*

Examples

- This example resizes a previously allocated segment:

```
// CA-Clipper include files
#include "vm.api"

#define VR_SHRANK     1
#define VR_GREW       2
#define VR_NOCHANGE   3
#define VR_ERROR      0

// Prototype
int VMResizeExample(HANDLE hSegment, unsigned uiNewSize);


int VMResizeExample(HANDLE hSegment, unsigned uiNewSize)
{
    int iResult = VR_ERROR;
    unsigned int uiOldSize;
    long lChange;

    uiOldSize = _xvsize(hSegment);

    // Attempt to resize and assign result
    //
    if (_xvrealloc(hSegment, uiNewSize, 0))
       {
       lChange = (_xvsize(hSegment) - uiOldSize);

       if (lChange > 0)
          iResult = VR_GREW;

       if (lChange == 0)
          iResult = VR_NOCHANGE;

       if (lChange < 0)
          iResult = VR_SHRANK;
       }

    return (iResult);
}
```

Files Library is CLIPPER.LIB, header file is Vm.api.

See Also _xvalloc(), _xvheapresize(), _xvsize()

_xvsize()

Determine the size of a VM segment

C Prototype

```
#include "vm.api"
USHORT _xvsize(
                    HANDLE hSegment
               )
```

Arguments

hSegment is the VM segment handle returned by _xvalloc().

Returns

_xvsize() returns the size of the VM segment, in bytes. The actual value returned is the original number of bytes specified in _xvalloc() that created the segment.

Description

_xvsize() returns the size of the VM segment. You can resize the segment if necessary using _xvrealloc().

Examples

- The following example resizes a previously allocated segment:

```
// CA-Clipper include files
#include "vm.api"

#define VR_SHRANK      1
#define VR_GREW        2
#define VR_NOCHANGE    3
#define VR_ERROR       0

// Prototype
int VMResizeExample(HANDLE hSegment, unsigned uiNewSize);

int VMResizeExample(HANDLE hSegment, unsigned uiNewSize)
{
   int iResult = VR_ERROR;
   unsigned int uiOldSize;
   long lChange;

   uiOldSize = _xvsize(hSegment);

   // Attempt to resize and assign result
   //
   if (_xvrealloc(hSegment, uiNewSize, 0))
      {
      lChange = (_xvsize(hSegment) - uiOldSize);

      if (lChange > 0)
         iResult = VR_GREW;

      if (lChange == 0)
         iResult = VR_NOCHANGE;

      if (lChange < 0)
         iResult = VR_SHRANK;
      }

   return (iResult);
}
```

Files Library is CLIPPER.LIB, header file is Vm.api.

See Also _xvalloc(), _xvlockcount(), _xvrealloc()

_xvunlock()

Unlock a VM segment

C Prototype

```
#include "vm.api"
void _xvunlock(
                    HANDLE hSegment
                )
```

Arguments

hSegment is the VM segment handle returned by _xvalloc().

Returns

_xvunlock() has no return value.

Description

_xvunlock() unlocks a VM segment locked by _xvlock() and invalidates any pointers to the segment. Unlocking the segment allows the VMM system to move or swap the segment if needed.

Note: If the segment's lock count (_xvlockcount()) is greater than one, the lock count is decremented but the segment remains locked. The segment will not be unlocked until its lock count is decremented to zero.

Warning! *After you unlock a VM segment, you should not attempt to access it with the far pointer returned by _xvlock() since there is no guarantee that the segment at that physical memory address is the segment you previously locked. If you need further access to the contents of the segment, lock it again and use the new pointer returned by _xvlock().*

Examples

- This example allocates a segment with _xvalloc() and locks it with _xvlock(). After a string is copied into the locked segment, the segment is unlocked and freed (with _xvunlock() and _xvfree()):

```
// CA-Clipper include files
#include "extend.api"
#include "vm.api"

// Microsoft C include files
#include "string.h"

// Prototype
Boolean VMExample(char * spSrc);


Boolean VMExample(char * spSrc)
{
    HANDLE hSegment;
    char * spString;
    Boolean bResult = FALSE;

    if (hSegment = _xvalloc(strlen(spSrc) + 1, 0))
       {
       spString = _xvlock(hSegment);
       if (spString != NULL)
          {
          strcpy(spString, spSrc);

          .
          . <statements>
          .

          bResult = TRUE;

          _xvunlock(hSegment);
          }
       _xvfree(hSegment);
       }

    return (bResult);
}
```

Files Library is CLIPPER.LIB, header file is Vm.api.

See Also _xvalloc(), _xvfree(), _xvlock(), _xvlockcount(), _xvunwire()

_xvunwire()

Unlock a wired VM segment

C Prototype

```
#include "vm.api"
void _xvunwire(
                    HANDLE hSegment
                )
```

Arguments

hSegment is the VM segment handle returned by _xvalloc().

Returns

_xvunwire() has no return value.

Description

_xvunwire() unlocks a VM segment locked by _xvwire() and invalidates any pointers to that segment. Unwiring a segment allows the VMM system to move and swap the segment as needed.

Note: If the segment's lock count (_xvlockcount()) is greater than one, the lock count is decremented but the segment remains locked. The segment will not be unlocked until its lock count is decremented to zero.

Warning! *After you unlock a VM segment, you should not attempt to access it with the far pointer returned by _xvwire() since there is no guarantee that the segment at that physical memory address is the segment you previously locked. If you need further access to the contents of the segment, lock it again and use the new pointer returned by _xvwire().*

Examples

- This example allocates a segment with _xvalloc() and locks it with _xvwire(). After a string is copied into the segment, it is unlocked and freed.

```
// CA-Clipper include files
#include "extend.api"
#include "vm.api"

// Microsoft C include files
#include "string.h"

// Prototypes
Boolean VMWireExSetup(char * spSrc);
void VMWireExExit(void);

static HANDLE hSegment;
static char * spString = NULL;

Boolean VMWireExSetup(char * spSrc)
{
    Boolean bResult = FALSE;

    if (hSegment = _xvalloc(strlen(spSrc) + 1, 0))
       {
       spString = _xvwire(hSegment);
       if (spString != NULL)
          {
          strcpy(spString, spSrc);

          bResult = TRUE;

          }
       else
          _xvfree(hSegment);
       }

    return (bResult);
}

void VMWireExExit(void)
{
    // Clean up if anything was allocated
    if (spString)
       {
       _xvunwire(hSegment);

       spString = NULL;

       _xvfree(hSegment);
       }

    return;
}
```

Files Library is CLIPPER.LIB, header file is Vm.api.

See Also _xvalloc(), _xvfree(), _xvlockcount(), _xvunlock(), _xvwire()

_xvwire()

Obtain a long-term lock on a VM segment

C Prototype

```
#include "vm.api"
FARP _xvwire(
                HANDLE hSegment
            )
```

Arguments

hSegment is the VM segment handle returned by _xvalloc().

Returns

_xvwire() returns a far pointer to the base of the locked VM segment, or a NULL pointer, if the segment pointer is not valid (i.e., does not represent an allocated segment).

Description

_xvwire() moves a segment to the low end of VM swap space and locks it, preventing the segment from being moved or swapped. Locking a VM segment allows access to it through a standard far pointer.

VM segments locked by _xvwire() must be unlocked by _xvunwire().

Warning! *Segments that you lock with _xvlock() may be placed in the middle of the swap space. Thus, if you hold the lock for a long period of time, the swap space can become fragmented, reducing the size of the largest contiguous memory block available.*

_xvwire(), on the other hand, moves the locked segment to the lowest possible position within the swap space. Thus, if you use _xvwire() to lock a segment for a long period of time, you avoid fragmentation.

Note: You may lock VM segment more than once. For each segment, the VMM system maintains a lock count (_xvlockcount()), which is incremented each time you lock the segment and decremented each time you unlock the segment. A segment is not physically unlocked until its lock count is zero.

Notes

- ***_xvwire() vs. _xvlock():*** The question of when to use _xvwire() and when to use _xvlock() is one of efficiency. _xvlock() is faster at locking segments, but you must unlock the segments relatively quickly or they will inhibit the VMM system.

  _xvwire() is slower at locking segments, but you can maintain the segment locks for a long period of time. This gives functions access to the wired segment without having to lock it.

  A good example of the need for _xvwire() is a serial communications buffer. The initialization routine is not affected by the extra time it takes to wire a segment, but the ISR for the communications port needs instant access to memory and, thus, cannot afford to wait for a segment to be locked.

- ***Error conditions:*** If the segment cannot be loaded into conventional memory because of insufficient VM swap space, the system raises an internal error and halts.

Examples

- This example allocates a segment with _xvalloc() and locks it with _xvwire(). After a string is copied into the segment, it is unlocked and freed.

```
// CA-Clipper include files
#include "extend.api"
#include "vm.api"

// Microsoft C include files
#include "string.h"

// Prototypes
Boolean VMWireExSetup(char * spSrc);
void VMWireExExit(void);

static HANDLE hSegment;
static char * spString = NULL;
```

```
Boolean VMWireExSetup(char * spSrc)
{
    Boolean bResult = FALSE;

    if (hSegment = _xvalloc(strlen(spSrc) + 1, 0))
       {
       spString = _xvwire(hSegment);
       if (spString != NULL)
          {
          strcpy(spString, spSrc);

          bResult = TRUE;

          }
       else
          _xvfree(hSegment);
       }

    return (bResult);
}

void VMWireExExit(void)
{
    // Clean up if anything was allocated
    if (spString)
       {
       _xvunwire(hSegment);

       spString = NULL;

       _xvfree(hSegment);
       }

    return;
}
```

Files Library is CLIPPER.LIB, header file is Vm.api.

See Also _xvalloc(), _xvfree(), _xvlock(), _xvlockcount(), _xvunwire()

Chapter 10
Error System Reference Listing

Chapter 10
Error System API Reference

The Error System API lets your Extend routines produce error "objects" that are compatible with the CA-Clipper Error system and call the current CA-Clipper error handler routine. This chapter is an alphabetical reference to all of the functions in the CA-Clipper Error System API.

Important! *The use of the term "object" in this reference simply refers to a piece of encapsulated data, not to the CA-Clipper object data type. The use of object-oriented terminology in this chapter is merely a convenience—the current implementation is not object-oriented. The Error System API does, however, echo the CA-Clipper predefined Error class.*

The prototypes for these functions are defined in the header file Error.api. The data types used in the prototypes are defined in the header file Clipdefs.h (automatically included by Error.api). Additional constants are defined in Error.ch. All of these header files are located in the \CLIP53\INCLUDE directory. The functions themselves are defined in CLIPPER.LIB, located in the \CLIP53\LIB directory.

Due to the extreme interdependency of the functions in this API, the examples in each function reference show only the basic syntax in a code fragment. An example of how all the functions work together is provided at the end of the chapter.

Warning! *Since the implementation can change in future versions, it is important to access this data via the public protocol presented in this chapter. Do not attempt to access the underlying data structure directly.*

_errGetDescription()

Get the value of *description*

C Prototype

```
#include "error.api"
BYTEP _errGetDescription(
                              ERRORP pError
                            )
```

Arguments

pError is a pointer to the Error object from which the value is to be retrieved.

Returns

_errGetDescription() returns a pointer to the string containing the error *description*.

Description

_errGetDescription() retrieves a pointer to the *description* string of an error. The *description* is a short error message that is displayed by the default CA-Clipper error handler routine to give the user an indication of the problem.

Note: This string may be empty if the *genCode* is defined in Error.ch (such as EG_OPEN). In this case, the proper *description* will automatically be supplied when the error is launched.

Examples

- This code fragment retrieves a pointer to the error *description*:

```
#include "error.api"
    .
    .
    .
BYTEP fpDescription;
fpDescription = _errGetDescription( pError );
    .
    .
    .
```

Files

Library is CLIPPER.LIB, header file is Error.api.

See Also

_errPutDescription()

_errGetFileName()

Get the *filename*

C Prototype

```
#include "error.api"
BYTEP _errGetFileName(
                        ERRORP pError
                      )
```

Arguments

pError is a pointer to the Error object from which the value is to be retrieved.

Returns

_errGetFileName() returns a pointer to the string containing the *filename*.

Description

_errGetFileName() retrieves a pointer to a string containing the name of the file being operated on, if any, when the error occurred. This value should be empty for operations that do not involve files.

Note: The _fsExtOpen() function automatically sets *filename* to the correct value, if an error occurs. See the "File System API Reference" chapter in this guide and the Usage Example at the end of this chapter for more information.

Examples

- This code fragment retrieves a pointer to the string *filename*:

```
#include "error.api"
    .
    .
    .
BYTEP fpFileName;
fpFileName = _errGetFileName( pError );
    .
    .
    .
```

Files

Library is CLIPPER.LIB, header file is Error.api.

See Also

_errPutFileName()

_errGetFlags()

Get the values of the error type *flags*

C Prototype

```
#include "error.api"
USHORT _errGetFlags(
                        ERRORP pError
                    )
```

Arguments

pError is a pointer to the Error object from which the value is to be retrieved.

Returns

_errGetFlags() returns the value of the error type *flags*.

Description

_errGetFlags() retrieves the current value of *flags*, which determines the actions that are allowed for this error. The value and meaning of the error type *flags* defined in Error.api are as follows:

Error flags Values

Constant	Value	Meaning
EF_CANRETRY	1	Advises the CA-Clipper error handler routine that the failed operation can be retried
EF_CANDEFAULT	4	Advises the CA-Clipper error handler routine that the failed operation can be safely ignored

Examples

- This code fragment retrieves the contents of *flags*:

```
#include "error.api"
    .
    .
    .
USHORT uiFlags;
uiFlags = _errGetFlags( pError );
    .
    .
    .
```

Files Library is CLIPPER.LIB, header file is Error.api.

See Also _errPutFlags()

_errGetGenCode()

Get the value of *genCode*

C Prototype

```
#include "error.api"
USHORT _errGetGenCode(
                        ERRORP pError
                      )
```

Arguments

pError is a pointer to the Error object from which the value is to be retrieved.

Returns

_errGetGenCode() returns a value indicating the general type of the operation taking place when the error occurred.

Description

_errGetGenCode() retrieves the value of *genCode* which represents the general category of operation taking place when the error occurred. The header file Error.ch contains many predefined *genCode* values that you may use to test for specific errors. See _errPutGenCode() for additional details about *genCode*.

Examples

- This code fragment retrieves the contents of *genCode*:

```
#include "error.api"
    .
    .
    .
    USHORT uiGenCode;
    uiGenCode = _errGetGenCode( pError );
    .
    .
    .
```

Files

Library is CLIPPER.LIB, header file is Error.api.

See Also

_errPutGenCode()

_errGetOperation()

Get the value of *operation*

C Prototype

```
#include "error.api"
BYTEP _errGetOperation(
                        ERRORP pError
                       )
```

Arguments

pError is a pointer to the Error object from which the value is to be retrieved.

Returns

_errGetOperation() returns a pointer to the string containing the *operation*.

Description

_errGetOperation() retrieves a pointer to the string indicating the *operation* at the time the error occurred. This is used to identify the CA-Clipper-level operation that failed, which might be an operator symbol (like "+" or "&") or a function name (like "STR()" or "TYPE()").

Note: The *operation* value that you, as a subsystem developer, use will normally be the name of your CA-Clipper-callable function.

Examples

- This code fragment retrieves a pointer to the string *operation*:

```
#include "error.api"
    .
    .
    .
    BYTEP fpOperation;
    fpOperation = _errGetOperation( pError );
    .
    .
    .
```

Files

Library is CLIPPER.LIB, header file is Error.api.

See Also

_errPutOperation()

_errGetOsCode()

Get the value of *osCode*

C Prototype

```
#include "error.api"
USHORT _errGetOsCode(
                      ERRORP pError
                      )
```

Arguments

pError is a pointer to the Error object from which the value is to be retrieved.

Returns

_errGetOsCode() returns the current value of *osCode*.

Description

_errGetOsCode() retrieves the value representing an operating system-specific error code. This code currently corresponds to the DOS error number since CA-Clipper presently will run only under the DOS operating system, a DOS window in Windows or OS/2, or a DOS emulator in other operating systems.

Examples

- This code fragment retrieves the contents of *osCode*:

```
#include "error.api"
    .
    .
    .
    USHORT uiOsCode;
    uiOsCode = _errGetOsCode( pError );
    .
    .
    .
```

Files

Library is CLIPPER.LIB, header file is Error.api.

See Also

_errPutOsCode()

_errGetSeverity()

Get the value of *severity*

C Prototype

```
#include "error.api"
USHORT _errGetSeverity(
                        ERRORP pError
                       )
```

Arguments

pError is a pointer to the Error object from which the value is to be retrieved.

Returns

_errGetSeverity() returns a flag indicating the severity of the error.

Description

_errGetSeverity() gets the value of *severity* in the Error object. You may use the *severity* level to generate warnings as well as errors.

Note: Manifest constants that you may use to set the *severity* are available in the Error.ch file and are shown in the _errPutSeverity() function reference.

Examples

- This code fragment retrieves the contents of *severity*:

```
#include "error.api"
    .
    .
    .
USHORT uiSeverity;
uiSeverity = _errGetSeverity( pError );
    .
    .
    .
```

Files

Library is CLIPPER.LIB, header file is Error.api.

See Also

_errPutSeverity()

_errGetSubCode()

Get the value of *subCode*

C Prototype

```
#include "error.api"
USHORT _errGetSubCode(
                        ERRORP pError
                      )
```

Arguments

pError is a pointer to the Error object from which the value is to be retrieved.

Returns

_errGetSubCode() returns the value of *subCode*.

Description

_errGetSubCode() retrieves a value that indicates the specific error number within the subsystem referenced by *subSystem*. The combination of the *subSystem* name and the *subCode* uniquely identifies a specific error. See _errPutSubCode() for additional details about *subCode*.

Examples

- This code fragment shows how to retrieve the contents of *subCode*:

```
#include "error.api"
    .
    .
    .
    USHORT uiSubCode;
    uiSubCode = _errGetSubCode( pError );
    .
    .
    .
```

Files

Library is CLIPPER.LIB, header file is Error.api.

See Also

_errGetSubSystem(), _errPutSubCode()

_errGetSubSystem()

Get the value of *subSystem*

C Prototype

```
#include "error.api"
BYTEP _errGetSubSystem(
                        ERRORP pError
                       )
```

Arguments

pError is a pointer to the Error object from which the value is to be retrieved.

Returns

_errGetSubSystem() returns a pointer to the string containing the *subSystem* name.

Description

_errGetSubSystem() retrieves a pointer to name of the *subSystem* in which the error occurred. The combination of the *subSystem* name and the *subCode* uniquely identifies a specific error. See _errPutSubSystem() for additional details about *subSystem*.

Examples

- This code fragment retrieves a pointer to the string *subSystem*:

```
#include "error.api"
    .
    .
    .
BYTEP fpSubSystem;
fpSubSystem = _errGetSubSystem( pError );
    .
    .
    .
```

Files

Library is CLIPPER.LIB, header file is Error.api.

See Also

_errGetSubCode(), _errPutSubSystem()

_errGetTries()

Get the value of the retry counter *tries*

C Prototype

```
#include "error.api"
USHORT _errGetTries(
                      ERRORP pError
                    )
```

Arguments

pError is a pointer to the Error object from which the value is to be retrieved.

Returns

_errGetTries() returns the value of *tries*.

Description

_errGetTries() retrieves the value of the *tries* counter, indicating how many times the operation has been tried. If the error can be rectified, *tries* is incremented each time the operation is retried. See _errPutTries() for additional details about the *tries* counter.

Examples

- This code fragment increments the retry counter *tries*:

```
#include "error.api"
    .
    .
    .
    _errPutTries( pError, _errGetTries( pError )++ );
    .
    .
    .
```

Files

Library is CLIPPER.LIB, header file is Error.api.

See Also

_errPutTries()

_errLaunch()

Launch the CA-Clipper error handler system

C Prototype

```
#include "error.api"
ERRCODE _errLaunch(
                     ERRORP pError
                   )
```

Arguments

pError is a pointer to a properly created and generated Error object.

Returns

_errLaunch() returns an error code indicating the action to take, or zero if no error occurred.

Description

_errLaunch() invokes the CA-Clipper error handler, using *pError* as an argument. This function handles errors that can *default* or *retry* (see _errPutFlags()).

The following table shows several manifest constants defined in Error.api that you may use to test the return value of this function. The error code returned indicates the value returned by the error handler. You should return an error code from each function in the subsystem so that you may test for an error after the function returns.

_errLaunch() Return Values

Constant	Value	Meaning	Action
E_BREAK	FFFFh	Error Handler issued a BREAK command or BREAK() function	Clean up and exit immediately
E_RETRY	1	Retry operation (occurs only if you specify EF_CANRETRY flag)	Retry the failed operation and increment the value of *tries*
E_DEFAULT	0	Ignore failure (occurs only if you specify EF_CANDEFAULT flag)	Ignore the failure, taking whatever default action is required for the subsystem to continue

Notice that E_DEFAULT is the same as the value for success (zero). E_DEFAULT literally means to pretend nothing happened and, therefore, the calling routine has no need to distinguish between E_DEFAULT and success. The most common use you will find for E_DEFAULT is to handle warnings. Never use E_DEFAULT if the error will affect the operation of any other part of the subsystem.

Examples

- This code fragment demonstrates launching an error and returning the error code to the calling function:

```
#include "error.api"

    .
    .   <an error occurs>
    .
    ERRORP pError;
    ERRCODE uiErrCode;

    pError = _errNew();
    .
    .   <put values in error>
    .
    uiErrCode = _errLaunch( pError );
    _errRelease( pError );

    return ( uiErrCode );
```


Files Library is CLIPPER.LIB, header file is Error.api.

See Also _errNew(), _errPutFlags(), _errPutTries(), _errRelease()

_errNew()

Create a new Error object

C Prototype

```
#include "error.api"
ERRORP _errNew(void)
```

Returns

_errNew() returns a pointer to a new Error object.

Description

_errNew() creates an empty Error object and returns a pointer to it. You may modify the variables within the object using the "_errPut" functions and inspect them using the "_errGet" functions. You may, then, send the object to the CA-Clipper error handler.

The values of the variables in the Error object created by _errNew() are compatible with the instance variables in CA-Clipper Error objects. This allows the use of the constants in the Error.ch include file.

Proper error handling is the responsibility of the subsystem programmer. The CA-Clipper API systems provide low-level services and normally provide only simple return values indicating success or failure. The exception is the _fsExtOpen() function from the File System API (demonstrated in the Usage Example at the end of this chapter), which can assign values to an Error object.

Note: When it is no longer required, release all memory allocated by the Error object using _errRelease().

Examples

- This code fragment demonstrates how to create an Error object:

```
#include "error.api"
    .
    .
    .
    ERRORP pError;

    pError = _errNew();
    .
    .
    .
    _errRelease( pError );
```

Files

Library is CLIPPER.LIB, header file is Error.api.

See Also

_errLaunch(), _errRelease()

_errPutDescription()

Set the value of *description*

C Prototype

```
#include "error.api"
ERRORP _errPutDescription(
                          ERRORP pError,
                          BYTEP  fpDescription
                         )
```

Arguments

pError is a pointer to the Error object that is to be set.

fpDescription is a pointer to a null-terminated string containing a short description of the specific error.

Returns

_errPutDescription() returns a pointer to the Error object (self).

Description

_errPutDescription() defines a short error *description* that is displayed by the default CA-Clipper error handler routine to give the user an indication of the problem. The message should correspond to the general problem (*genCode*), not the specific failure (*subCode*). "Open error" or "Argument Error" are typical *description* messages.

Note: If you use one of the *genCodes* defined in Error.ch (such as EG_OPEN), you do not have to supply a *description*. The proper *description* will be used automatically when the error is launched.

Examples

- This code fragment shows the setting of *description* (you should try to be more descriptive than the example):

```
#include "error.api"
    .
    .
    .
    _errPutDescription( pError, "Something happened" );
    uiErrCode = _errLaunch( pError );
    .
    .
    .
```

Files

Library is CLIPPER.LIB, header file is Error.api.

See Also

_errGetDescription(), _errPutGenCode(), _errPutSubCode()

_errPutFileName()

Set the *filename*

C Prototype

```
#include "error.api"
ERRORP _errPutFileName(
                        ERRORP pError,
                        BYTEP  fpFileName
                       )
```

Arguments

pError is a pointer to the Error object that is to be set.

fpFileName is a pointer to a null-terminated string containing the name of the file being accessed.

Returns

_errPutFileName() returns a pointer to the Error object (self).

Description

_errPutFileName() assigns a string to the error containing the name of the file being operated on, if any. This value has no meaning on operations that do not involve files.

Note: The _fsExtOpen() function automatically sets *filename* to the correct value if an error occurs. See the "File System API Reference" chapter in this guide and the Usage Example at the end of this chapter for more information.

Examples

- This code fragment shows the setting of *filename*:

```
#include "error.api"
    .
    .
    .
    _errPutFileName( pError, "Foo.txt" );
    uiErrCode = _errLaunch( pError );
    .
    .
    .
```

Files Library is CLIPPER.LIB, header file is Error.api.

See Also _errGetFileName()

_errPutFlags()

Set the value of the error type *flags*

C Prototype

```
#include "error.api"
ERRORP _errPutFlags(
                        ERRORP pError,
                        USHORT uiFlags
                      )
```

Arguments

pError is a pointer to the Error object that is to be set.

uiFlags contains a value indicating the allowable actions for this error.

Returns

_errPutFlags() returns a pointer to the Error object (self).

Description

_errPutFlags() assigns a value to *flags* to specify the actions that are allowed for this error. You, as the subsystem designer, are responsible for determining what actions are permissible, and the following general guidelines should help:

- If an error can be rectified by the user, you should allow a retry. For example, if a file open error occurs because the device is not ready (such as an open floppy drive door), the user can correct the problem and retry the operation.
- If the error is caused by something that cannot be remedied by the user, you will gain nothing by allowing a retry.
- If the error will not cause a failure in other parts of the system, such as the failure to set a screen color, you may safely ignore the error (the default behavior).
- You may also ignore the error if there is a valid alternative to the routine that failed. For example, if a routine results in an error because of failure to retrieve an environment variable that holds the user's name, you can execute a function to retrieve the name from the user before returning to the calling routine (which would have no idea that an error even occurred).

The value and meaning of the error type *flags* defined in Error.api are as follows:

Error flags Values

Constant	Value	Meaning
EF_CANRETRY	1	Advises the CA-Clipper Error Handler routine that the failed operation can be retried
EF_CANDEFAULT	4	Advises the CA-Clipper Error Handler routine that the failed operation can be safely ignored

Warning! *The most common use you will find for EF_CANDEFAULT is to handle warnings. Never use EF_CANDEFAULT if the error will affect the operation of any other part of the subsystem.*

Examples

- This code fragment shows the setting of *flags*:

```
#include "error.api"

    .
    .
    .
    _errPutFlags( pError, EF_CANRETRY );
    uiErrCode = _errLaunch( pError );
    .
    .
    .
```

Files

Library is CLIPPER.LIB, header file is Error.api.

See Also

_errGetFlags(), _errLaunch(), _errPutGenCode()

_errPutGenCode()

Set the value of *genCode*

C Prototype

```
#include "error.api"
ERRORP _errPutGenCode(
                        ERRORP pError,
                        USHORT uiGenCode
                      )
```

Arguments

pError is a pointer to the Error object that is to be set.

uiGenCode is a code specifying the general type of operation taking place when the error occurred.

Returns

_errPutGenCode() returns a pointer to the Error object (self).

Description

_errPutGenCode() assigns a value to *genCode,* indicating the general category of operation being performed when the error occurred. It is the *subSystem* and *subCode* combination that will identify your particular error among all the possible errors sharing a *genCode.*

The header file Error.ch contains many predefined *genCodes.* You should use these codes whenever possible to be consistent with other subsystems.

If you use a one of the predefined *genCodes* from Error.ch and do not supply a *description,* a predefined *description* will automatically be generated by the error handling system. Using the system supplied *description* has two advantages: the *description* will be consistent with other subsystems, and the supplied *description* in international versions will be translated into the target language.

Examples

- This code fragment shows the setting of *genCode*:

```
#include "error.api"
#include "error.ch"

    .
    .
    .
    _errPutGenCode( pError, EG_OPEN );
    uiErrCode := _errLaunch( pError );
    .
    .
    .
```

Files Library is CLIPPER.LIB, header file is Error.api.

See Also _errGetGenCode(), _errPutDescription(), _errPutSubCode(), _errPutSubSystem()

_errPutOperation()

Set the value of *operation*

C Prototype

```
#include "error.api"
ERRORP _errPutOperation(
                              ERRORP pError,
                              BYTEP fpOperation
                            )
```

Arguments

pError is a pointer to the Error object that is to be set.

fpOperation is a pointer to a null-terminated string containing the operation in progress when the error occurred.

Returns

_errPutOperation() returns a pointer to the Error object (self).

Description

_errPutOperation() assigns a string to the error, indicating the current *operation*. This is used to identify the CA-Clipper-level operation that failed which might be an operator symbol (like "+" or "&") or a function name (like "STR()" or "TYPE()").

Note: The *operation* value that you, as a subsystem developer, use will normally be the name of your CA-Clipper-callable function. All standard operators are handled automatically by CA-Clipper.

Examples

- This code fragment shows the setting of *operation* and assumes that the CA-Clipper-callable function name is OClone():

```
#include "error.api"

    .
    .  <error occurs cloning the object>
    .
    _errPutOperation( pError, "OCLONE()" );
    uiErrCode = _errLaunch( pError );
    .
    .
    .
```

Files

Library is CLIPPER.LIB, header file is Error.api.

See Also

_errGetOperation()

_errPutOsCode()

Set the value of *osCode*

C Prototype

```
#include "error.api"
ERRORP _errPutOsCode(
                          ERRORP pError,
                          USHORT uiOsCode
                        )
```

Arguments

pError is a pointer to the Error object that is to be set.

uiOsCode is the operating system error code.

Returns

_errPutOsCode() returns a pointer to the Error object (self).

Description

_errPutOsCode() sets the value of the operating system-specific error code. This code corresponds to the DOS error number since current CA-Clipper versions run only under the DOS operating system, a DOS window in Windows or OS/2, or a DOS emulator in other operating systems.

Examples

- This code fragment shows the setting of *osCode*:

```
#include "error.api"
#include "filesys.api"
    .
    .  <a file related error>
    .
    _errPutOsCode( pError, _fsError() );
    uiErrCode = _errLaunch( pError );
    .
    .
    .
```

Files

Library is CLIPPER.LIB, header file is Error.api.

See Also

_errGetOsCode()

_errPutSeverity()

Set the value of *severity*

C Prototype

```
#include "error.api"
ERRORP _errPutSeverity(
                        ERRORP pError,
                        USHORT uiSeverity
                       )
```

Arguments

pError is a pointer to the Error object that is to be set.

uiSeverity is a flag indicating the severity of the error.

Returns

_errPutSeverity() returns a pointer to the Error object (self).

Description

_errPutSeverity() sets the value of *severity* in the Error object. You may use the *severity* level to generate warnings as well as errors.

Manifest constants that you may use to set the *severity* are available in the Error.ch file and are shown below:

Error severity Values

Constant	Value	Meaning	Usage
ES_WHOCARES	0	Undefined	An undefined value that can be used to send an informative message
ES_WARNING	1	Warning	A noncritical anomaly that should allow DEFAULT, as well as RETRY, if appropriate
ES_ERROR	2	Error	Should only allow DEFAULT if the consequences of ignoring the error are well known. Should allow RETRY, if appropriate.
ES_CATASTROPHIC	3	Severe Error	Catastrophic error value; should not allow any action other than application shutdown

Examples

- This code fragment shows the setting of *severity*:

```
#include "error.api"
#include "error.ch"

    .
    .
    .
    _errPutSeverity( pError, ES_ERROR );
    uiErrCode = _errLaunch( pError );
    .
    .
    .
```

Files Library is CLIPPER.LIB, header file is Error.api.

See Also _errGetSeverity(), _errLaunch(), _errPutFlags()

_errPutSubCode()

Set the value of *subCode*

C Prototype

```
#include "error.api"
ERRORP _errPutSubCode(
                          ERRORP pError,
                          USHORT uiSubCode
                        )
```

Arguments

pError is a pointer to the Error object that is to be set.

uiSubCode is a value indicating the specific error number within the subsystem referenced by *subSystem.*

Returns

_errPutSubCode() returns a pointer to the Error object (self).

Description

_errPutSubCode() allows the assignment of a *subCode* to an error. It is the combination of the *subSystem* name and the *subCode* that uniquely identifies a specific error.

You should make *subCodes* consistent between subsystems that are identical in functionality. For example, the CA-Clipper DBFNTX and DBFNDX drivers share the same *subCodes* wherever possible. Designers of RDD subsystems should attempt to use the same *subCodes* as the CA-Clipper RDDs in order to provide consistency to the end user.

Examples

- This code fragment shows the setting of *subCode*:

```
#include "error.api"

    .
    .
    .
    _errPutSubCode( pError, 1210 );
    uiErrCode = _errLaunch( pError );
    .
    .
    .
```

Files

Library is CLIPPER.LIB, header file is Error.api.

See Also

_errGetSubCode(), _errPutSubSystem()

_errPutSubSystem()

Set the value of *subSystem*

C Prototype

```
#include "error.api"
ERRORP _errPutSubSystem(
                              ERRORP pError,
                              BYTEP fpSubSystem
                            )
```

Arguments

pError is a pointer to the Error object that is to be set.

fpSubSystem is a pointer to a null-terminated string indicating the subsystem in which the error occurred.

Returns

_errPutSubSystem() returns a pointer to the Error object (self).

Description

_errPutSubSystem() allows the assignment of the *subSystem* name to an error. It is the combination of *subSystem* name and the *subCode* that uniquely identifies a specific error.

You should make *subCodes* consistent between subsystems that are identical in functionality. For example, the CA-Clipper DBFNTX and DBFNDX share the same *subCodes* wherever possible. Designers of RDD subsystems should attempt to use the same *subCodes* as the CA-Clipper RDDs in order to provide consistency to the end user.

Examples

- This code fragment shows the setting of *subSystem*:

```
#include "error.api"

    .
    .
    .
    _errPutSubSystem( pError, "MYDRIVER" );
    uiErrCode = _errLaunch( pError );
    .
    .
    .
```

Files

Library is CLIPPER.LIB, header file is Error.api.

See Also

_errGetSubSystem(), _errPutSubCode()

_errPutTries()

Set the value of the counter *tries*

C Prototype

```
#include "error.api"
ERRORP _errPutTries(
                      ERRORP pError,
                      USHORT uiTries
                    )
```

Arguments

pError is a pointer to the Error object that is to be set.

uiTries is a counter that indicates how many times an operation has been tried.

Returns

_errPutTries() returns a pointer to the Error object (self).

Description

_errPutTries() sets the value of the counter *tries*, indicating how many times the operation has been tried. If you designate an operation as retriable, you should increment *tries* each time the error occurs before calling _errLaunch(). This allows the error handler to determine how many times to allow a retry of an operation before finally giving up.

Note: The _fsExtOpen() function automatically updates the *tries* counter each time it is executed. See the "File System API Reference" chapter in this guide for more information.

Examples

- This code fragment increments the *tries* counter:

```
#include "error.api"

    .
    .
    .
    _errPutTries( pError, _errGetTries( pError )++ );
    uiErrCode = _errLaunch( pError );
    .
    .
    .
```

Files

Library is CLIPPER.LIB, header file is Error.api.

See Also

_errGetTries(), _errLaunch(), _errPutFlags()

_errRelease()

Destroy an Error object

C Prototype

```
#include "error.api"
void _errRelease(
                    ERRORP pError
                  )
```

Arguments

pError is a pointer to the Error object to be destroyed.

Returns

_errRelease() has no return value.

Description

_errRelease() destroys the Error object *pError* and releases the memory used by it.

Warning! *_errRelease() does not release memory referenced by the pointers it contains. You as the subsystem programmer must free this memory as necessary.*

Examples

- This code fragment shows the destruction of an Error object:

```
#include "error.api"
   ERRORP pError;
   pError = _errNew();
   .
   .
   .
   _errRelease( pError );
```

Files

Library is CLIPPER.LIB, header file is Error.api.

See Also

_errLaunch(), _errNew()

Usage Example

- This example demonstrates how to handle a retriable error by producing, launching, and destroying an Error object.

  The first function is a CA-Clipper-callable interface function which calls a second C routine to actually open the file.

  The isolated C function opens the file, retrying the operation as many times as the CA-Clipper error handler indicates. The example uses the function _fsExtOpen() to perform the low-level file I/O (See the "File System API Reference" chapter in this guide for more information).

```
/***
*   FILEOPEN()
*
*   CA-Clipper-callable function that attempts to open a file
*   for reading and writing using FileOpener() and returns the
*   file handle.  The first parameter is the name of the file to
*   open and the second is a logical value indicating if the
*   file should be opened shared (default is false).  If the
*   filename is passed by reference it will be changed to the
*   fully qualified name of the file if the open is successful,
*   or the name that was attempted if unsuccessful.
*
*   Note: the subsystem code is set in this function since the
*   lower-level function FileOpener() is designed to be called
*   from many different routines.
*/

#include "extend.api"

CLIPPER FILEOPEN()
{

    BYTEP fpFileName;
    FHANDLE hFile = FS_ERROR;
    USHORT uiFlags = FO_READWRITE
    ERRORP pError;

    if ( ISCHAR( 1 ) )
    {
       fpFileName = _parc( 1 );
       uiFlags |= ( ISLOG( 2 ) && _parl( 2 ) )
         ? FO_SHARED : FO_EXCLUSIVE;

       pError = _errNew()

       _errPutSubCode( pError, 1111 ); // subCode determined by
                                       // caller
```

```
      hFile = FileOpener( fpFileName, uiFlags, pError );

      if ( ISBYREF( 1 ) )
         _storc( fpFileName, 1 );

   {

   _retni( hFile );
}

/***
*   FileOpener()
*
*   Attempts to open a file whose name is passed as an argument
*   along with the open mode flags and a pointer to an allocated
*   Error object.  A default extension of ".TXT" will be assumed
*   if the file name does not include an extension.
*
*   The function will generate an error if the open fails and
*   will continue to retry for as long as the Error Handler
*   returns E_RETRY.
*
*   The return value will be a valid file handle if successful,
*   otherwise FS_ERROR is returned.
*/

#include "error.api"
#include "error.ch"
#include "filesys.api"

FHANDLE FileOpener( BYTEP fpFileName, USHORT uiFlags,
                    ERRORP pError )
{
   FHANDLE hHandle;
   BOOL    retry;

   _errPutSeverity( pError, ES_ERROR );    // we determine the
                                           // severity
   _errPutFlags( pError, EF_CANRETRY );    // and the allowable
                                           // action
   _errPutGenCode( pError, EG_OPEN );      // open error

   // will be modified by _fsExtOpen() to the opened file name
   _errPutFileName( pError, fpFileName )

   do
   {
      hHandle = _fsExtOpen( fpFileName, ".TXT", uiFlags,
         NULL, pError );

      retry = FALSE;

      if (hHandle == FS_ERROR)
      {
         retry = ( _errLaunch( pError) == E_RETRY );
      }

   } while (retry);

   fpFileName = _errGetFileName( pError );

   return (handle);
}
```

Chapter 11
File System Reference Listing

Chapter 11
File System API Reference

The File System (Filesys) API gives your Extend routines access to CA-Clipper's low-level file routines. This allows you to create, access, and delete files without the added overhead of the C library I/O functions or custom Assembler routines. This chapter is an alphabetical reference to all of the functions in the CA-Clipper File System API.

The prototypes for these functions are defined in the header file Filesys.api, located in the \CLIP53\INCLUDE directory. The data types used in the function prototypes are defined in the header file Clipdefs.h, also located in the \CLIP53\INCLUDE directory. The functions themselves are defined in CLIPPER.LIB, located in the \CLIP53\LIB directory.

Note: CA-Clipper uses the Microsoft C large model calling conventions.

_fsChDir()

Change the current DOS directory

C Prototype

```
#include "filesys.api"
BOOL _fsChDir(
                BYTEP fpDirName
              )
```

Arguments

fpDirName is the name of the directory to change to, including an optional drive. If you do not specify a drive, the current one is used.

Returns

_fsChDir() returns true (.T.) if successful; otherwise, it returns false (.F.).

Description

This function changes the current DOS directory. This function may also be used to determine whether or not a directory exists. It is equivalent to the CA-Clipper function DIRCHANGE(). If an error occurs, _fsError() will return the number of the error.

Examples

- The following code fragment illustrates the use of _fsChDir():

```
#include "filesys.api"
.
.
.
if (! _fsChDir( "c:\dos" ) )
{
   // Report the error condition using _fsError()
}
```

- You may also use something like this:

```
_fsChDir( "..\..\test" )
```

Files

Library is CLIPPER.LIB, header file is Filesys.api.

See Also

_fsError()

_fsChDrv()

Change the current DOS disk drive

C Prototype

```
#include "filesys.api"
USHORT _fsChDrv(
                BYTE nDrive
               )
```

Arguments

nDrive is the number of the disk drive that is to be the current disk drive.

Returns

_fsChDrv() returns zero if successful; otherwise, FS_ERROR.

Description

This function changes the current DOS drive. It is equivalent to the CA-Clipper function DISKCHANGE().

Examples

- The following code fragment illustrates the use of _fsChDrv():

```
#include "filesys.api"
.
.
.
BYTE cDrv = 'd';
if ( _fsChDrv( (BYTE) ( cDrv - 'a' ) ) )
{
   // Report the error condition
}
```

Files

Library is CLIPPER.LIB, header file is Filesys.api.

_fsClose()

Close a file

C Prototype

```
#include "filesys.api"
void _fsClose(
                FHANDLE hFileHandle
              )
```

Arguments

hFileHandle is a valid DOS reference to the file to be closed.

Returns

_fsClose() has no return value.

Description

This function closes the file indicated by *hFileHandle*. It is equivalent to the CA-Clipper function FCLOSE(). Use the _fsError() function to determine if the file close operation resulted in an error.

Examples

- The following code fragment illustrates the use of _fsClose(). Note that error handling should be performed at each step. Refer to the "Error System API Reference" chapter of this guide for details on communicating with the CA-Clipper Error system:

```
#include "filesys.api"

FHANDLE hFile;

hFile = _fsOpen( "Foo", FO_READWRITE | FO_EXCLUSIVE );

if (! _fsError() )
{
    .
    .
    .
    _fsClose( hFile );
}
```

Files

Library is CLIPPER.LIB, header file is Filesys.api.

See Also

_fsCreate(), _fsError(), _fsOpen(), _fsRead(), _fsSeek(), _fsWrite()

_fsCommit()

Flush a buffer to disk

C Prototype

```
#include "filesys.api"
void _fsCommit(
                    FHANDLE hFileHandle
                )
```

Arguments

hFileHandle is a valid DOS reference to the file to be committed to disk.

Returns

_fsCommit() has no return value.

Description

This function flushes the file buffer of the file whose handle is specified by *hFileHandle*. It is equivalent to the CA-Clipper function DBCOMMIT().

Examples

- The following code fragment illustrates the use of _fsCommit():

```
#include "filesys.api"

FHANDLE hFile;

// Open a file for reading and writing
hFile = _fsOpen( "Foo", FO_READWRITE | FO_EXCLUSIVE );

if (! _fsError() )
{
    .
    .
    .
    _fsCommit( hFile );  // Flush buffers to disk
    .
    .
    .
}
```

Files

Library is CLIPPER.LIB, header file is Filesys.api.

See Also

_fsClose()

_fsCreate()

Create a file

C Prototype

```
#include "filesys.api"
FHANDLE _fsCreate(
                        BYTEP fpFilename,
                        USHORT uiAttribute
                    )
```

Arguments

fpFilename is the name of the file to create, specified as a null-terminated string. The *fpFilename* must be fully specified, including the drive letter, path, and extension.

uiAttribute is one of the DOS archive attributes shown in the table below:

File Create Flags

Constant	Flag	Description
FC_NORMAL	0x0000	No file attributes are set
FC_READONLY	0x0001	Read-only file attribute is set
FC_HIDDEN	0x0002	Hidden file attribute is set
FC_SYSTEM	0x0004	System file attribute is set

Returns

_fsCreate() returns a DOS handle to the newly created file, or FS_ERROR, if an error occurs.

Description

This function creates a file named *fpFilename* according to the value of *uiAttribute*. It is equivalent to the CA-Clipper function FCREATE().

Examples

- The following code fragment illustrates the use of _fsCreate(). Note that error handling should be performed at each step. Refer to the "Error System API Reference" chapter of this guide for details on communicating with the CA-Clipper Error system:

```
#include "filesys.api"

FHANDLE hFile;

// Create a normal read/write file
hFile = _fsCreate( "Foo", FC_NORMAL );

if (! _fsError() )
{
    .
    .
    .
    _fsClose( hFile );
}
```

Files Library is CLIPPER.LIB, header file is Filesys.api.

See Also _fsClose(), _fsError(), _fsOpen()

_fsCurDir()

Return a name of the current directory

C Prototype

```
#include "filesys.api"
BYTEP _fsCurDir(
                    USHORT uiDrive
                )
```

Arguments

uiDrive is the number of the disk drive to check, where 0 = current drive, 1 = A, 2 = B, 3 = C, etc.

Returns

_fsCurDir() returns a name of the current directory. If an error occurs, or the current directory of the specified drive is the root directory, _fsCurDir() returns a null string ("").

Description

This function gets a name of the current directory on a specified disk drive.

Examples

- The following code fragment illustrates the use of _fsCurDir():

```
#include "filesys.api"
.
.
.
BYTEP cDir;
cDir = _fsCurDir( 0 )      // Checks the current drive
```

Files

Library is CLIPPER.LIB, header file is Filesys.api.

See Also

_fsError()

_fsCurDrv()

Return the current DOS drive

C Prototype

```
#include "filesys.api"
BYTE _fsCurDrv(void)
```

Returns

_fsCurDrv() returns the current DOS drive code, where 0 = A, 1 = B, 2 = C, etc.

Description

This function gets the current DOS drive code. Please refer to the documentation of the CA-Clipper DISKNAME() function for more information.

Examples

- The following code fragment illustrates the use of _fsCurDrv():

```
#include "filesys.api"
.
.
.
BYTE cDrvName;
cDrvName = 'a' + _fsCurDrv();
```

Files

Library is CLIPPER.LIB, header file is Filesys.api.

_fsDelete()

Delete a file

C Prototype

```
#include "filesys.api"
void _fsDelete(
                        BYTEP fpFilename
                    )
```

Arguments

fpFilename is the name of the file to delete specified as a null-terminated string. The *fpFilename* must be fully specified, including the drive letter, path, and extension.

Returns

_fsDelete() has no return value.

Description

This function deletes the file *fpFilename*. It is equivalent to the CA-Clipper function FERASE(). Use the _fsError() function to determine if the file delete operation resulted in an error.

Examples

- The following code fragment illustrates the use of _fsDelete(). Note that error handling should be performed at each step. Refer to the "Error System API Reference" chapter of this guide for details on communicating with the CA-Clipper Error system:

```
#include "filesys.api"

FHANDLE hFile;

// Create a temporary read/write file
hFile = _fsCreate( "Temp", FC_NORMAL );

if (! _fsError() )
{
    .
    .
    .
    _fsClose( hFile );  // Can't delete an open file
    _fsDelete( "Temp" );
}
```

Files

Library is CLIPPER.LIB, header file is Filesys.api.

See Also

_fsError(), _fsRename()

_fsError()

Return the number of the last DOS error

C Prototype

```
#include "filesys.api"
USHORT _fsError(void)
```

Returns

_fsError() returns the DOS error code (summarized in the table below) for the last Filesys function. If there is no error, _fsError() returns zero.

_fsError() Return Values

Error	Meaning
0	Successful
2	File not found
3	Path not found
4	Too many files open
5	Access denied
6	Invalid handle
8	Insufficient memory
15	Invalid drive specified
19	Attempted to write to a write-protected disk
21	Drive not ready
23	Data CRC error
29	Write fault
30	Read fault
32	Sharing violation
33	Lock Violation

Description

This function returns the last DOS error that occurred. Each of the Filesys functions saves its error status to a static variable that is retrieved by this function. This function is equivalent to the CA-Clipper function FERROR().

Examples

- The following code fragment illustrates the use of _fsError() to determine if the last operation resulted in an error:

```
#include "filesys.api"

FHANDLE hFile;

hFile = _fsCreate( "Foo", FC_NORMAL );

// Check for an open error before using the file
if (! _fsError() )
{
    .
    .
    .
    _fsClose( hFile );
}
```

Files Library is CLIPPER.LIB, header file is Filesys.api.

See Also _fsClose(), _fsCreate(), _fsDelete(), _fsOpen(), _fsRead(), _fsRename(), _fsWrite()

_fsExtOpen()

Extended file open

C Prototype

```
#include "filesys.api"
FHANDLE _fsExtOpen(
                        BYTEP fpFilename,
                        BYTEP fpDefExt,
                        USHORT uiFlags,
                        BYTEP fpPaths,
                        ERRORP pError
                    )
```

Arguments

fpFilename is the name of the file to open, specified as a null-terminated string. It may include the path and file extension.

fpDefExt is the default extension to use, which must include the (.) extension separator.

uiFlags are the open mode flags, which may consist of both extended open mode and normal open mode flags.

fpPaths are the paths to be searched when opening *fpFilename*.

pError is a pointer to an Error object created via the _errNew() function (see the "Error System API Reference" chapter of this guide).

Returns

_fsExtOpen() returns a DOS handle to the newly created file or FS_ERROR if an error occurs.

Description

This function opens the file specified by *fpFilename*. If *fpFilename* does not include a file extension, the default extension *fpDefExt*, if supplied, will be used.

Note: If an extension is supplied, it must contain the (.) extension separator (i.e., ".txt").

The value of the specified *uiFlags* determines the mode in which the file is opened. In addition to the open mode flags shown in the _fsOpen() entry, *uiFlags* also accommodates the extended mode flags shown in the following table. To specify multiple flags, combine them with the bitwise OR operator (e.g., FXO_DEFAULT | FO_EXCLUSIVE | FO_INHERITED).

Extended Mode Flags

Constant	Flag	Description
FXO_TRUNCATE	0x0100	Create (truncate file if it exists)
FXO_APPEND	0x0200	Create (append to file if it exists)
FXO_FORCEEXT	0x0800	Force default extension
FXO_DEFAULTS	0x1000	Use SET command defaults
FXO_DEVICERAW	0x2000	Open devices in raw mode

FXO_TRUNCATE creates the file *fpFilename*, unless it already exists. If it already exists, it is truncated to zero-length.

FXO_APPEND opens *fpFilename*, if it exists; otherwise it creates *fpFilename*. If the file is successfully opened, and the file is not empty, the file pointer is positioned to the last data byte in the file. This file positioning only applies to files, not devices.

Note: Files created by either FXO_TRUNCATE or FXO_APPEND are created using the FC_NORMAL attribute (see _fsCreate()).

If *fpFilename* includes a path, no other path is ever tried. If *fpFilename* does not include a path, the file will be searched for, based on the following rules:

- FXO_DEFAULTS, if specified when opening a file, causes _fsExtOpen() to search the paths specified by SET DEFAULT and SET PATH. Paths specified by the *fpPaths* argument are ignored.

  If specified when creating a file, FXO_DEFAULTS causes _fsExtOpen() to create the file in the path specified by SET DEFAULT.

- If FXO_DEFAULTS is not specified, _fsExtOpen() uses the *fpPaths* argument (if not NULL) as a search path for opening files. For creating files, the *fpFilename* is created using the current drive/directory.

- FXO_FORCEEXT causes the *fpDefExt* argument (if not NULL) to be used as the file extension, even if *fpFilename* includes an extension. Otherwise, the extension specified by *fpDefExt* is used, only if *fpFilename* does not include an extension.

- FXO_DEVICERAW sets the device referenced by *fpFilename* to raw (binary) mode. This allows low-order or high-order characters to be passed to devices. This flag affects only devices—it has no effect on files.

The pointer *pError*, if not NULL, must point to a properly created error object. The Error object's *filename* member (if not NULL) must contain a buffer of at least 80 bytes. If *pError* is not NULL, _fsExtOpen will assign values to the Error object as follows:

- If _fsExtOpen() is successful, the fully qualified file specification (i.e., including the drive, path, file name, and extension) will be copied into the buffer pointed to by the Error object's *filename*.

- If _fsExtOpen() is not successful, the file name specifying the file that it attempted to open will be copied into the buffer pointed to by the Error object's *filename*. The *filename* may or may not be fully qualified depending on the path determination rules above. Also, *osCode* will be set to the appropriate DOS error number and *genCode* will be set to EG_CREATE if the operation was an FXO_TRUNCATE, or EG_OPEN if the operation was an FXO_APPEND.

Warning! *Failure to provide a buffer of at least 80 bytes of memory will result in a memory overwrite, which will likely cause major system problems.*

Examples

- The following example illustrates _fsExtOpen(). Screen output is performed using the GT subsystem. See the "General Terminal API Reference" chapter of this guide for more information:

```
#include "filesys.api"
#include "error.api"

#define FILE_LEN 80


CLIPPER TEST( void )
{
   USHORT  uiFlags;
   BYTE    buff[FILE_LEN];
   FHANDLE hFileHandle;
   ERRORP  pError;
   ERRCODE uiError = 0;  // Initialize to 0 (no error)

   pError = _errNew();
   _errPutFileName( pError, buff );

   /*
       uiFlags:    access          = FO_READWRITE
                   sharing         = FO_EXCLUSIVE
                   inheritance     = FO_PRIVATE
                   extended        = FXO_DEFAULTS
   */

   uiFlags = FO_READWRITE | FO_EXCLUSIVE | FO_PRIVATE |
             FXO_DEFAULTS;

   hFileHandle = _fsExtOpen(
             "Test",  /* File name will use default extension */
             ".txt",  /* Default extension to use             */
             uiFlags, /* Open mode flags shown above          */
             NULL,    /* Path: ignored with FXO_DEFAULTS      */
             pError   /* Full file spec returned              */
              );

   if (hFileHandle == FS_ERROR)
   {
       uiError = _errLaunch( pError );
   }

   else
   {
       _gtWriteCon( "Opened file: ", 13 );  /* Display status */
       _gtWriteCon( buff, strlen( buff ) ); /* using GT API   */
       _gtWriteCon( "\r\n", 2 );
       _retni( hFileHandle );
   }

   _errRelease( pError );
   _errPost( uiError );
}
```

Files Library is CLIPPER.LIB, header file is Filesys.api.

See Also _fsCreate(), _fsOpen()

_fsIsDrv()

Check if a disk drive is available

C Prototype

```
#include "filesys.api"
USHORT _fsIsDrv(
                  BYTE nDrive
                )
```

Arguments

nDrive is the number of the disk drive to check.

Returns

_fsIsDrv() returns zero if successful; otherwise, FS_ERROR.

Description

This function checks if a disk drive is available. It is equivalent to the CA-Clipper function ISDISK().

Examples

- The following code fragment illustrates the use of _fsIsDrv():

```
#include "filesys.api"
.
.
.
BYTEP AllDrives()
{
    BYTE i, nDrives, cDrives[27];

    nDrives = 0;
    for ( i = 0, i < 26, i++)
    {
       if(! _fsIsDrv( i ) )
          cDrives[nDrives++] = 'a' + i;
    }
    cDrives[nDrives++] = '\0';
    return( cDrives );
}
```

Files

Library is CLIPPER.LIB, header file is Filesys.api.

_fsLock()

Lock or unlock a portion of a file

C Prototype

```
#include "filesys.api"
BOOL _fsLock(
                FHANDLE hFileHandle,
                ULONG ulStart,
                ULONG ulLength,
                USHORT uiMode
            )
```

Arguments

hFileHandle is a valid DOS reference to the file to be locked or unlocked.

ulStart is the offset from the start of the file that specifies the starting location of the portion of the file to be locked or unlocked.

ulLength is the number of bytes to lock or unlock.

uiMode is a flag that specifies whether the file is to be locked or unlocked. The following constants are defined for *uiMode*:

File Lock Flags

Constant	Flag	Description
FL_LOCK	0x0000	Lock the region
FL_UNLOCK	0x0001	Unlock the region

Returns

_fsLock() returns a boolean value indicating success or failure of the specified operation.

Description

This function locks or unlocks a region of the file whose handle is specified by *hFileHandle*. The region starts at offset *ulStart* and continues for *ulLength* bytes. The flag *uiMode* specifies whether the area is to be locked or unlocked. This function has no CA-Clipper equivalent.

Examples

- The following code fragment illustrates the use of _fsLock() to both lock and unlock a part of the file:

```
#include "filesys.api"

FHANDLE hFile;

hFile = _fsCreate( "Foo", FC_NORMAL );

// Check for an open error before using the file
if (! _fsError() )
{
    if (_fsLock( hFile, 0, 1, FL_LOCK ) ) // Lock first byte
    {
        .
        .
        .
        _fsLock( hFile, 0, 1, FL_UNLOCK ); // Unlock first byte
    }
    .
    .
    .
        _fsClose( hFile );
}
```

Files

Library is CLIPPER.LIB, header file is Filesys.api.

_fsMkDir()

Create a directory

C Prototype

```
#include "filesys.api"
BOOL _fsMkDir(
                  BYTEP fpDirName
              )
```

Arguments

fpDirName is the name of the directory to create, including an optional drive. If you do not specify a drive, the current one is used.

Returns

_fsMkDir() returns true (.T.) if successful; otherwise, it returns false (.F.).

Description

This function creates a specified directory. You must have sufficient rights to create a directory. To create nested subdirectories, you must create each subdirectory separately, starting from the top-level directory that you want to create. This function is equivalent to the CA-Clipper function DIRMAKE(). If an error occurs, _fsError() will return the number of the DOS error.

Examples

- The following code fragment illustrates the use of _fsMkDir():

```
#include "filesys.api"
.
.
.
if (! _fsMkDir( "c:\test" ) )
{
   // Report the error condition using _fsError()
}
```

- You may also use something like this:

```
_fsMkDir( "..\test" )
```

Files

Library is CLIPPER.LIB, header file is Filesys.api.

See Also

_fsError()

_fsOpen()

Open a file

C Prototype

```
#include "filesys.api"
FHANDLE _fsOpen(
                    BYTEP fpFilename,
                    USHORT uiFlags
                )
```

Arguments

fpFilename is the name of the file to open specified as a null-terminated string. The *fpFilename* must be fully qualified, including the drive letter, path, and extension.

uiFlags contains the file open attribute bits that indicate how the file is to be opened. The following constants are defined for *uiFlags*:

Access Flags

Constant	Flag	Description
FO_READ	0x0000	File is opened for reading
FO_WRITE	0x0001	File is opened for writing
FO_READWRITE	0x0002	File is opened for reading and writing

Sharing Flags

Constant	Flag	Description
FO_COMPAT	0x0000	No sharing specified
FO_COMPAT	0x0000	No sharing specified
FO_EXCLUSIVE	0x0010	Subsequent attempts to open the file are not allowed
FO_DENYWRITE	0x0020	Deny subsequent attempts to open the file for writing
FO_DENYREAD	0x0030	Deny subsequent attempts to open the file for reading
FO_DENYNONE	0x0040	Do not deny subsequent attempts to open the file in shared mode
FO_SHARED	0x0040	Same as FO_DENYNONE

Inheritance Flags

Constant	Flag	Description
FO_INHERITED	0x0000	Spawned processes can inherit
FO_PRIVATE	0x0080	Spawned processes cannot inherit

To specify multiple flags, combine them with the bitwise OR operator (e.g., FO_READWRITE | FO_EXCLUSIVE | FO_INHERITED).

Returns

_fsOpen() returns a DOS handle to the newly created file or FS_ERROR, if an error occurs.

Description

This function opens the file specified by *fpFilename*. The value of the specified *uiFlags* determines the mode in which the file is opened. _fsOpen() is equivalent to the CA-Clipper function FOPEN().

Examples

- The following code fragment illustrates the use of _fsOpen(). Note that error handling should be performed at each step. Refer to the "Error System API Reference" chapter of this guide for details on communicating with the CA-Clipper Error system:

```
#include "filesys.api"

FHANDLE hFile;

hFile = _fsOpen( "Foo", FO_READWRITE | FO_EXCLUSIVE );

// Check for an open error before using the file
if (! _fsError() )
{
    .
    .
    .
    _fsClose( hFile );
}
```

Files

Library is CLIPPER.LIB, header file is Filesys.api.

See Also

_fsClose(), _fsCreate(), _fsError(), _fsExtOpen()

_fsRead()

Read a file

C Prototype

```
#include "filesys.api"
USHORT _fsRead(
                    FHANDLE hFileHandle,
                    BYTEP fpBuff,
                    USHORT uiCount
                )
```

Arguments

hFileHandle is a valid DOS reference to the file to be read.

fpBuff is a pointer to an allocated memory buffer.

uiCount is the number of bytes to be read from the file.

Returns

_fsRead() returns the number of bytes read from the file.

Description

_fsRead() reads characters from the file specified by *hFileHandle* into the buffer area specified by *fpBuff*. It reads from the file starting at the current DOS file pointer position, advancing the file pointer by the number of bytes read. If the return value is not equal to the *uiCount* specified, use _fsError() to determine the nature of the read error.

This function is equivalent to the CA-Clipper FREAD() function.

Examples

- The following code fragment illustrates the use of _fsRead(). Note that error handling should be performed at each step. Refer to the "Error System API Reference" chapter of this guide for details on communicating with the CA-Clipper Error system:

```
#include "filesys.api"

FHANDLE hFile;
BYTEP fpBuff[15];

hFile = _fsOpen( "Foo", FO_READWRITE | FO_EXCLUSIVE );

// Check for an open error before using the file
if (! _fsError() )
{
    // Seek to EOF
    _fsSeek( hFile, 0, FS_END );

    _fsRead( hFile, fpBuff, 15 );

    _fsClose( hFile );
}
```

Files Library is CLIPPER.LIB, header file is Filesys.api.

See Also _fsClose(), _fsCreate(), _fsError(), _fsOpen(), _fsSeek(), _fsWrite()

_fsRmDir()

Remove a directory

C Prototype

```
#include "filesys.api"
BOOL _fsRmDir(
                BYTEP fpDirName
              )
```

Arguments

fpDirName is the name of the directory to remove, including an optional drive. If you do not specify a drive, the current one is used.

Returns

_fsRmDir() returns true (.T.) if successful; otherwise, it returns false (.F.).

Description

This function removes a specified directory. You must have sufficient rights to delete a directory. A directory must be empty in order to be deleted; therefore, to delete a directory that contains subdirectories, you must first delete the subdirectories. This function is equivalent to the CA-Clipper function DIRREMOVE(). If an error occurs, _fsError() will return the number of the error.

Examples

- The following code fragment illustrates the use of _fsRmDir():

```
#include "filesys.api"
.
.
.
if (! _fsRmDir( "c:\test\one" ) )
{
   // Report the error condition using _fsError()
}
```

Files

Library is CLIPPER.LIB, header file is Filesys.api.

See Also

_fsError()

_fsRename()

Rename a file

C Prototype

```
#include "filesys.api"
void _fsRename(
                    BYTEP fpOldName,
                    BYTEP fpNewName
                )
```

Arguments

fpOldName is the current name of the file.

fpNewName is the new name for the file.

Returns

_fsRename() has no return value.

Description

This function renames the file, *fpOldName*, to the new name, *fpNewName*. This function is equivalent to the CA-Clipper function FRENAME(). If an error occurs, _fsError() will return the number of the DOS error.

Examples

- The following code fragment illustrates the use of _fsRename(). Note that error handling is important when renaming a file as the function can fail for a number of reasons:

```
#include "filesys.api"
.
.
.
_fsRename( "Foo", "Temp" );

// Always check for errors when renaming a file
if (_fsError() )
{
    // Report the error condition
}
```

Files

Library is CLIPPER.LIB, header file is Filesys.api.

See Also

_fsDelete(), _fsError()

_fsSeek()

Reposition the pointer within a file

C Prototype

```
#include "filesys.api"
ULONG _fsSeek(
                FHANDLE hFileHandle,
                LONG lOffset,
                USHORT uiMode
              )
```

Arguments

hFileHandle is a valid DOS reference to the file to be repositioned.

lOffset is the memory location to set the file pointer to.

uiMode specifies the starting point for the seek. The following constants are defined for *uiMode*:

File Mode Flags

Constant	Flag	Description
FS_SET	0x0000	Seek from beginning of file
FS_RELATIVE	0x0001	Seek from current file pointer
FS_END	0x0002	Seek from end of file

Returns

_fsSeek() returns the new file pointer position.

Description

This function repositions the file pointer in the file whose handle is specified by *hFileHandle*. The file pointer is positioned based on *lOffset* and *uiMode*. This function is equivalent to the CA-Clipper function FSEEK(). If an error occurs, _fsError() will return the number of the DOS error.

Examples

- The following example uses _fsSeek() to determine the size of a file by seeking to the end:

```
#include "filesys.api"

ULONG FileSize( BYTEP fpFileName )
{

    FHANDLE hFile;
    ULONG ulSize = 0;

    hFile = _fsOpen( "Foo", FO_READWRITE | FO_EXCLUSIVE );

    // Check for an open error before using the file
    if (! _fsError() )
    {
       // Determine size by seeking to EOF
       ulSize = _fsSeek( hFile, 0, FS_END );

       _fsClose( hFile );
    }

    return (ulSize);
}
```

Files Library is CLIPPER.LIB, header file is Filesys.api.

See Also _fsClose(), _fsCreate(), _fsError(), _fsOpen(), _fsRead(), _fsWrite()

_fsWrite()

Write a file

C Prototype

```
#include "filesys.api"
USHORT _fsWrite(
                    FHANDLE hFileHandle,
                    BYTEP fpBuff,
                    USHORT uiCount
                )
```

Arguments

hFileHandle is a valid DOS reference to the file to be written.

fpBuff is a pointer to an allocated memory buffer.

uiCount is the number of bytes to write to the file.

Returns

_fsWrite() returns the number of bytes written to the file.

Description

This function attempts to write *uiCount* bytes from the memory buffer *fpBuff* to the file specified by *hFileHandle*. It is equivalent to the CA-Clipper function FWRITE().

This function returns the number of bytes actually written. If this return value is not equal to *uiCount*, use _fsError() to determine the nature of the error.

Examples

- The following code fragment illustrates the use of _fsWrite(). Note that error handling should be performed at each step. Refer to the "Error System API Reference" chapter of this guide for details on communicating with the CA-Clipper Error system:

```
#include "filesys.api"

FHANDLE hFile;
BYTEP fpBuff[15];

hFile = _fsOpen( "Foo", FO_READWRITE | FO_EXCLUSIVE );

// Check for an open error before using the file
if (! _fsError() )
{
    // Seek to EOF
    _fsSeek( hFile, 0, FS_END );

    _fsWrite( hFile, fpBuff, 15 );

    _fsClose( hFile );
}
```

Files Library is CLIPPER.LIB, header file is Filesys.api.

See Also _fsClose(), _fsCreate(), _fsError(), _fsOpen()

Chapter 12
General Terminal Reference Listing

_gtBox()
_gtBoxD()
_gtBoxS()
_gtColorSelect()
_gtDispBegin()
_gtDispCount()
_gtDispEnd()
_gtGetColorStr()
_gtGetCursor()
_gtGetPos()
_gtIsColor()
_gtMaxCol()
_gtMaxRow()
_gtPostExt()
_gtPreExt()
_gtRectSize()
_gtRepChar()
_gtRest()
_gtSave()
_gtScrDim()
_gtScroll()
_gtSetBlink()
_gtSetColorStr()
_gtSetCursor()
_gtSetMode()
_gtSetPos()
_gtSetSnowFlag()
_gtWrite()
_gtWriteAt()
_gtWriteCon()

Chapter 12
General Terminal API Reference

The General Terminal (GT) API gives your Extend routines access to the CA-Clipper General Terminal system. This chapter is an alphabetical reference to all of the functions in the CA-Clipper General Terminal API.

The prototypes for these functions are defined in the header file Gt.api, located in the \CLIP53\INCLUDE directory. The data types used in the prototypes are defined in the header file Clipdefs.h (automatically included by Gt.api), also located in \CLIP53\INCLUDE. The functions themselves are defined in CLIPPER.LIB, located in the \CLIP53\LIB directory.

_gtBox()

Draw a box on the screen

C Prototype

```
#include "gt.api"
ERRCODE _gtBox(
                    USHORT uiTop,
                    USHORT uiLeft,
                    USHORT uiBottom,
                    USHORT uiRight,
                    BYTEP fpBoxString
                  )
```

Arguments

uiTop, ***uiLeft***, ***uiBottom***, and ***uiRight*** define the coordinates of the box. Allowable row values are from zero to _gtMaxRow() and allowable column values are from zero to _gtMaxCol(). Specifying coordinates that are not within this range will result in that area being drawn off the screen.

fpBoxString defines a string of eight border characters and a fill character to be used in drawing the box.

Returns

_gtBox() returns zero if successful. Any other value indicates an error.

Description

_gtBox() draws a box at the specified display coordinates. _gtBox() draws the box using the characters in *fpBoxString* starting from the upper left-hand corner, proceeding clockwise and filling the screen region with the ninth character. If the ninth character is not specified, the screen region within the box is not painted. Existing text and color remain unchanged.

After _gtBox() executes, the cursor is located in the upper corner of the boxed region (*uiTop* + 1 and *uiLeft* + 1). The internal row and column coordinates are updated so the CA-Clipper functions, ROW() and COL(), will reflect the new cursor position.

The following table shows the manifest constants defined in Gt.api that you can use with this function to facilitate its use:

Manifest Constants for Box Drawing

Gt.api	Description
_B_SINGLE	Single-line box
_B_DOUBLE	Double-line box
_B_SINGLE_DOUBLE	Single horizontal lines, double vertical lines
_B_DOUBLE_SINGLE	Double horizontal lines, single vertical lines

Examples

- This example draws framed text at a specified screen row and column. The frame is made up of single horizontal lines and double vertical lines:

```
#include "gt.api"

void boxText( USHORT uiRow, USHORT uiCol, BYTEP fpStr );

void boxText( USHORT uiRow, USHORT uiCol, BYTEP fpStr )
{
    _gtBox( uiRow, uiCol, uiRow + 2, uiCol + strlen( fpStr ) + 2,
            _B_SINGLE_DOUBLE );
    _gtWrite( fpStr, strlen( fpStr ) );
}
```

Files

Library is CLIPPER.LIB, header file is Gt.api.

See Also

_gtBoxD(), _gtBoxS(), _gtMaxCol(), _gtMaxRow(), _gtWrite(), _gtWriteAt()

_gtBoxD()

Draw a double-line box on the screen

C Prototype

```
#include "gt.api"
ERRCODE _gtBoxD(
                    USHORT uiTop,
                    USHORT uiLeft,
                    USHORT uiBottom,
                    USHORT uiRight
                  )
```

Arguments

uiTop, ***uiLeft***, ***uiBottom***, and ***uiRight*** define the coordinates of the box. Allowable row values are from zero to _gtMaxRow() and allowable column values are from zero to _gtMaxCol(). Specifying coordinates that are not within this range will result in that area being drawn off the screen.

Returns

_gtBoxD() returns zero if successful. Any other value indicates an error.

Description

_gtBoxD() draws a double-line box at the specified coordinates on the screen. Existing text and color remain unchanged.

After _gtBoxD() executes, the cursor is located in the upper corner of the boxed region (*uiTop* + 1 and *uiLeft* + 1). The internal row and column coordinates are updated so the CA-Clipper functions, ROW() and COL(), will reflect the new cursor position.

Examples

- This example draws text framed by a double-line box on the screen:

```
#include "gt.api"

void boxTextD( USHORT uiRow, USHORT uiCol, BYTEP fpStr );

void boxTextD( USHORT uiRow, USHORT uiCol, BYTEP fpStr )
{
    _gtBoxD( uiRow, uiCol, uiRow + 2,
             uiCol + strlen( fpStr ) + 2 );
    _gtWrite( fpStr, strlen( fpStr ) );

}
```

Files

Library is CLIPPER.LIB, header file is Gt.api.

See Also

_gtBox(), _gtBoxS(), _gtMaxCol(), _gtMaxRow(), _gtWrite(), _gtWriteAt()

_gtBoxS()

Draw a single-line box on the screen

C Prototype

```
#include "gt.api"
ERRCODE _gtBoxS(
                    USHORT uiTop,
                    USHORT uiLeft,
                    USHORT uiBottom,
                    USHORT uiRight
                  )
```

Arguments

uiTop, ***uiLeft***, ***uiBottom***, and ***uiRight*** define the coordinates of the box. Allowable row values are from zero to _gtMaxRow() and allowable column values are from zero to _gtMaxCol(). Specifying coordinates that are not within this range will result in that area being drawn off the screen.

Returns

_gtBoxS() returns zero if successful. Any other value indicates an error.

Description

_gtBoxS() draws a single-line box at the specified coordinates on the screen. Existing text and color remain unchanged.

After _gtBoxS() executes, the cursor is located in the upper corner of the boxed region (*uiTop* + 1 and *uiLeft* + 1). The internal row and column coordinates are updated so the CA-Clipper functions, ROW() and COL(), will reflect the new cursor position.

Examples

- This example draws text framed by a single-line box on the screen:

```
#include "gt.api"

void boxTextS( USHORT uiRow, USHORT uiCol, BYTEP fpStr );

void boxTextS( USHORT uiRow, USHORT uiCol, BYTEP fpStr )
{
    _gtBoxS( uiRow, uiCol, uiRow + 2,
             uiCol + strlen( fpStr ) + 2 );
    _gtWrite( fpStr, strlen( fpStr ) );
}
```

Files Library is CLIPPER.LIB, header file is Gt.api.

See Also _gtBox(), _gtBoxD(), _gtMaxCol(), _gtMaxRow(), _gtWrite(), _gtWriteAt()

_gtColorSelect()

Activate an attribute in the current CA-Clipper color setting

C Prototype

```
#include "gt.api"
ERRCODE _gtColorSelect(
                              USHORT uiColorIndex
                            )
```

Arguments

uiColorIndex is the zero-based ordinal position in the current list of CA-Clipper color attributes.

Returns

_gtColorSelect() returns zero if successful. Any other value indicates an error.

Description

_gtColorSelect() sets the color with which writes are displayed. The specified color pair is obtained from the current list of color attributes. Manifest constants for specifying *uiColorIndex* values, defined in the header file Color.ch, are shown in the table below:

Color.ch Manifest Constants

Constant Name	Value	Color Setting
CLR_STANDARD	0	Standard
CLR_ENHANCED	1	Enhanced
CLR_BORDER	2	Border
CLR_BACKGROUND	3	Background
CLR_UNSELECTED	4	Unselected

Examples

- This example creates two functions that change the active color:

```
#include "gt.api"

void colorStandard( void );
void colorEnhanced( void );


void colorStandard( )
{
    _gtColorSelect( CLR_STANDARD );
}

void colorEnhanced( )
{
    _gtColorSelect( CLR_ENHANCED );
}

.
.
.
    _gtSetColorStr( "W+/B, B/W" );

    colorEnhanced( );
    _gtWrite( "Enhanced color (B/W)", 20 );

    colorStandard( );
    _gtWrite( "Standard Color (W+/B)", 21 );
.
.
.
```

Files Library is CLIPPER.LIB, header file is Gt.api.

See Also _gtGetColorStr(), _gtIsColor(), _gtSetColorStr(), _gtWrite(), _gtWriteAt()

_gtDispBegin()

Begin buffering screen output

C Prototype

```
#include "gt.api"
ERRCODE _gtDispBegin(void)
```

Returns

_gtDispBegin() returns zero if successful. Any other value indicates an error.

Description

_gtDispBegin() informs the CA-Clipper display system of the start of a series of display operations that are to be buffered. Display output that occurs after _gtDispBegin(), but before _gtDispEnd(), accumulates in internal buffers. All screen updates are performed after _gtDispEnd(). This can enhance the performance of applications with complex screen displays.

_gtDispBegin() and _gtDispEnd() calls are optional.

Notes

- ***Nested calls:*** _gtDispBegin() calls are nested internally. If you issue several _gtDispBegin() calls, buffering occurs until you issue a corresponding number of _gtDispEnd() calls.
- ***Guaranteed operations:*** Display updates performed between _gtDispBegin() and _gtDispEnd() are not guaranteed to be buffered. Some updates may become visible before _gtDispEnd() is called. However, all updates are guaranteed to be visible after the closing call to _gtDispEnd().
- ***Terminal operations:*** CA-Clipper terminal input operations such as INKEY() and READ should not be performed between _gtDispBegin() and _gtDispEnd(). Doing this may cause input or display output to be lost.
- ***Incompatible operations:*** Display output other than by the CA-Clipper display functions (e.g., by add-on libraries or by DOS via the OUTSTD() function) may not be compatible with _gtDispBegin() and _gtDispEnd(). Output may be lost.

Examples

- This example buffers screen output, updates the screen, then displays the buffered screen output:

```
#include "gt.api"
.
.
.
    _gtDispBegin();      // Start screen buffering

    _gtSetPos(10, 10);
    _gtWrite( "A display update", 17 );
    _gtSetPos(11, 10)
    _gtWrite( "Another display update", 22 )

    _gtDispEnd()         // Display buffered screen data
.
.
.
```

Files Library is CLIPPER.LIB, header file is Gt.api.

See Also _gtDispCount(), _gtDispEnd(), _gtGetPos(), _gtSetPos(), _gtWrite()

_gtDispCount()

Return the number of pending _gtDispEnd() requests

C Prototype

```
#include "gt.api"
USHORT _gtDispCount(void)
```

Returns

_gtDispCount() returns the number of pending _gtDispEnd() requests as an unsigned short integer value.

Description

_gtDispCount() determines the number of nested gtDispBegin() calls that have been made. Since you may nest _gtDispBegin(), ... ,_gtDispEnd() calls, use _gtDispCount() to determine whether there are pending screen refresh requests.

Examples

- This example releases all pending display contexts:

```
#include "gt.api"

void ForceDisplay( void );

void ForceDisplay( void )
{
   // Display all pending buffers

   while ( _gtDispCount() > 0 )
      _gtDispEnd();

}
```

Files

Library is CLIPPER.LIB, header file is Gt.api.

See Also

_gtDispBegin(), _gtDispEnd()

_gtDispEnd()

Release the display buffer

C Prototype

```
#include "gt.api"
ERRCODE _gtDispEnd(void)
```

Returns

_gtDispEnd() returns zero if successful. Any other value indicates an error.

Description

_gtDispEnd() informs the CA-Clipper display system of the end of a series of display operations that have been buffered. Any pending screen updates are performed after _gtDispEnd().

_gtDispBegin() and _gtDispEnd() calls are optional. For more information, refer to _gtDispBegin().

Examples

- This example buffers screen output, updates the screen, then displays the buffered screen output:

```
#include "gt.api"
.
.
.
    _gtDispBegin();      // Start screen buffering

    _gtSetPos(10, 10);
    _gtWrite( "A display update", 17 );
    _gtSetPos(11, 10)
    _gtWrite( "Another display update", 22 )

    _gtDispEnd()         // Display buffered screen data
.
.
.
```

Files

Library is CLIPPER.LIB, header file is Gt.api.

See Also

_gtDispBegin(), _gtDispCount(), _gtGetPos(), _gtSetPos(), _gtWrite()

_gtGetColorStr()

Get the CA-Clipper color attributes setting

C Prototype

```
#include "gt.api"
ERRCODE _gtGetColorStr(
                              BYTEP fpColorString
                            )
```

Arguments

fpColorString is a null-terminated character string of at least 64 bytes that receives the current CA-Clipper color attributes setting. The constant CLR_STRLEN can be used to allocate a string of correct size.

Returns

_gtGetColorStr() returns zero if successful. Any other value indicates an error.

Description

_gtGetColorStr() saves the current CA-Clipper color attributes setting as a null-terminated character string, *fpColorString*. You can later use *fpColorString* with _gtSetColorStr() to return the color setting to its prior value. For information on how *fpColorString* is formatted and the meanings of the various color codes, see the _gtSetColorStr() entry in this chapter.

Examples

- This example saves the current color setting before changing it, displays a string using the new color setting, and restores the original color setting:

```
#include "gt.api"

void PrintStr( USHORT uiRow, USHORT uiCol, BYTEP fpStr,
               BYTEP fpColor );

void PrintStr( USHORT uiRow, USHORT uiCol, BYTEP fpStr,
               BYTEP fpColor )
{
   BYTE cSavColor[ CLR_STRLEN ];
   USHORT uiSavRow;
   USHORT uiSavCol;

   _gtGetPos( &uiSavRow, &uiSavCol ); // Save coordinates
   _gtGetColorStr( cSavColor );       // Save color setting

   _gtSetColorStr( fpColor );
   _gtSetPos( uiRow, uiCol );
   _gtWrite( fpStr, strlen( fpStr ) );

   _gtSetColorStr( cSavColor );       // Restore color setting
   _gtSetPos( uiSavRow, uiSavCol );   // Restore coordinates

}
```

Files

Library is CLIPPER.LIB, header file is Gt.api.

See Also

_gtColorSelect(), _gtGetPos(), _gtIsColor(), _gtSetColorStr(), _gtSetPos(), _gtWrite()

_gtGetCursor()

Get the current cursor shape

C Prototype

```
#include "gt.api"
ERRCODE _gtGetCursor(
                          USHORTP uipCursorShape
                        )
```

Arguments

uipCursorShape receives a number indicating the shape of the cursor.

Returns

_gtGetCursor() returns zero if successful. Any other value indicates an error.

Description

_gtGetCursor() saves the current cursor shape as a numeric value, *uipCursorShape*. You can later use the value referenced by *uipCursorShape* with _gtSetCursor() to return the cursor to its prior shape.

Examples

- This example illustrates how to use _gtGetCursor() and _gtSetCursor() to save the current cursor shape, change it, and restore it to its original shape:

```
#include "gt.api"
.
.
.

   USHORT uiSavCursor;

   _gtGetCursor( &uiSavCursor );    // Save cursor shape
   _gtSetCursor( SC_SPECIAL1 );     // Change cursor to a block
.
.
.
   _gtSetCursor( uiSavCursor );     // Restore cursor shape
}
```

Files

Library is CLIPPER.LIB, header file is Gt.api.

See Also

_gtGetPos(), _gtSetCursor(), _gtSetPos()

_gtGetPos()

Get the cursor location

C Prototype

```
#include "gt.api"
ERRCODE _gtGetPos(
                        USHORTP uipRow,
                        USHORTP uipCol
                    )
```

Arguments

uipRow receives the row position of the cursor. The value received may range from zero to _gtMaxRow().

uipCol receives the column position of the cursor. The value received may range from zero to _gtMaxCol().

Returns

_gtGetPos() returns zero if successful. Any other value indicates an error.

Description

_gtGetPos() saves the current cursor position as a pair of numeric coordinates, *uipRow* and *uipCol*. You can later use the coordinates with _gtSetPos() to return the cursor to its prior location.

Examples

- This example saves the current cursor location before displaying output at a specified location. The cursor is then restored to its original position:

```
#include "gt.api"

void PrintStr( USHORT uiRow, USHORT uiCol, BYTEP fpStr );

void PrintStr( USHORT uiRow, USHORT uiCol, BYTEP fpStr )
{
    USHORT uiSavRow;
    USHORT uiSavCol;

    _gtGetPos( &uiSavRow, &uiSavCol );     // Save position
    _gtSetPos( uiRow, uiCol );             // Set new position
    _gtWrite( fpStr, strlen( fpStr ) );
    _gtSetPos( uiSavRow, uiSavCol );       // Restore position

}
```

Files

Library is CLIPPER.LIB, header file is Gt.api.

See Also

_gtGetCursor(), _gtMaxCol(), _gtMaxRow(), _gtSetCursor(), _gtSetPos(), _gtWrite(), _gtWriteAt()

_gtIsColor()

Determine if the current screen display has color capability

C Prototype

```
#include "gt.api"
BOOL _gtIsColor(void)
```

Returns

_gtIsColor() returns true (.T.) if a color screen display is detected; otherwise, the function returns false (.F.).

Description

_gtIsColor() lets you make decisions about the type of screen attributes to assign (color or monochrome) by giving you information about the type of screen display being used. Note that some monochrome adapters with graphics capability return true (.T.).

Examples

- This example uses _gtIsColor() in an if decision block to display the statement "Color system" on color screens only:

```
#include "gt.api"
.
.
.
   if ( _gtIsColor() )
   {
      _gtWriteAt( 0, 0, "Color screen",  12 );
   }
.
.
.
```

Files Library is CLIPPER.LIB, header file is Gt.api.

See Also _gtColorSelect(), _gtGetColorStr(), _gtSetColorStr()

_gtMaxCol()

Determine the maximum screen column

C Prototype

```
#include "gt.api"
USHORT _gtMaxCol(void)
```

Returns

_gtMaxCol() returns an unsigned integer representing the maximum column of the display screen.

Description

_gtMaxCol() determines the maximum visible column of the screen. Row and column numbers start at zero.

Examples

- This example uses _gtMaxCol() to display the string "Right" on the right-hand side of the screen:

```
#include "gt.api"
.
.
.
   _gtWriteAt( 1 , _gtMaxCol() - 5, "Right", 5 );
.
.
.
```

Files

Library is CLIPPER.LIB, header file is Gt.api.

See Also

_gtMaxRow(), _gtWriteAt()

_gtMaxRow()

Determine the maximum screen row

C Prototype

```
#include "gt.api"
USHORT _gtMaxRow(void)
```

Returns

_gtMaxRow() returns an unsigned integer representing the maximum row of the display screen.

Description

_gtMaxRow() determines the maximum visible row of the screen. Row and column numbers start at zero.

Examples

- This example uses _gtMaxRow() to display the string "Bottom" on the last row of the screen:

```
#include "gt.api"

.
.
.
   _gtWriteAt( _gtMaxRow() , 1, "Bottom", 6 );
.
.
.
```

Files

Library is CLIPPER.LIB, header file is Gt.api.

See Also

_gtMaxCol(), _gtWriteAt()

_gtPostExt()

Return control to the General Terminal system

C Prototype

```
#include "gt.api"
ERRCODE _gtPostExt(void)
```

Returns

_gtPostExt() returns zero if successful. Any other value indicates an error.

Description

_gtPostExt() returns control to the General Terminal system after direct video access.

Warning! *Always call _gtPostExt() after direct video access.*

Examples

- This example uses _gtPostExt() to re-engage the General Terminal system after attempting to reverse the screen attribute of a passed coordinate using direct video access:

```
/***
*
*   Reverse text Attribute for passed coordinates
*
*/

#include "extend.api"
#include "gt.api"

#define VIDSEG  (_gtIsColor() ? 0xB800 : 0xB000 )
#define MK_FP( seg, off ) \
               ((FARP) ( ((ULONG)(seg)<<16) | (USHORT)(off) ));


CLIPPER ReverseAttribute()
{

   USHORT uiRow         = _parni(1);
   USHORT uiCol         = _parni(2)
   USHORT uiNumCols     = _gtMaxCol() + 1;
   BYTEP *scrPtr;

   _gtPreExt();              // Release display context!
```

```
    // Make a far pointer to row/col in video memory
    scrPtr = MK_FP( VIDSEG, ( ( uiRow * uiNumCols ) +
                      uiCol ) * 2 );

    // Skip one to the attribute
    scrPtr++;

    // Reverse the attribute
    *scrPtr = (*scrPtr >> 4) | ((*scrPtr & 0x07) << 4) |
              (*scrPtr & 0x08) | (*scrPtr & 128);


    _gtPostExt();              // Signal end of external code.

    _ret();                    // Post a NIL return value

}
```

Files Library is CLIPPER.LIB, header file is Gt.api.

See Also _gtPreExt()

_gtPreExt()

Prepare the area manager for external manipulation of the terminal

C Prototype

```
#include "gt.api"
ERRCODE _gtPreExt(void)
```

Returns

_gtPreExt() returns zero if successful. Any other value indicates an error.

Description

_gtPreExt() readies the General Terminal system for external access. All pending writes (i.e., display buffers called from C or CA-Clipper) are written to the physical screen.

Warning! *Always call _gtPreExt() before attempting direct video access.*

Examples

- This example uses _gtPreExt() to flush any pending screen writes from the display buffers before attempting to reverse the screen attribute of a passed coordinate using direct video access:

```
/***
*
*   Reverse text Attribute for passed coordinates
*
*/

#include "extend.api"
#include "gt.api"


#define VIDSEG  (_gtIsColor() ? 0xB800 : 0xB000 )
#define MK_FP( seg, off ) \
               ((FARP) ( ((ULONG)(seg)<<16) | (USHORT)(off) ));


CLIPPER ReverseAttribute()
{

    USHORT uiRow        = _parni(1);
    USHORT uiCol        = _parni(2)
    USHORT uiNumCols    = _gtMaxCol() + 1;
    BYTEP *scrPtr;

    _gtPreExt();              // Release display context!
```

```
        // Make a far pointer to row/col in video memory
        scrPtr = MK_FP( VIDSEG, ( ( uiRow * uiNumCols ) +
                          uiCol ) * 2 );

        // Skip one to the attribute
        scrPtr++;

        // Reverse the attribute
        *scrPtr = (*scrPtr >> 4) | ((*scrPtr & 0x07) << 4) |
                  (*scrPtr & 0x08) | (*scrPtr & 128);


        _gtPostExt();              // Signal end of external code.

        _ret();                    // Post a NIL return value

    }
```

Files Library is CLIPPER.LIB, header file is Gt.api.

See Also _gtPostExt()

_gtRectSize()

Determine the buffer size needed to save a specified screen region

C Prototype

```
#include "gt.api"
ERRCODE _gtRectSize(
                          USHORT  uiTop,
                          USHORT  uiLeft,
                          USHORT  uiBottom,
                          USHORT  uiRight,
                          USHORTP uipBuffSize
                        )
```

Arguments

uiTop, ***uiLeft***, ***uiBottom***, and ***uiRight*** define the coordinates of the rectangular region to be calculated. If all coordinates are NULL, the entire display region is used.

uipBuffSize is the size of the buffer needed to contain the specified region.

Returns

_gtRectSize() returns zero if successful. Any other value indicates an error.

Description

_gtRectSize() analyzes the specified screen region, determines the buffer size needed to save it, and saves the buffer size as a numeric value.

Examples

- This example uses _gtRectSize() to calculate the memory needed by _xvalloc() (see the "Virtual Memory API Reference" chapter of this guide) to properly save the screen buffer:

```
#include "gt.api"
#include "vm.api"
.
.
.
    BYTEP    psBuff;
    USHORT   uiBuffSize;
    HANDLE   hVM;

    _gtRectSize( 1, 1, 10, 25, &uiBuffSize );

    hVM      = _xvalloc( uiBuffSize, 0 );
    scrBuff  = _xvlock( hVM );
    _gtSave( 1, 1, 10, 25, fpBuff );// Save area under rect
    .
    . <manipulate screen>
    .
    _gtRest( 1, 1, 10, 25, fpBuff );// Restore screen

    _xvunlock( hVM );
    _xvfree( hVM );
.
.
.
```

Files Library is CLIPPER.LIB, header file is Gt.api.

See Also _gtRest(), _gtSave()

_gtRepChar()

Replicate a character to the screen

C Prototype

```
#include "gt.api"
ERRCODE _gtRepChar(
                        USHORT uiRow,
                        USHORT uiCol,
                        USHORT uiChar,
                        USHORT uiCount
                    )
```

Arguments

uiRow and ***uiCol*** are the screen coordinates at which to replicate the character. *uiRow* may be in the range of zero to _gtMaxRow(). *uiCol* may be in the range of zero to _gtMaxCol().

uiChar is the ASCII value of the character to replicate.

uiCount is the number of times to replicate the character.

Returns

_gtRepChar() returns zero if successful. Any other value indicates an error.

Description

_gtRepChar() lets you replicate a single character to the screen a specified number of times. The row that you specify stays the same while the column position is incremented by one for each replication of the character. The internal row and column coordinates are updated so the CA-Clipper functions, ROW() and COL(), will reflect the new cursor position.

Examples

- In this example, _gtRepChar() draws a single-line horizontal divider:

```
#include "gt.api"

#define HBAR_LEFT    195
#define HBAR_MIDDLE  196
#define HBAR_RIGHT   180

void drawHBarS( USHORT uiR, USHORT uiL, USHORT uiW );

void drawHBarS( USHORT uiR, USHORT uiL, USHORT uiW )
{
    _gtWriteAt( uiR, uiL, HBAR_LEFT );
    _gtRepChar( uiR, uiL + 1, HBAR_MIDDLE, uiW - 2 );
    _gtWrite( HBAR_RIGHT );
}
```

Files

Library is CLIPPER.LIB, header file is Gt.api.

See Also

_gtMaxCol(), _gtMaxRow(), _gtRest(), _gtSave(), _gtWrite(), _gtWriteAt(), _gtWriteCon()

_gtRest()

Display a saved screen region at a specified location

C Prototype

```
#include "gt.api"
ERRCODE _gtRest(
                    USHORT uiTop,
                    USHORT uiLeft,
                    USHORT uiBottom,
                    USHORT uiRight,
                    FARP vlpScrBuff
                  )
```

Arguments

uiTop, ***uiLeft***, ***uiBottom***, and ***uiRight*** define the coordinates where the saved screen region will be restored. If *uiBottom* is greater than _gMaxRow() or *uiRight* is greater than _gtMaxCol(), the screen is clipped.

vlpScrBuff is a far pointer to a character that, in most cases, is a value returned from _gtSave().

Returns

_gtRest() returns zero if successful. Any other value indicates an error.

Description

_gtRest() restores a screen region previously saved with _gtSave(). The target screen location may be the same as or different than the original location when the screen region was saved. If you specify a new screen location, the new screen region must be the same size or you will get ambiguous results.

Some examples of when you might use _gtSave() and _gtRest() are to display a pop-up menu or to drag a screen object.

Warning! *Like SAVE SCREEN, RESTORE SCREEN, SAVESCREEN(), and RESTSCREEN(), _gtRest() and _gtSave() are supported when using the default (IBM PC memory-mapped) terminal driver. Other terminal drivers may not support saving and restoring screens.*

Examples

- In this example, _gtRest() and _gtSave() create a see-through shadow on the screen. By not manipulating the video memory directly, the subroutine RevForeAttr() ensures the integrity of the General Terminal system and does not destroy any screen buffers that may be in use. To insure proper memory usage, RevForeAttr() uses the Virtual Memory API for screen saves (see the "Virtual Memory API Reference" chapter of this guide):

```
#include "vm.api"
#include "gt.api"


void near Shadow( USHORT uiTRow, USHORT uiLCol,
                  USHORT uiBRow, USHORT uiRCol );

HIDE void near RevForeAttr( USHORT uiTRow,
                            USHORT uiLCol,
                            USHORT uiBRow,
                            USHORT uiRCol );

void near Shadow( USHORT uiTRow, USHORT uiLCol,
                  USHORT uiBRow, USHORT uiRCol )
{
    /* Draw shadow on right side */
    RevForeAttr( uiTRow+1, uiRCol+1, uiBRow+1, uiRCol+2 );

    /* Draw shadow on bottom */
    RevForeAttr( uiBRow+1, uiLCol+2, uiBRow+1, uiRCol );

}



/***
*    Reverse text Attribute for
*    passed coordinates
*/

HIDE void near RevForeAttr( USHORT uiTRow,
                            USHORT uiLCol,
                            USHORT uiBRow,
                            USHORT uiRCol )

{
    FARP vlpScreen;
    HANDLE hVM;
    USHORT uiBuffSize;

    USHORT uiRow;
    USHORT uiCol;
    USHORT i;
```

```
   _gtRectSize( uiTRow, uiLCol, uiBRow, uiRCol, &uiBuffSize );

   if !( hVM = _xvalloc( uiBuffSize ) )
      return;

   vlpScreen = _xvlock( hVM );

   _gtSave( uiTRow, uiLCol, uiBRow, uiRCol, vlpScreen );

   for ( uiRow = uiTRow; uiRow <= uiBRow; ++uiRow )
   {
      i = ( (uiRow - uiTRow) * (uiRCol - uiLCol + 1)
            * 2 ) + 1;
      for ( uiCol = uiLCol; uiCol <= uiRCol; ++uiCol, i += 2 )
      {
         vlpScreen[i] &= 0x0007;
         if( !vlpScreen[i] ) vlpScreen[i] = 0x0008;
      }
   }
   _gtRest( uiTRow, uiLCol, uiBRow, uiRCol, vlpScreen );
}
```

Files Library is CLIPPER.LIB, header file is Gt.api.

See Also _gtMaxCol(), _gtMaxRow(), _gtRectSize(), _gtSave()

_gtSave()

Save a screen region for later display

C Prototype

```
#include "gt.api"
ERRCODE _gtSave(
                    USHORT uiTop,
                    USHORT uiLeft,
                    USHORT uiBottom,
                    USHORT uiRight,
                    FARP vlpScrBuff
                  )
```

Arguments

uiTop, uiLeft, uiBottom, and ***uiRight*** define the coordinates of the screen region to save. If *uiBottom* is greater than _gMaxRow() or *uiTop* is greater than _gtMaxCol(), the screen is clipped.

vlpScrBuff is a far pointer to a character string. To make sure the string is large enough to hold the screen region, use _gtRectSize().

Returns

_gtSave() returns zero if successful. Any other value indicates an error.

Description

_gtSave() saves a screen region to a character string. Later, you can redisplay the saved screen image to the same location, or to a new location using _gtRest().

Some examples of when you might use _gtSave() and _gtRest() are to display a pop-up menu or to drag a screen object.

Warning! *Like SAVE SCREEN, RESTORE SCREEN, SAVESCREEN(), and RESTSCREEN(), _gtRest() and _gtSave() are supported when using the default (IBM PC memory mapped) terminal driver. Other terminal drivers may not support saving and restoring screens.*

Examples

- In this example, _gtRest() and _gtSave() create a see-through shadow on the screen. By not manipulating the video memory directly, the subroutine RevForeAttr() ensures the integrity of the General Terminal system and does not destroy any screen buffers that may be in use. To insure proper memory usage, RevForeAttr() uses the Virtual Memory API for screen saves (see the "Virtual Memory API Reference" chapter of this guide):

```
#include "vm.api"
#include "gt.api"


void near Shadow( USHORT uiTRow, USHORT uiLCol,
                  USHORT uiBRow, USHORT uiRCol );

HIDE void near RevForeAttr( USHORT uiTRow,
                            USHORT uiLCol,
                            USHORT uiBRow,
                            USHORT uiRCol );

void near Shadow( USHORT uiTRow, USHORT uiLCol,
                  USHORT uiBRow, USHORT uiRCol )
{
   /* Draw shadow on right side */
   RevForeAttr( uiTRow+1, uiRCol+1, uiBRow+1, uiRCol+2 );

   /* Draw shadow on bottom */
   RevForeAttr( uiBRow+1, uiLCol+2, uiBRow+1, uiRCol );

}


/***
*   Reverse text Attribute for
*   passed coordinates
*/

HIDE void near RevForeAttr( USHORT uiTRow,
                            USHORT uiLCol,
                            USHORT uiBRow,
                            USHORT uiRCol )

{
   FARP vlpScreen;
   HANDLE hVM;
   USHORT uiBuffSize;

   USHORT uiRow;
   USHORT uiCol;
   USHORT i;
```

```
   _gtRectSize( uiTRow, uiLCol, uiBRow, uiRCol, &uiBuffSize );

   if !( hVM = _xvalloc( uiBuffSize ) )
      return;

   vlpScreen = _xvlock( hVM );

   _gtSave( uiTRow, uiLCol, uiBRow, uiRCol, vlpScreen );

   for ( uiRow = uiTRow; uiRow <= uiBRow; ++uiRow )
   {
      i = ( (uiRow - uiTRow) * (uiRCol - uiLCol + 1)
            * 2 ) + 1;
      for ( uiCol = uiLCol; uiCol <= uiRCol; ++uiCol, i += 2 )
      {
         vlpScreen[i] &= 0x0007;
         if( !vlpScreen[i] ) vlpScreen[i] = 0x0008;
      }
   }
   _gtRest( uiTRow, uiLCol, uiBRow, uiRCol, vlpScreen );
}
```

Files Library is CLIPPER.LIB, header file is Gt.api.

See Also _gtMaxCol(), _gtMaxRow(), _gtRectSize(), _gtRest()

_gtScrDim()

Get the current screen dimensions

C Prototype

```
#include "gt.api"
ERRCODE _gtScrDim(
                        USHORTP uipHeight,
                        USHORTP uipWidth
                    )
```

Arguments

uipHeight receives the screen height.

uipWidth receives the screen width.

Returns

_gtScrDim() returns zero if successful. Any other value indicates an error.

Description

_gtScrDim() saves the current screen dimensions in *uipWidth* and *uipHeight*.

Examples

- In this example, a function uses _gtScrDim() to get the screen dimensions then returns the height:

```
#include "gt.api"

USHORT scrGetHeight( void );

USHORT scrGetHeight( )

{
   USHORT uiHeight;
   USHORT uiWidth;

   _gtScrDim( &uiHeight, &uiWidth );

   return (uiHeight);
}
```

Files

Library is CLIPPER.LIB, header file is Gt.api.

See Also

_gtSetMode()

_gtScroll()

Scroll a region of the screen

C Prototype

```
#include "gt.api"
ERRCODE _gtScroll(
                        USHORT uiTop
                        USHORT uiLeft,
                        USHORT uiBottom,
                        USHORT uiRight,
                        SHORT iRows,
                        SHORT iCols
                    )
```

Arguments

uiTop, ***uiLeft***, ***uiBottom***, and ***uiRight*** define the scroll region coordinates. Row and columns values may range from (0, 0) to (_gtMaxRow(), _gtMaxCol()).

iRows defines the number of rows to scroll. A value greater than zero scrolls the specified screen area up by the specified number of rows. A value less than zero scrolls the specified screen area down by the specified number of rows.

iCols defines the number of columns to scroll. A value greater than zero scrolls the specified screen area left by the specified number of columns. A value less than zero scrolls the specified screen area right by the specified number of columns.

If only one of the parameters *iRows* or *iCols* is supplied a zero value, the specified screen area will not be scrolled in that direction. If both *iRows* and *iCols* are supplied zero values, the specified screen region will be blanked (cleared).

Returns

_gtScroll() returns zero if successful. Any other value indicates an error.

Description

_gtScroll() scrolls a screen region up, down , left, or right.

When a screen region scrolls up the first line of the region is erased, all other lines are moved up, and a blank line is displayed in the current standard color on the bottom line of specified region. When a screen region scrolls down, the orientation is reversed. If the screen region is scrolled more than one line, this process is repeated.

When a screen region scrolls right the first column of the region is erased, all other columns are moved right, and a blank column is displayed in the current standard color on the left of specified region. When a screen region scrolls left, the orientation is reversed. If the screen region is scrolled more than one column, this process is repeated.

The screen can be scrolled vertically or horizontally. Additionally, the screen can be scrolled both directions at once causing the screen to be scrolled diagonally. Also, the screen region can be blanked (cleared).

Examples

- The following examples demonstrate the effects of the different parameter combinations.

```
#include "gt.api"

.
.
.
   /*
      Demonstrate how to scroll the contents of a screen area
      (5, 5 to 20, 50) in different ways
   */

   _gtScroll( 5, 5, 20, 50,  1,  0 );     // up
   _gtScroll( 5, 5, 20, 50, -1,  0 );     // down
   _gtScroll( 5, 5, 20, 50,  0,  1 );     // left
   _gtScroll( 5, 5, 20, 50,  1, -1 );     // right
   _gtScroll( 5, 5, 20, 50,  1,  1 );     // up and left
   _gtScroll( 5, 5, 20, 50, -1, -1 );     // down and right
   _gtScroll( 5, 5, 20, 50,  0,  0 );     // clears screen area
.
.
.
```

Files

Library is CLIPPER.LIB, header file is Gt.api.

See Also

_gtMaxCol(), _gtMaxRow(), _gtWriteAt()

_gtSetBlink()

Toggle asterisk (*) interpretation in color strings between blinking and background intensity

C Prototype

```
#include "gt.api"
ERRCODE _gtSetBlink(
                      BOOL bBlink
                    )
```

Arguments

bBlink toggles the meaning of the asterisk (*) character in a color string. Specifying true (.T.) causes the asterisk to mean blinking and false (.F.) causes it to mean high intensity background. The default is true (.T.).

Returns

_gtSetBlink() returns zero if successful. Any other value indicates an error.

Description

_gtSetBlink() toggles the blinking/background intensity attribute. When you call _gtSetBlink() with *bBlink* equal to true (.T.), you can make characters written to the screen blink by including an asterisk (*) in the color string passed to _gtSetColorStr(). When you call _gtSetBlink() with *bBlink* equal to false (.F.), the asterisk (*) causes the background color to be intensified instead. Thus, blinking and background intensity attributes are not available at the same time.

Note: This function is meaningful only on IBM PC or compatible computers with CGA, EGA, or VGA display hardware.

Examples

- This example enables high intensity backgrounds:

```
#include "gt.api"

.
.
.
   _gtSetBlink( FALSE )
.
.
.
```

Files

Library is CLIPPER.LIB, header file is Gt.api.

See Also

_gtSetColorStr()

_gtSetColorStr()

Change the CA-Clipper color attributes setting

C Prototype

```
#include "gt.api"
ERRCODE _gtSetColorStr(
                              BYTEP fpColorString
                            )
```

Arguments

fpColorString is a null-terminated character string containing a list of color attribute settings. The following table shows each color setting with its scope:

Color Settings

Setting	Scope
Standard	All screen output commands and functions
Enhanced	GETs and selection highlights
Border	Border around screen, not supported on EGA and VGA
Background	Not supported
Unselected	Unselected GETs

The order of the settings in the table is the order in which they must be specified in *fpColorString*. Each setting is a foreground and background color separated by a slash (/) character and followed by a comma.

All settings are optional. If a setting is skipped, its previous value is retained with only new values set. Settings may be skipped within the list or left off the end as illustrated in the example below.

Returns

_gtSetColorStr() returns zero if successful. Any other value indicates an error.

Description

_gtSetColorStr() changes the current CA-Clipper color attributes to those specified in *fpColorString*. Before using this function, you may want to save the current color setting with _gtGetColorStr() so that you can restore the prior color setting.

fpColorString is made up of several color settings, each color corresponding to a different region of the screen. As stated above, each setting is made up of a foreground and background color pair. Foreground defines the color of characters displayed on the screen. Background defines the color displayed behind the character. Spaces and nondisplay characters displayable as background only.

Standard: The standard setting governs all console, full-screen, and interface commands and functions when displaying to the screen. This includes commands such as @...PROMPT, @...SAY, and ?, and functions such as ACHOICE(), DBEDIT(), and MEMOEDIT().

Enhanced: The enhanced setting governs highlighted displays. This includes GETs with INTENSITY ON, and the MENU TO, DBEDIT(), and ACHOICE() selection highlights.

Border: The border is an area around the outside of the screen to which you cannot write.

Background: The background is not supported at this time.

Unselected: The unselected setting indicates that a GET no longer has input focus. The current GET (the GET with input focus) is displayed in the enhanced color while all other GETs are displayed in the unselected color.

In addition to colors, foreground settings can have high intensity and/or blinking attributes. With a monochrome display, high intensity enhances brightness of painted text. With a color display, high intensity changes the hue of the specified color. For example, "N" displays foreground text as black where "N+" displays the same text as gray. High intensity is denoted by "+".

The blinking attribute (controlled by _gtSetBlink()) causes the foreground text to flash on and off at rapid intervals. Blinking is denoted with "*". The attribute character can occur anywhere in the setting string but is always applied to the foreground color regardless of where it occurs.

The following colors are supported:

List of Colors

Color	Letter	Monochrome
Black	N, Space	Black
Blue	B	Underline
Green	G	White
Cyan	BG	White
Red	R	White
Magenta	RB	White
Brown	GR	White
White	W	White
Gray	N+	White
Bright Blue	B+	Bright Underline
Bright Green	G+	Bright White
Bright Cyan	BG+	Bright White
Bright Red	R+	Bright White
Bright Magenta	RB+	Bright White
Yellow	GR+	Bright White
Bright White	W+	Bright White
Black	U	Underline
Inverse Video	I	Inverse Video
Blank	X	Blank

Examples

- These two examples demonstrate specifying _gtSetColorStr() with missing settings:

```
#include "gt.api"
.
.
.
    // Settings left off the end
    _gtSetColorStr( "W/N, BG+/B" );
    //
    // Settings skipped within the list
    _gtSetColorStr("W/N, BG+/B,,,W/N");
.
.
.
```

Files

Library is CLIPPER.LIB, header file is Gt.api.

See Also

_gtColorSelect(), _gtGetColorStr(), _gtSetBlink()

_gtSetCursor()

Set the cursor shape

C Prototype

```
#include "gt.api"
ERRCODE _gtSetCursor(
                          USHORT uiCursorShape
                        )
```

Arguments

uiCursorShape is a number indicating the shape of the cursor.

Returns

_gtSetCursor() returns zerc if successful. Any other value indicates an error.

Description

_gtSetCursor() changes the current cursor shape. Before using this function, you may want to save the current cursor shape with _gtGetCursor(), so that you can restore the cursor to its prior shape.

CA-Clipper supplies constants in the Setcurs.ch header file (shown in the table below) that assign descriptive names to various cursor shapes. Using the constants in Setcurs.ch will simplify setting the cursor shape in your programs.

Cursor Shapes

Shape	Value	Setcurs.ch
None	0	SC_NONE
Underline	1	SC_NORMAL
Lower half block	2	SC_INSERT
Full block	3	SC_SPECIAL1
Upper half block	4	SC_SPECIAL2

Examples

- This example uses _gtGetCursor() and _gtSetCursor() to save the current cursor shape, change it, and restore it to its original shape:

```
#include "gt.api"
.
.
.
   USHORT uiSavCursor;

   _gtGetCursor( &uiSavCursor );    // Save cursor shape
   _gtSetCursor( SC_SPECIAL1 );     // Change cursor to a block
   .
   .
   .
  _gtSetCursor( uiSavCursor );      // Restore cursor shape
.
.
.
```

Files Library is CLIPPER.LIB, header file is are Gt.api.

See Also _gtGetCursor()

_gtSetMode()

Change display mode to a specified number of rows and columns

C Prototype

```
#include "gt.api"
ERRCODE _gtSetMode(
                        USHORT uiRows,
                        USHORT uiCols
                    )
```

Arguments

uiRows is the number of rows in the desired display mode.

uiCols is the number of columns in the desired display mode.

Returns

_gtSetMode() returns zero if successful. Any other value indicates an error.

Description

_gtSetMode() attempts to change the mode of the display hardware to match the number of rows and columns specified. The change in screen size is reflected in the values returned by _gtMaxRow() and _gtMaxCol().

Examples

- This example displays a message after successfully selecting 43-line mode:

```
#include "gt.api"
.
.
.
   if ( _gtSetMode(43, 80) = 0 )
      _gtWriteAt( 0, 0, "43-line mode set", 16 );
.
.
.
```

Files Library is CLIPPER.LIB, header file is Gt.api.

See Also _gtMaxCol(), _gtMaxRow(), _gtScrDim()

_gtSetPos()

Move the cursor to a new position

C Prototype

```
#include "gt.api"
ERRCODE _gtSetPos(
                        USHORT uiRow,
                        USHORT uiCol
                    )
```

Arguments

uiRow represents the horizontal position of the cursor. The value may range from zero to _gtMaxRow().

uiCol represents the vertical position of the cursor. The value may range from zero to _gtMaxCol().

Returns

_gtSetPos() returns zero if successful. Any other value indicates an error.

Description

_gtSetPos() moves the cursor to a new position on the screen. After the cursor is positioned, the CA-Clipper functions ROW() and COL() are updated accordingly. Before using this function, you may want to save the current cursor position with _gtGetPos() so that you can restore the cursor to its original position.

To control the shape and visibility of the cursor, use _gtSetCursor().

Examples

- This example creates a simple function that displays a string at a specific screen location:

```
#include "gt.api"

void PrintStr( USHORT uiRow, USHORT uiCol, BYTEP fpStr );

void PrintStr( USHORT uiRow, USHORT uiCol, BYTEP fpStr )
{

   _gtSetPos( uiRow, uiCol );                // Set new position
   _gtWrite( fpStr, strlen( fpStr ) );       // Write string
}
```

Files

Library is CLIPPER.LIB, header file is Gt.api.

See Also

_gtGetCursor(), _gtGetPos(), _gtSetCursor(), _gtWrite(), _gtWriteAt()

_gtSetSnowFlag()

Toggle snow suppression

C Prototype

```
#include "gt.api"
ERRCODE _gtSetSnowFlag(
                              BOOL bNoSnow
                            )
```

Arguments

bNoSnow toggles the current state of snow suppression. A value of true (.T.) enables snow suppression, while a value of false (.F.) disables snow suppression. The default is false (.F.).

Returns

_gtSetSnowFlag() returns zero if successful. Any other value indicates an error.

Description

Use _gtSetSnowFlag() to suppress snow in applications that will be run using a CGA monitor.

Examples

- This simple example turns enables snow suppression:

```
#include "gt.api"
.
.
.
   _gtSetSnowFlag( TRUE );
.
.
.
```

Files

Library is CLIPPER.LIB, header file is Gt.api.

See Also

_gtIsColor()

_gtWrite()

Write a string to the current screen location

C Prototype

```
#include "gt.api"
ERRCODE _gtWrite(
                    BYTEP fpStr,
                    USHORT uiLen
                  )
```

Arguments

fpStr is the string to write.

uiLen is the length of the string.

Returns

_gtWrite() returns zero if successful. Any other value indicates an error.

Description

_gtWrite() displays a string to the screen at the current cursor location. The internal row and column coordinates are updated so the CA-Clipper functions, ROW() and COL() will reflect the new cursor position.

Examples

- This example creates a simple function that displays a null-terminated string at a specific screen location:

```
#include "gt.api"

void PrintStr( USHORT uiRow, USHORT uiCol, BYTEP fpStr );

void PrintStr( USHORT uiRow, USHORT uiCol, BYTEP fpStr )
{

   _gtSetPos( uiRow, uiCol );                // Set new position
   _gtWrite( fpStr, strlen( fpStr ) );       // Write string
   return;
}
```

Files

Library is CLIPPER.LIB, header file is Gt.api.

See Also

_gtGetPos(), _gtRepChar(), _gtSetPos(), _gtWriteAt(), _gtWriteCon()

_gtWriteAt()

Write a string at the specified screen location

C Prototype

```
#include "gt.api"
ERRCODE _gtWriteAt(
                        USHORT uiRow,
                        USHORT uiCol,
                        BYTEP fpStr,
                        USHORT uiLen
                    )
```

Arguments

uiRow and ***uiCol*** are the screen coordinates at which to write the string. *uiRow* may be in the range of zero to _gtMaxRow(). *uiCol* may be in the range of zero to _gtMaxCol().

fpStr is the string to write.

uiLen is the length of the string.

Returns

_gtWriteAt() returns zero if successful. Any other value indicates an error.

Description

_gtWriteAt() writes a string to the screen at the specified coordinates. After the string is written, the internal row and column coordinates are updated so the CA-Clipper functions, ROW() and COL(), will reflect the new cursor position.

Examples

- In this example, the function PrintStr() uses _gtWriteAt() to display a null-terminated string in a specified color:

```
#include "gt.api"

void PrintStr( USHORT uiRow, USHORT uiCol, BYTEP fpStr,
                  BYTEP fpColor );

void PrintStr( USHORT uiRow, USHORT uiCol, BYTEP fpStr,
                  BYTEP fpColor )
{
   _gtSetColorStr( fpColor );
   _gtWriteAt( uiRow, uiCol, fpStr, strlen( fpStr ) );
}
```

Files Library is CLIPPER.LIB, header file is Gt.api.

See Also _gtGetPos(), _gtRepChar(), _gtSetColorStr(), _gtSetPos(), _gtWrite(), _gtWriteCon()

_gtWriteCon()

Perform a console-style write to the screen

C Prototype

```
#include "gt.api"
ERRCODE _gtWriteCon(
                        BYTEP fpStr,
                        USHORT uiLen
                    )
```

Arguments

fpStr is the string to write.

uiLen is the length of the string.

Returns

_gtWriteCon() returns zero if successful. Any other value indicates an error.

Description

_gtWriteCon() writes a character string at the current cursor position using "console" style output. This includes processing control characters, wrapping lines at the last column, and scrolling the screen at the last row. This is similar to the CA-Clipper QQOUT() console function except that the output is always sent to the console. The internal row and column coordinates are updated so the CA-Clipper functions, ROW() and COL(), will reflect the new cursor position.

Examples

- In this simple example, _gtWriteCon() displays the string "Hello" on the screen:

```
_gtWriteCon( "Hello", 5 )
```

Files

Library is CLIPPER.LIB, header file is Gt.api.

See Also

_gtRepChar(), _gtWrite(), _gtWriteAt()

Chapter 13
Replaceable Database Driver API Reference

CA-Clipper provides a default database driver, DBFNTX, and several other RDDs to give you access to the database, memo, and index file formats of many popular database software products. This capability allows CA-Clipper applications to access and manipulate files created by other database engines. The RDD API is a system that lets you create your own RDDs.

In This Chapter

This chapter serves as a reference to all of the structures and methods in the CA-Clipper Replaceable Database Driver (RDD) API. This API gives your C and Assembly routines access to the CA-Clipper Replaceable Database Driver system.

This chapter is broken up into the following sections that categorize the structures and methods. The items in each section are arranged in alphabetical order:

- ***Data Structures:*** Definitions for all of the data structures used by this API, including a description of each structure element and a cross reference to methods and other structures that make use of the current structure. Structure names are defined using all uppercase letters.
- ***Work Area Methods:*** Methods designed for movement and positioning of the work area record pointer.
- ***Data Management Methods:*** Methods designed for managing data at the field/record level.

- ***Work Area/Database Management Methods:*** Methods designed for use at the work area level.
- ***Relational Operation Methods:*** Methods designed for creating and manipulating relations between work areas.
- ***Order Management Methods:*** Methods designed for ordering the records in a work area.
- ***Filter and Scoping Methods:*** Methods designed for setting filters and specifying record scopes in a work area.
- ***Network Operation Methods:*** Methods designed for performing locks/unlocks in a network environment.
- ***Memo File Methods:*** Methods designed for creating and manipulating memo files.
- ***Miscellaneous Methods:*** Remaining methods in the RDD API.

The prototypes for these structures and methods are defined in the header file Rdd.api, located in the \CLIP53\INCLUDE directory. The data types used in the prototypes are defined in the header file Clipdefs.h (automatically included by Rdd.api) or one of the .api header files, also located in \CLIP53\INCLUDE.

Data Structures

AREA structure
DBEVALINFO structure
DBFIELDINFO structure
DBFILTERINFO structure
DBLOCKINFO structure
DBOPENINFO structure
DBORDERCONDINFO structure
DBORDERCREATEINFO structure
DBORDERINFO structure
DBRELINFO structure
DBSCOPEINFO structure
DBSORTINFO structure
DBSORTITEM structure
DBTRANSINFO structure
DBTRANSITEM structure
FIELD structure
RDDFUNCS structure

AREA structure

Information to administrate the work area

Structure

```
typedef struct _AREA
{
   struct _RDDFUNCS far * lprfsHost;

   USHORT uiArea;
   FARP   atomAlias;

   USHORT  uiFieldExtent;
   USHORT  uiFieldCount;
   LPFIELD lpFields;
   FARP    lpFieldExtents;

   ITEM valResult;

   BOOL fTop;
   BOOL fBottom;
   BOOL fBof;
   BOOL fEof;
   BOOL fFound;

   DBSCOPEINFO  dbsi;
   DBFILTERINFO dbfi;

   LPDBORDERCONDINFO lpdbOrdCondInfo;

   LPDBRELINFO lpdbRelations;
   USHORT      uiParents;

   HANDLE heap;
   USHORT heapSize;

   USHORT rddID;

} AREA;

typedef AREA far * LPAREA;
```

Elements

lprfsHost

Contains a pointer to the virtual method table for this work area (see RDDFUNCS).

uiArea

Contains a numeric value representing the number assigned to this work area. This number is analogous to the value returned by the CA-Clipper SELECT() function.

atomAlias

Contains a character data type value representing the CA-Clipper ALIAS of the work area.

uiFieldExtent

Contains a numeric value specifying the total number of columns allocated for the work area. Columns are allocated by a call to setFieldExtent().

uiFieldCount

Contains a numeric value specifying the total number of columns currently in use by the work area. *uifieldCount* is incremented with each call to addField().

lpFields

Contains a pointer to an array of FIELD items. *lpFields* defines all the fields used by this work area including type, length, and name (see FIELD).

lpFieldExtents

Contains a pointer to an array of additional field properties.

valResult

Contains an item that is used on occasion by the CA-Clipper runtime system and may be used by the RDD developer as an all purpose result holder for an operation.

fTop

Contains a boolean value that is set to true (.T.) if the record pointer is on the first logical record.

fBottom

Contains a boolean value that is set to true (.T.) if the record pointer is on the last logical record.

fBof

Contains a boolean value that is set to true (.T.) if logical beginning of file is reached.

fEof

Contains a boolean value that is set to true (.T.) if logical end of file is reached.

fFound

Contains a boolean value that is set to true (.T.) when a seek() operation successfully locates a key value.

dbsi

Contains a data structure specifying the scoping condition used by the CA-Clipper LOCATE command (see DBSCOPEINFO).

dbfi

Contains a data structure specifying the filter condition to use for the work area (see DBFILTERINFO).

lpdbOrdCondInfo

Contains a pointer to a data structure specifying the order condition to use for the work area (see DBORDERCONDINFO).

lpdbRelations

Contains information about relations currently in use by the work area (see DBRELINFO).

uiParents

Contains a numeric value specifying the number of parent work areas currently in effect for this work area.

heap

Contains a virtual memory handle that can be used by the RDD developer to allocate a VM heap and make use of the VM heap suballocation routines available through the Virtual Memory API.

heapSize

Contains a numeric value indicating the size of the *heap*.

rddID

Reserved.

Note: In addition to being used by the data structures in the Used By section below, the AREA structure is used by all methods in the RDD API.

Files Header file is Rdd.api.

Used By DBRELINFO, DBTRANSINFO

DBEVALINFO structure

Information needed for a code block evaluation on each row (record) of the work area

Structure

```
typedef struct
{
   ITEM        itmBlock;
   DBSCOPEINFO dbsci;
} DBEVALINFO;

typedef DBEVALINFO far * LPDBEVALINFO;
```

Elements

itmBlock

Contains a code block to be evaluated (with evalBlock()) on each row of the work area that is in the range defined by *dbsci*.

dbsci

Contains a data structure limiting the evaluation of *itmBlock* (see DBSCOPEINFO).

Warning! *You should perform code block evaluations with great caution. Save any states that are likely to change as a result of re-entrance, so you can restore them after the code evaluation.*

Files Header file is Rdd.api.

Used By dbEval()

DBFIELDINFO structure

Information needed to define a column (field) to the work area

Structure

```
typedef struct
{
   BYTEP  atomName;
   USHORT uiType;
   USHORT typeExtended;
   USHORT uiLen;
   USHORT uiDec;
} DBFIELDINFO;

typedef DBFIELDINFO far * LPDBFIELDINFO;
```

Elements

atomName

Contains a pointer to the null-terminated symbol name to assign to the column. When the CA-Clipper runtime system encounters this symbol, it assumes a reference to the column defined by this structure. The format of this column must conform to the symbolic representation rules of CA-Clipper.

uiType

Contains a literal constant indicating the column's CA-Clipper data type. The numeric constants used to indicate the data types are defined in Extend.api and have the following meanings:

DBFIELDINFO Field Data Types

Constant	Meaning
CHARACTER	A CA-Clipper character value
DATE	A CA-Clipper date value
DOUBLE	A CA-Clipper numeric value stored as an XDOUBLE by the Extend System
LOGICAL	A CA-Clipper logical value
MEMO	A CA-Clipper memo value
NUMERIC	A CA-Clipper numeric value stored as a LONG by the Extend System
UNDEF	The CA-Clipper NIL value

typeExtended

Contains a numeric value indicating a user-defined column type. This member is used to provide support for column types not supported by CA-Clipper's default database (.DBF) file format. The intrinsically supported column types are Character, Numeric, Date, Logical, and Memo.

uiLen

Contains a numeric value representing the overall length of the column. RDDs supporting floating point column types should include the decimal point in this value. By convention, variable length columns should store a zero in this element.

uiDec

Contains a numeric value indicating the number of places to the right of the decimal point. This element is only meaningful when the column is a floating point value.

Files Header file is Rdd.api.

Used By addField()

DBFILTERINFO structure

Specifies a filter condition to hide specified rows (records) from access

Structure

```
typedef struct
{
   ITEM itmCobExpr;
   ITEM abFilterText;
   BOOL fFilter;
} DBFILTERINFO;

typedef DBFILTERINFO far * LPDBFILTERINFO;
```

Elements

itmCobExpr / abFilterText

Contain a code block and a character string, respectively, representing the condition that is evaluated at each cursor location. If the result of the evaluation is false (.F.), the cursor location requested is invalid according to the current filter condition.

fFilter

Contains a boolean value that is set to true (.T.) if the filter is active.

Warning! *You should perform code block evaluations with great caution. Save any states that are likely to change as a result of re-entrance, so you can restore them after the code evaluation.*

Files

Header file is Rdd.api.

Used By

AREA, setFilter()

DBLOCKINFO structure

Information necessary to lock a row or table for exclusive access

Structure

```
typedef struct
{
   ULONG  itmRecID;
   USHORT uiMethod;
   BOOL   fResult;
} DBLOCKINFO;

typedef DBLOCKINFO far * LPDBLOCKINFO;
```

Elements

itmRecID

Contains a numeric value indicating the row to lock. This element is only meaningful if *uimethod* is set to DBLM_EXCLUSIVE or DBLM_MULTIPLE.

uiMethod

Contains a literal constant indicating the type of lock to obtain. The numeric constants used for *uimethod* are defined in Rdd.api and have the following meanings:

DBLOCKINFO Lock Type Values

Constant	Meaning
DBLM_EXCLUSIVE	Lock a row, releasing currently locked rows
DBLM_MULTIPLE	Lock a row, maintaining currently locked rows
DBLM_FILE	Lock a table, releasing locks currently held

fResult

Contains a boolean value specifying the success of the lock operation.

Files

Header file is Rdd.api.

Used By

lock()

DBOPENINFO structure

Information about a data store (table) to be opened in the work area

Structure

```
typedef struct
{
   USHORT uiArea;
   BYTEP  abName;
   BYTEP  atomAlias;
   BOOL   fShared;
   BOOL   fReadonly;
   FARP   lpdbHeader;
} DBOPENINFO;

typedef DBOPENINFO far * LPDBOPENINFO;
```

Elements

uiArea

Contains a numeric value representing the work area number where the data store will be opened. This number corresponds directly to the CA-Clipper-language level SELECT() area.

abName

Contains a pointer to a null-terminated string identifying the data store. In most cases, it is a file name.

atomAlias

Contains a pointer to a null-terminated string used from the CA-Clipper-language level to access the new work area. The *atomAlias* element must follow CA-Clipper-level rules for valid symbolic references (i.e., a valid CA-Clipper identifier).

fShared

Contains a boolean value that is set to true (.T.) if the data store is to be opened in a shared (network) mode.

fReadonly

Contains a boolean value that is set to true (.T.) if the data store is to be opened in a read-only mode.

lpdbHeader

Contains a pointer to a header of the data store.

Files Header file is Rdd.api.

Used By create(), open()

DBORDERCONDINFO structure

Information needed for the creation of a conditional order

Structure

```
typedef struct _DBORDERCONDINFO
{
   BOOL  fActive;
   BYTEP abFor;
   ITEM  itmCobFor;
   ITEM  itmCobWhile;
   ITEM  itmCobEval;
   LONG  lStep;
   LONG  lStartRecno;
   LONG  lNextCount;
   LONG  lRecno;
   BOOL  fRest;
   BOOL  fDescending;
   BOOL  fScoped;
   BOOL  fAll;
   BOOL  fAdditive;
   BOOL  fUseCurrent;
   BOOL  fCustom;
   BOOL  fNoOptimize;
   FARP  lpvCargo;
} DBORDERCONDINFO;

typedef DBORDERCONDINFO far * LPDBORDERCONDINFO;
```

Elements

fActive

Contains a boolean value indicating whether one or more valid conditions have been specified in the structure.

abFor / itmCobFor

Contain a CA-Clipper-level reference to a string and a matching code block containing the key expression defining the FOR condition to be used for the creation and maintenance of the order.

itmCobWhile

Contains a CA-Clipper-level reference to a code block defining the WHILE condition to be used for the creation of the order. If NIL, no WHILE condition is being specified.

`itmCobEval`

Contains a CA-Clipper-level reference to a code block defining the expression to be evaluated every *lStep* rows (records) during the creation of the order. The code block referenced should always evaluate to either true (.T.), indicating that creation of the order should continue normally, or false (.F.), indicating that order creation should terminate). A return value of false (.F.) from *itmCobEval* does not constitute an error condition but rather is the mechanism by which an RDD is informed that no more rows are to be included in the order.

`lStep`

Contains a numeric value determining the frequency of the evaluation of *itmCobEval*.

`lStartRecno`

Contains a numeric value indicating the row at which to begin processing when either the *lNextCount* or *fRest* scoping options are specified.

`lNextCount`

Contains a numeric value indicating the number of rows to process for order creation.

`lRecno`

Contains a numeric value indicating a single row to include in the order.

`fRest`

Contains a boolean value indicating that only the rows specified by *lStartRecno* through end of file (EOF) are to be included in the order.

`fDescending`

Contains a boolean value indicating whether the order should be created in descending order.

fScoped

Contains a boolean value indicating whether the order is to be scoped. *fScoped* will be true (.T.) if any of *itmCobWhile, lNextCount, lRecno, fRest*, or *fAll* are specified.

fAll

Contains a boolean value indicating whether all rows are to be processed during order creation.

fAdditive

Contains a boolean value indicating whether open orders should remain open while the new order is being created.

fUseCurrent

Contains a boolean value indicating whether only rows in the controlling order will be included in this order.

fCustom

Contains a boolean value indicating whether the new order will be a custom-built order.

fNoOptimize

Contains a boolean value indicating whether the FOR condition will be optimized.

lpvCargo

Contains a pointer to a value of any data type provided as a user-definable slot. It may be used by the RDD developer as an all-purpose holder.

Files Header file is Rdd.api.

Used By DBORDERCREATEINFO, orderCreate()

DBORDERCREATEINFO structure

Information needed to create a new order for the work area

Structure

```
typedef struct
{
   LPDBORDERCONDINFO lpdbOrdCondInfo;
   BYTEP             abBagName;
   BYTEP             atomBagName;
   ITEM              itmOrder;
   BOOL              fUnique;
   ITEM              itmCobExpr;
   ITEM              abExpr;
} DBORDERCREATEINFO;

typedef DBORDERCREATEINFO far * LPDBORDERCREATEINFO;
```

Elements

lpdbOrdCondInfo

Contains a pointer to information about the condition (if any) for the order.

abBagName

Contains a pointer to a null-terminated string containing the index file name to create.

atomBagName

Reserved CA-Clipper compatibility slot.

itmOrder

Contains a CA-Clipper-level reference to a null-terminated string containing the order name or number to create in *abBagName*.

fUnique

Contains a boolean value specifying whether the order should contain only unique keys.

itmCobExpr / abExpr

Contain a CA-Clipper-level reference to a code block and a matching string containing the key expression defining the order imposed on the work area.

Warning! *You should perform code block evaluations with great caution. Save any states that are likely to change as a result of re-entrance, so you can restore them after the code evaluation.*

Files Header file is Rdd.api.

Used By orderCreate()

DBORDERINFO structure

Information needed to open/address an order in the work area

Structure

```
typedef struct
{
   ITEM atomBagName;
   ITEM itmOrder;
   ITEM itmCobExpr;
   ITEM itmResult;
   BOOL fAllTags;

} DBORDERINFO;

typedef DBORDERINFO far * LPDBORDERINFO;
```

Elements

atomBagName

Contains a CA-Clipper-level reference to a character value indicating the name of the order bag.

itmOrder

Contains a CA-Clipper-level reference to a character value indicating the name of the order.

itmCobExpr

Contains a CA-Clipper-level reference to a code block containing the key expression defining the order imposed on the work area.

itmResult

Contains an item that is used on occasion by the CA-Clipper runtime system and may be used by the RDD developer as an all-purpose result holder.

fAllTags

Contains a boolean value specifying whether all tags of the index file must be opened.

Files

Header file is Rdd.api.

Used By

orderListAdd(), orderInfo(), orderListFocus()

DBRELINFO structure

List of relational information

Structure

```
typedef struct _DBRELINFO
{
   ITEM itmCobExpr;
   ITEM abKey;

   struct _AREA far * lpaParent;
   struct _AREA far * lpaChild;

   struct _DBRELINFO far * lpdbriNext;

} DBRELINFO;

typedef DBRELINFO far * LPDBRELINFO;
```

Elements

itmCobExpr / abKey

Contain a code block and a character expression, respectively, used to reposition the cursor of the child database when this relation is resolved.

lpaParent

Contains a pointer to the parent work area of this relation. When you move the cursor of a parent work area to a new position, you must move the cursor of each of its children to corresponding positions (usually with forceRel()). Find the corresponding position by performing a seek() for the result of the *abKey*.

lpaChild

Contains a pointer to the work area containing the child in this relation.

lpdbriNext

Contains a pointer identifying the next field in the field list. Note that a null pointer in this field indicates the end of the list. By convention, this list is not circular.

Warning! *You should perform code block evaluations with great caution. Save any states that are likely to change as a result of re-entrance, so you can restore them after the code evaluation.*

Files Header file is Rdd.api.

Used By AREA, childEnd(), childStart(), relEval(), setRel()

DBSCOPEINFO structure

References to all of the CA-Clipper Xbase-style scope clause expressions

Structure

```
typedef struct
{
   ITEM itmCobFor;
   ITEM lpstrFor;
   ITEM itmCobWhile;
   ITEM lpstrWhile;
   ITEM lNext;
   ITEM itmRecID;
   ITEM fRest;

   BOOL fIgnoreFilter;
   BOOL fIncludeDeleted;
   BOOL fLast;
   BOOL fIgnoreDuplicates;
} DBSCOPEINFO;

typedef DBSCOPEINFO far * LPDBSCOPEINFO;
```

Elements

itmCobFor / lpstrFor

Contain a code block and a character expression, respectively, representing the conditional FOR clause. FOR expressions are, essentially, filters that hide rows (records) which evaluate to false (.F.). The character value is provided for storage, while the code block is provided as a parameter for the evalBlock() method.

itmCobWhile / lpstrWhile

Contain a code block and a character expression, respectively, representing the conditional WHILE clause. WHILE clauses permit continuation of a process that steps through rows until its expression evaluates to false (.F.). The character value is provided for storage, while the code block is provided as a parameter for the evalBlock() method.

lNext

Contains a numeric value representing the CA-Clipper-level NEXT *lNext* scoping clause. This permits continuation of a process for the NEXT *lNext* rows, while obeying FOR and WHILE clauses.

`itmRecID`

Contains a numeric value representing the CA-Clipper-level RECORD *itmRecID* scoping clause. This permits continuation of a process for a single record, while obeying FOR and WHILE clauses.

`fRest`

Contains a CA-Clipper-level boolean value that is set to true (.T.) if a process should continue stepping through data from the current work area cursor position until logical end of file.

`fIgnoreFilter`

Contains a CA-Clipper-level boolean value that is set to true (.T.) if a process should ignore any filter condition imposed on the current work area.

`fIncludeDeleted`

Contains a CA-Clipper-level boolean value that is set to true (.T.) if a process should include deleted rows.

`fLast`

Contains a CA-Clipper-level boolean value that is set to true (.T.) if the last record of the current scope is required.

`fIgnoreDuplicates`

Contains a CA-Clipper-level boolean value that is set to true (.T.) if a process should ignore duplicate key values.

Warning! *You should perform code block evaluations with great caution. Save any states that are likely to change as a result of re-entrance, so you can restore them after the code evaluation.*

Files Header file is Rdd.api.

Used By AREA, DBEVALINFO, DBTRANSINFO, setLocate()

DBSORTINFO structure

Information needed to perform a physical sort on the work area

Structure

```
typedef struct
{
   DBTRANSINFO  dbtri;
   LPDBSORTITEM lpdbsItem;
   USHORT       uiItemCount;

} DBSORTINFO;

typedef DBSORTINFO far * LPDBSORTINFO;
```

Elements

dbtri

Contains a data structure holding the destination work area, field transfer information, control and optimization flags, and scoping information for the sort() method (see DBTRANSINFO).

lpdbsItem

Contains an array of DBSORTITEM items (rows) which compose the key values for the sort (see DBSORTITEM).

uiItemCount

Contains the number of DBSORTITEM items allocated to *lpdbsItem*.

Files

Header file is Rdd.api.

Used By

sort()

DBSORTITEM structure

Array of items indicating the key values, in order, to use when sorting data

Structure

```
typedef struct
{
   USHORT uiField;
   USHORT uiFlags;
} DBSORTITEM;

typedef DBSORTITEM far * LPDBSORTITEM;
```

Elements

uiField

Contains a numeric value; an index into the *AREA->lpFields* structure indicating the field on which the sort is based.

uiFlags

Contains literal constants that function as sort optimization and control flags. They are passed to your RDD sort() routine from the high-level wrapper function for the CA-Clipper SORT command. The flag values are defined in Rdd.api and have the following meanings:

DBSORTITEM Sort Flag Values

Constant	Meaning
SF_ASCEND	An ascending sort
SF_CASE	A case-insensitive sort
SF_DESCEND	A descending sort
SF_NUM	The sort is on numeric data only
SF_DOUBLE	The sort is to be done exclusively with floating point numbers
SF_LONG	The sort is to be done exclusively with integers

Files

Header file is Rdd.api.

Used By

DBSORTINFO

DBTRANSINFO structure

Define a global transfer of data items from one work area to another

Structure

```
typedef struct
{
   struct _AREA   far * lpaSource;
   struct _AREA   far * lpaDest;
   DBSCOPEINFO    dbsci;
   USHORT         uiFlags;
   USHORT         uiItemCount;
   LPDBTRANSITEM lpTransItems;
} DBTRANSINFO;

typedef DBTRANSINFO far * LPDBTRANSINFO;
```

Elements

lpaSource

Contains a pointer to the source work area.

lpaDest

Contains a pointer to the destination work area.

dbsci

Contains a data structure describing the limits of the scope of the transfer (see DBSCOPEINFO).

uiFlags

Contains literal constants specifying transfer attributes. The flag values are defined in Rdd.api and have the following meanings:

DBTRANSINFO Transfer Flag Values

Constant	Meaning
DBTF_MATCH	Both this work area and the destination work area have identical row structures (i.e., all fields match)
DBTF_PUTREC	The RDD has the ability to transfer an entire row

uiItemCount

Contains a numeric value specifying the number of items in the *lpTransItems* array. (This is usually the number of fields to be transferred).

lpTransItems

An array of character data type values, assumed to be the items to transfer to the destination work area. *lpTransItems* is usually a list of field mappings from the source to the destination (see DBTRANSITEM).

Files Header file is Rdd.api.

Used By DBSORTINFO, trans(), transRec()

DBTRANSITEM structure

Define a single transfer item (usually a field) from one work area to another

Structure

```
typedef struct
{
   USHORT uiSource;
   USHORT uiDest;
} DBTRANSITEM;

typedef DBTRANSITEM far * LPDBTRANSITEM;
```

Elements

uiSource

Contains a numeric value; an index into the *AREA->lpFields* structure indicating the field that is to be transferred to the destination work area.

uiDest

Contains a numeric value; an index into the *AREA->lpfields* structure indicating the field that is to be the recipient of the row being transferred from the source work area.

Files Header file is Rdd.api.

Used By DBTRANSINFO

FIELD structure

Field (column) definitions

Structure

```
typedef struct _FIELD
{
   USHORT uiType;
   USHORT uiTypeExtended;
   USHORT uiLen;
   USHORT uiDec;
   USHORT uiArea;
   FARP   sym;

   struct _FIELD * lpfNext;

} FIELD;

typedef FIELD far * LPFIELD;
```

Notes

The pointer to an array of FIELD structures is contained in the basic AREA structure. Each field is created by a call to the addField() method for a given RDD.

Elements

uiType

Contains a literal constant indicating the column's CA-Clipper data type. The numeric constants used to indicate the data types are defined in Extend.api and have the following meanings:

FIELD Data Types

Constant	Meaning
CHARACTER	A CA-Clipper character value
DATE	A CA-Clipper date value
DOUBLE	A CA-Clipper numeric value stored as an XDOUBLE by the Extend System
LOGICAL	A CA-Clipper logical value
MEMO	A CA-Clipper memo value
NUMERIC	A CA-Clipper numeric value stored as a LONG by the Extend System
UNDEF	The CA-Clipper NIL value

uiTypeExtended

Contains a numeric value indicating a user-defined column type. This member is used to provide support for column types not supported by CA-Clipper's default database (.DBF) file format. The intrinsically supported column types are character, numeric, date, logical, and memo. A non-zero value in this field indicates that the column type is not one of the intrinsically supported types. This allows the RDD to support column types that are not intrinsically understood by CA-Clipper. Extended types must be translated into CA-Clipper types by the RDD and *uiTypeExtended* values are defined by the RDD.

uiLen

Contains a numeric value representing the overall length of the column. RDDs supporting floating point column types should include the decimal point in this value. By convention, variable length columns should store a zero (0) in this element.

uiDec

Contains a numeric value indicating the number of places to the right of the decimal point. This element is only meaningful when the column is a floating point value.

uiArea

Contains a numeric value indicating the work area in which the field resides. The CA-Clipper runtime system uses this field to reference the appropriate work area.

sym

Contains a character data type value representing the column and work area to the runtime system. The CA-Clipper runtime system will interpret runtime requests (get or set actions) to this symbol as a call to your driver.

lpfNext

Reserved.

Files Header file is Rdd.api.

Used By AREA

RDDFUNCS structure

The virtual method table for the work area

Structure

```
typedef struct _RDDFUNCS
{

   /* Movement and positioning methods */

   DBENTRYP_SP  bof;
   DBENTRYP_SP  eof;
   DBENTRYP_SP  found;
   DBENTRYP_V   goBottom;
   DBENTRYP_L   go;
   DBENTRYP_I   goToId;
   DBENTRYP_V   goTop;
   DBENTRYP_SI  seek;
   DBENTRYP_L   skip;
   DBENTRYP_L   skipFilter;
   DBENTRYP_L   skipRaw;

   /* Data management */

   DBENTRYP_VP  addField;
   DBENTRYP_S   append;
   DBENTRYP_I   createFields;
   DBENTRYP_V   delete;
   DBENTRYP_SP  deleted;
   DBENTRYP_SP  fieldCount;
   DBENTRYP_VP  fieldDisplay;
   DBENTRYP_SSI fieldInfo;
   DBENTRYP_SVP fieldName;
   DBENTRYP_V   flush;
   DBENTRYP_PP  getRec;
   DBENTRYP_SI  getValue;
   DBENTRYP_SVP getVarLen;
   DBENTRYP_V   goCold;
   DBENTRYP_V   goHot;
   DBENTRYP_VP  putRec;
   DBENTRYP_SI  putValue;
   DBENTRYP_V   recall;
   DBENTRYP_LP  reccount;
   DBENTRYP_ISI recInfo;
   DBENTRYP_I   recno;
   DBENTRYP_S   setFieldExtent;

   /* Workarea/Database management */

   DBENTRYP_VP  alias;
   DBENTRYP_V   close;
   DBENTRYP_VP  create;
   DBENTRYP_SI  info;
   DBENTRYP_V   new;
```

```
DBENTRYP_VP  open;
DBENTRYP_V   release;
DBENTRYP_SP  structSize;
DBENTRYP_VP  sysName;
DBENTRYP_VP  dbEval;
DBENTRYP_V   pack;
DBENTRYP_LSP packRec;
DBENTRYP_VP  sort;
DBENTRYP_VP  trans;
DBENTRYP_VP  transRec;
DBENTRYP_V   zap;

/* Relational Methods */

DBENTRYP_VP  childEnd;
DBENTRYP_VP  childStart;
DBENTRYP_VP  childSync;
DBENTRYP_V   syncChildren;
DBENTRYP_V   clearRel;
DBENTRYP_V   forceRel;
DBENTRYP_SVP relArea;
DBENTRYP_VP  relEval;
DBENTRYP_SVP relText;
DBENTRYP_VP  setRel;

/* Order Management */

DBENTRYP_VP  orderListAdd;
DBENTRYP_V   orderListClear;
DBENTRYP_VP  orderListDelete;
DBENTRYP_VP  orderListFocus;
DBENTRYP_V   orderListRebuild;

DBENTRYP_VP  orderCondition;
DBENTRYP_VP  orderCreate;
DBENTRYP_VP  orderDestroy;
DBENTRYP_SVP orderInfo;

/* Filters and Scope Settings */

DBENTRYP_V    clearFilter;
DBENTRYP_V    clearLocate;
DBENTRYP_V    clearScope;
DBENTRYP_VPLP countScope;
DBENTRYP_VP   filterText;
DBENTRYP_SI   scopeInfo;
DBENTRYP_VP   setFilter;
DBENTRYP_VP   setLocate;
DBENTRYP_VP   setScope;
DBENTRYP_VLP  skipScope;

/* Miscellaneous */

DBENTRYP_VP  compile;
DBENTRYP_VP  error;
DBENTRYP_I   evalBlock;
```

```
   /* Network operations */

   DBENTRYP_VSP rawLock;
   DBENTRYP_VP  lock;
   DBENTRYP_L   unlock;

   /* Memo file methods */

   DBENTRYP_V    closeMemFile;
   DBENTRYP_VP   createMemFile;
   DBENTRYP_SVP  getValueFile;
   DBENTRYP_VP   openMemFile;
   DBENTRYP_SVP  putValueFile;

   /* Database file header methods */

   DBENTRYP_V    readDBHeader;
   DBENTRYP_V    writeDBHeader;


   /* Special and reserved methods */

   DBENTRYP_SVP  whoCares;

} RDDFUNCS;

typedef RDDFUNCS  far * LPRDDFUNCS;
typedef LPRDDFUNCS far * LPLPRDDFUNCS;
```

Elements

The elements of the RDDFUNCS structure define the *slots* of the virtual method table. Each slot in the table represents the location of a specific method (i.e., the address of the function that acts as the *pseudomethod*). Methods that will not be supported by the RDD should have a NULL placed in its corresponding slot. When the RDD is initialized, CA-Clipper will use the default behavior of the superclass instead. If no functionality is provided by the superclass, CA-Clipper will generate an appropriate "unsupported" error.

Warning! *Under no circumstance should the virtual method table be expanded. The size and ordinal positioning of the methods in the table is critical. Failure to adhere to this arrangement will cause unpredictable results and potential memory corruption.*

Files

Header file is Rdd.api.

Used By

AREA

Work Area Methods

bof( ) method
eof( ) method
found( ) method
go( ) method
goBottom( ) method
goToId() method
goTop( ) method
seek( ) method
skip( ) method
skipFilter( ) method
skipRaw( ) method

bof() method

Determine logical beginning of file

Prototype

```
ERRCODE bof(
              AREAP wa,
              BOOLP isBof
            )
```

Arguments

wa is a pointer to self.

isBof is used to determine the logical beginning of file.

Description

bof() indicates whether the current cursor for *wa* is at the logical beginning of file. This method provides the return value of the CA-Clipper BOF() function.

Default Behavior

Copies *wa->fBof* to *isBof*.

Implementation Notes

- Determine *wa->fBof* status and then call SUPER_BOF().
- If your RDD supports relations, update relations with a forceRel() prior to determining *wa->fBof* status.

Warning! isBof *must point to sizeof(BOOL) bytes of allocated memory.*

Files Header file is Rdd.api.

See Also AREA, eof(), forceRel()

eof() method

Determine logical end of file

Prototype

```
ERRCODE eof(
              AREAP wa,
              BOOLP isEof
            )
```

Arguments

wa is a pointer to self.

isEof determines the logical end of file.

Description

eof() indicates whether the current cursor for *wa* is at the logical end of file. This method provides the return value of the CA-Clipper EOF() function.

Default Behavior

Copies *wa->Eof* to *isEof*.

Implementation Notes

- Determine *wa->Eof* status and then call SUPER_EOF().
- If your RDD supports relations, update relations with a forceRel() prior to determining *wa->Eof* status.

Warning! isEof *must point to one byte of allocated memory.*

Files

Header file is Rdd.api.

See Also

AREA, bof(), forceRel()

found() method

Determine outcome of the last search operation

Prototype

```
ERRCODE found(
                    AREAP wa,
                    BOOLP isFound
                  )
```

Arguments

wa is a pointer to self.

isFound determines the status of the last search operation.

Description

found() obtains the status of the last search operation for *wa*. In the default database driver, search operations are defined as seek() calls. This method provides the return value of the CA-Clipper FOUND() function.

Default Behavior

Copies *wa->fFound* to *isFound*.

Implementation Notes

- Determine the setting of *wa->fFound* (if it is not automatically set by your seek() implementation) and then call SUPER_FOUND().
- If your RDD supports relations, update relations with a forceRel() prior to determining *wa->fFound* status.

Warning! isFound *must point to one byte of allocated memory.*

Files Header file is Rdd.api.

See Also AREA, forceRel(), seek()

go() method

Position cursor at a specific physical identity (record)

Prototype

```
ERRCODE go(
                AREAP wa,
                ULONG row
            )
```

Arguments

wa is a pointer to self.

row is the row number of the new cursor position.

Description

go() repositions the work area cursor for *wa* to the physical row number *row*.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- Since go() repositions the pointer to a *physical* row number, you need not consider active scopes or filters. If the *row* is not in the physical range of rows available, you may generate an error or position the pointer at the physical end of file.
- In CA-Clipper, the idea of a *phantom row* is vital to the performance of certain functions. The go() method should permit access to the phantom row. The phantom row is defined as the last row + 1 and is the position at which end of file is set to true (.T.). All columns of the phantom row are empty. A *row* value greater than the last row in the table should position the pointer to the phantom row. If you do not support the concept of the phantom row in your driver, you must document this fact as an incompatibility of your driver with the high-level CA-Clipper syntax.

Files Header file is Rdd.api.

See Also AREA, goBottom(), goTop()

goBottom() method

Position cursor at the last logical identity (record)

Prototype

```
ERRCODE goBottom(
                    AREAP wa
                  )
```

Arguments

wa is a pointer to self.

Description

goBottom() repositions the work area cursor for *wa* to the last logical data item.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- When positioning to the last *logical* data item, goBottom() should respect any active scopes or filters. This method emulates the CA-Clipper GOTO BOTTOM command.
- If an order is attached to your implementation of work area, you must use the bottom data item based on the correct row order and respect any scoping or filtering.

Files Header file is Rdd.api.

See Also AREA, go(), goTop()

goTold() method

Position the cursor to a specific, physical identity

Prototype

```
ERRCODE goToId(
                    AREAP wa,
                    ITEM itmRecID
                )
```

Arguments

wa is a pointer to self.

itmRecID is the physical identity of the row. This may be a row number or some other unique identifier.

Description

goTold() repositions the work area cursor for *wa* to the physical row identifier *itmRecID*.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- This method is used to implement the DBGOTO() function. It is designed so that the driver can identify rows using a means other than the physical row number.
- Since goToId() repositions the cursor to a physical row number, you need not consider active scopes, filters, or orders. If *lRec* is not in the range of available rows, you can generate an error or position the cursor to the physical end of file.
- In CA-Clipper, the idea of a phantom row is vital to the performance of certain functions. The goToId() method should permit access to the phantom row. The phantom row is defined as the last row + 1 and is the position at which end of file is set to true (.T.). All columns of the phantom row are empty. An *lRec* value greater than the last row in the table should position the cursor to the phantom row. If you do not support the concept of the phantom row in your driver, you must document this fact as an incompatibility between your driver and the high-level CA-Clipper functionality.

Files Header file is Rdd.api.

See Also AREA, go(), goBottom(), goTop()

goTop() method

Position cursor at the first logical identity (record)

Prototype

```
ERRCODE goTop(
                    AREAP wa
                )
```

Arguments

wa is a pointer to self.

Description

goTop() repositions the work area cursor for *wa* to the first logical data item. This method emulates the CA-Clipper GOTO TOP command.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- When positioning to the first *logical* data item, goTop() should respect any active scopes or filters.
- If an order is attached to your implementation of work area, you must use the top data item based on the correct row order and respect any scoping or filtering.

Files Header file is Rdd.api.

See Also AREA, goBottom(), go()

seek() method

Position cursor at first row with matching key value

Prototype

```
ERRCODE seek(
                AREAP wa,
                BOOL isSoft,
                ITEM keyVal
            )
```

Arguments

wa is a pointer to self.

isSoft is set when a softseek is to be performed.

keyVal is the search value.

Description

seek() positions the cursor for *wa* to the row whose key value matches *keyVal.*

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- The definition of seek() assumes that an order has been imposed on the work area. You must match *keyVal* against the key value of this ordering.
- Once the seek() has been performed, your implementation of seek() should set the *wa->fFound* flag, if possible. If you cannot set this flag at the time of the seek(), you should investigate subclassing the found() method to tailor it to your specific needs.
- When the CA-Clipper SOFTSEEK-style SEEK is required, the boolean value *isSoft* should be true (.T.). See SET SOFTSEEK in the *Reference Guide, Volumes 1 and 2* for more information.

Files Header file is Rdd.api.

See Also AREA, found()

skip() method

Reposition cursor relative to current position

Prototype

```
ERRCODE skip(
                    AREAP wa,
                    LONG count
                )
```

Arguments

wa is a pointer to self.

count is the number of rows to skip.

Description

skip() skips *count* number of rows (*count* may be negative, indicating reverse traversal), obeying filter and scope conditions. This method is intended to provide the functionality of the CA-Clipper SKIP command.

Default Behavior

The default implementation of skip() calls skipFilter() and skipRaw() to successfully navigate the work area. skip() also sets *wa->fTop*, *wa->fBottom*, *wa->fBof*, and *wa->fEof*, when appropriate.

Note: As implemented, skip() may pass a zero (0) value to skipRaw() indicating that a COMMIT (flush()) is to be performed.

Implementation Notes

- If an order is attached to your implementation of work area, skip() should support this new ordering as well as any condition or filtering behavior.
- If your RDD supports relations, update them with a forceRel() prior to attempting relative movement.

Files Header file is Rdd.api.

See Also AREA, flush(), forceRel(), go(), skipFilter(), skipRaw()

skipFilter() method

Reposition cursor respecting any filter setting

Prototype

```
ERRCODE skipFilter(
                        AREAP wa,
                        LONG count
                      )
```

Arguments

wa is a pointer to self.

count is the number of rows to skip.

Description

skipFilter() skips to the next visible record, obeying filter and scope conditions as well as the deleted row settings. skipFilter() is provided for skip() optimization.

Default Behavior

Sets *wa->fBof* and *wa->fEof*, as appropriate.

Implementation Notes

- If your RDD supports relations, update relations with a forceRel() prior to attempting relative movement.

Files Header file is Rdd.api.

See Also AREA, forceRel(), skip(), skipRaw()

skipRaw() method

Reposition cursor, regardless of filter

Prototype

```
ERRCODE skipRaw(
                    AREAP wa,
                    LONG numToSkip
                )
```

Arguments

wa is a pointer to self.

numToSkip is the number of rows to skip. A positive number indicates moving forward in the table, while a negative number indicates backwards movement.

Description

skipRaw() relatively repositions the work area cursor *numToSkip* rows regardless of filter settings. skipRaw() is the base movement method for all relative work area movements.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- If an order is imposed on the work area by an index or other such structure, skipRaw() should always follow that order.
- In some systems, an absolute reposition using go() may be an appropriate implementation of skipRaw().
- skipRaw() must be able to accept a zero value for *numToSkip*, indicating that the row at the current cursor position should be committed (see flush()).

Files Header file is Rdd.api.

See Also AREA, flush(), go(), skip(), skipFilter()

Data Management Methods

addField( ) method
append( ) method
createFields( ) method
delete( ) method
deleted( ) method
fieldCount( ) method
fieldDisplay( ) method
fieldInfo( ) method
fieldName( ) method
flush( ) method
getRec( ) method
getValue( ) method
getVarLen( ) method
goCold( ) method
goHot( ) method
putRec( ) method
putValue( ) method
recall( ) method
reccount( ) method
recInfo( ) method
recno( ) method
setFieldExtent( ) method

addField() method

Add a column (field) to the work area

Prototype

```
ERRCODE addField(
                    AREAP wa,
                    LPDBFIELDINFO lpdbFieldInfo
                  )
```

Arguments

wa is a pointer to self.

lpdbFieldInfo is a pointer to a structure containing information about a column in the work area.

Description

addField() adds a column specified by *lpdbFieldInfo* to the *wa* work area.

Default Behavior

addField() inserts the column referred to by *lpdbFieldInfo* into the data table referenced by the AREA structure. The new column is added to the end of the *wa->lpFields* structure. *wa->uiFieldCount* is updated to reflect the addition.

Note: Call setFieldExtent() prior to adding fields to the AREA structure. setFieldExtent() sizes and clears the *wa->lpFields* structure, readying it for use.

***Warning! wa->uiFieldCount** must be less than **wa->uiFieldExtent** or an assertion error will be raised.*

Implementation Notes

- Most implementations should use the default behavior for addField(). However, if you change the addField() implementation, be aware of the order imposed on adding fields to the work area structures.

Files Header file is Rdd.api.

See Also AREA, DBFIELDINFO, createFields(), setFieldExtent()

append() method

Append a row (record) to the work area

Prototype

```
ERRCODE append(
                    AREAP wa
                )
```

Arguments

wa is a pointer to self.

Description

append() appends a row to the work area defined by *wa*.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- append() allows the addition of a blank record to the work area defined in *wa* and places the work area cursor at the new row position.
- append() should make a call to goCold() before appending a row and resolve (or cancel) pending relational moves.
- On work areas marked as *wa->fShared*, pay strict attention to network etiquette during the append. On work areas marked as *wa->fReadonly*, raise a recoverable error.

Files

Header file is Rdd.api.

See Also

AREA, goCold()

createFields() method

Add all columns defined in an array to the work area

Prototype

```
ERRCODE createFields(
                        AREAP wa,
                        FARP  lpvStru
                      )
```

Arguments

wa is a pointer to self.

lpvStru is a pointer to an array containing the column definitions to add. The length of this array is equal to the number of columns to add. Each element of *lpvStru* is a subarray containing information about a single column.

Default Behavior

createFields() is used to implement the CA-Clipper DBCREATE() function. It adds the columns, in the order specified by *lpvStru*, by calling addField() for each subarray.

Implementation Notes

- Most implementations should use the default behavior for createFields(). However, if you change the createFields() implementation, be aware of the order imposed on adding fields to the work area structures.

Files Header file is Rdd.api.

See Also AREA, DBFIELDINFO, addField(), setFieldExtent()

delete() method

Mark a row (record) for deletion

Prototype

```
ERRCODE delete(
                  AREAP wa
               )
```

Arguments

wa is a pointer to self.

Description

delete() marks the row at the current cursor location in *wa* for deletion.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- The delete() method marks the data item at the work area's cursor for deletion. The delete() mechanism for databases must deal with the current work area cursor after the deletion.
- If your RDD supports relations, update them with a forceRel() prior to attempting the deletion.
- On work areas marked as *wa->fReadonly*, raise a recoverable error. On work areas marked as *wa->fShared*, you must address network contingencies.
- Some RDDs may want to perform a physical table deletion.

Files

Header file is Rdd.api.

See Also

AREA, deleted(), forceRel(), pack(), recall()

deleted() method

Determine deleted status for a row (record)

Prototype

```
ERRCODE deleted(
                    AREAP wa,
                    BOOLP isDeleted
                  )
```

Arguments

wa is a pointer to self.

isDeleted is a pointer to a boolean value that determines whether a row is deleted.

Description

deleted() queries the row at the current work area cursor to determine if it has been marked for deletion.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- If your RDD supports relations, update them with a forceRel() prior to attempting relative movement.

Warning! isDeleted *must point to sizeof(BOOL) bytes of allocated memory.*

Files Header file is Rdd.api.

See Also AREA, delete(), forceRel(), recall()

fieldCount() method

Determine the number of columns (fields) in the work area

Prototype

```
ERRCODE fieldCount(
                        AREAP wa,
                        USHORTP fieldCount
                      )
```

Arguments

wa is a pointer to self.

fieldCount is a pointer to an unsigned short integer value that specifies the number of columns in the work area.

Description

fieldCount() gets the physical column count for *wa*.

Default Behavior

Copies *wa->uiFieldCount* into *fieldCount*.

Implementation Notes

- If the number of columns is dynamic, fieldCount() should return the count for the current row (record).

Warning! fieldCount *must point to sizeof(USHORT) bytes of allocated memory.*

Files Header file is Rdd.api.

See Also AREA, fieldName(), reccount()

fieldDisplay() method

This method is reserved for future use.

fieldInfo() method

Retrieve information about a column.

Prototype

```
ERRCODE fieldInfo(
                        AREAP  wa,
                        USHORT uiFieldNum,
                        USHORT uiInfoType,
                        ITEM   itmInfo
                      )
```

Arguments

wa is a pointer to self.

uiFieldNum specifies the ordinal position of the column from which information will be retrieved.

uiInfoType specifies the type of the information to be provided.

itmInfo is a pointer to a CA-Clipper item which will contain the field information.

Description

fieldInfo() retrieves information about the column. The information requested is defined by the value passed in *uiInfoType*. The field information that is available is defined by the RDD. In the DBF work area model, this is limited to the information stored in the DBF file structure (that is, name, length, number of decimals, and data type).

Default Behavior

The fieldInfo() method is used to implement the CA-Clipper DBFIELDINFO() function. In the default implementation, there are four properties (shown in the table below) defined for each column (the numeric constants are defined in the header file Rdd.api):

fieldInfo() Information types

Constant	Meaning
DBS_NAME	Obtain the field's name
DBS_LEN	Obtain the field's length
DBS_DEC	Obtain the number of decimal places
DBS_TYPE	Obtain the field's type

Warning! ***itmInfo*** *must be a valid item.*

Implementation Notes

- You must implement new behavior for this method only if your driver requires properties in addition to those listed in the previous table. Note that the properties listed in the table are required for every implementation.
- If your implementation of fieldInfo() cannot determine the return value based on the value of *uiInfoType*, you should allow the work area default implementation to attempt it by calling SUPER_FIELDINFO().
- If *itmInfo* contains a value other than NIL, it is the new value for the property. If your implementation warrants, you can change *itmInfo* by assigning the new value to it.
- The first 1000 possible values for *uiInfoType* are reserved by CA-Clipper.

Files Header file is Rdd.api.

See Also AREA, DBFieldInfo(), info(), orderInfo(), recInfo()

fieldName() method

Determine the name associated with a column (field) number

Prototype

```
ERRCODE fieldName(
                  AREAP wa,
                  USHORT fieldNum,
                  BYTEP name
                  )
```

Arguments

wa is a pointer to self.

fieldNum specifies the ordinal position of the column whose name is to be obtained.

name is a pointer to a buffer used to store the field name.

Description

fieldName() obtains the symbolic name associated with column *fieldNum*.

Default Behavior

fieldName() copies the *name* element of the symbol from the *wa->lpFields* structure at *fieldNum* into *name*. Since *fieldNum* is the one-based CA-Clipper *fieldNum* column (field) number, we must subtract one to correctly reference the C (zero-based) column array.

Warning! *The character buffer referenced by* ***name*** *must be allocated to at least 11 bytes prior to calling this method.*

Implementation Notes

- If you choose to reimplement this method in your subclass, you must adhere to the CA-Clipper-based symbol type. That is, symbols that represent fields in work areas must be 10 or fewer characters in length.

Files Header file is Rdd.api.

See Also AREA, fieldCount()

flush() method

Write data buffer to the data store

Prototype

```
ERRCODE flush(
                    AREAP wa
                )
```

Arguments

wa is a pointer to self.

Description

flush() writes active data and buffers to the work area's data store (usually a disk).

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- All implementations of flush() should perform a call to goCold() before physically updating the work area.
- Use writeDBHeader() to write the contents of the header record to disk.

Files Header file is Rdd.api.

See Also AREA, goCold(), writeDBHeader()

getRec() method

Retrieve the pointer to the RDD's record data buffer

Prototype

```
ERRCODE getRec(
                    AREAP wa,
                    BYTEPP lplpBuffer
                )
```

Arguments

wa is a pointer to self.

lplpBuffer is a pointer to a pointer to the current record data buffer.

Description

getRec() retrieves the pointer to the current record data buffer. This method is most notably used to transfer data between two similar work areas.

Default Behavior

You must implement the default behavior of this method through a subclass.

Files Header file is Rdd.api.

See Also AREA, putRec(), transRec()

getValue() method

Obtain the current value of a column (field)

Prototype

```
ERRCODE getValue(
                    AREAP wa,
                    USHORT fieldNum,
                    ITEMP value
                  )
```

Arguments

wa is a pointer to self.

fieldNum specifies the ordinal position of the column whose value is to be obtained.

value is a pointer to a CA-Clipper item which will contain the field's value.

Description

getValue() provides the mechanism by which the values of a row are obtained for use by the CA-Clipper runtime system.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- getValue() should check *wa->lpFields[fieldNum]* to retrieve the column information. This permits the implementation of getValue() to translate the value from the data store into an appropriate CA-Clipper type.
- getValue() is one of the few work area methods that the CA-Clipper runtime calls without benefit of a higher-level wrapper function. A simple column access generates a call to this method, as in:

```
cBobDobbs := MYDATA->BobDobbs
```

Warning! ***value*** *must point to a valid item.*

Files Header file is Rdd.api.

See Also AREA, getVarLen(), putValue()

getVarLen() method

Obtain the length of a column (field) value

Prototype

```
ERRCODE getVarLen(
                        AREAP wa,
                        USHORT fieldNum,
                        ULONGP length
                    )
```

Arguments

wa is a pointer to self.

fieldNum specifies the ordinal position of the column whose length is to be obtained.

length is a pointer to an unsigned long integer that determines the length of the field.

Description

getVarLen() retrieves the length of the specified column for use by the CA-Clipper runtime system.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- In some cases, getting the length of the column *wa->lpFields[fieldNum]* may be as simple as assigning *wa->lpFields[fieldNum].uiLen* to *length*. In other cases, where columns are of varying length, you may have to perform a physical data read to correctly determine the length of the data.

Warning! ***length*** *must point to sizeof(ULONG) bytes of allocated memory.*

Files

Header file is Rdd.api.

See Also

AREA, getValue()

goCold() method

Perform a write of work area memory to the data store

Prototype

```
ERRCODE goCold(
                  AREAP wa
               )
```

Arguments

wa is a pointer to self.

Description

goCold() forces the RDD to ensure that any data contained in memory matches the data store (usually a disk).

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- The concept of *hot* and *cold* refers to the current state of information in memory as it relates to information on disk. If the information in memory has changed since last written to disk, it is considered hot. A cold work area's memory is in sync with its disk-based counterpart.

  In any implementation of goCold(), you should determine whether the buffer holding the current cursor's data has been written to. If so, it is hot and must be turned cold by flushing or writing it to disk.

- *wa* should remain cold until it is marked hot by goHot().

Files Header file is Rdd.api.

See Also AREA, append(), flush(), goHot()

goHot() method

Mark the work area data buffer as hot

Prototype

```
ERRCODE goHot(
                    AREAP wa
                )
```

Arguments

wa is a pointer to self.

Description

goHot() informs the RDD that data in memory is not guaranteed to match data in the data store (usually a disk).

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- The concept of *hot* and *cold* refers to the current state of information in memory as it relates to information on disk. If the information in memory has changed since last written to disk, it is considered hot. A cold work area's memory is in sync with its disk-based counterpart.
- When a work area goes hot in a shared environment, you must be certain that all necessary locks are in place. Going hot indicates that a write is about to be performed on data.
- You must never allow work areas marked as *wa->fReadonly* to call goHot(). This is an ideal place to generate runtime recoverable errors indicating that locks must be present or the work area must not be in a read-only mode.
- *wa* should remain hot until it is marked cold by goCold().

Files

Header file is Rdd.api.

See Also

AREA, goCold()

putRec() method

Replace the current row (record)

Prototype

```
ERRCODE putRec(
                  AREAP wa,
                  BYTEP buffer
                )
```

Arguments

wa is a pointer to self.

buffer is the buffer at the work area cursor.

Description

putRec() replaces the current row's data buffer with the contents of *buffer*. This method is most notably used to transfer data between two similar work areas.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- It is assumed that *buffer* points to the buffer at the work area cursor. putRec() should call goHot() prior to flushing the buffer contents to disk.

Files Header file is Rdd.api.

See Also AREA, flush(), goHot(), transRec()

putValue() method

Assign a value to the current column (field)

Prototype

```
ERRCODE putValue(
                    AREAP wa,
                    USHORT fieldNum,
                    ITEM value
                  )
```

Arguments

wa is a pointer to self.

fieldNum specifies the ordinal position of the column whose value is to be assigned.

value is a CA-Clipper item which contains the value to place at the current cursor position.

Description

putValue() places a value in the column at the current work area cursor position.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- putValue() should check *wa->lpFields[fieldNum]* to retrieve the column information. This allows your implementation of putValue() to translate the CA-Clipper value to the RDD's native data store format.
- putValue() must address the shared and read-only status of the work area. Calling putValue() must trigger a call to goHot().
- putValue() is one of the few work area methods that the CA-Clipper runtime system calls without benefit of a higher-level wrapper function. The following code generates a call to this method:

```
MYDATA->BobDobbs := cBobDobbs
```

- If your RDD supports relations, update them with a forceRel() prior to placing a value in the buffer.

Files Header file is Rdd.api.

See Also AREA, forceRel(), getValue()

recall() method

Undelete the current row (record)

Prototype

```
ERRCODE recall(
                  AREAP wa
               )
```

Arguments

wa is a pointer to self.

Description

recall() recalls the row marked for deletion at the work area cursor position.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- Assuming that your RDD supports *virtual* deletes, recall() should bring the deleted record at the current work area cursor position back from its deleted status (i.e. remove the deleted flag). If your RDD does not support *virtual* deletes, you should generate a "not supported" error.
- A recall() usually involves a write to the data store. This means that shared and read-only access must be considered, and a call to goHot() is in order before performing the recall.
- If your RDD supports relations, update them with a forceRel() prior to attempting relative movement.

Files Header file is Rdd.api.

See Also AREA, delete(), deleted(), pack()

reccount() method

Obtain the number of rows (records) in the work area's table

Prototype

```
ERRCODE reccount(
                    AREAP wa,
                    ULONGP numRecords
                  )
```

Arguments

wa is a pointer to self.

numRecords is a pointer to an unsigned long integer that determines the actual number of records in the work area.

Description

reccount() retrieves the actual number of rows in the work area.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- reccount() should always return the physical number of rows in the work area regardless of conditional or filtered settings.

Warning! numRecords *must point to sizeof(ULONG) bytes of allocated memory.*

Files Header file is Rdd.api.

See Also AREA, fieldCount(), recno()

recInfo() method

Retrieve the information about a row.

Prototype

```
ERRCODE recInfo(
                    AREAP wa,
                    ITEM itmRecID,
                    USHORT uiInfoType,
                    ITEM itmInfo
                  )
```

Arguments

wa is a pointer to self.

itmRecID is a pointer to an item that determines the current row identifier and will normally be a number indicating the row number.

uiInfoType is a value that determines the type of the information to be provided.

itmInfo is a pointer to a CA-Clipper item which will contain the required information. The data type of *itmInfo* depends on the value of *uiInfoType*.

Description

recInfo() retrieves information about the state of a record (row). The information requested is defined by the value passed in *uiInfoType*. The record information that is available is defined by the RDD. Some suggested values for *uiInfoType* (constants with "DBRI_" prefix) and their associated meanings are provided in the header file Rdd.api.

Default Behavior

recInfo() is used to implement the DBRECORDINFO() function and is called any time information about the row is needed to perform an operation. In the default implementation, there are five properties (shown in the following table) defined for each row (the constants are defined in Rdd.api):

recInfo() Information types

Constant	Meaning
DBRI_DELETED	Check the 'deleted' status of the row
DBRI_LOCKED	Check if the row is locked
DBRI_RECNO	Get the row position number
DBRI_RECSIZE	Get the record size
DBRI_UPDATED	Check if the row was updated

Warning! ***itmInfo*** *must be a valid item.*

Implementation Notes

- You must implement new behavior for this method only if your driver requires properties in addition to those listed in the previous table.
- If your implementation of recInfo() cannot determine the return value based on the value of *uiInfoType*, you should allow the work area default implementation to attempt it by calling SUPER_RECINFO().
- If *itmInfo* contains a value other than NIL, it is the new value for the property. If your implementation warrants, you can change *itmInfo* by assigning the new value to it.
- The first 1000 possible values for *uiInfoType* are reserved by CA-Clipper.

Files Header file is Rdd.api.

See Also AREA, DBRecordInfo(), fieldInfo(), info(), orderInfo()

recno() method

Obtain the physical row (record) number at the current work area cursor position

Prototype

```
ERRCODE recno(
                  AREAP wa,
                  ITEM record
              )
```

Arguments

wa is a pointer to self.

record is a pointer to an item that determines the current row identifier, and will normally be a number indicating the row number.

Description

recno() retrieves the physical row identifier at the current work area cursor position for use by the CA-Clipper runtime system. If your driver operates on a database that does not contain record numbers, you must provide a unique value to be used to identify each row.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- If your RDD supports relations, update them with a forceRel() prior to attempting relative movement.

Files Header file is Rdd.api.

See Also AREA, forceRel(), reccount()

setFieldExtent() method

Establish the extent of the array of fields for a work area

Prototype

```
ERRCODE setFieldExtent(
                        AREAP wa,
                        LONG extent
                       )
```

Arguments

wa is a pointer to self.

extent is the maximum number of fields that will be used by the work area.

Description

setFieldExtent() establishes the maximum number of fields that the work area will contain.

Default Behavior

Assigns *extent* to *wa->uiFieldExtent* and allocates memory for *extent* number of fields in *wa->lpFields.*

Implementation Notes

- Most implementations should use the default behavior of setFieldExtent(). However, if your RDD allows dynamic addition of fields in the table, this method should set *wa->uiFieldExtent* to the maximum number of fields allowed for the work area (or provide for resizing of *wa->lpFields*) and allocate enough memory (in *wa->lpFields*) to store *wa->uiFieldExtent* number of fields.

Files Header file is Rdd.api.

See Also AREA, addField()

Work Area/Database Management Methods

alias( ) method
close( ) method
create( ) method
dbEval( ) method
info( ) method
new( ) method
open( ) method
pack( ) method
packRec( ) method
readDBHeader( ) method
release( ) method
sort( ) method
structSize( ) method
sysName( ) method
trans( ) method
transRec( ) method
writeDBHeader( ) method
zap( ) method

alias() method

Obtain the alias of the work area

Prototype

```
ERRCODE alias(
                  AREAP wa,
                  BYTEP alias
                )
```

Arguments

wa is a pointer to self.

alias is a pointer to a buffer that is assigned the alias name.

Description

alias() provides the alias name of the work area.

Default Behavior

Places the name of the work area from *wa->atomAlias* into *alias* as a null-terminated string.

Implementation Notes

Warning! *You must allocate at least 11 bytes to the character buffer referenced by* ***alias*** *prior to calling this function.*

Files Header file is Rdd.api.

See Also AREA, open()

close() method

Close the table in the work area

Prototype

```
ERRCODE close(
                    AREAP wa
                )
```

Arguments

wa is a pointer to self.

Description

The close() method closes the data table in the work area defined by *wa*.

Default Behavior

close() clears the work area referenced by *wa*, and performs the following additional functions:

1. Calls clearFilter() and clearLocate() passing *wa* to each.
2. If *wa* has any parent work areas, any relations pointing to the work area are killed.
3. Fields are unlinked from the symbol table, and the *wa->atomAlias* symbol is reset.

Implementation Notes

- The default implementation of close() should deallocate system information hooked into the AREA structure. Your implementation of close() should clear relations, if necessary (with clearRel()), flush buffers, close physical files, call writeDBHeader(), and then call SUPER_CLOSE().
- If a memo file is involved, close() should call closeMemFile().

Files

Header file is Rdd.api.

See Also

AREA, clearRel(), closeMemFile(), open(), release(), writeDBHeader()

create() method

Create a data store (table) in the specified work area

Prototype

```
ERRCODE create(
                  AREAP wa,
                  LPDBOPENINFO lpdbOpenInfo
                )
```

Arguments

wa is a pointer to self.

lpdbOpenInfo is a pointer to a structure containing information about the work area and the table to be created.

Description

The create() method creates an empty data store in the work area referenced by *wa*.

Default Behavior

At the work area level, create() maps directly to open().

Implementation Notes

- If creation of the new data store fails, create() should ensure that the work area is in a usable state by calling SELF_CLOSE() before you generate a recoverable error.
- If a memo file is involved, create() should call createMemFile().
- Use writeDBHeader() to write the contents of the header record to disk.

Files Header file is Rdd.api.

See Also AREA, DBOPENINFO, createMemFile(), open(), writeDBHeader()

dbEval() method

Evaluate code block for each row (record) in work area

Prototype

```
HIDE ERRCODE dbEval(
                      AREAP wa,
                      LPDBEVALINFO lpdbEvalInfo
                    )
```

Arguments

wa is a pointer to self.

lpdbEvalInfo is a pointer to a structure containing information necessary for a code block evaluation.

Description

dbEval() evaluates a code block for each row in the scope in the work area referenced by *wa*.

Default Behavior

dbEval() emulates the CA-Clipper DBEVAL() function, obeying the "normal" Xbase scoping conditions. dbEval() calls evalBlock() for each row in the database, using the *wa*'s skip() method to traverse the data. Both *wa* and *lpdbEvalInfo->itmBlock* are passed to each call of evalBlock() while the following conditions are met:

- If *lpdbEvalInfo->dbsci.itmRecID* is set to a number, evalBlock() is called once with that physical record number.
- If *lpdbEvalInfo->dbsci.lNext* is set to a number *n*, then *n* iterations occur.
- Otherwise, evalBlock() is called for each record while *wa->fEof* is not true (.T.)and the *lpdbEvalInfo->dbsci.itmCobWhile* block evaluates to true (.T.).

Implementation Notes

- If this method is reimplemented, care should be taken to completely emulate the behavior of Xbase scoping. This will offer the CA-Clipper developer maximum compatibility with the Xbase DML. Note especially that several of the scoping conditions exclude each other, and many opportunities exist for optimization. The default behavior of this method provides a very high level of optimization with the default Xbase scoping.

Files Header file is Rdd.api.

See Also AREA, DBEVALINFO, skip()

info() method

Retrieve information about the current driver

Prototype

```
ERRCODE info(
             LPAREA wa,
             USHORT uiInfoType,
             ITEM itmInfo
            )
```

Arguments

wa is a pointer to self.

uiInfoType is a value that determines the type of information to be provided.

itmInfo is a CA-Clipper item which contains the information that corresponds to *uiInfoType*.

Description

info() obtains the information about the current work area.

Default Behavior

The info() method is used to implement the CA-Clipper DBINFO() function and is called any time information about the work area is needed to perform an operation. It returns items of data in response to some basic questions concerning the driver's functionality. Those questions are defined by the value passed in *uiInfoType*. Some suggested values for *uiInfoType* (constants with "DBI_" prefix) and their associated meanings are provided in the header file Rdd.api.

Warning! itmInfo *must be a valid item.*

Implementation Notes

- If your implementation of info() cannot determine the return value based on the value of *uiInfoType*, you should allow the work area default implementation to attempt it by calling SUPER_INFO().
- DBI_USER is a constant (defined in Rdd.api) that returns the minimum value that third-party developers can use for defining new *uiInfoType* parameters. Values less than DBI_USER are reserved by CA-Clipper.

Files Header file is Rdd.api.

See Also AREA, sysName()

new() method

Clear the work area for use

Prototype

```
ERRCODE new(
                AREAP wa
            )
```

Arguments

wa is a pointer to self.

Description

Clears a work area for use.

Default Behavior

new() assures that *wa* is ready to be put into use.

Implementation Notes

- Most implementations should use the default implementation of new(). Any behavior added to a subclassed version should provide any initialization of the work area needed for the RDD.

Warning! *You should never break the inheritance chain with this method. In other words, always call SUPER_NEW(wa) from any RDD you design.*

Files Header file is Rdd.api.

See Also AREA, open(), release()

open() method

Open a data store (table) in the work area

Prototype

```
ERRCODE open(
             AREAP wa,
             LPDBOPENINFO lpdbOpenInfo
           )
```

Arguments

wa is a pointer to self.

lpdbOpenInfo is a pointer to a structure containing information about the work area and the data store to be opened.

Description

The open() method opens the data store referenced by *lpdbOpenInfo*.

Default Behavior

The work area open() method concludes the process of opening a database file by performing the following tasks:

- Sets up the *wa->atomAlias* (checking for duplicate aliases or bad alias names)
- Links the fields to the symbol table
- Adds the work area to the list of work areas that are in use

Note: The work area must not already be in use.

Implementation Notes

- open() assumes that the RDD has already opened the data store and set up the *wa->lpFields* array prior to calling SUPER_OPEN().
- If a memo file involved, open() should call openMemFile().

Files Header file is Rdd.api.

See Also AREA, DBOPENINFO, close(), create(), new(), openMemFile()

pack() method

Remove rows (records) marked for deletion from a database

Prototype

```
ERRCODE pack(
                    AREAP wa
                )
```

Arguments

wa is a pointer to self.

Description

pack() physically removes rows marked for deletion from the database.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- Since a pack operation must traverse the entire data store, the buffer for the current work area must call goCold() before any pack() operation can take place. Further, in a shared environment, you can only implement this method after the proper locks are established.
- If your RDD supports relations, resolve any pending relational moves (perhaps by simply removing them) before attempting the pack() operation.

Files Header file is Rdd.api.

See Also AREA, delete(), deleted(), goCold(), packRec(), recall(), zap()

packRec() method

Copy a single row back to the current work area

Prototype

```
ERRCODE packRec(
                    AREAP wa,
                    LONG lRecno,
                    USHORTP lpfWritten
                )
```

Arguments

wa is a pointer to self.

lRecno is the number of a record which has to remain within the file.

lpfWritten is a pointer to a boolean flag indicating whether the record will remain within the file.

Description

packRec() copies a single row back to the work area and is called from within the pack() method.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- Within the pack() method, call packRec() for each single row that you want to copy back to the original database file.

Files Header file is Rdd.api.

See Also AREA, pack()

readDBHeader() method

Read the database file header record in the work area

Prototype

```
ERRCODE readDBHeader(
                        AREAP wa
                    )
```

Arguments

wa is a pointer to self.

Description

The readDBHeader() method reads the database file header in the work area referenced by *wa*.

Files

Header file is Rdd.api.

See Also

AREA, headerLock(), writeDBHeader()

release() method

Release all references to a work area

Prototype

```
ERRCODE release(
                    AREAP wa
                )
```

Arguments

wa is a pointer to self.

Description

The release() method releases the work area referenced by *wa*.

Default Behavior

release() releases the work area referenced by *wa*, and performs the following tasks:

- Releases memory used by *wa->lpFields*
- Removes the referenced work area from the list of open work areas
- If the referenced work area is the current work area, the current work area symbol is set to NULL
- Releases the memory held by the current AREA structure

Implementation Notes

- It is highly recommended that you do not break the inheritance chain of release() (i.e., any implementation of release() in your RDD should call SUPER_RELEASE()).

Files Header file is Rdd.api.

See Also AREA, close(), new()

sort() method

Physically reorder a database

Prototype

```
ERRCODE sort(
              AREAP wa,
              LPDBSORTINFO lpdbSortInfo
            )
```

Arguments

wa is a pointer to self.

lpdbSortInfo is a pointer to a structure containing information on how to sort the work area table.

Description

sort() performs a physical reordering of the database by copying records from the current work area to another database file in sorted order, as specified in *lpdbSortInfo*. sort() performs as much of its operation as possible in memory, then it spools to a uniquely named temporary disk file. This temporary file can be as large as the size of the source database file. It then overwrites the original database with the newly sorted data.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- Since a sort operation must traverse the entire data store, the buffer for the current work area must call goCold() before any sort() operation can take place. Further, in a shared environment, you can only implement this method after the proper locks are established.

Files Header file is Rdd.api.

See Also AREA, DBSORTINFO, goCold()

structSize() method

Retrieve the size of the work area structure

Prototype

```
ERRCODE structSize(
                        AREAP wa,
                        USHORTP size
                    )
```

Arguments

wa is a pointer to self.

size is a pointer to a value which is assigned the size of the work area structure.

Description

Obtains the size of the work area structure.

Default Behavior

The size of AREA is copied to *size*.

Implementation Notes

- If your RDD changes the size of the AREA structure, you *must* reimplement this method to return the true size of the new structure.

Warning! ***size*** *must point to a valid USHORT.*

Files

Header file is Rdd.api.

See Also

AREA, open()

sysName() method

Obtain the name of the replaceable database driver (RDD) subsystem

Prototype

```
ERRCODE sysName(
                    AREAP wa,
                    BYTEP name
                  )
```

Arguments

wa is a pointer to self.

name is a pointer to a buffer that is assigned the name of the RDD.

Description

sysName() obtains the name of the driver subsystem.

Default Behavior

You must implement the default behavior of this method through a subclass.

Warning! ***name*** *must point to a preallocated buffer of at least 16 bytes.*

Implementation Notes

- Your RDD should implement sysName() by simply returning the name of your driver. For example, the DBFNTX driver copies "DBFNTX" into *name* and returns. There is no need to maintain the inheritance chain for sysName().

Files Header file is Rdd.api.

See Also AREA, info()

trans() method

Copy one or more rows (records) from one work area to another

Prototype

```
ERRCODE trans(
                    AREAP wa,
                    LPDBTRANSINFO lpdbTransInfo
                  )
```

Arguments

wa is a pointer to self.

lpdbTransInfo is a pointer to a structure containing information about the transfer of data.

Description

trans() copies one or more rows between two tables in two different work areas.

Default Behavior

Copies multiple rows by passing each *wa* record to transRec() while the following conditions are met (these are the same conditions as dbEval()):

- If *lpdbTransInfo->dbsci.itmRecID* is set to a number, evalBlock() is called once with that physical row (record) number
- If *lpdbTransInfo->dbsci.lNext* is set to a number, *n*: *n* iterations occur.

  Otherwise, evalBlock() is called for each record while *wa->fEof* is not true (.T.) and the *lpdbTransInfo->dbsci.itmCobWhile* block evaluates to true (.T.).

Implementation Notes

- If you perform multiple iterations of this method, be careful to completely emulate the behavior of Xbase scoping. This will offer the CA-Clipper developer maximum compatibility with the Xbase DML. Note especially that several of the scoping conditions exclude each other, so there are many opportunities for optimization. The default behavior of this method provides a very high level of optimization with the default Xbase scoping.

Files Header file is Rdd.api.

See Also AREA, DBTRANSINFO, transRec()

transRec() method

Copy a single row (record) to another work area

Prototype

```
ERRCODE transRec(
                    AREAP wa,
                    LPDBTRANSINFO lpdbTransInfo
                    )
```

Arguments

wa is a pointer to self.

lpdbTransInfo is a pointer to a structure containing information about the transfer of data.

Description

transRec() copies a single row to another table.

Default Behavior

Copies the current row from *wa* to the database specified by *lpdbTransInfo.*

Note: If an error occurs, the new record in *lpdbTransInfo->lpaDest* is deleted by a call to SELF_DELETE().

Implementation Notes

- If your RDD supports relations, update them in the source work area with a forceRel() prior to attempting to perform the copy.
- You should append the new copy to the target work area.
- A reimplementation of transRec() should check locks and read-only status in the target database.

Files Header file is Rdd.api.

See Also AREA, DBTRANSINFO, forceRel(), trans()

writeDBHeader() method

Write the database file header record in the work area back to disk

Prototype

```
ERRCODE writeDBHeader(
                          AREAP wa
                      )
```

Arguments

wa is a pointer to self.

Description

writeDBHeader() writes the contents of the header record to the work area referenced by *wa*.

Implementation Notes

- Normally, before writing the contents of the header record using writeDBHeader(), call headerLock() to obtain a lock on the header record.

Files Header file is Rdd.api.

See Also AREA, headerLock(), readDBHeader()

zap() method

Physically remove all rows (records) from the data store (table)

Prototype

```
ERRCODE zap(
                AREAP wa
            )
```

Arguments

wa is a pointer to self.

Description

zap() physically removes all rows from the data store referenced by *wa*.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- You may only implement zap() with the appropriate network locks in place while the work area is not in read-only mode.

Files Header file is Rdd.api.

See Also AREA, pack()

Relational Operation Methods

childEnd( ) method
childStart( ) method
childSync( ) method
clearRel( ) method
forceRel( ) method
relArea( ) method
relEval( ) method
relText( ) method
setRel( ) method
syncChildren( ) method

childEnd() method

Report end of relation

Prototype

```
ERRCODE childEnd(
                    AREAP wa,
                    LPDBRELINFO lpdbRelInfo
                  )
```

Arguments

wa is a pointer to self.

lpdbRelInfo is a pointer to a structure containing information on a relation.

Description

childEnd() indicates that one of *wa*'s parent work areas is no longer related.

Default Behavior

childEnd() indicates that the referenced work area is related to one less parent by decrementing *wa->uiParents*. This method does not physically unhook a child from a parent.

Warning! *There is no check to guarantee that* ***wa->uiParents*** *will wrap from 0 to 255 after the decrement.*

Implementation Notes

- If your RDD supports relations, ensure that the child work area (denoted by *wa*) does not have a pending relational move that relies upon the existence of the parent referenced by *lpdbRelInfo*. If such a movement exists, it must be resolved by a call to SELF_FORCEREL() prior to removal.
- In most driver implementations, parent-child relations will be stored exclusively in the parent work area. clearRel() removes all child relations from a parent; there is no way to remove a single relation from a parent's relation list using the CA-Clipper DML.

Files Header file is Rdd.api.

See Also AREA, DBRELINFO, childStart(), clearRel()

childStart() method

Report initialization of a relation

Prototype

```
ERRCODE childStart(
                        AREAP wa,
                        LPDBRELINFO lpdbRelInfo
                    )
```

Arguments

wa is a pointer to self.

lpdbRelInfo is a pointer to a structure containing information on a relation.

Description

childStart() informs the child work area (referenced by *wa*) that it will be attached to the parent specified in *lpdbRelInfo*.

Default Behavior

Increments the *wa->uiParents* field. Parents can be sent relation information for initialization with the setRel() method.

Implementation Notes

- A call to SELF_CHILDSYNC() should be made within any implementation of this method to force the child work area to begin life in sync with its parent.

Files

Header file is Rdd.api.

See Also

AREA, DBRELINFO, childEnd(), setRel()

childSync() method

Post a pending relational movement

Prototype

```
ERRCODE childSync(
                    AREAP wa,
                    LPDBRELINFO lpdbRelInfo
                   )
```

Arguments

wa is a pointer to self.

lpdbRelInfo is a pointer to a structure containing information on a relation.

Description

childSync() posts *lpdbRelInfo* as a pending relational movement indicating that the child work area referenced by *wa* has been affected by a parental movement.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- Note that grandchildren must have relative movements posted in them after the child moves. This means that any implementation of this method should make a call to syncChildren() after doing its business.
- childSync() does not physically move the child's cursor. childSync() should simply post notification that a relational movement is pending. It can later be resolved (if necessary) by forceRel().

Files Header file is Rdd.api.

See Also AREA, DBRELINFO, forceRel(), syncChildren()

clearRel() method

Clear all relations in the specified work area

Prototype

```
ERRCODE clearRel(
                    AREAP wa
                  )
```

Arguments

wa is a pointer to self.

Description

clearRel() clears all relations currently in use by a specified work area.

Default Behavior

clearRel() removes all relations from *wa->lpdbRelations*. clearRel() traverses the relation list and releases the memory used by each relation structure (DBRELINFO) in *wa->lpdbRelations*. For each relation killed, childEnd() is called to notify the child of its parent's abandonment.

Implementation Notes

- If this method is replaced, be certain to properly deallocate items and memory used by DBRELINFO.

Files Header file is Rdd.api.

See Also AREA, DBRELINFO, childEnd(), setRel()

forceRel() method

Force relational seeks in the specified work area

Prototype

```
ERRCODE forceRel(
                    AREAP wa
                  )
```

Arguments

wa is a pointer to self.

Description

forceRel() causes any pending relational seeks in the work area referenced by *wa* to be performed immediately.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- The ideal implementation for forceRel() would have a pending relation structure (DBRELINFO) defined as a part of the subclassed AREA structure. Using this information, forceRel() should ensure that the work area buffer is not hot and call SELF_RELEVAL() to position the work area.
- This is one of the rare instances when *wa->fFound* should be set if the relational move is successful. While this is not a seek(), part of the specification of the CA-Clipper implementation of FOUND() is to return true (.T.) if a relational seek is successful.

Files

Header file is Rdd.api.

See Also

AREA, DBRELINFO, relEval()

relArea() method

Obtain the logical number of the specified work area

Prototype

```
ERRCODE relArea(
                    AREAP wa,
                    USHORT relNum,
                    USHORTP nArea
                )
```

Arguments

wa is a pointer to self.

relNum is an unsigned short integer that specifies the relation number for which to obtain the work area number.

nArea is a pointer to an unsigned short value that determines the number of the work area.

Description

relArea() obtains the CA-Clipper SELECT() area of the child relation referenced by *relNum*. *relNum* is a one-based number.

Default Behavior

relArea() implements the behavior of the CA-Clipper function DBRSELECT(). It scans relations for the work area at *wa* and returns the area number of the child work area.

Implementation Notes

- If this method is reimplemented, be careful to return the area number for the relation at *relNum*-1 since *relNum* is a one-based number.

Files Header file is Rdd.api.

See Also AREA, relText()

relEval() method

Evaluate a block against the relation in the specified work area

Prototype

```
ERRCODE relEval(
                    AREAP wa,
                    LPDBRELINFO lpdbRelInfo
                )
```

Arguments

wa is a pointer to self.

lpdbRelInfo is a pointer to a structure containing information on a relation.

Description

relEval() performs a relational seek from the parent work area referenced by *wa* to the child work area specified by *lpdbRelInfo.*

Default Behavior

Evaluates the block held in the structure pointed to by *lpdbRelInfo->itmCobExpr*. The expression resulting from the evaluation is placed in *wa->valResult* and the return code indicates success or failure.

Note: The selected work area is saved and restored.

Warning! *If the item held in* ***lpdbRelInfo->itmCobExpr*** *is not a code block, a BASE/1004 "No exported method" error is generated.*

Implementation Notes

- If any global state necessary for the operation of your driver could be modified by the code block in *lpdbRelInfo->itmCobExpr* (i.e., any CA-Clipper code), you should save that state, call SUPER_RELEVAL(), and then restore the state.

Files Header file is Rdd.api.

See Also AREA, DBRELINFO, forceRel()

relText() method

Obtain the character expression of the specified relation

Prototype

```
ERRCODE relText(
                    AREAP wa,
                    USHORT relNum,
                    BYTEP cExpr
                )
```

Arguments

wa is a pointer to self.

relNum is a numeric value that specifies for which relation the character expression is to be obtained.

cExpr is a pointer to a buffer that contains the character expression of relation *relNum*.

Description

relText() obtains a string that describes the relationship of the *relNum* relation to the work area referenced by *wa*. *relNum* is a one-based number.

Default Behavior

relText() implements the behavior of the CA-Clipper function DBRELATION(). It scans relations for the work area at referenced by *wa* and returns the key text for the child work area.

Implementation Notes

- If this method is reimplemented, be careful to return the key text for the relation at *relNum*-1 since *relNum* is a one-based number.

Files Header file is Rdd.api.

See Also AREA, relArea()

setRel() method

Set a relation in the parent file

Prototype

```
ERRCODE setRel(
                  AREAP wa,
                  LPDBRELINFO lpdbRelInfo
                )
```

Arguments

wa is a pointer to self.

lpdbRelInfo is a pointer to a structure containing information on a relation.

Description

setRel() adds *lpdbRelInfo* to the *wa->lpdbRelations* list.

Default Behavior

Sets a relation—as defined by *lpdbRelInfo*—in the parent file *wa*. A copy of the *lpdbRelInfo* structure is placed in the *wa->lpdbRelations* list.

setRel() notifies the child of its additional parent by calling childStart().

Warning! *The* ***lpdbRelInfo->lpaParent*** *pointer must be the same as the work area pointer that is passed as the first argument.*

Implementation Notes

- If you replace this method in your RDD, locate the end of the *wa->lpdbRelations* chain and mark the *lpdbriNext* element to point to your new relation.

Files

Header file is Rdd.api.

AREA, DBRELINFO, childStart(), clearRel()

syncChildren() method

Force relational movement in child work areas

Prototype

```
ERRCODE syncChildren(
                          AREAP wa
                     )
```

Arguments

wa is a pointer to self.

Description

syncChildren() synchronizes all child work areas in *wa->lpdbRelations*.

Default Behavior

syncChildren() traverses the relation list *wa->lpdbRelations* and calls childSync(), passing the child's work area structure and the current DBRELINFO structure from the parent.

Note: childSync() must be implemented by a subclass.

Implementation Notes

- You should only replace this method with extreme care because recursive use of childSync() may call the code in syncChildren(). Also, error codes (return value from childSync()) may cause the premature end of processing in this method.

Files Header file is Rdd.api.

See Also AREA, childSync()

Order Management Methods

orderCondition() method

Set or delete a condition for subsequent order creation in the specified work area

Prototype

```
ERRCODE orderCondition(
                        AREAP wa,
                        LPDBORDERCONDINFO
                           lpdbOrdCondInfo
                        )
```

Arguments

wa is a pointer to self.

lpdbOrdCondInfo is a pointer to a structure containing information for the order condition.

Description

orderCondition() sets or deletes an order condition *lpdbOrdCondInfo* in *wa*.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- For DBF-like RDDs, setting and resetting of the order condition is managed by the super driver, DBF.RDD. You can use this method by calling SUPER_ORDSETCOND().

Files Header file is Rdd.api.

See Also AREA, DBORDERCONDINFO, orderCreate()

orderCreate() method

Create new order

Prototype

```
ERRCODE orderCreate(
                        AREAP wa,
                        LPDBORDERCREATEINFO
                           lpdbCreateOrderInfo
                        )
```

Arguments

wa is a pointer to self.

lpdbCreateOrderInfo is a pointer to a structure containing information for order creation.

Description

orderCreate() creates a new order in *wa.*

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- Before you create and attach a new order to the work area, call goCold().
- orderCreate() provides the functionality of the CA-Clipper INDEX ON command. In most implementations, orderCreate() should close any currently open orders before creating the new order. After order creation, the closed orders may then be reopened or left closed, as dictated by the RDD implementation.
- The base ordering procedure requires key evaluation on an empty *phantom* row. To reposition to a phantom row, perform a SELF_GOTO(). See go() for more information.

Warning! *If you choose to allow orders to remain open during order creation, and your implementation provides for scoped or conditional orders, keep in mind that future order creations may be affected by such orders.*

Files Header file is Rdd.api.

See Also AREA, DBORDERCREATEINFO, goCold(), go(), orderListRebuild()

orderInfo() method

Provides information about order management

Prototype

```
ERRCODE orderInfo(
                    AREAP wa,
                    USHORT message,
                    FARP value
                  )
```

Arguments

wa is a pointer to self.

message is a value that determines the type of information to be provided.

value is a pointer to some data type structure indicating the status of *message*. The data type pointed to by *value* depends on the value of *message*.

Description

orderInfo() returns information about the current order. The information requested is defined by the value passed in *message*. Some suggested values for *message* and their associated meanings are provided in the header file Rdd.api.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- If your implementation of orderInfo() cannot determine the return value based on the value of *message*, you should allow the work area default implementation to attempt it by calling SUPER_ORDINFO().
- The first 1000 possible values for *message* are reserved by CA-Clipper.

Files Header file is Rdd.api.

See Also AREA, DBORDERINFO

orderListAdd() method

Opens an order bag in the indicated work area

Prototype

```
ERRCODE orderListAdd(
                        AREAP wa,
                        LPDBORDERINFO lpdbOrderInfo
                    )
```

Arguments

wa is a pointer to self.

lpdbOrderInfo is a pointer to a structure containing information about the Order Bag to be opened.

Description

orderListAdd() opens an order bag with all its associated orders in the work area. This functionality is analogous to the CA-Clipper SET INDEX TO...ADDITIVE command.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- Before an order can be attached to the work area, the work area's buffer should be written to disk in a SELF_GOCOLD() call.
- It is recommended that you place a limit on the number of open orders per work area since memory is a finite resource, even if you tap the power of the Virtual Memory API to supply you with memory.
- By convention, once the order is set, a SELF_GOTOP() should be issued to remain consistent with the CA-Clipper DML.

Files Header file is Rdd.api.

See Also AREA, DBORDERINFO, goCold(), goTop(), orderListClear(), orderListFocus()

orderListClear() method

Clear the current order list

Prototype

```
ERRCODE orderListClear(
                              AREAP wa
                            )
```

Arguments

wa is a pointer to self.

Description

orderListClear() clears the order list currently in use by the work area referenced by *wa*.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- Before you clear orders, bring the database to a cold state with the SELF_GOCOLD() method. Also, flush to disk and deallocate any memory buffers maintained by the order lists.

Files Header file is Rdd.api.

See Also AREA, goCold(), orderListAdd()

orderListFocus() method

Select the controlling order

Prototype

```
ERRCODE orderListFocus(
                        AREAP wa,
                        USHORT order
                      )
```

Arguments

wa is a pointer to self.

order is a numeric value that indicates the order number to select.

Description

orderListFocus() selects the logical order of the table based on *order*.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- Before you can attach an order to the work area, write the work area's buffer to disk through a SELF_GOCOLD() call.
- The number in *order* is the position of the order in the controlling key list.

Files Header file is Rdd.api.

See Also AREA, goCold(), orderListAdd()

orderListRebuild() method

Rebuild all orders in the specified work area

Prototype

```
ERRCODE orderListRebuild(
                                AREAP wa
                              )
```

Arguments

wa is a pointer to self.

Description

orderListRebuild() causes all Orders in use by *wa* to be rebuilt.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- Prior to a orderListRebuild() call, the work area buffer should be written to disk with a call to SELF_GOCOLD().
- In a shared or read-only work area, care should be taken to observe network etiquette.
- orderListRebuild() should build the orders from scratch without depending on any calculated information already available in order to ensure maximum integrity of data.
- Once orderListRebuild() is complete, the rebuilt order should be in control and a SELF_GOTOP() should be issued to remain consistent with the CA-Clipper DML.

Files Header file is Rdd.api.

See Also AREA, goCold(), goTop(), orderCreate()

Filter and Scoping Methods

clearFilter( ) method
clearLocate( ) method
clearScope( ) method
filterText( ) method
setFilter( ) method
setLocate( ) method

clearFilter() method

Clear the active filter expression

Prototype

```
ERRCODE clearFilter(
                      AREAP wa
                    )
```

Arguments

wa is a pointer to self.

Description

clearFilter() clears the filter expression currently in use for a specified work area.

Default Behavior

clearFilter() removes the filter expression in *wa->dbfi*. The items held in *wa->dbfi.itmCobExpr* and *wa->dbfi.abFilterText* are both released and set to NULL.

Implementation Notes

- Verify that all items are properly released upon any reimplementation of clearFilter().

Files Header file is Rdd.api.

See Also AREA, setFilter()

clearLocate() method

Clear the active locate expression

Prototype

```
ERRCODE clearLocate(
                        AREAP wa
                    )
```

Arguments

wa is a pointer to self.

Description

clearLocate() clears the locate expression currently in use for a specified work area.

Default Behavior

clearLocate() removes the locate expressions in *wa->dbsi*. The items held by *wa->dbsi.itmCobFor*, *wa->dbsi.lpstrFor*, *wa->dbsi.itmCobWhile*, *wa->dbsi.lpstrWhile*, *wa->dbsi.lNext*, *wa->dbsi.itmRecID*, and *wa->dbsi.fRest* are all released and set to NULL.

Implementation Notes

- Verify that all items are properly released upon any reimplementation of clearLocate().

Files Header file is Rdd.api.

See Also AREA, setLocate()

clearScope() method

Clear the scope setting for the specified work area

Prototype

```
ERRCODE clearScope(
                        AREAP wa
                    )
```

Arguments

wa is a pointer to self.

Description

clearScope() clears the scope information currently in use for a specified work area.

Note: This method is reserved for future implementation.

Files Header file is Rdd.api.

See Also AREA, DBSCOPEINFO, setScope(), clearScope(), skipScope()

filterText() method

Return filter condition of the specified work area

Prototype

```
ERRCODE filterText(
                        AREAP wa,
                        BYTEP cExpr
                    )
```

Arguments

wa is a pointer to self.

cExpr is a pointer to a buffer containing the text expression.

Description

filterText() returns a text string indicating the current filtering condition set for the data store referenced by *wa*.

Default Behavior

The work area implementation returns the value at *wa->dbfi.abFilterText*.

Implementation Notes

- This method emulates CA-Clipper DBFILTER() function. Refer to this function for more information regarding the rules of implementation.

Files Header file is Rdd.api.

See Also AREA, setFilter()

setFilter() method

Set the filter condition for the specified work area

Prototype

```
ERRCODE setFilter(
                        AREAP wa,
                        LPDBFILTERINFO lpdbFilterInfo
                      )
```

Arguments

wa is a pointer to self.

lpdbFilterInfo is a pointer to a structure containing information about the filter condition set for the work area.

Description

setFilter() sets a table filtering condition for the work area referenced by *wa*.

Default Behavior

setFilter() sets the *wa->dbfi* to the filter expressions held in *lpdbFilterInfo.*

Implementation Notes

- Prior to setting the new filter, setFilter() must perform a clearFilter(), which is defined as releasing the items held by the *wa->dbfi* structure.

Files Header file is Rdd.api.

See Also AREA, DBFILTERINFO, clearFilter(), filterText()

setLocate() method

Set the locate scope for the specified work area

Prototype

```
ERRCODE setLocate(
                        AREAP wa,
                        LPDBSCOPEINFO lpdbScopeInfo
                    )
```

Arguments

wa is a pointer to self.

lpdbScopeInfo is a pointer to a structure containing information about the locate scope for the work area.

Description

setLocate() sets the locate scope for the work area referenced by *wa*.

Default Behavior

setLocate() sets the locate scope for *wa* from *lpdbScopeInfo*.

Implementation Notes

- Prior to setting the new locate scope, setLocate() must perform a clearLocate() which releases the items held by the *wa->dbsi* structure.

Files Header file is Rdd.api.

See Also AREA, DBSCOPEINFO, clearLocate()

Network Operation Methods

lock( ) method
rawLock( ) method
unlock( ) method

lock() method

Perform a network lock in the specified work area

Prototype

```
ERRCODE lock(
                AREAP wa,
                LPDBLOCKINFO lpLockInfo
            )
```

Arguments

wa is a pointer to self.

lpLockInfo is a pointer to a structure containing information about the lock to be obtained.

Description

lock() performs a network lock in the work area. The type of lock to be obtained is defined in *lpLockInfo*. See DBLOCKINFO for more information on the lock types available.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- Your implementation of lock() should determine the type of lock to be obtained by checking *lpLockInfo->uiMethod* and return a success code in *lpLockInfo->fResult* as TRUE if successful or FALSE if not.

Files Header file is Rdd.api.

See Also AREA, DBLOCKINFO, unlock()

rawLock() method

Perform various locks and unlocks

Prototype

```
ERRCODE rawLock(
                    AREAP wa,
                    USHORT action,
                    LONG lrecord
                )
```

Arguments

wa is a pointer to self.

action is a numeric value that represents the type of locking or unlocking to perform. (See the definitions below.)

lRecord is a numeric value that represents the record number on which to perform the lock or unlock.

Description

rawLock() performs one of the following actions: file lock, file unlock, record lock, record unlock, header lock, header unlock, append lock, or append unlock.

The lock or unlock performed is determined by the action performed. The action parameter can be one of the following values:

Action Value	Action Performed
FILE_LOCK	File lock
FILE_UNLOCK	File unlock
REC_LOCK	Record lock of record number lrecord
REC_UNLOCK	Unlock record number lrecord
HEADER_LOCK	Header lock
HEADER_UNLOCK	Header unlock
APPEND_LOCK	Locks a newly appended record
APPEND_UNLOCK	Unlocks a newly appended record

Files

Header file is Rdd.api.

See Also

AREA, DBLOCKINFO, lock(), unlock()

unlock() method

Release network locks in the specified work area

Prototype

```
ERRCODE unlock(
                  AREAP wa,
                  ULONG record
                )
```

Arguments

wa is a pointer to self.

record is a numeric value that indicates the row to unlock. *record* is only meaningful if a lock on a record exists.

Description

The unlock() method unlocks any network locks held on *wa*.

Default Behavior

You must implement the default behavior of this method through a subclass.

Implementation Notes

- Your implementation should flush the data buffer by calling SELF_GOCOLD() prior to releasing any network locks. *record* is provided should your RDD support multiple row locks. By convention a *record* value of zero (0) indicates that all rows should be unlocked.

Files Header file is Rdd.api.

See Also AREA, DBLOCKINFO, lock()

Memo File Management Methods

closeMemFile( ) method
createMemFile( ) method
getValueFile( ) method
openMemFile( ) method
putValueFile( ) method

closeMemFile() method

Close a memo file in the specified work area

Prototype

```
ERRCODE closeMemFile(
                          AREAP wa
                        )
```

Arguments

wa is a pointer to self.

Description

The closeMemFile() method closes the memo file in the work area referenced by *wa*.

Implementation Notes

- closeMemFile() isolates the code necessary to close the memo file. It is normally called by close() when the database has an associated memo file.

Files Header file is Rdd.api.

See Also AREA, close(), createMemFile(), openMemFile()

createMemFile() method

Create a memo file in the work area

Prototype

```
ERRCODE createMemFile(
                        AREAP wa,
                        LPDBOPENINFO lpdboi
                      )
```

Arguments

wa is a pointer to self.

lpdboi is a pointer to a structure containing information about the work area and the table to be created.

Description

The createMemFile() method creates a memo file in the work area referenced by *wa*.

Implementation Notes

- createMemFile() isolates the code necessary to create the memo file. It is normally called by create() when the database has an associated memo file.

Files Header file is Rdd.api.

See Also AREA, DBOPENINFO, create(), closeMemFile(), openMemFile()

getValueFile() method

Retrieve the current value of a column and put it into a file

Prototype

```
ERRCODE getValueFile(
                          AREAP wa,
                          USHORT uiIndex,
                          BYTEP lpbFile
                     )
```

Arguments

wa is a pointer to self.

uiIndex is the ordinal position of the column (memo field) whose value is to be obtained.

lpbFile is a pointer to a string representing the file name.

Description

getValueFile() retrieves the current value of the column referenced by *uiIndex* in the work area referenced by *wa*. The current value is put into a file referenced by *lpbFile*.

Files Header file is Rdd.api.

See Also AREA, putValueFile()

openMemFile() method

Open a memo file in the specified work area

Prototype

```
ERRCODE openMemFile(
                          AREAP wa,
                          LPDBOPENINFO lpdboi
                        )
```

Arguments

wa is a pointer to self.

lpdboi is a pointer to a structure containing information about the work area and the table to be opened.

Description

The openMemFile() method opens a memo file in the work area referenced by *wa*.

Implementation Notes

- openMemFile() isolates the code necessary to open the memo file. It is normally called by open() when the database has an associated memo file.

Files Header file is Rdd.api.

See Also AREA, DBOPENINFO, closeMemFile(), createMemFile(), open()

putValueFile() method

Assign a value to a specified column at the current cursor position based on the contents of a file

Prototype

```
ERRCODE putValueFile(
                        AREAP wa,
                        USHORT uiIndex,
                        BYTEP lpbFile
                    )
```

Arguments

wa is a pointer to self.

uiIndex is the ordinal position of the column (memo field) whose value is to be assigned.

lpbFile is a pointer to a string representing the file name.

Description

putValueFile() places the contents of a file referenced by *lpbFile* into column *uiIndex* in the work area referenced by *wa*.

Files Header file is Rdd.api.

See Also AREA, getValueFile()

Miscellaneous Methods

compile( ) method
error( ) method
evalBlock( ) method

compile() method

Compile a character expression

Prototype

```
ERRCODE compile(
                    AREAP wa,
                    BYTEP cExpr
                  )
```

Arguments

wa is a pointer to self.

cExpr is the character expression to be compiled.

Description

The compile() method compiles a CA-Clipper expression, passed as a character string, into a code block.

Default Behavior

The compile() macro compiles the character expression *cExpr*, a null-terminated string.

The returned ERRCODE indicates success or failure. An item to the resulting code block is placed in *wa->valResult*.

Warning! *If the expression in* ***cExpr*** *is NULL or larger than the legal limit for macro-compiled expressions (in the RDD system, this limit is 256 characters), a runtime error is generated.*

Warning! *The item in* ***wa->valResult*** *must be deallocated later.*

Implementation Notes

- One common reason to subclass the compile() method is to save the item held in *wa->valResult* in case it is needed for some other process.

Files Header file is Rdd.api.

See Also AREA

error() method

Raise a runtime error

Prototype

```
ERRCODE error(
                  AREAP wa,
                  ERRORP error
                )
```

Arguments

wa is a pointer to self.

error is a pointer to an Error object containing information about the runtime error to be generated. (See the "Error System API Reference" chapter for more information.)

Description

error() raises a CA-Clipper runtime error.

Default Behavior

You can use this method to generate a CA-Clipper runtime error. This method creates a CA-Clipper-level Error object and passes it to the currently posted ERRORBLOCK(). The structure referred to by *error* contains members that are copied into the CA-Clipper Error object.

The return value of the CA-Clipper error system is mirrored in the return value of the error() method. error() returns E_BREAK, E_RETRY, or E_DEFAULT, depending on the user's response to the available choices as defined in the Error object. Note that substitution return values are not supported in this method. (See the "Error System API Reference" chapter for more information.)

If a subsystem name is not specified (by a call to _errPutSubSystem()), a value is retrieved from sysName(). If sysName() remains unimplemented or yields an empty value, "???DRIVER" is used for the subsystem name.

Implementation Notes

- Subclassing or replacing this method is not recommended.

Files Header file is Rdd.api.

See Also AREA, sysName()

evalBlock() method

Evaluate a code block

Prototype

```
ERRCODE evalBlock(
                    AREAP wa,
                    ITEM block
                  )
```

Arguments

wa is a pointer to self.

block is the code block to be evaluated.

Description

The evalBlock() method evaluates a specified code block.

Default Behavior

Evaluates the block specified by the item *block*, placing the resultant item in *wa->valResult*. The returned error code indicates success or failure.

Warning! *If* ***block*** *is not a code block, a BASE/1004 "No exported method" error is generated.*

Warning! *The item in* ***wa->valResult*** *must be deallocated later.*

Implementation Notes

- You should save any global state necessary to your driver that might be modified by the evaluated code block (i.e., any CA-Clipper code), by calling SUPER_EVALBLOCK(). Restore the state after implementing the evalBlock() method.

Files Header file is Rdd.api.

See Also AREA

Chapter 14
Light Lib Graphics API Reference Listing

Chapter 14
Light Lib Graphics API Reference

The Light Lib Graphics API gives your Extend routines the ability to manipulate output in different video modes (both text and graphic). This chapter is an alphabetical reference to all of the functions in the Light Lib Graphics API.

The prototypes for these functions and the data structures are defined in the header file Llibg.api, located in the \CLIP53\INCLUDE directory. Data types used in the prototypes are defined in the header file Clipdefs.h or one of the .api header files, also located in \CLIP53\INCLUDE. The functions themselves are defined in LLIBG.LIB, located in the \CLIP53\LIB directory.

_mClipErase()

Erase the currently defined mouse clipping region

C Prototype

```
#include "llibg.api"
void _mClipErase(void)
```

Description

_mClipSet() and _mClipErase() control mouse pointer movements. You use _mClipSet() to restrict movement to a rectangular area of the screen, called a clipping region. When a clipping region is defined and the user tries to move the mouse pointer out of the rectangle, the mouse pointer remains stuck at the edge of the region but is still visible. Then, when the clipping region is no longer needed, you use _mClipErase() to remove it and resume using the entire screen.

Files Library is LLIBG.LIB, header file is Llibg.api.

See Also _mClipSet()

_mClipGet()

Retrieve the mouse clipping region

C Prototype

```
#include "llibg.api"
ERRCODE _mClipGet(
                        LLG_LPRECT lpClipRectGet
                    )
```

Arguments

lpClipRectGet is a pointer to an LLG_RECT structure that you must allocate prior to calling this function. _mClipGet() writes information regarding the current mouse clipping region—as set by the most recent call to _mClipSet()—to this structure.

Returns

SUCCESS if successful; otherwise, an error code.

Files

Library is LLIBG.LIB, header file is Llibg.api.

See Also

_mClipSet(), _gClipGet(), LLG_RECT

_mClipSet()

Set the mouse clipping region

C Prototype

```
#include "llibg.api"
ERRCODE _mClipSet(
                        LLG_LPRECT lpClipRectSet
                    )
```

Arguments

lpClipRectSet is a pointer to an LLG_RECT structure that you must allocate and initialize prior to calling this function. _mClipSet() uses the members of this structure to set the current mouse clipping region.

Returns

SUCCESS if successful; otherwise, an error code.

Description

_mClipSet() and _mClipErase() control mouse pointer movements. You use _mClipSet() to restrict movement to a rectangular area of the screen, called a clipping region. When a clipping region is defined and the user tries to move the mouse pointer out of the rectangle, the mouse pointer remains stuck at the edge of the region but is still visible. Then, when the clipping region is no longer needed, you use _mClipErase() to remove it and resume using the entire screen.

Note: To retrieve the current mouse clipping region without changing it, call _mClipGet().

Files Library is LLIBG.LIB, header file is Llibg.api.

See Also _mClipErase(), _mClipGet(), _gClipSet(), LLG_RECT

_mCol()

Retrieve the column position of the mouse pointer

C Prototype

```
#include "llibg.api"
int _mCol(void)
```

Returns

The mouse's column position as a text-based coordinate.

Files

Library is LLIBG.LIB, header file is Llibg.api.

See Also

_mRow(), _mPixX(), _mPixY()

_mHide()

Hide the mouse pointer

C Prototype

```
#include "llibg.api"
void _mHide(void)
```

Description

_mHide() can be used with _mShow() when updating the screen. For example, you may want to hide the mouse pointer before changing the screen display, then show it again after the change.

Files Library is LLIBG.LIB, header file is Llibg.api.

See Also _mShow()

_mPixPos()

Set the X and Y pixel-based coordinates of the mouse pointer

C Prototype

```
#include "llibg.api"
void _mPixPos(
                int iX,
                int iY
                )
```

Arguments

iX is the new pixel-based X coordinate.

iY is the new pixel-based Y coordinate.

Description

_mPixPos() moves the mouse pointer to a new screen location that you specify in pixel-based coordinates. After the mouse pointer is positioned, _mPixX(), _mPixY(), _mRow() and _mCol() are updated accordingly.

Files Library is LLIBG.LIB, header file is Llibg.api.

See Also _mPixX(), _mPixY()

_mPixX()

Retrieve the pixel-based X coordinate of the mouse pointer

C Prototype

```
#include "llibg.api"
int _mPixX(void)
```

Returns

The mouse's pixel-based X coordinate position.

Files

Library is LLIBG.LIB, header file is Llibg.api.

See Also

_mCol(), _mPixPos(), _mPixY(), _mRow()

_mPixY()

Retrieve the pixel-based Y coordinate of the mouse pointer

C Prototype

```
#include "llibg.api"
int _mPixY(void)
```

Returns

The mouse's pixel-based Y coordinate position.

Files

Library is LLIBG.LIB, header file is Llibg.api.

See Also

_mCol(), _mPixPos(), _mPixX(), _mRow()

_mRow()

Retrieve the row position of the mouse pointer

C Prototype

```
#include "llibg.api"
int _mRow(void)
```

Returns

The mouse's row position as a text-based coordinate.

Files

Library is LLIBG.LIB, header file is Llibg.api.

See Also

_mCol(), _mPixX(), _mPixY()

_mShow()

Display a new mouse pointer at a particular location

C Prototype

```
#include "llibg.api"
int _mShow(
                int iCursorShape,
                int iRow,
                int iCol
                )
```

Arguments

iCursorShape is the mouse pointer shape, which you specify using one of the following values:

Mouse Pointer Shape Constants

Constant	Mouse Pointer Shape
LLM_CURSOR_ARROW	Standard pointer
LLM_CURSOR_CROSS	Cross
LLM_CURSOR_FINGER	Hand with pointing index finger
LLM_CURSOR_HAND	Hand
LLM_CURSOR_SIZE_NE_SW	North-East South-West arrow
LLM_CURSOR_SIZE_NS	North South arrow
LLM_CURSOR_SIZE_NW_SE	North-West South-East arrow
LLM_CURSOR_SIZE_WE	West East arrow
LLM_CURSOR_WAIT	Hourglass

iRow is the text-based row at which to display the mouse pointer.

iCol is the text-based column at which to display the mouse pointer.

Returns

The cursor shape of the previous mouse pointer (see *iCursorShape* above for a table of possible values).

Description

_mShow() serves two purposes. You can use it after calling _mHide() to redisplay a mouse pointer that was previously hidden, or you can use it to change the current mouse pointer shape without first hiding it. _mCol(), _mRow(), _mPixX(), and _mPixY() are updated accordingly after calling _mShow().

Files Library is LLIBG.LIB, header file is Llibg.api.

See Also _mHide(), _mState()

_mState()

Retrieve the current mouse state

C Prototype

```
#include "llibg.api"
ERRCODE _mState(
                                BOOL bReset,
                                LLG_LPMOUSESTATE lpMState
                                )
```

Arguments

bReset is a flag that resets the number of left and right mouse clicks if you specify true (.T.). If you specify false (.F.), these values are not reset.

lpMState is a pointer to an LLG_MOUSESTATE structure that you must allocate prior to calling this function. _mState() writes information regarding the current mouse state to this structure.

Returns

SUCCESS if successful; otherwise, an error code.

Files Library is LLIBG.LIB, header file is Llibg.api.

See Also LLG_MOUSESTATE

_gBmpDisp()

Display a bitmap or icon previously loaded with _gBmpLoad()

C Prototype

```
#include "llibg.api"
void _gBmpDisp(
                ITEM itmArrayBmp,
                int iXStart,
                int iYStart,
                DWORD dwTransColor
                )
```

Arguments

itmArrayBmp is an array item returned by _gBmpLoad().

iXStart is the pixel-based X coordinate at which to begin the display.

iYStart is the pixel-based Y coordinate at which to begin the display.

dwTransColor is the outline color. The range of valid values is limited to the number of colors available in the selected video mode. For example, in 16-color modes, valid values are between zero and 15, and in 256-color modes, valid values are between zero and 255. Note that you can specify LLG_NO_COLOR for no color.

Notes

- ***Restrictions***: You cannot use this function unless you have set the screen to one of the graphic modes using _gModeSet(). This function respects the screen clipping region as set by _gClipSet().

Files Library is LLIBG.LIB, header file is Llibg.api.

See Also _gBmpLoad(), _gClipSet(), _gModeSet()

_gBmpLoad()

Load a bitmap (.BMP) or icon (.ICO) file into memory

C Prototype

```
#include "llibg.api"
ITEM _gBmpLoad(
                char far * fpFileName
                )
```

Arguments

fpFileName is a pointer to the .BMP or .ICO file name that you want to load.

Returns

An array item which is the pointer to the VMM region containing the .BMP or .ICO (not a black and white icon). The first two elements of the returned array contain the width and height of the .BMP or .ICO in pixels.

Important! *Do not modify this array before passing it to _gBmpDisp().*

Files

Library is LLIBG.LIB, header file is Llibg.api.

See Also

_gBmpDisp()

_gClipGet()

Retrieve the screen clipping region

C Prototype

```
#include "llibg.api"
ERRCODE _gClipGet(
                  LLG_LPRECT lpClipRectGet
                  )
```

Arguments

lpClipRectGet is a pointer to an LLG_RECT structure that you must allocate prior to calling this function. _gClipGet() writes information regarding the current screen clipping region—as set by the most recent call to _gClipSet()—to this structure.

Returns

SUCCESS if successful; otherwise, an error code.

Files

Library is LLIBG.LIB, header file is Llibg.api.

See Also

_mClipGet(), _gClipSet(), LLG_RECT

_gClipSet()

Set the screen clipping region

C Prototype

```
#include "llibg.api"
ERRCODE _gClipSet(
                    LLG_LPRECT lpClipRectSet
                    )
```

Arguments

lpClipRectSet is a pointer to an LLG_RECT structure that you must allocate and initialize prior to calling this function. _gClipSet() uses the members of this structure to set the current screen clipping region.

Returns

SUCCESS if successful; otherwise, an error code.

Description

_gClipSet() limits the active display for Light Lib Graphic API functions to a portion of the screen, called a clipping region.

Note: To retrieve information regarding the current clipping region without changing it, call _gClipGet().

Files Library is LLIBG.LIB, header file is Llibg.api.

See Also _mClipSet(), _gClipGet(), LLG_RECT

_gEllipse()

Draw an ellipse

C Prototype

```
#include "llibg.api"
void _gEllipse(
                int iXCenter,
                int iYCenter,
                int iRadiusX,
                int iRadiusY,
                int iStartAngle,
                int iEndAngle,
                int iMode,
                BOOL bFilled,
                DWORD dwFillColor,
                BOOL bOutlined,
                DWORD dwOutlineColor,
                int iHeight3D
                )
```

Arguments

iXCenter is the pixel-based X coordinate of the center point.

iYCenter is the pixel-based Y coordinate of the center point.

iRadiusX is the length of the radius, in pixels, along the X axis.

iRadiusY is the length of the radius, in pixels, along the Y axis.

iStartAngle is the starting angle in degrees.

iEndAngle is the final angle in degrees.

iMode is the display mode, which you specify using one of the following values:

Display Mode Constants

Constant	Display Mode
LLG_MODE_SET	Overwrite existing pixels that are beneath where this object will be displayed. This is the most common display mode.
LLG_MODE_AND	Perform a logical AND on existing pixels—and on the display color—that are beneath where this object will be displayed.
LLG_MODE_OR	Perform a logical OR on existing pixels—and on the display color—that are beneath where this object will be displayed.
LLG_MODE_XOR	Perform a logical XOR on existing pixels—and on the display color—that are beneath where this object will be displayed (see Note below).

Note: LLG_MODE_XOR allows you to move objects around on the screen without damaging the background. To retrieve the initial background, simply repeat the function call using the XOR display mode.

bFilled is a flag that you specify as true (.T.) to draw a filled ellipse.

dwFillColor is the fill color.

bOutlined is a flag that you specify as true (.T.) to draw an outline around the ellipse.

dwOutlineColor is the outline color.

The range of valid values for all color parameters is limited to the number of colors available in the selected video mode. For example, in 16-color modes, valid values are between zero and 15, and in 256-color modes, valid values are between zero and 255.

iHeight3D is a value representing the height of the 3-D effect in pixels.

Notes

- ***Restrictions***: You cannot use this function unless you have set the screen to one of the graphic modes using _gModeSet(). This function respects the screen clipping region as set by _gClipSet().
- ***Drawing a circle***: To draw a circle (which is just a special case of an ellipse), set the *iRadiusX* and *iRadiusY* parameters to the same value.
- ***Drawing arcs***: Changing the values of *iStartAngle* and *iEndAngle* allows you to draw arcs of a circle or ellipse and sections of a pie chart.

Files

Library is LLIBG.LIB, header file is Llibg.api.

See Also

_gClipSet(), _gFrame(), _gLine(), _gModeSet(), _gPolygon(), _gRect()

_gExclCountGet()

Retrieve the number of exclusion areas currently in effect

C Prototype

```
#include "llibg.api"
int _gExclCountGet(void)
```

Returns

The number of exclusion areas that have been set so far using _gExclSet(). Zero indicates no exclusion areas are set.

Files

Library is LLIBG.LIB, header file is Llibg.api.

See Also

_gExclSet()

_gExclErase()

Delete all exclusion areas previously defined with _gExclSet()

C Prototype

```
#include "llibg.api"
void _gExclErase(void)
```

Description

After calling _gExclErase(), the return value of _gExclCountGet() is reset and no portion of the screen is excluded from use.

Files

Library is LLIBG.LIB, header file is Llibg.api.

See Also

_gExclCountGet(), _gExclSet()

_gExclGet()

Retrieve an exclusion area

C Prototype

```
#include "llibg.api"
ERRCODE _gExclGet(
                    LLG_LPRECT lpExclGet,
                    int iZoneNum
                    )
```

Arguments

lpExclGet is a pointer to a LLG_RECT structure that you must allocate prior to calling this function. _gExclGet() writes information regarding the specified exclusion area to this structure.

Note: The members of *lpExclGet* are always pixel-based, regardless of the status of the *bIsGraphic* parameter when you created the exclusion area with _gExclSet().

iZoneNum is the number of the exclusion area, beginning with one (1), that you want to query. The numbers are assigned based on the order in which they are added with _gSetExcl().

Returns

SUCCESS if successful; otherwise, an error code.

Files Library is LLIBG.LIB, header file is Llibg.api.

See Also _gExclCountGet(), _gExclSet(), LLG_RECT

_gExclSet()

Set a specified screen region as an exclusion area

C Prototype

```
#include "llibg.api"
ERRCODE _gExclSet(
                  LLG_LPRECT lpExclSet,
                  BOOL bIsGraphic
                  )
```

Arguments

lpExclSet is a pointer to an LLG_RECT structure that you must allocate and initialize prior to calling this function. _gExclSet() uses the members of this structure to set an exclusion area.

bIsGraphic is a flag that you specify as true (.T.) if the coordinates specified in *lpExclSet* are pixel-based and false (.F.) if they are text-based.

Returns

SUCCESS if successful; otherwise, an error code.

Description

This function is used to prevent output from being displayed in a defined region of the screen, such as when you have multiple, cascading windows and do not want to write to an area that is currently covered by another window.

Note: To retrieve information about an existing exclusion area without changing it, call _gExclGet().

Files Library is LLIBG.LIB, header file is Llibg.api.

See Also _gExclCountGet(), _gExclErase(), _gExclGet(), LLG_RECT

_gFntClipGet()

Get the clipping region for the active font

C Prototype

```
#include "llibg.api"
ERRCODE _gFntClipGet(
                        LLG_LPFNTCLIP lpFntClipGet
                        )
```

Arguments

lpFntClipGet is a pointer to an LLG_FNTCLIP structure that you must allocate prior to calling this function. _gFntClipGet() writes information regarding the clipping region for the active font—as set by most recent calls to _gFntClipSet() and _gFntSet()—to this structure.

Returns

SUCCESS if successful; otherwise, an error code.

Files Library is LLIBG.LIB, header file is Llibg.api.

See Also _gFntClipSet(), _gFntSet(), LLG_FNTCLIP

_gFntClipSet()

Set the clipping region for the active font

C Prototype

```
#include "llibg.api"
ERRCODE _gFntClipSet(
                     LLG_LPFNTCLIP lpFntClipSet
                     )
```

Arguments

lpFntClipSet is a pointer to an LLG_FNTCLIP structure that you must allocate and initialize prior to calling this function. _gFntClipSet() uses the members of this structure to set the clipping region for the active font—as set by the most recent call to _gFntSet().

Returns

SUCCESS if successful; otherwise, an error code.

Files Library is LLIBG.LIB, header file is Llibg.api.

See Also _gFntClipGet(), _gFntSet(), LLG_FNTCLIP

_gFntErase()

Erase a font previously loaded with _gFntLoad() from memory

C Prototype

```
#include "llibg.api"
void _gFntErase(
                int iFontID
                )
```

Arguments

iFontID is the font handle as returned by _gFntLoad().

Files

Library is LLIBG.LIB, header file is Llibg.api.

See Also

_gFntLoad()

_gFntGet()

Retrieve the active font handle

C Prototype

```
#include "llibg.api"
int _gFntGet(void)
```

Returns

The active font handle as set by the most recent call to _gFntSet().

Files

Library is LLIBG.LIB, header file is Llibg.api.

See Also

_gFntSet()

_gFntLoad()

Load a font file into memory

C Prototype

```
#include "llibg.api"
int _gFntLoad(
              char far * fpFontFile
              )
```

Arguments

fpFontFile is a pointer to the .FND or .FNT file name that you want to load.

Returns

The handle for the loaded font.

Files

Library is LLIBG.LIB, header file is Llibg.api.

See Also

_gFntErase(), _gFntSet()

_gFntSet()

Set a font previously loaded with _gFntLoad() as active

C Prototype

```
#include "llibg.api"
ERRCODE _gFntSet(
                    int iFontID
                    )
```

Arguments

iFontID is the font handle as returned by _gFntLoad().

Returns

SUCCESS if successful; otherwise, an error code.

Files

Library is LLIBG.LIB, header file is Llibg.api.

See Also

_gFntClipSet(), _gFntGet(), _gFntLoad()

_gFrame()

Draw a frame with a 3-D look

C Prototype

```
#include "llibg.api"
void _gFrame(
             int iLeft,
             int iTop,
             int iRight,
             int iBottom,
             DWORD dwBackgroundColor,
             DWORD dwBrightColor,
             DWORD dwDarkColor,
             int iTopWidth,
             int iLeftWidth,
             int iBottomWidth,
             int iRightWidth,
             int iMode,
             BOOL bFilled
             )
```

Arguments

iLeft is the pixel-based X coordinate of the top left corner of the frame.

iTop is the pixel-based Y coordinate of the top left corner of the frame.

iRight is the pixel-based X coordinate of the bottom right corner of the frame.

iBottom is the pixel-based Y coordinate of the bottom right corner of the frame.

dwBackgroundColor is the frame border's background color.

dwBrightColor is the frame border's bright color.

dwDarkColor is the frame border's dark color.

The range of valid values for all color parameters is limited to the number of colors available in the selected video mode. For example, in 16-color modes, valid values are between zero and 15, and in 256-color modes, valid values are between zero and 255.

iTopWidth is the thickness of the top frame border.

iLeftWidth is the thickness of the left frame border.

iBottomWidth is the thickness of the bottom frame border.

iRightWidth is the thickness of the right frame border.

iMode is the display mode (see _gEllipse() for a table of possible values).

bFilled is a flag that you specify as true (.T.) to draw a filled frame.

Notes

- ***Restrictions:*** You cannot use this function unless you have set the screen to one of the graphic modes using _gModeSet(). This function respects the screen clipping region as set by _gClipSet().

Files

Library is LLIBG.LIB, header file is Llibg.api.

See Also

_gClipSet(), _gEllipse(), _gLine(), _gModeSet(), _gPolygon(), _gRect()

_gLine()

Draw a line

C Prototype

```
#include "llibg.api"
void _gLine(
            int iXStart,
            int iYStart,
            int iXEnd,
            int iYEnd,
            DWORD dwColor,
            int iMode
            )
```

Arguments

iXStart is the pixel-based X coordinate of the starting point.

iYStart is the pixel-based Y coordinate of the starting point.

iXEnd is the pixel-based X coordinate of the ending point.

iYEnd is the pixel-based Y coordinate of the ending point.

dwColor is the line color. The range of valid values is limited to the number of colors available in the selected video mode. For example, in 16-color modes, valid values are between zero and 15, and in 256-color modes, valid values are between zero and 255.

iMode is the display mode (see _gEllipse() for a table of possible values).

Notes

- ***Restrictions:*** You cannot use this function unless you have set the screen to one of the graphic modes using _gModeSet(). This function respects the screen clipping region as set by _gClipSet().

Files Library is LLIBG.LIB, header file is Llibg.api.

See Also _gClipSet(), _gEllipse(), _gFrame(), _gModeSet(), _gPolygon(), _gRect()

_gModeGet()

Retrieve parameters for the current video mode

C Prototype

```
#include "llibg.api"
ERRCODE _gModeGet(
                    int iMode,
                    LLG_LPVIDEOMODE lpVideoModeGet
                    )
```

Arguments

iMode is the video mode, which you specify using one of the following values:

Video Mode Constants

Constant	Video Mode
LLG_VIDEO_TXT	Text
LLG_VIDEO_VESA_80_60	VESA text 80 x 60
LLG_VIDEO_VESA_132_25	VESA text 132 x 25
LLG_VIDEO_VESA_132_43	VESA text 132 x 43
LLG_VIDEO_VESA_132_50	VESA text 132 x 50
LLG_VIDEO_VESA_132_60	VESA text 132 x 60
LLG_VIDEO_VESA_800_592_16	VESA 800 x 592 x 16 colors
LLG_VIDEO_VESA_1024_768_16	VESA 1024 x 768 x 16 colors
LLG_VIDEO_VESA_1280_1024_16	VESA 1280 x 1024 x 16 colors
LLG_VIDEO_VESA_640_480_256	VESA 640 x 480 x 256 colors
LLG_VIDEO_VESA_800_592_256	VESA 800 x 592 x 256 colors
LLG_VIDEO_VESA_1024_768_256	VESA 1024 x 768 x 256 colors
LLG_VIDEO_VESA_1280_1024_256	VESA 1280 x 1024 x 256 colors
LLG_VIDEO_VESA_640_480_32K	VESA 640 x 480 x 32 K colors
LLG_VIDEO_VESA_800_592_32K	VESA 800 x 592 x 32 K colors
LLG_VIDEO_VESA_1024_768_32K	VESA 1024 x 768 x 32 K colors
LLG_VIDEO_VESA_1280_1024_32K	VESA 1280 x 1024 x 32 K colors
LLG_VIDEO_VESA_640_480_64K	VESA 640 x 480 x 64 K colors
LLG_VIDEO_VESA_800_592_64K	VESA 800 x 592 x 64 K colors
LLG_VIDEO_VESA_1024_768_64K	VESA 1024 x 768 x 64 K colors

Video Mode Constants (cont.)

Constant	Video Mode
LLG_VIDEO_VESA_1280_1024_64K	VESA 1280 x 1024 x 64 K colors
LLG_VIDEO_VESA_640_480_16M	VESA 640 x 480 x 16 M colors
LLG_VIDEO_VESA_800_592_16M	VESA 800 x 592 x 16 M colors
LLG_VIDEO_VESA_1024_768_16M	VESA 1024 x 768 x 16 M colors
LLG_VIDEO_VESA_1280_1024_16M	VESA 1280 x 1024 x 16 M colors
LLG_VIDEO_VGA_640_480_16	VGA 640 x 480 x 16 colors

lpVideoModeGet is a pointer to an LLG_VIDEOMODE structure that you must allocate prior to calling this function. _gModeGet() writes information regarding the specified video mode to this structure.

Returns

SUCCESS if successful; otherwise, an error code.

Files Library is LLIBG.LIB, header file is Llibg.api.

See Also _gModeSet(), LLG_VIDEOMODE

_gModeSet()

Switch the video mode

C Prototype

```
#include "llibg.api"
ERRCODE _gModeSet(
                  int iMode
                  )
```

Arguments

iMode is the video mode (see _gModeGet() for a table of possible values).

Returns

SUCCESS if successful; otherwise, an error code.

Files

Library is LLIBG.LIB, header file is Llibg.api.

See Also

_gModeGet()

_gPalGet()

Retrieve the current color palette

C Prototype

```
#include "llibg.api"
ERRCODE _gPalGet(
                    LLG_LPPALETTE lpPaletteGet
                    )
```

Arguments

lpPaletteGet is a pointer to an LLG_PALETTE structure that you must allocate prior to calling this function. _gPalGet() writes information regarding the current color palette—as set by the most recent call to _gPalSet()—to this structure.

Returns

SUCCESS if successful; otherwise, an error code.

Files

Library is LLIBG.LIB, header file is Llibg.api.

See Also

_gPalSet(), LLG_PALETTE

_gPalSet()

Set the current color palette

C Prototype

```
#include "llibg.api"
ERRCODE _gPalSet(
                    LLG_LPPALETTE lpPaletteSet
                    )
```

Arguments

lpPaletteSet is a pointer to an LLG_PALETTE structure that you must allocate and initialize prior to calling this function. _gPalSet() uses the members of this structure to define the current color palette.

Returns

SUCCESS if successful; otherwise, an error code.

Description

This function is used to set an entire palette. To use it, you must first set up an array of LLG_RGB structures—one for each color in the palette. The palette can consist of 16, 256, 32 K, 64 K, or 16 M colors, depending on the video mode specified using _gModeSet().

Then, you define an LLG_PALETTE structure that points to the LLG_RGB structure array and defines a few additional palette parameters, and pass a pointer to this structure to _gPalSet().

Note: To retrieve information about an existing palette, call _gPalGet().

Files

Library is LLIBG.LIB, header file is Llibg.api.

See Also

_gPalGet(), LLG_PALETTE

_gPixelGet()

Retrieve the color of a specified pixel

C Prototype

```
#include "llibg.api"
DWORD _gPixelGet(
                    int iX,
                    int iY
                    )
```

Arguments

iX is the pixel-based X coordinate.

iY is the pixel-based Y coordinate.

Returns

The color of the specified pixel.

Files Library is LLIBG.LIB, header file is Llibg.api.

See Also _gPixelSet()

_gPixelSet()

Draw a pixel

C Prototype

```
#include "llibg.api"
void _gPixelSet(
                int iX,
                int iY,
                DWORD dwColor,
                init iMode
                )
```

Arguments

iX is the pixel-based X coordinate.

iY is the pixel-based Y coordinate.

dwColor is the pixel color. The range of valid values is limited to the number of colors available in the selected video mode. For example, in 16-color modes, valid values are between zero and 15, and in 256-color modes, valid values are between zero and 255.

iMode is the display mode (see _gEllipse() for a table of possible values).

Files Library is LLIBG.LIB, header file is Llibg.api.

See Also _gEllipse(), _gPixelGet()

_gPolygon()

Draw a polygon

C Prototype

```
#include "llibg.api"
void _gPolygon(
                int iVertCount,
                LLG_LPPOINT lpVertexArray,
                BOOL bFilled,
                DWORD dwOutlineColor,
                DWORD dwFillColor,
                int iMode
                )
```

Arguments

iVertCount is the number of vertices, beginning with one (1).

lpVertexArray is a pointer to an array of LLG_POINT structures in which you define the coordinates of each of the vertices. This array must have *iVertCount* elements defined.

bFilled is a flag that you specify as true (.T.) to draw a filled polygon.

dwOutlineColor is the outline color.

dwFillColor is the fill color.

The range of valid values for all color parameters is limited to the number of colors available in the selected video mode. For example, in 16-color modes, valid values are between zero and 15, and in 256-color modes, valid values are between zero and 255.

iMode is the display mode (see _gEllipse() for a table of possible values).

Notes

- ***Restrictions***: You cannot use this function unless you have set the screen to one of the graphic modes using _gModeSet(). This function respects the screen clipping region as set by _gClipSet().

Files

Library is LLIBG.LIB, header file is Llibg.api.

See Also

_gClipSet(), _gFrame(), _gEllipse(), _gLine(), _gModeSet(), _gRect(), LLG_POINT

_gRect()

Draw a rectangle

C Prototype

```
#include "llibg.api"
void _gRect(
             int iLeft,
             int iTop,
             int iRight,
             int iBottom,
             BOOL bFilled,
             DWORD dwColor,
             int iMode
             )
```

Arguments

iLeft is the pixel-based X coordinate of the top left corner of the rectangle.

iTop is the pixel-based Y coordinate of the top left corner of the rectangle.

iRight is the pixel-based X coordinate of the bottom right corner of the rectangle.

iBottom is the pixel-based Y coordinate of the bottom right corner of the rectangle.

bFilled is a flag that you specify as true (.T.) to draw a filled rectangle.

dwColor is the rectangle color. The range of valid values is limited to the number of colors available in the selected video mode. For example, in 16-color modes, valid values are between zero and 15, and in 256-color modes, valid values are between zero and 255.

iMode is the display mode (see _gEllipse() for a table of possible values).

Notes

- ***Restrictions***: You cannot use this function unless you have set the screen to one of the graphic modes using _gModeSet(). This function respects the screen clipping region as set by _gClipSet().

Files Library is LLIBG.LIB, header file is Llibg.api.

See Also _gClipSet(), _gEllipse(), _gFrame(), _gLine(), _gModeSet(), _gPolygon()

_gRGBColorGet()

Retrieve the color for a particular palette index

C Prototype

```
#include "llibg.api"
DWORD _gRGBColorGet(
                        int iPalNum
                        )
```

Arguments

iPalNum is the palette index of the color, beginning with one (1), that you want to retrieve.

Returns

The color of the specified palette index.

Description

_gRGBColorGet() retrieves a single color for the specified palette. This function is similar to _gPalGet() but instead of retrieving the entire palette, it retrieves only one color.

Files

Library is LLIBG.LIB, header file is Llibg.api.

See Also

_gPalGet(), _gRGBColorSet()

_gRGBColorSet()

Set the color for a particular palette index

C Prototype

```
#include "llibg.api"
ERRCODE _gRGBColorSet(
                        int iPalNum,
                        DWORD dwColorSet
                        )
```

Arguments

iPalNum is the palette index for color, beginning with one (1), that you want to define.

dwColorSet is the color. The range of valid values is limited to the number of colors available in the selected video mode. For example, in 16-color modes, valid values are between zero and 15, and in 256-color modes, valid values are between zero and 255.

Returns

SUCCESS if successful; otherwise, an error code.

Description

_gRGBColorSet() sets a single color for the specified palette. This function is similar to _gPalSet() but instead of setting the entire palette, it sets only one color.

Files

Library is LLIBG.LIB, header file is Llibg.api.

See Also

_gPalSet(), _gRGBColorGet()

_gScreenRest()

Restore a screen region previously saved with _gScreenSave()

C Prototype

```
#include "llibg.api"
void _gScreenRest(
                  int iTop,
                  int iLeft,
                  int iBottom,
                  int iRight,
                  ITEM itmSavedScreen
                  )
```

Arguments

iTop is the row coordinate of the top left corner of the screen region.

iLeft is the column coordinate of the top left corner of the screen region.

iBottom is the row coordinate of the bottom right corner of the screen region.

iRight is the column coordinate of the bottom right corner of the screen region.

itmSavedScreen is an item containing a screen region previously saved with _gScreenSave().

Files Library is LLIBG.LIB, header file is Llibg.api.

See Also _gScreenSave()

_gScreenSave()

Save a screen region

C Prototype

```
#include "llibg.api"
ITEM _gScreenSave(
                  int iTop,
                  int iLeft,
                  int iBottom,
                  int iRight
                  )
```

Arguments

iTop is the row coordinate of the top left corner of the screen region.

iLeft is the column coordinate of the top left corner of the screen region.

iBottom is the row coordinate of the bottom right corner of the screen region.

iRight is the column coordinate of the bottom right corner of the screen region.

Returns

In text mode, this function returns a string item identical to the CA-Clipper SAVESCREEN() function. In graphic mode, it returns an array item in which each element is a string.

Files

Library is LLIBG.LIB, header file is Llibg.api.

See Also

_gScreenRest()

_gWriteAt()

Draw graphic text without background

C Prototype

```
#include "llibg.api"
int _gWriteAt(
                int iX,
                int iY,
                DWORD dwFontColor,
                int iFontID,
                char far * fpString,
                int iMode,
                int iAttribute
                )
```

Arguments

iX is the pixel-based X coordinate of the starting point.

iY is the pixel-based Y coordinate of the starting point.

dwFontColor is the font color. The range of valid values is limited to the number of colors available in the selected video mode. For example, in 16-color modes, valid values are between zero and 15, and in 256-color modes, valid values are between zero and 255.

iFontID is the font handle.

fpString is a pointer to the display text.

iMode is the display mode, which you specify using one of the following values:

Display Mode Constants

Constant	Display Mode
LLG_MODE_SET	Overwrite existing pixels that are beneath where the string will be displayed. This is the most common display mode.
LLG_MODE_AND	Perform a logical AND on existing pixels—and on the display color—that are beneath where the string will be displayed.
LLG_MODE_OR	Perform a logical OR on existing pixels—and on the display color—that are beneath where the string will be displayed.

Display Mode Constants (cont.)

Constant	Display Mode
LLG_MODE_XOR	Perform a logical XOR on existing pixels—and on the display color—that are beneath where the string will be displayed (see Note below).
LLG_MODE_NO_DISPLAY	Compute the width of the string without displaying anything on the screen. Be aware that .FNT fonts are proportional (e.g., an "m" and an "i" do not use the same number of pixels).

Note: LLG_MODE_XOR allows you to move objects around on the screen without damaging the background. To retrieve the initial background, simply repeat the function call using the XOR display mode.

iAttribute is the font attribute, which you specify using one or more of the following values:

Font Attribute Constants

Constant	Font Attribute
LLG_FONT_BOLD	Bold
LLG_FONT_ITALIC	Italic
LLG_FONT_UNDERLINE	Underline

To specify more than one attribute, simply add the values together. Note also that you can specify a value of zero for this parameter if you do not want to specify any font attributes.

Returns

The length of **fpString* in pixels.

Notes

- ***Restrictions:*** You cannot use this function unless you have set the screen to one of the graphic modes using _gModeSet(). This function respects the screen clipping region as set by _gClipSet().

Files

Library is LLIBG.LIB, header file is Llibg.api.

See Also

_gClipSet(), _gFntLoad(), _gFntSet(), _gModeSet()

LLG_FNTCLIP structure

Structure for font clipping information

Structure

```
typedef struct
{
   int   iPixTop;
   int   iPixBot;
} LLG_FNTCLIP;

typedef LLG_FNTCLIP far * LLG_LPFNTCLIP;
```

Elements

iPixTop is the starting row number, in pixels, of the font matrix. Values can range from one to the height of the font.

iPixBot is the ending row number, in pixels, of the font matrix. Values can range from one to the height of the font.

Files Header file is Llibg.api.

See By _gFntClipGet(), _gFntClipSet()

LLG_MOUSESTATE structure

Structure for mouse state information

Structure

```
typedef struct
{
   int   iX;
   int   iY;
   int   iRow;
   int   iCol;
   int   iLeft;
   int   iRight;
   int   iVisible;
   int   iDriverVersion;
   int   iCursorShape;
   int   iClicksLeft;
   int   iClicksRight;
} LLG_MOUSESTATE;

typedef LLG_MOUSESTATE far * LLG_LPMOUSESTATE;
```

Elements

iX is the pixel-based X coordinate.

iY is the pixel-based Y coordinate.

iRow is the text-based row position.

iCol is the text-based column position.

iLeft is the state of the left mouse button, which is either LLM_BUTTON_DOWN if the button is down and LLM_BUTTON_UP if it is up.

iRight is the state of the right mouse button, which is either LLM_BUTTON_DOWN if the button is down and LLM_BUTTON_UP if it is up.

iVisible is the current state of the mouse pointer. 1 means that it is visible, and 0 means that it is hidden.

iDriverVersion is the version number of the mouse driver.

iCursorShape is the shape of the mouse pointer (see _mShow() for a table of possible values).

iClicksLeft is the number of left clicks since the value was last reset with _mState().

iClicksRight is the number of right clicks since the value was last reset with _mState().

Files Header file is Llibg.api.

See By _mState()

LLG_PALETTE structure

Structure for palette information

Structure

```
typedef struct
{
   WORD     wVersion;
   WORD     wNumEntries;
   LLG_RGB dwPalEntry[1];
} LLG_PALETTE;

typedef LLG_PALETTE far * LLG_LPPALETTE;
```

Elements

wVersion is the version reserved for future use.

wNumEntries is the number of entries in the palette.

dwPalEntry is a pointer to an array of LLG_RGB structures—one element for each color in the palette. The palette can consist of 16, 256, 32 KB, 64 KB, or 16 MB colors, depending on the video mode specified using _gModeSet().

Files Header file is Llibg.api.

See By _gPalGet(), _gPalSet()

LLG_POINT structure

Structure for specifying coordinates of a point

Structure

```
typedef struct
{
   int   iX;
   int   iY;
} LLG_POINT;

typedef LLG_POINT far * LLG_LPPOINT;
```

Elements

iX is the pixel-based X coordinate.

iY is the pixel-based Y coordinate.

Files Header file is Llibg.api.

See By _gPolygon()

LLG_RECT structure

Structure for screen region information

Structure

```
typedef struct
{
   int   iLeft;
   int   iTop;
   int   iRight;
   int   iBottom;
} LLG_RECT;

typedef LLG_RECT far * LLG_LPRECT;
```

Elements

iLeft is the X coordinate of the top left corner of the screen region.

iTop is the Y coordinate of the top left corner of the screen region.

iRight is the X coordinate of the bottom right corner of the screen region.

iBottom is the Y coordinate of the bottom right corner of the screen region.

Note: These coordinates are pixel-based unless you specify the *blsGraphic* parameter as false (.F.) when you call _gExclSet().

Files

Header file is Llibg.api.

See By

_mClipGet(), _mClipSet(), _gClipGet(), _gClipSet(), _gExclGet(), _gExclSet()

LLG_RGB structure

Structure for color information

Structure

```
typedef union
{
   struct
   {
      BYTE bRed;
      BYTE bGreen;
      BYTE bBlue;
      BYTE bFlags;
   } b;
   DWORD  dwColor;
} LLG_RGB;

typedef LLG_RGB far * LLG_LPRGB;
```

Elements

bRed is a value from zero to 255 defining the red component of *dwColor*.

bGreen is a value from zero to 255 defining the green component of *dwColor*.

bBlue is a value from zero to 255 defining the blue component of *dwColor*.

bFlags can be set to true (.T.), in which case the LLG_RGB structure will be set to 32 bits. Otherwise, it will be set to 16 bits.

dwColor is the color whose RGB components you are specifying. The range of valid values is limited to the number of colors available in the selected video mode. For example, in 16-color modes, valid values are between zero and 15, and in 256-color modes, valid values are between zero and 255.

Files Header file is Llibg.api.

See By LLG_PALETTE

LLG_VIDEOMODE structure

Structure for video mode information

Structure

```
typedef struct
{
   int     iTextRow;
   int     iTextCol;
   int     iGraphRow;
   int     iGraphCol;
   int     iFontRow;
   int     iFontCol;
   DWORD   dwColorNb;
   int     iVideoMode;
   int     iLibVer;
   int     iLastMode;
   DWORD   dwLastColor;
} LLG_VIDEOMODE;

typedef LLG_VIDEOMODE far * LLG_LPVIDEOMODE;
```

Elements

iTextRow is the number of text rows on the screen.

iTextCol is the number of text columns on the screen.

iGraphRow is the number of graphic rows on the screen.

iGraphCol is the number of graphic columns on the screen.

iFontRow is the number of graphic rows per character.

iFontCol is the number of graphic columns per character.

dwColorNb is the number of available colors.

iVideoMode is the video mode (see _gModeGet() for a table of possible values).

iLibVer is the version number of LLIBG.LIB.

iLastMode is the last display mode setting (see _gEllipse() for a table of possible values).

dwLastColor is the last color used in a Light Lib Graphics API function call.

Files Header file is Llibg.api.

See By _gModeGet()

Chapter 15
CA-Clipper/Exospace API Reference Listing

ExoFreeSelector()
ExoIsDPMI()
ExoIsExoSpace()
ExoIsPM()
ExoIsVMM()
ExoProtectedPtr()
ExoRealPtr()
ExoReside()
ExoRMInterrupt()
ExoSegCSAlias()
ExoSegDSAlias()
_xalloclow()
_xfreelow()

Chapter 15
CA-Clipper/Exospace API Reference

The CA-Clipper/Exospace API allows your Extend routines to manipulate and query information regarding the Exospace link process. This chapter provides an alphabetical reference to all of the functions in the CA-Clipper/Exospace API.

The prototypes for these functions are defined in the header file Exospace.api, located in the CLIP53\INCLUDE directory. The functions themselves are located in the EXOSPACE.LIB, located in the CLIP53\LIB directory.

ExoFreeSelector()

Cancels a protected mode segment descriptor created by ExoProtectedPtr(), ExoSegCSAlias(), or ExoSegDSAlias()

C Prototype

```
int ExoFreeSelector(unsigned int selector)
```

Returns

This function always returns 0.

Description

This function is a primitive function—it corresponds to the DPMI function that cancels a descriptor.

ExoFreeSelector() must be passed only the selector part of a pointer created with ExoProtectedPtr(), ExoSegCSAlias(), or ExoSegDSAlias().

Because selectors are a limited resource in protected mode (there are "usually" around 8000 available for code and data segments), you should free selectors you have created with ExoProtectedPtr(), ExoSegCSAlias(), or ExoSegDSAlias() as soon as you are done with them.

Examples

This code creates a protected mode pointer to video color memory, does some unknown operation with it and then frees the protected mode selector.

```
void *rmvideo;
void *pmvideo;

/* set up pointer to point to real mode address 0xB800:0 which
   is the address of the color video card */
FP_SEG(rmvideo) = 0xB800;
FP_OFF(rmvideo) = 0;

/* create protected mode pointer for use in protected mode */
pmvideo = ExoProtectedPtr(rmvideo, 0x8000);

... more code using pmvideo pointer would normally be here ...

/* free up selector used by pmpointer now that we are done
   with it */

ExoFreeSelector(FP_SEG(pmvideo));
```

Files

Library is EXOSPACE.LIB, header file is Exospace.api.

See Also

ExoProtectedPtr(), ExoSegCSAlias(), or ExoSegDSAlias()

ExoIsDPMI()

Determines if running in a DPMI host environment

C Prototype

```
int ExoIsDPMI(void)
```

Returns

This function returns nonzero (true (.T.)) if the program is running in a DPMI host. If the program is not running under DPMI, ExoIsDPMI returns zero.

Description

Use this function to determine whether your program is running under a DPMI host, such as Windows 3.x in enhanced mode.

Examples

This program fragment prints a message telling whether or not the program executing is running under DPMI.

```
If (ExoIsDPMI())
   printf("Program is running under DPMI");
else
   printf("Program is not running under DPMI");
```

Files

Library is EXOSPACE.LIB, header file is Exospace.api.

See Also

ExoIsExoSpace(), ExoIsPM()

ExoIsExoSpace()

Determines if running under CA-Clipper/Exospace

C Prototype

```
int ExoIsExoSpace(void)
```

Returns

This function returns nonzero (true (.T.)) if an CA-Clipper/Exospace program is running.

Description

Use this function to determine whether your program is running under CA-Clipper/Exospace. If the program has been linked with the real mode "stub" library, this function will return zero.

Examples

This program fragment prints a message telling whether or not the program executing is an CA-Clipper/Exospace application.

```
if (ExoIsExoSpace())
   printf("ExoSpace program is running");
else
   printf("ExoSpace program is not running");
```

Files

Library is EXOSPACE.LIB, header file is Exospace.api.

See Also

ExoIsDPMI(), ExoIsPM()

ExoIsPM()

Determines if the CPU is running in protected mode

C Prototype

```
int ExoIsPM(void)
```

Returns

This function returns nonzero (true (.T.)) if the CPU is executing in protected mode, otherwise it returns 0.

Description

Use this function to determine whether the processor is running in protected mode.

Examples

This program fragment prints a message telling whether or not the CPU is in protected mode.

```
If (ExoIsPM())
  printf("cpu is in protected mode");
else
  printf("cpu not in protected mode");
```

Files

Library is EXOSPACE.LIB, header file is Exospace.api.

See Also

ExoIsDPMI(), ExoIsExoSpace()

ExoIsVMM()

Determines if the CA-Clipper/Exospace program is using the CA-Clipper/Exospace VMM system

C Prototype

```
int ExoIsVMM(void)
```

Returns

This function returns nonzero (true (.T.)) if the program is using the CA-Clipper/Exospace VMM system, otherwise it returns 0.

Description

Use this function to determine whether the CA-Clipper/Exospace VMM system is active. It will not be active if the option "EXOSPACE PACKAGE NOVM" was specified during the link or if you are running under a DPMI server such as a Windows or OS/2 DOS box.

Examples

This program fragment prints a message telling whether or not the program is using the CA-Clipper/Exospace VMM system.

```
if (ExoIsVMM())
   printf("program is using the ExoSpace VMM system");
else
   printf("program is not using the ExoSpace VMM system");
```

Files Library is EXOSPACE.LIB, header file is Exospace.api.

See Also ExoIsDPMI(), ExoIsExoSpace()

ExoProtectedPtr()

Creates a protected mode pointer from a real mode pointer

C Prototype

```
void *ExoProtectedPtr(void *rm_ptr, unsigned int size)
```

Returns

This function returns the protected mode pointer (*pm_ptr*), or NULL if the pointer cannot be allocated. A new descriptor is allocated.

Description

The protected mode data pointer created by this function points to a segment of size bytes. Since a value of 64KB is too large for an unsigned variable, use 0 to specify 64KB for size. The offset of *pm_ptr* is always 0. The base physical address of *pm_ptr*'s segment is the absolute address of *rm_ptr*, which is segment(*rm_ptr*) * 16 + offset(*rm_ptr*). Because ExoProtectedPtr() is run with interrupts enabled, realtime applications should not call it from an external interrupt handler.

The offset of the pointer returned by ExoProtectedPtr() will ALWAYS be 0.

ExoFreeSelector() *must* be called to free the selector allocated with ExoProtectedPtr(). Selectors are a limited resource and repeated calls to ExoProtectedPtr() as well as ExoSegCSAlias() and ExoSegDSAlias() could use up all the available selectors causing the program to crash. Once you are done with a pointer created with ExoProtectedPtr(), free the selector unless you will be using the pointer throughout the program..

Examples

See example for ExoFreeSelector().

Files Library is EXOSPACE.LIB, header file is Exospace.api.

See Also ExoRealPtr(), ExoFreeSelector()

ExoRealPtr()

Creates a real mode pointer from a protected mode pointer

C Prototype

```
void *ExoRealPtr(void *pm_ptr)
```

Returns

This function returns the real mode pointer, or NULL if *pm_ptr* is in extended memory (not accessible in real mode).

Description

The *pm_ptr* argument is the protected mode pointer to be converted. Note that if the real mode pointer is 0:0, the returned value is the same as an error value.

Examples

See example for _xalloclow().

Files

Library is EXOSPACE.LIB, header file is Exospace.api.

See Also

ExoProtectedPtr()

ExoReside()

Flags a VMM segment to remain resident in memory

C Prototype

```
int ExoReside(void *pmptr)
```

Returns

This function returns -1 on error, 0 or 1 otherwise.

Description

When the VMM system is active a code or data segment may be swapped out at any time. When data segments are swapped out they are always written to the swap file, but code segments are never written to the swap file when they are swapped out unless a data segment alias created by ExoSegDSAlias() exists at the time the code segment is swapped out. If you write to a data segment alias of a code segment when CA-Clipper/Exospace's VMM is active, you will need to mark the segment as resident before freeing the data segment alias with ExoFreeSelector().

Examples

See example for ExoSegDSAlias().

Files Library is EXOSPACE.LIB, header file is Exospace.api.

See Also ExoSegDSAlias()

ExoRMInterrupt()

Sets the registers, then signals a real mode interrupt

C Prototype

```
int ExoRMInterrupt(int intno, EXOREGS *inregs,
                   EXOREGS *outregs)
```

Returns

This function returns the flag's register.

Description

Use ExoRMInterrupt() to signal a real mode interrupt that requires you to set registers. This function allows you to set real mode segment register values from protected mode. When you use regular passdown interrupts, you cannot set real mode register values.

A call to ExoRMInterrupt() sets the registers to the values you provide in the structure to which *inregs* points; then it invokes interrupt *intno*. After the interrupt is processed, ExoRMInterrupt() stores the register values in the structure pointed to by *outregs*. The structures pointed to by *inregs* and *outregs* are both type EXOREGS. Note that you cannot set the SS and SP registers from an EXOREGS structure.

Before CA-Clipper/Exospace signals the specified interrupt, it sets the CPU flags to the value they had when you called ExoRMInterrupt() in protected mode. Notice that the value of the flags register after the real mode interrupt call is made is returned by this function.

When you use ExoRMInterrupt(), there must be a return from the real mode interrupt to match every call (unless your program is terminating). Also, if re-entrance on ExoRMInterrupt() is possible, the interrupt should use less than 256 bytes of stack. You can get around this limit by switching to a different stack in the real mode handler/function.

Examples

This example signals DOS function 2Ch to get the time of day. As required for that call, the high order byte of the AX register is set to 2Ch. Note that the structure EXOREGS does not contain a field AH, so you must set the high order byte using the shift operator (<<). After the call to ExoRMInterrupt(), this example displays the contents of the *outreg* registers. The display shows the time in hours, minutes and seconds. Since the hours and seconds are stored in the high order byte, the shift operator is used again.

```
int main()
{
   EXOREGS inreg;
   EXOREGS outreg;

    inreg.ax = 0x2C << 8;

    /* DOS function "get time" - signal int 21h */
    ExoRMInterrupt(0x21, &inreg, &outreg);

    printf("The time is %d:%02d:%02d\n",
      outreg.cx >> 8, outreg.cx&0xFF, outreg.dx >> 8);
}
```

Files

Library is EXOSPACE.LIB, header file is Exospace.api.

ExoSegCSAlias()

Creates a code descriptor for the given data segment

C Prototype

```
void *ExoSegCSAlias(void *pm_ptr)
```

Returns

This function returns a function pointer to the same linear address as *pm_ptr*, or NULL if an error occurred.

Description

The new descriptor created by ExoSegCSAlias points to the segment base of *pm_ptr*. This function enables you to execute code in a data segment. To store data in a code segment, use ExoSegDSAlias(). Because ExoSegCSAlias() is run with interrupts enabled, realtime applications should not call it from an external interrupt handler.

The offset of the alias will ALWAYS equal the offset of the original pointer.

ExoFreeSelector() MUST be called to free the selector allocated with ExoSegCSAlias(). Selectors are a limited resource and repeated calls to ExoSegCSAlias() as well as ExoSegDSAlias() and ExoProtectedPtr() could use up all the available selectors causing the program to crash. Once you are done with a pointer created with ExoSegCSAlias(), free the selector unless you will be using the pointer throughout the program.

Examples

This code fragment calls executable machine code written by the program to a data buffer.

```
char writablecode[100];
void (*funcptr(void));

/* during execution the program would write executable machine
   code to the 100 byte writablecode buffer */

/* create a selector that is a code segment alias to the data
   segment containing the variable writablecode */

funcptr = ExoSegCSAlias(writablecode);

/* set the offset of the function pointer to the offset of the
   writablecode buffer */
FP_OFF(funcptr) = FP_OFF(writablecode);

/* call the code written to the writablecode buffer */
*funcptr();

/* cancel the code segment selector */
ExoFreeSelector(FP_SEG(funcptr));
```

Files

Library is EXOSPACE.LIB, header file is Exospace.api.

See Also

ExoSegDSAlias()

ExoSegDSAlias()

Creates a data descriptor for the given code segment

C Prototype

```
void *ExoSegDSAlias(void *pm_ptr)
```

Returns

This function returns a data pointer to the same linear address as *pm_ptr*, or NULL if an error occurred.

Description

This function creates a new data descriptor with the same base as the segment specified by *pm_ptr*. After the call to ExoSegDSAlias(), the descriptor indicated in *pm_ptr* still exists and retains its original segment type. The offset of the alias will ALWAYS equal the offset of the original pointer.

ExoFreeSelector() MUST be called to free the selector allocated with ExoSegDSAlias(). Selectors are a limited resource and repeated calls to ExoSegDSAlias() as well as ExoSegCSAlias() and ExoProtectedPtr() could use up all the available selectors causing the program to crash. Once you are done with a pointer created with ExoSegDSAlias(), free the selector unless you will be using the pointer throughout the program.

Examples

The example below consists of two modules: one in assembly language and one in C. The int_instruction label in the assembly code marks the location of an instruction signaling Interrupt 0h. The C code overwrites this instruction so that a different interrupt is signaled.

The C code calls ExoSegDSAlias() to get an alias data pointer to int_instruction. The example then increments the pointer by one byte to point to the location of "0h" in the instruction. The example overwrites this location with "0x88" and frees the data descriptor with ExoFreeSelector() since it is no longer needed.

```
public _do_int, int_instruction
_do_int:
...
int_instruction:
int 0h
...

char *data;
extern int_instruction();

/* create data segment alias and write interrupt number 0x88 */
data = ExoSegDSAlias(int_instruction);
*(data + 1) = 0x88;

/* mark code segment as resident so the VMM system won't swap
   it out and lose the changes made to it */
ExoReside(int_instruction);

/* free the selector created by ExoSegDSAlias() since we don't
   need it anymore */
ExoFreeSelector(FP_SEG(data));
```

Files

Library is EXOSPACE.LIB, header file is Exospace.api.

See Also

ExoSegCSAlias()

_xalloclow()

Allocates low DOS memory for use with real mode interrupts that must be passed buffers within the first 1 MB

C Prototype

```
void *_xalloclow(unsigned int sizebytes)
```

Returns

This function returns a protected mode pointer to the allocated memory.

Description

This function returns a protected mode pointer to the allocated memory. Use this pointer to write to or read from the allocated buffer. When calling a real mode interrupt you must get a real mode pointer to the allocated buffer with the ExoRealPtr() function and pass that address in the EXOREGS structure of a ExoRMInterrupt() call.

You can allocate a 64K buffer by passing a size of 0.

You must use _xfreelow() to free memory allocated with this function. Do not use xfree()!

It is suggested that you free the allocated low DOS memory as soon as possible to keep down the amount of low DOS memory that may be allocated at the same time not only by your functions but also by other C/ASM code or by other third party libraries.

CA-Clipper/Exospace has added the LOWMEM parameter to the CLIPPER environment variable to allow a developer to set aside low DOS memory for allocation with _xalloclow(). The LOWMEM parameter does not have to be specified in order for _xalloclow() to work; if there is no available low DOS memory when _xalloclow() is called, CA-Clipper/Exospace will free up low DOS memory it uses in order to meet the request. At this time because of the current design of CA-Clipper/Exospace's memory allocation scheme we recommend that you ignore the LOWMEM parameter. Future revisions or individual end-user needs may make the LOWMEM parameter more necessary.

Examples

This example maps a fake root directory of a specified drive under Novell NetWare. The function below accepts a drive number (0=default, 1=A:, ...) and path for fake root; it returns zero on success and an error code if the function failed.

```
int maproot(int drive, char *path)
{
   int returnvalue = 0xFFFF;
   int flags;
   char *buffer;
   char *realptr;
   EXOREGS inoutregs;

   /* allocate low memory buffer to store path */
   if ((buffer = _xalloclow(strlen(path) + 1)) != NULL)
   {


      /* store path in low memory buffer */
      strcpy(buffer, path);

      /* get real mode pointer to low memory buffer */
      if ((realptr = ExoRealPtr(buffer)) != NULL)
      {

         /* fill in EXOREGS structure */
         inoutregs.ax = 0xE905;
         inoutregs.bx = drive;
         inoutregs.ds = FP_SEG(realptr);
         inoutregs.dx = FP_OFF(realptr);

         /* call the real mode interrupt */
         flags = ExoRMInterrupt(0x21, &inoutregs, &inoutregs);

         /* see if carry flag is set to indicate error */
         if (flags & 0x0001)
            /* return error code in al */
            returnvalue = (inoutregs.ax & 0x00FF);
         else
            /* return no error code */
            returnvalue = 0;
}

      /* free low memory buffer */
      _xfreelow(buffer);
   }
}
```

Files Library is EXOSPACE.LIB, header file is Exospace.api.

See Also ExoRealPtr(), ExoRMInterrupt(), _xfreelow()

_xfreelow()

Frees memory allocated by _xalloclow()

C Prototype

```
void _xfreelow(void *lowmemory)
```

Returns

This function does not return a value.

Description

You must use this function to free memory allocated by _xalloclow(). Do not use xfree() to free memory allocated by _xalloclow()!

Examples

See _xalloclow() example.

Files

Library is EXOSPACE.LIB, header file is Exospace.api.

See Also

_xalloclow()

Index

1

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Z