WINDCREST/McGRAW-HILL

The Clipper® Interface Handbook

- Master Clipper through 5.01
- Develop interfaces to C, assembly, DOS & PC-MOS
- Program customized user menus & mouse support

TAB ASIAN STUDENT EDITION

John Mueller

The Clipper® Interface Handbook

John Mueller

Windcrest®/McGraw-Hill

FIRST EDITION
FIRST PRINTING

Library of Congress Cataloging-in-Publication Data

Mueller, John, 1958–
The Clipper interface handbook / by John Mueller.
p. cm.
Includes index.
ISBN 0-8306-3532-7 (pbk.)
1. Compilers (Computer programs) 2. Clipper (Computer program)
I. Title.
QA76.76.C65M83 1991
005.75'65—dc20 91-34294
CIP

TAB Books offers software for sale. For information and a catalog, please contact TAB Software Department, Blue Ridge Summit, PA 17294-0850.

Acquisitions Editor: Brad Schepp
Book Editor: Kellie Hagan
Director of Production: Katherine G. Brown
Series Design: Jaclyn J. Boone WP1
Cover: Sandra Blair Design, and Brent Blair Photography, Harrisburg, PA

Acknowledgments

My friend Wallace Wang helped get me started writing and encourages me to do so with a fervor unmatched by anyone. His gift to me has always been laughter in the face of overwhelming odds. Everyone should have someone like this to cheer them on.

A good technical editor is hard to find, at least that is what I've heard. Dian Schaffhauser has been an example to me of what an editor should be. Never harsh, and always willing to talk, Dian led me on the chase for perfection. She encouraged me to find that elusive thing called truth.

I want to thank David Frier for technically reviewing this book. I can't help but feel his contribution has made this book better than I could have alone.

I also want to acknowledge the contributions of those people who made parts of this book possible. Tanya VanDam of Microsoft Corporation graciously helped me obtain the best information possible about Macro Assembler and the C compiler. Frances Jackson and Craig Ogg of Nantucket Corporation assisted me in every way possible to obtain and understand the Clipper compiler. Artist Graphics provided the Artist TI12, which I used to document the TMS34010 processor. And finally, I want to thank all those technicians, developers, and consultants who cheered me on, provided criticism and ideas, and generally helped me shape this book.

Introduction

This book conveys four main ideas that anyone using Clipper needs to consider. First, "What does Clipper provide in the way of programming tools?" The Introduction looks at what you should expect from Clipper as a compiler. Second, "How does Clipper interface with Assembler and C?" This is the basis for building your own tools to enhance the Clipper programming environment. Third, "How can you enhance Clipper-provided programming tools?" Once you understand the application program interface (API), you can begin to modify your working environment. Doing so will increase your productivity while decreasing the amount of work required to create a program. Fourth, "How can I create entirely new programming tools?" Enhancements are often not enough to give a developer an edge over the competition. Programming tools are the means by which you cannot only reduce programming time, but differentiate your product from someone else's. This book assumes that you are an intermediate to advanced programmer, so actual implementation of an idea takes precedence over the theory behind the idea.

What is in this book?

The *Clipper Interface Handbook* begins with a brief discussion of all the Clipper commands and functions already available. This is important because some readers might not know that a particular capability already exists or that there are problems with a particular capability. The main reason for this lack of user knowledge is the relatively poor documentation supplied by Nantucket with Clipper. The newest release does not even supply hardcopy documentation. As a result, there are new features that many people are not familiar with.

Part 1: Clipper Introduction This introductory section contains the first four chapters. Chapters 1 and 2 discuss Clipper commands and functions, and each description highlights any anomalies you should expect to see when using the command or function. It also tells you when the command or function first appeared in Clipper. This will help you determine when to use an advanced feature and when to maintain compatibility with older versions of Clipper. Chapter 3 discusses the preprocessor and how to use code blocks. Both of these features are new to version 5.0 of Clipper. Understanding what they can do for you is paramount to tool building. Chapter 4 is a detailed discussion of the object-oriented features in Clipper. In many cases, these new object-oriented features allow you to redefine your programming environment without resorting to C or assembler add-on functions.

Part 2: Interfacing Clipper The second section of the book discusses the Clipper interface to other programming languages. It is divided into two chapters, one for Assembly Language and another for C. Each chapter discusses Nantucket-supplied interface aids, user additions to these interfaces, and new interface aids that you can create. Many of the differences between C and assembler add-ons are discussed as well, for example, using assembler or C to create an add-on.

Part 3: Clipper add-ons The third section of the book discusses Clipper add-ons, in other words, using the supplied functions as a base and adding to them. This section is also split into two chapters. The first chapter covers standard functions and the second covers network-related functions. Enhancing existing commands and functions is often easier than programming them from the beginning. This section will help you determine when you can use this relatively simple technique in place of a full-blown C or assembler add-on.

Part 4: New functions The fourth and final section looks at creating entirely new functions. Once you determine that Clipper does not supply a required function and that an existing function cannot be modified, you can use this book to create entirely new functions. This section contains four chapters, and is therefore a major portion of the book. Chapter 9 discusses Clipper interfaces to other programs. This is intended to illustrate how to create a data (versus programming language) interface. For example, you could write a program where the client wants a program to call a cash register, download information, convert it to dBASE III format, and then analyze the receipt and employee information. Without the proper interface to the cash register, this program couldn't work. Chapter 10 discusses how to create libraries of routines. This is important in a group programming environment. Using a library of routines is far more convenient than using individual object modules. Chapters 11 and 12 discuss low-level access to DOS routines not exploited by Clipper, specifically mouse routines and graphic displays.

Appendices The seven appendices contain reference material that you might need to create a program. This includes information about Clipper responses to the keyboard, a summary of commands for both Assembly Language and C, and two appendices describing how to directly access the computer's peripheral chips.

Why is this book unique?

Most Clipper programming books leave you stranded with untested code fragments that may or may not work when used. This book provides complete programs as examples. Therefore, you not only see how to include a feature as part of a program, but you know that the code works as well. Also, no other book attempts to cover low-level programming techniques in the detail that this book does. Even though some books tell you that you can access hardware, they fail to show you how. The *Clipper Interface Handbook* provides you with complete coverage of how to use Clipper to your best advantage.

Programming conventions

There are several programming conventions used throughout this book. An understanding of these conventions will help you receive more information from the examples in this book and from the Nantucket manuals in general. In addition, these same concepts are equally applicable to Clipper, C, and assembly code. Many of these conventions have been discussed by developers at conferences and on bulletin board systems (BBSs).

The first stage of development for this system was started by Charles Simonyi of Microsoft Corporation. He called his system *Hungarian notation*. There are many places that you can obtain a copy of his work, including many BBSs. His work was further enhanced by developers in close association with the Nantucket Corporation. A final copy of the enhancement to the original Hungarian notation was published by Robert A. Difalco of Fresh Technologies. You can find his work on many BBSs as well, including the Nantucket-supported forum on CompuServe.

Much of the information in this section can be found in one of the two previously mentioned documents in one form or another. The purpose in presenting them here is to make you aware of the exact nature of the conventions and show you how to use them to your best advantage. There are four reasons why you should use these naming conventions in your programs:

Mnemonic value This allows the programmer to remember the name of a variable more easily, an important consideration for team projects.

Suggestive value You might not be the only person modifying your code. If you're working on a team project, others in the team will most likely look

at the code you've written. Using these conventions will help others understand your work.

Consistency A programmer's ability is often evaluated on the basis of not only how efficiently he programs or how well the programs he creates function, but also how easily another programmer can read his code. Using these conventions will help you maintain uniform code from one project to another. Other programmers will be able to anticipate the value or function of a section of code simply by the conventions you use.

Speed of decision In the business world, the speed at which you can create and modify code will often determine how successful a particular venture will be. Using consistent code will reduce the time you spend trying to decide what someone meant when creating a variable or function. This reduction in decision time will increase the amount of time you have available for productive work.

Procedure and function naming conventions

This book uses the same conventions for naming both functions and procedures. To make the text easier to understand, I will refer to both procedures and functions as functions. The following rules will help you understand the conventions used to name functions throughout the book.

- Some languages allow you to type a function by the value it returns. This is not the case with Clipper, however, because it is not a strongly typed language. Because you can't rely on a specific return value from a function, you can't use an indicator to show its return type. For example, a variable returning a numeric value would begin with a lowercase *n* to show its type. Therefore, all Clipper functions begin with an uppercase character for an external third-party function, a lowercase letter for a native external function, or an underscore for an internal function.
- You can further differentiate between native and third-party functions because native functions use all lowercase letters. A third-party function might use a combination of upper- and lowercase, providing the first character is always uppercase.
- In many cases, a function converts one value to another value. To differentiate these functions from functions that perform a more generalized task, you type the input value, a 2, and then the output value. For example, if you wanted to create a function for converting a value from the frequency domain to the time domain, you could name the function Freq2Time.
- Even though you can't type a function to return a specific value, there are instances where the purpose of a function is clearly outlined and you know that it returns a specific value. In those cases you can use a standard qualifier to help define the function. A list-

ing of these standard qualifiers appears in the section on variable naming conventions.

- Always define a function using only one or two standard qualifiers. Some programmers use so many qualifiers to define a function name that they actually make their code less rather than more descriptive. The purpose of a function can become difficult to determine if you use too many qualifiers.
- It is always convenient to be able to quickly find where a function is defined within the source code. To help in this, always capitalize the keywords PROCEDURE, FUNCTION, and RETURN. This will make the task of finding function definitions easier.
- To help differentiate native standard functions from those using the object-oriented programming (OOP) exported method, use a combination of upper- and lowercase letters for the OOP function, and always make the first letter of the function lowercase. For example, oBrowse:goBottom( ) is an exported instance variable of the TBrowse class.

Using these seven rules will make it much easier to determine the purpose and origin of the functions you use within a program. The following examples illustrate the seven rules.

```
SomeFunc( )              // third-party function
cls                      // native external function
__retni( )               // native internal function
Num2Char( )              // third-party conversion function
Bin2W( )                 // native conversion function
SetColor( )              // qualified function
FUNCTION SetColor( )     // keyword capitalization
oGet:KillFocus           // OOP function
```

DBF and field-related naming conventions

One of the things that differentiates database programming from other types of programming is the use of databases and indexes. These file structures rely on the contents of fields within the structure. Because these are of such importance in a Clipper program, you need to differentiate between a database, field, index, and standard variable. Often, a database is assigned an alias when the user opens it. For the purpose of this discussion, aliases will have the same naming conventions as database files.

One way to make a piece of code stand out from the code around it is by using capitalization. To differentiate the three parts of a database from the rest of the code, the database filenames, index filenames, and field names will always be expressed in capital letters.

Just like other variables, it makes sense to give database variables log-

ical names. Whenever possible, use the same standard qualifiers for database filenames, index filenames, and field names as you use for standard variables.

Even with the precaution of using all capital letters to reference a database filename and field name, it might still be possible for someone to confuse the two variables within your code. Because of this and also the possibility of confusing the compiler, always reference a field name with an alias. This will ensure that anyone reading your code will instantly recognize a field. In addition, you'll ensure that the compiler knows that you're referring to a field name rather than a variable. An example of this approach is SOMEDATA→FIELD NAME, where SOMEDATA is an alias for a database file, and FIELD NAME is the name of a field.

It's always best to show an association where one exists. For this reason, when a variable is used to store the contents of a field, it's best to give it the same name as the field, with the addition of a standard prefix. For example, if you have a field named FIELDNAME, and that field stores character values, then the variable name would be cFieldName. A list of variable prefixes is provided in the variable naming convention section.

There is also the question of exactly how to name a database file. One strategy is to build a database filename out of various parts. For example, if you had a database that was used for accounting, you might start each file specifically for that purpose with the prefix ACT. That way, you could easily separate the accounting files from other files in the directory. Another typical database is one that contains customer addresses and other information. You could give these files a prefix of CUST.

Just as a variable can be associated with a specific field in a database, index files are always associated with a specific database file. Because you want to be able to see a relationship where it occurs, you should always use the same name for an index file as you do for the database file. However, then you have to handle databases with multiple indexes. In these cases, it's better to give the database file a seven-character name and sequentially number the index files. For example, the first index file of a database used to store customer address information might be CUSTADR1.NTX. This approach is much better than trying to indicate the indexing scheme as part of the index filename.

Variable naming conventions

Variables are one of the hardest parts of a program to understand. Unlike functions and procedures, variables are not defined in the manual anywhere and few programs have published data dictionaries. As a result, there is often a lot of confusion about the exact meaning of a variable. There are several ways to understand the variables you use within a program.

Always prefix a variable with a single lowercase letter, indicating its type. In most cases this is the first letter of the variable type, so it's easy to

remember what letter to use. The following examples show the most common prefixes. Note that these prefixes are also the values returned by the ValType( ) function described in chapter 2.

Prefix	Variable type
a	Array
b	Code block
c	Character
d	Date
h	Handle
l	Logical
n	Numeric
o	Object
x	Variable type (macro or changing value)

Some variables represent the state of a database, or store the state of another variable. You can identify these variables using a three-character state qualifier. The following examples represent the most common state qualifiers.

Qualifier	State
New	a new state
Sav	a saved state
Tem	a temporary state

A standard qualifier can help someone see the purpose of a variable almost instantly. For example, using the Clr qualifier tells you that this variable is used in some way with color. You can even combine the qualifiers to amplify their effect and describe how the variable is used. For example, cClrCrs is a character variable that determines the color of the cursor. Using from one to three of these qualifiers is usually sufficient to describe the purpose of a variable. The following standard qualifiers are examples of the more common types.

Qualifier	Type	Qualifier	Type
Ar	Array	Msg	Message
Attr	Attribute	Name	Name
B	Bottom	Ntx	Index File
Clr	Color	R	Right
Col	Column	Rec	Record number
Crs	Cursor	Ret	Return value
Dbf	Database file	Scr	Screen
F	First	Str	String
File	File	T	Top
Fld	Field	X	Row
L	Last/left	Y	Column

Use the following specifications to refer to optional pointer references:

Qualifier	Reference
1,2,3	State pointer references, as in cSavClr1, cSavClr2, etc.
Max	Strict upper limit, as in nFldMax, maximum number of fields
Min	Strict lower limit, as in nRecMin, minimum number of records

Other conventions

Besides programming conventions, there are some book conventions I use to illustrate examples more clearly. These conventions are not part of the programs, but are used to illustrate the use of a command, function, or procedure. These book conventions are as follows:

<EXP> A standard expression of no particular type. This usually refers to a command or function that accepts multiple types as input.

<aEXP> An array is required as input. The command or function description will tell you if it is a single or multidimensional array. It will also tell you the variable type required as input.

<bEXP> A code block is required as input. The command or function description will provide the parameters that the code block must meet. The example section of the description will show how to format the code block.

<cEXP> A character expression is required as input.

<dEXP> A date expression is required as input.

<hEXP> A file handle is required as input. The description will tell you how the handle is obtained and for what purpose it is used.

<lEXP> A logical expression is required as input. This is always the Boolean operators .T. or .F., or some expression that equates to a Boolean.

<nEXP> A numeric expression is required as input.

<oEXP> An object is required as input.

<xEXP> Some type of variable expression is required as input. The command or procedure description will tell you what types of input you can provide.

[<EXP>] The requested expression is optional. This usually means that the command or function will perform the specified task correctly without this input. The optional expression merely enhances program operation in some way.

EXP LIST A list of expressions separated by commas or combined with math operators is required. A single expression constitutes a subset of the list in most cases.

KeyWord Type this word exactly as written in the heading of the description, using the proper capitalization (as described in the previous section) within your program.

<**SCOPE**> This usually appears in conjunction with a database or array command or function. It indicates that you can specify the range of records on which the command or function operates.

<**FIELD**> You must supply a field expression as input. This usually means that the command or function works directly with the database instead of variables.

<**CONDITION**> An expression that equates to a Boolean output. It is used in conjunction with the scope clause of a command or function to further define the range of records on which the command or function operates.

The book uses other conventions in special cases. In these instances, the exact meaning of the convention is stated in the description of the command or function.

1
Clipper commands

This chapter contains an alphabetical listing of all Clipper commands. In many cases, the use and execution of these commands vary widely from other *x*base (generic database) dialects. The preprocessor found in versions 5.0 and above of Clipper will allow you to modify this behavior. To find out how to enhance the capabilities provided by these versions, read chapter 3.

Each command description in this chapter also provides you with the version in which the command first appeared. If the command appeared in more than one version (Summer 87, 5.0, or 5.01), then the version information will tell you how the command changed. In many cases it will become apparent that version 5.0 enhanced the language, while 5.01 increased compatibility. Some of the version-specific information provides warnings about using certain commands under specific conditions. These warnings include unimplemented features between versions, enhancements, anomalies, and fixes that change the behavior of the command.

This chapter uses the conventions explained in the introduction to the book. Reading the introduction will increase your understanding of the examples provided as part of each description.

Each command entry contains the command line interface for the command and a description of how to use it. Notice how each version changes the behavior of some commands. These differences determine how you should use the command while programming. In this chapter, I assume that you're using the default Clipper configuration, libraries, and other support files.

?/?? <EXP LIST>

Summer 87, 5.0, 5.01

A change in versions 5.0 and 5.01 is that the preprocessor actually substitutes the QOut( ) function for the ? command, and the QQOut( ) function for the ?? command. This command is provided for compatibility reasons only.

This command displays the results of the expression list. It does not provide any method of formatting the output. The output appears at the current cursor position unless the expression list contains the control codes necessary to change the cursor position. In most cases, use the @ Say/Get command in place of this command for standard output. Use the QOut( ) or QQOut( ) function within code blocks. Version 5.01 provides the Alert( ) function for error messages.

The ? command displays a single line of text or numbers. It always ends the text with a carriage return and line feed combination. The ?? places the text or numbers on the current line. EXAMPLE:

```
? "Some Text"
```

@ <nEXP1>, <nEXP2>, <nEXP3>, <nEXP4> Box <cEXP1> [Color <cEXP2>]

Summer 87, 5.0, 5.01

5.01 provides a new function, DispBox( ), in place of this command. The DispBox( ) function provides greater flexibility and is easier to use than the @...Box command. In addition, both 5.0 and 5.01 allow you to use a numeric argument in place of cEXP1. A value of 1 draws a single box, while 2 draws a double box. 5.01 also adds cEXP2, which allows you to change the color of the box.

Use this command to draw a box around a display area. nEXP1 contains the top row offset. nEXP2 contains the left column offset, nEXP3 contains the bottom row offset, and nEXP4 contains the right column offset. cEXP1 contains a string of nine characters. The first eight characters are the line-drawing characters used for corners and sides of the box. Clipper begins a box at the upper left corner of the defined area. The ninth character is the box fill character. cEXP2 sets the border color of the box. It uses the same colors described for the SetColor( ) function. The optional Color argument provides the means to set the colors for that specific say or get. It uses the same color setup described in chapter 2 for the SetColor( ) function. EXAMPLE:

```
cFILL = chr(218) + chr(196) + chr(191) + chr(179) + chr(217) + ;
      chr(196) + chr(192) + chr(179) + chr(176)

@ 01, 01, 23, 79 box cFILL
```

@ 01, 01, 23, 79 box cFILL
@ <nEXP1>, <nEXP2> Clear [To <nEXP3>, <nEXP4>]

Summer 87, 5.0, 5.01

5.01 uses a combination of the Scroll( ) and SetPos( ) functions to accomplish this command. Depending on the effect you want to create, it might be more efficient to use a combination of these two commands rather than using @...Clear.

Use this command to clear a rectangular area of a display. nEXP1 contains the starting row and nEXP2 contains the starting column. If you don't specify the optional To argument, Clipper clears a single line starting at nEXP2 and going to column 79. nEXP3 specifies the ending row and nEXP4 specifies the ending column. This command removes windows of information from the display. EXAMPLE:

```
* Clear the upper left corner of the display.

@ 01, 01 clear to 15, 24
```

@ 01, 01 clear to 15, 24
@ <nEXP1>, <nEXP2> Prompt <cEXP1> [Message <cEXP2>]

Summer 87, 5.0, 5.01

Starting with version 5.0, the number of prompts has been increased from 32 to 4,096.

Create menus using this command in conjunction with the MENU TO command. nEXP1 contains the prompt row, nEXP2 contains the prompt column, and cEXP1 contains the menu prompt. Specify an optional help message using the Message argument. If you specify this argument, Clipper displays the cEXP2 associated with the currently highlighted prompt. Set the message location using the Set Message To command. EXAMPLE:

```
@ 01, 20 prompt "Display Database"
@ 02, 20 prompt "Print Database"
@ 03, 20 prompt "Exit Program"
menu to SELECTION
```

@ <nEXP1>, <nEXP2> [Say <EXP> [Picture <cEXP1>]] [Color <cEXP2>] [Get <VARIABLE> [Picture <cEXP3>] [Color <cEXP4>] [When <lEXP>] [Range <nEXP3>, <nEXP4>] [Valid <lEXP>]]

Summer 87, 5.0, 5.01

The When clause was added in version 5.0 and the Color clause in 5.01. The Valid and Range clauses are mutually exclusive in versions 5.0 and above. 5.0 and 5.01 handle incorrect date entry differently than Sum-

mer 87. They don't restore the original date as did Summer 87. Instead, the newer versions home the cursor in the get field and allow the user to edit the incorrect date. Pressing Escape in a get field restores the original value to the field and redisplays the value on-screen. Summer 87 doesn't redisplay the value. The 5.0 and 5.01 versions also clip any gets or says that extend past the end of line rather than continuing them on the next line of the screen. Most of these changes are due to the fact that both versions 5.0 and 5.01 provide enhanced get handling through internal routines. Using the get object provides additional flexibility over the get command at the expense of increased complexity. Chapter 4 describes get objects in detail. In addition, both versions use two new functions, DevOut( ) and DevPos( ), to handle says. These functions are not documented in the 5.0 manual, but you can use DevOut( ) to perform any full-screen display task.

This command performs three separate and independent functions. The first function positions the cursor. nEXP1 is the cursor row and nEXP2 is the cursor column. Using the optional Say argument displays EXP at the specified location. Using the optional Get argument obtains keyboard input up to the length of VARIABLE at the specified location. In both cases, the optional Picture argument formats the input or output. TABLE 1-1 provides a listing of the legal picture functions and templates. A function is a shorthand method of formatting the picture string. Precede each picture function with the @ symbol. A template specifies the format of each character separately. The optional Color argument provides the means to set the colors for that specific say or get. It uses the same color setup described in chapter 2 for the SetColor( ) function. The optional When argument allows you to display the say and get portions of the command, but does not allow the user to enter information unless lEXP is satisfied. The optional Range argument determines the range of numbers Clipper accepts for numeric input. nEXP3 determines the lower limit and nEXP4 determines the upper limit. The optional Valid argument determines legal input criteria. Clipper accepts input that results in a true condition. This allows you to create UDFs for processing data input by the user. However, at this level of complexity it's easier to use a get object in place of the @...Get command when using version 5.0 or above. EXAMPLE:

```
@ 15, 15 say "Enter your age: " get AGE picture '999' range 5, 100
```

@ <nEXP1>, <nEXP2>
To <nEXP3>, <nEXP4> [Double] [Color <cEXP>]
Summer 87, 5.0, 5.01
5.01 provides a new function, DispBox( ), in place of this command. The DispBox( ) function provides greater flexibility and is easier to use than the @...Box command. The Color clause was added in version 5.01.