# XMENU

# XMENU/REXX Interface User's Guide and Reference

**Version 2 Release 3**
**October 1990**

Computer Associates™

030510510501

# Preface

The *XMENU/REXX Interface User's Guide and Reference* contains information for use with the Relay Technology product XMENU.

## Audience

This manual is intended for application programmers who design and implement 327*x* full-screen applications. The *XMENU/REXX Interface User's Guide and Reference* describes how to use an XMENU menu in EXEC applications: how to control and fine-tune how a menu is displayed, how to transfer data back and forth between the menu and the application, how to check user input to ensure it is valid, how to display error messages and HELP menus, how to create and use windows, and so on.

The *XMENU/REXX Interface User's Guide and Reference* also includes a complete reference section for the MENUEXEC utility, which allows you to use XMENU menus in EXECs, and for other XMENU/REXX Interface utilities you can use in your EXEC applications.

A basic understanding of CMS EXEC languages, principally REXX, is assumed for anyone using this manual to develop applications. If you have never worked with the XMENU/REXX Interface, a good place to start is with *XMENU in Minutes: A guide to using menus with EXECs*. This quick tutorial will give you some helpful, basic information on and experience with the XMENU/REXX Interface (MENUEXEC).

## How this manual is organized

This manual contains the following parts, chapters, and appendixes:

- Part 1, The XMENU/REXX Interface User's Guide

  Chapter 1, "Introduction to the XMENU/REXX Interface" on page 3, provides a general overview of the XMENU/REXX Interface and what it can do in your EXEC.

  Chapter 2, "Using menus with an EXEC" on page 5, begins with a general description of the MENUEXEC command, its options and subcommands, as well as the role EXEC variables play in transferring data to and from the menu and your application. It then discusses and provides examples of the more commonly-used options and subcommands, grouped by function.

- Part 2, The XMENU/REXX Interface Utilities Reference

  Chapter 3, "MENUEXEC" on page 41, details the command syntax, options, subcommands, messages and return codes for MENUEXEC, the XMENU/REXX Interface.

  Chapter 4, "MENUCTRL" on page 73, explains the use of and command syntax for the MENUCTRL utility, which allows you to move data to and from MENUCTRL files and menu variables.

Chapter 5, "MENUEXAM" on page 75, explains the use of and command syntax for the MENUEXAM utility, which allows the creation of dynamically-sized menus for use with MENUEXAM.

Chapter 6, "KOLVSUB" on page 79, details the command format for the utility and describes how to generate forms, mailing labels, JCL, and so on, based on input from a skeleton input file.

Appendix A, "Sample XMENU/REXX programs" on page 81, includes many helpful XMENU/REXX sample programs.

## Other manuals you should have

In addition to this manual, you should also have the following XMENU documents:

- *XMENU in Minutes: A guide to using menus with EXECs*
- *XMENU Utilities Reference*

You should also have the reference manuals for any CMS EXEC language you are using.

# Contents

# Part 1. The XMENU/REXX Interface User's Guide

This section provides a complete guide for using MENUEXEC, the XMENU/REXX Interface. Here's a "road map" of the major topics discussed in this section:

# Chapter 1. Introduction to the XMENU/REXX Interface

This chapter introduces the MENUEXEC utility, which enables your EXEC-language application to use XMENU menus, and introduces the other XMENU/REXX Interface utilities that can be used with your EXEC-language applications to perform a variety of other related functions.

## Overview

After you have designed your application and created the required menus, you need to write a program that will do two basic things:

- Perform each function of your application
- Use the menus you created for your application

We are not concerned in this manual with how to write an EXEC. We are concerned, however, with successfully integrating XMENU menus into your full-screen applications.

Most EXEC applications that use XMENU menus can be thought of as consisting of code to perform the following functions:

- Move data to the menu, if required

  For example, on a Read Mail/Reply menu you could move the userid of a person who sent a mail message into the **Reply:** field. Then, the person reading mail can reply to the sender without having to type the sender's userid.

- Display the menu

  You can display a menu on the screen and control such things as whether an alarm sounds when the menu is displayed, where the cursor is positioned, and so on.

- Move user input from the menu to the application

  You can move the data entered by users and information about the status of the menu from the menu into the application. For example, your application can receive data entered into fields, information about the position of the cursor, and the interrupt-generating key that was pressed.

- Validate user input and display error messages and HELP menus

  You can set a variety of different requirements for a field to ensure that only correct data is accepted. You can also display error messages to help users correct invalid data.

- Display windows associated with the menu, if required

  You can associate one or more windows with a menu and then display them on top of the menu, one at a time or all at once, partially overlapping, etc.

With the MENUEXEC utility of the XMENU/REXX Interface, your EXEC-language application can perform these functions and more. In addition to providing a flexible,

full-function EXEC language interface in MENUEXEC, the XMENU/REXX Interface comes complete with these utilities:

**MENUEXEC** Allows you to use XMENU menus under REXX, EXEC 2, and EXEC.

**MENUCTRL** Allows you to move data from MENUCTRL files to EXEC variables and back again.

**MENUEXAM** Allows you to create a dynamic menu for use by MENUEXEC.

**KOLVSUB** Allows you substitute the contents of EXEC variables into skeleton files to produce customized output, such as form letters, JCL, or mailing labels.

# How MENUEXEC works

In its most simple invocation, shown below, MENUEXEC can be issued in your application EXEC to enable output (display the menu) and input (receive entered data) to be shared between XMENU and your EXEC.

```
"MENUEXEC menuname"
```

MENUEXEC command options, subcommands, and menu data expand on this basic capability and perform a variety of additional functions. These basic and additional functions can be sent to and received by MENUEXEC from several sources:

1. **EXEC and MENUEXEC variables**—XMENU field names and their EXEC variable names are one and the same; so EXEC variables can be easily used to move data directly to and from menu fields.

   Additionally, MENUEXEC creates a set of useful EXEC variables that you can use in your application to determine which PF key was pressed, where the cursor was located when an interrupt-generating key was pressed, and so on.

2. **MENUEXEC command options**—These specify global options affecting the menu display, for example, whether to sound an alarm when the menu is displayed, whether to clear the screen before you display the menu, and whether to have PF13-24 and PF25-36 act the same as, or be "mapped to" PF1-12.

3. **MENUEXEC subcommands**—These allow dynamic and extensive changes to be made to menus. The kinds of changes you can make include adding and deleting fields, protecting and unprotecting fields, and displaying field-specific HELP messages.

# Chapter 2. Using menus with an EXEC

We begin this chapter with an explanation of using EXEC variables with MENUEXEC and continue with a discussion of how to include MENUEXEC command line options and subcommands in your EXEC. Then, the most frequently-used MENUEXEC options and subcommands are described, with a sample of each presented. These descriptions are grouped, with functionally-related options and subcommands presented together.

## Using EXEC variables with MENUEXEC

If a user fills in or changes a named field on a menu, an EXEC variable with the same name as the field is either created or modified. Hence, after user input, each variable contains the contents of the identically-named menu field that the user modified.

The concept that XMENU field names are directly associated with and are used as variable names in your EXEC programs is an important one. For example, the contents of a field named "FIELD1" on an XMENU menu are read or changed by your EXEC program as the variable FIELD1. When MENUEXEC is called, it checks each named field on the menu before displaying it. If a variable matching a field name exists, MENUEXEC replaces the contents of the field with the contents of the matching variable. If no variable exists, the contents of the field are not changed. This concept not only simplifies your program but allows menus to be re-designed and fields to be moved without requiring a change to your program.

Supported variable/field names are alphanumeric with an alphabetic first character (periods are also permitted), which do not have special or pre-defined uses; for example, control words like &INDEX or &RETCODE and the MENUEXEC-defined variables that are introduced below cannot be used.

Be sure not to assign menu fields with the names of EXEC-language control words, such as SELECT. If you do, the control words will no longer work as defined in the EXEC specifications, and the remainder of the EXEC execution will be unpredictable. This is because EXEC languages perform symbol substitution from the user variable list before checking for special variable names.

MENUEXEC also creates EXEC variables that you can use to move data to and from menu fields. After MENUEXEC exits, these variables can be used to help you determine what the user did on the menu (for example, what key was pressed and the location of the cursor when the user completed input). Here is a list of the MENUEXEC-created variable names:

- ACTION
- CURCOL
- CURFNAM
- CURFOFF
- CURLINE
- CURSOR
- CURSTR
- CURWRD

- LGHTPEN
- MENU
- PFK
- PFKSTR
- SMSG
- SMSGUSR
- VARCHNG

For a complete description of what each variable returns to your EXEC, see "MENUEXEC EXEC variables" on page 63.

All EXEC variables that are read and modified by MENUEXEC are taken from the last EXEC level called before MENUEXEC. For example, if EXEC "A" calls EXEC "B", which calls a program that calls MENUEXEC, EXEC variables from EXEC "B" are used; those from EXEC "A" are ignored.

There may be some editing of user input:

- If the program calling MENUEXEC is written in the original CMS EXEC language, the user's input is truncated to eight characters.
- For EXEC 2 processing, the maximum supported field size is 255 characters.
- There is no variable length limit in REXX.
- If a user clears a field (uses the ERASE EOF key to clear the entire field), the corresponding variable is set to a null string. If called from REXX with the MENUEXEC DROP option, it no longer exists to REXX (see "MENUEXEC options" on page 42 for information on DROP).

Because of the way variables are used, if an EXEC calls the same menu twice, data that the user entered into fields on the first menu call will be transferred to the menu in the second call because the data was passed by the EXEC variables. Therefore, for the duration of an EXEC, the menu fields have "memory" of the last user input. This also applies to different menus called from the same EXEC, if these menus share the same field names.

Field names and their same-named EXEC variables can also be used, if desired, to simulate user input, or, in other words, to "prime" menus with field entries. One way to do this is to have data that is associated with field names transferred from a MENUCTRL file to the named menu fields. (see "Using MENUEXEC subcommands" on page 7 for an explanation and example of using a MENUCTRL file to "prime" a menu). Another way to simulate user input is to manipulate the fields' Modified Data Tags (or MDTs). This is described in "MDT and NOMDTRS—Supplying default values for menu fields" on page 20.

# Using MENUEXEC options

You will use the MENUEXEC command in your EXEC any time you want to display a menu. The options you choose to add to the MENUEXEC command line affect the specified menu's use and display in a *global* way; that is, MENUEXEC options set conditions affecting the whole menu.

MENUEXEC options are specified in the same way as CMS command options: they follow an open parenthesis and can be used in any order:

**Sample MENUEXEC call with options**

```
"MENUEXEC menuname ( CLEAR MAP EMSG PA1"
```

A discussion of these and other commonly-used MENUEXEC options is presented later in this chapter.

# Using MENUEXEC subcommands

MENUEXEC subcommands are used to make specific, dynamic, and more involved changes to menus or parts of menus than could be effected with MENUEXEC command options. Many subcommands affect specific fields, as opposed to options, which affect the entire menu.

There are three ways to present subcommands to MENUEXEC: they can be passed in the CMS stack, via the CMS SUBCOM interface, or in MENUCTRL files. Each method is discussed below and a sample for each is presented. A discussion of each MENUEXEC option and subcommand used in the following examples is presented later in this chapter.

Because the SUBCOM interface is the most flexible and direct method for passing subcommands, we selected that method for all the sample invocations in this chapter. Circumstances that may affect specific applications, personal preferences, or individual site standards will affect which method or methods you use.

## The stack interface

The stack interface to MENUEXEC is invoked by having "MENUCMDS" as the first subcommand in the stack when MENUEXEC is called. MENUEXEC subcommands are then read from the stack until the "MENUEND" subcommand is encountered or when the stack is empty.

**Sample MENUEXEC call using the CMS stack to pass subcommands**

```
/* Simple EXEC */
queue "MENUCMDS"
queue "CHANGE FIELD1 BLUE REVERSE"
queue "CHANGE FIELD2 RED BLINK"
queue "REQUIRE FIELD1 ALPHA"
queue "MENUEND"
"MENUEXEC MENU1 ( CLEAR MAP EMSG"
```

This sample includes subcommands that would change the attributes of FIELD1 and FIELD2 before the menu is displayed to make them BLUE REVERSE and RED BLINK, respectively, and would verify, or require, that any user input into FIELD1 was alphabetic.

## The SUBCMDS option to enable the SUBCOM interface

The most flexible way to invoke subcommands is via the SUBCOM interface. The SUBCMDS ***option*** will process MENUEXEC subcommands using the CMS SUBCOM interface. In this way, subcommand return codes can be processed individually and subcommand execution can be made conditionally. When the SUBCMDS option is used, the stack is not checked.

Whenever you want to pass subcommands for a menu display, include the SUBCMDS option on the MENUEXEC command line. Then, after the MENUEXEC call, between `address MENUCMDS` and `"MENUEND"`, enter the desired MENUEXEC subcommands.

**Sample MENUEXEC call using the SUBCMDS option**

```
"MENUEXEC MENU1 ( CLEAR MAP EMSG SUBCMDS"
address MENUCMDS
"CHANGE FIELD1 BLUE REVERSE"
"CHANGE FIELD2 RED BLINK"
"REQUIRE FIELD1 ALPHA"
"MENUEND"
address CMS
```

The MENUCMDS environment does not have to follow the MENUEXEC call immediately, as in this example; it can be placed anywhere in your program logic after the MENUEXEC call, and if desired, can be invoked conditionally.

When the MENUEND command is received, MENUEXEC ends the subcommand environment and continues processing. If a return is made to CMS command level without MENUEXEC first receiving the MENUEND subcommand, MENUEXEC automatically terminates the MENUCMDS subcommand environment without any subsequent display.

## MENUCTRL files

You can create a file with the filetype MENUCTRL that includes a list of MENUEXEC subcommands and field data assignments. This file can be used as read/only or read/write file. To process a MENUCTRL file, include the MENUEXEC FILE *filename* ***option*** on the MENUEXEC command line, as shown below:

**Sample invocation**

```
"MENUEXEC MENU1 ( CLEAR MAP EMSG FILE CNTRLNM"
```

In this example, the file CNTRLNM MENUCTRL will be processed and its contents will be used to update the menu, MENU1.

**Read/Only MENUCTRL files:** An example of the contents of a MENUCTRL file that is being used in read/only mode follows:

**Sample MENUCTRL R/O file**

```
MENUCMDS
CHANGE FIELD1 BLUE REVERSE
CHANGE FIELD2 RED BLINK
REQUIRE FIELD1 ALPHA
MENUEND
```

**Read/Write MENUCTRL files:** Variable data can also be written to MENUCTRL files. As described in "Using EXEC variables with MENUEXEC" on page 5, MENUEXEC provides "user-input memory" to an EXEC. In normal MENUEXEC processing, if the same menu is called more than once from the same EXEC, the user input from the first display is placed into EXEC variables, and is re-displayed on following menu displays (assuming the variable was not changed between MENUEXEC calls).

A MENUCTRL data file can provide the same sort of "user-input memory", but by capturing user input in a CMS file, the data can be referred to across IPLs, from several different EXECs, and from several different menus, if desired.

The MENUCTRL utility, described in more detail in Chapter 4, "MENUCTRL" on page 73, can be invoked to write variable data to a MENUCTRL file. A sample invocation of the MENUCTRL utility that shows how to capture user input into FIELD1 on MENU1 into the file CNTRLNM MENUCTRL follows:

**Sample MENUCTRL invocation**

```
"MENUEXEC MENU1 ( CLEAR MAP EMSG FILE CNTRLNM"
"MENUCTRL UPDTFILE CNTRLNM FIELD1"
```

If a user typed "BLIVET" into FIELD1 when MENU1 was displayed, CNTRLNM MENUCTRL would be updated as follows:

**Sample MENUCTRL R/W file**

```
MENUCMDS
CHANGE FIELD1 BLUE REVERSE
CHANGE FIELD2 RED BLINK
REQUIRE FIELD1 ALPHA
FIELD1 'BLIVET'
MENUEND
```

There is additional flexibility with MENUCTRL files when the FILE ***subcommand*** is used. This subcommand can be used to specify a MENUCTRL file to process, which can result in nested MENUCTRL file processing when issued from within a MENUCTRL file.

If you want to write variable data to a MENUCTRL file, investigate the use of the MENUCTRL utility by referring to Chapter 4, "MENUCTRL" on page 73.

# Displaying a menu

You can use a variety of options to control how a menu is displayed. For example, you can use MENUEXEC command line options to perform the following functions:

- Clear the screen before a menu is displayed. When the first display of a menu involves a transition from line mode to full-screen mode, you can prevent the screen from entering the MORE... condition on the first MENUEXEC call.
- Sound the terminal alarm when a menu is displayed. Sounding the terminal alarm is particularly useful when the user makes an error and you want to redisplay the menu.
- Display a menu without waiting for user input. You can show the user menus that require no input, and exit immediately without waiting for the user to press a key.
- Don't unlock the keyboard. This is particularly useful for showing application title screens or other screens where user input is not desired.
- Clear a menu after a specified time period if a user does not enter anything within that time period. You can clear a menu with sensitive information, for example, if a user does nothing in a set amount of time.
- Position the cursor when the menu is displayed. You can place the cursor in a specific location or field, and can override any previously set initial cursor position.
- Don't change the cursor position when a menu is displayed using the saved environment. You can write applications that modify the screen while the user is still typing.
- Move the cursor past protected fields. You can save the user time by having the cursor skip over protected fields that will not accept data entry.
- Map PF13 through PF24, and PF25 through PF36, to PF1 through PF12. If you only need to use 12 or fewer PF keys, you can have PF keys 13 through 24, and 25 through 36, returned as PF keys 1 through 12.

The MENUEXEC options associated with each of these functions are described below.

## CLEAR—Clearing the screen before a menu is displayed

When the first display of a menu involves a transition from normal screen mode (line mode) to full-screen mode, use the CLEAR option to prevent the MORE... condition on the first MENUEXEC call. Using CLEAR allows the user to see the menu without having to press PA2 or CLEAR.

Usually you should only use the CLEAR option the first time full-screen mode is entered. Refer to "Basic menu display program" on page 82, where an example is presented showing how to do this when one menu is repeatedly displayed in a loop. If you are displaying several menus from one or more EXECs, you could use the CLEAR option with each MENUEXEC call, but it would introduce additional processing and screen I/O.

**Sample invocation**

```
"MENUEXEC MENU1 ( CLEAR"
```

## ALARM—Sounding the terminal alarm when a menu is displayed

Use the ALARM option to sound the terminal alarm when a menu is displayed. This option is especially effective when used while redisplaying a menu after a user makes an input error.

**Sample invocation**

```
"MENUEXEC MENU1 ( ALARM"
```

## NOWAIT—Displaying a menu without waiting for user input

Use NOWAIT to display menus without waiting for user input. For example, you may want to display a menu that only gives information and requires no input. With NOWAIT, you could present a menu that lets a user know the status of certain processing operations, or the result of a previous operation.

When you use NOWAIT to show the user a menu, make sure you give the user enough time to read the menu. Any additional output sent to the terminal will remove the original menu.

**Sample invocation**

```
"MENUEXEC MENU1 ( NOWAIT"
```

## NOUNLOCK—Leaving the keyboard locked

As mentioned above, when you show the user a menu that only gives information and requires no input, you use NOWAIT. Use NOUNLOCK to tell XMENU not to unlock the keyboard. This is particularly useful when displaying application title screens.

When you use NOUNLOCK, the user can still use the RESET key to unlock the keyboard when the terminal is attached to certain 327*x* controllers.

**Sample invocation**

```
"MENUEXEC MENU1 ( NOUNLOCK"
```

## WAIT—Clearing a menu after a set time period without user input

You might want to clear a menu if the user does not enter anything within a certain time period. For example, a menu might display sensitive information that you do not want to remain on the screen. To clear a menu if a time period passes without user input, use WAIT *time*.

Specify the length of time to display the menu in 1/10ths of a second. If there is no user input within the time period, control is returned to the EXEC and the variable PFK is set to "TIMER".

If you use WAIT on a non-XA system, the virtual machine must have ECMODE set to ON. You can set ECMODE to ON using the CP command `SET ECMODE ON`, or by an `OPTION ECMODE` statement in the user's CP directory entry.

**Sample invocation**

```
"MENUEXEC MENU1 ( WAIT 20"
```

## CURSOR—Positioning the cursor when a menu is displayed

Use CURSOR to tell XMENU where to position the cursor when a menu is displayed. If you do not use the CURSOR option, XMENU places the cursor on the position specified when the menu was created.

You can use the CURSOR *fieldname* option with either:

- A symbolic field name, to place the cursor on a particular field
- A numeric offset from the upper left corner of the menu (position zero), to place the cursor on a particular spot on the menu

**Sample invocation**

```
"MENUEXEC MENU1 ( CURSOR FIELD1"
```

## NOCURSOR—Leaving the cursor alone when a menu is displayed

Use NOCURSOR with the SAVE option (described in "Using the saved menu environment" on page 14) to tell XMENU not to modify the cursor position when a menu is written.

With NOCURSOR you can write applications that modify the screen while the user is still typing, like performance monitors might. Because XMENU does not move the cursor, the user does not see the cursor move back to a previous location.

If you use NOCURSOR and the terminal screen has to be erased because you also used the CLEAR option, or because a non-full-screen message was displayed between menu displays, the cursor is positioned in the upper left corner of the screen.

**Sample invocation**

```
"MENUEXEC MENU1 ( SAVE NOCURSOR"
```

## SKIP—Moving the cursor past protected fields

When an application uses menus that contain a lot of fields, some protected and others requiring user input, you can help the user save time by having the cursor skip over protected fields, which are only informational and do not accept data entry anyway. Use the SKIP option to skip the cursor past protected fields.

**Sample invocation**

```
"MENUEXEC MENU1 ( SKIP"
```

### MAP—Mapping PF13 through PF36 to PF1 through PF12

If you only need to use 12 or fewer PF keys in your application, and you want to let the users use the most convenient key on their keyboards (no matter whether they have 12, 24, or 36 PF keys), you can use the MAP option. With this option, you can have PF keys 13 through 24, and PF keys 25 through 36, returned as PF keys 01 through 12.

The MAP option also lets you avoid having to run three checks in an EXEC for PF keys that perform the same function.

**Sample invocation**

```
"MENUEXEC MENU1 ( MAP"
```

## Retaining control upon interrupts

You can have MENUEXEC return control to the caller on any of these various interrupts:

- The PA1 key is pressed. You can return the PA1 key to the EXEC rather than placing the user in CP READ.
- A message is received. You can have control returned to the EXEC if an SMSG interrupt is received before the user enters any data on the menu.
- A card reader interrupt occurs. You can return control to the EXEC if a virtual card reader interrupt at address 00C is received before the user enters any data on the menu.
- An interrupt from a device occurs. You can return control to the EXEC if an interrupt is received from a specific device before the user enters any data on the menu.

The MENUEXEC options associated with each of these functions are described below.

### PA1—Trapping PA1

Use the PA1 option to keep the console from entering CP READ when the PA1 key is pressed. When you use the PA1 option and the user presses PA1, MENUEXEC will trap the key on input and return it to the user as PA1 in EXEC variable PFK.

**Sample invocation**

```
"MENUEXEC MENU1 ( PA1"
```

Because the PA1 option remains in effect only for the duration of the current menu display, you may want to ensure that users avoid entering CP READ for the duration of your EXEC execution. To do this, include the command `CP TERM BRKKEY NONE` in your EXEC.

### SMSG—Return control to the EXEC upon message interrupts

Use SMSG to return control to the EXEC if an SMSG interrupt is received before the user enters any data on the menu. If an SMSG interrupt is received, the variable PFK is set to SMSG, variable SMSG is set to the received message string, and SMSGUSR is set to the userid of the message sender.

**Sample invocation**

```
"MENUEXEC MENU1 ( SMSG"
```

### RDR—Return control to the EXEC upon card reader interrupts

Use RDR to return control to the EXEC if a virtual card reader interrupt at address 00C is received before the user enters any data on the menu. If a virtual card reader interrupt is received, the variable PFK is set to RDR.

**Sample invocation**

```
"MENUEXEC MENU1 ( RDR"
```

### DEV xxx—Return control to the EXEC upon device interrupts

You can use DEV *xxx* to return control to the EXEC if an interrupt is received from a specific device before the user enters any data on the menu. The device address is specified as *xxx*. If an interrupt is received, the variable PFK is set to DEVICE.

**Sample invocation**

```
"MENUEXEC MENU1 ( DEV 00C"
```

## Using the saved menu environment

MENUEXEC offers a saved menu environment that improves performance if your application uses one or more menus repeatedly.

- You can keep a copy of a frequently-displayed menu in storage by saving a menu for a later MENUEXEC call.
- Because MENUEXEC can't know if another application has changed the menu since its last display in the saved environment, you can reshow the menu with expected results.
- You can delete a menu from storage when it will no longer be called by purging the saved menu at the end of the MENUEXEC call.

The MENUEXEC options associated with each of these functions are described below.

## SAVE—Saving a displayed menu for a later MENUEXEC call

If your application uses one or more menus repeatedly, you can improve performance by keeping copies of the menus in storage. When a subsequent MENUEXEC call redisplays a saved menu, no file access is necessary because you are reusing a menu in storage and not loading a fresh copy from disk.

Use SAVE to keep a copy of a displayed menu in storage. When you use SAVE, EXEC variable MENU is given a unique value for passing the saved menu to subsequent MENUEXEC calls that also use SAVE.

**Sample invocation**

```
menu = "MENU1"
"MENUEXEC" menu "( SAVE"        (first call to MENUEXEC)

⋮

"MENUEXEC" menu "( SAVE"        (later calls to MENUEXEC)
```

A menu passed between MENUEXEC calls differs from a menu loaded from disk on each MENUEXEC call. For more information see "Some points to consider when using SAVE." You should also refer to "RESHOW—Redisplaying an entire menu and not only the changed fields" on page 16.

Follow these steps to use SAVE:

**Basic steps to using SAVE**

1. Assign the menuname to be saved to the EXEC variable MENU.

   For example: `menu = "MENU1"`

2. Call MENUEXEC, passing the EXEC variable MENU as the menuname, and passing SAVE as one of the MENUEXEC command line options.

   For example: `"MENUEXEC" menu "(SAVE"`

   MENUEXEC will change the contents of the EXEC variable MENU to the internal name assigned to the in-storage copy of the menu.

3. Code subsequent calls to MENUEXEC the same way as in Step 2 (or loop to Step 2). **Each** of these subsequent calls **must** use SAVE.

To operate with several saved menus, follow the steps above, but save the contents of the EXEC variable MENU when changing from one saved menu to another.

XMENU windows implicitly use the saved menu environment; this is how previous menus are displayed in background viewports. See "Using windows" on page 32 for more information on using windows.

**Some points to consider when using SAVE:** Menus saved in storage function differently than menus loaded from disk. Keep in mind the following points when you use SAVE:

- Do not confuse the menus saved in storage with SAVE with the in-storage menus loaded by XMENU's XMENUINS utility. You can use XMENUINS to permanently load frequently-used menus into storage. Refer to the *XMENU Subroutine Library Reference* for more information on this utility.
- Remember that saved menus are not refreshed when they are redisplayed. Hence, any data placed into a menu on a previous call, either from user input or from data moved from a MENUCTRL file or EXEC variable, becomes the default data on the current call **unless** a MENUCTRL file or EXEC variable moves data to the menu.

  For example, if on the first display of the menu the user changes a blank field to "ABCDE", and the second call to redisplay the menu does not change the variable associated with this field, then the field will still contain the string "ABCDE".

  In addition, keep in mind the following points:

  - If, on the first display of the menu, field attributes were changed with a CHANGE subcommand, those attributes remain changed when the menu is redisplayed.
  - If a prior MENUEXEC call passed MDT or SKIP as an option, the fields modified by these options remain modified in subsequent calls.
  - If a prior MENUEXEC call added or deleted fields from the menu, those fields remain added or deleted on subsequent displays of the menu.
- You should respecify all other MENUEXEC options and subcommands for each menu redisplay, including field validation (REQUIRE).
- You can remove a menu from storage after its last redisplay by using the MENUEXEC PURGE option.
- If you return to the CMS command environment, or if an abnormal termination (ABEND) occurs in your application, all saved menus are purged.
- If you mix calls of saved MENUEXEC and non-saved MENUEXEC, you should pass RESHOW (see the next section) as a MENUEXEC command line option the next time you call the saved environment. This is also true if any other full-screen display is used between saved MENUEXEC calls (including any XMENU application). This must be done because MENUEXEC doesn't know if a foreign application has changed the screen.

  Typical symptoms of MENUEXEC (or other XMENU applications) not knowing that a foreign application updated the screen include either garbled screens and I/O errors (if the previous application changed the terminal to a smaller size).

## RESHOW—Redisplaying an entire menu and not only the changed fields

If a full-screen write is performed between MENUEXEC SAVE calls (including a call to an XMENU application or to MENUEXEC without SAVE), XMENU may not know that the contents of the screen were changed. RESHOW, which you use with SAVE, forces MENUEXEC to display the entire menu. RESHOW specifies that the entire menu should be displayed, not just those fields changed since the last display of the menu. You must use RESHOW if you call an application between two MENUEXEC SAVE calls. When you use the SAVE option, XMENU only sends changed fields to the screen on redisplayed menus. Hence, the second MENUEXEC SAVE call may only send part of the menu to the screen, which is a significant performance boost.

**Sample invocation**

```
"MENUEXEC" menu "( SAVE RESHOW"
```

### PURGE—Deleting a menu from virtual storage at the end of a call

Use PURGE with SAVE to specify that at the end of a MENUEXEC call, a menu saved in storage (with SAVE) should be purged from storage.

**Sample invocation**

```
"MENUEXEC" menu "( SAVE PURGE"
```

## Moving data between menus and your application

As described in "Using EXEC variables with MENUEXEC" on page 5, field name variables are both read and set by MENUEXEC. On input, the contents of the EXEC variable replace the contents of the identically named menu field. After user input, each variable contains the contents of the identically named menu field that the user modified or whose Modified Data Tag (MDT) was set.

EXEC variables relating to menu fields are ignored if either the MENUEXEC command line option NOXVARS or CTRLVARS is set or if the NOXVARS subcommand was issued for this menu field. Refer to "MENUEXEC options" on page 42 for more information on these.

The options and subcommands described in this section allow additional flexibility when data is being read or written by MENUEXEC. The MENUEXEC command line options and subcommands we discuss here perform these functions:

- Display a menu and ignore user input
- Create a list of fields changed by the user
- Add a prefix string to EXEC variables
- Ignore subcommands that refer to non-existent fields
- Ignore certain keys when no function is assigned to them
- Don't validate fields when certain keys are pressed
- Don't update MENUCTRL files with data when certain keys are pressed
- "Prime" menus with simulated user input without using MENUCTRL files

### TEST—Displaying a menu and ignoring user input

Use TEST to look at a menu and check its input areas. When you use TEST, the menu is displayed and input is prompted for, but the input is ignored. In addition, no EXEC variables, stacked data, or updated file data are inspected or modified.

When you use TEST, you implicitly use CLEAR and EMSG. When a menu is displayed using TEST, any interrupt-producing key will exit MENUEXEC.

**Sample invocation**

```
"MENUEXEC MENU1 ( TEST"
```

## LISTVARS—Listing only fields changed by the user

Use LISTVARS to tell MENUEXEC to place the names of the fields changed by the user into an EXEC variable array named VARCHNG (refer to "MENUEXEC EXEC variables" on page 63 for more information on VARCHNG).

VARCHNG.1 contains the first changed field name, VARCHNG.2 the second, and so on.

VARCHNG.0 contains the number of changed fields. You can use VARCHNG.0 as a loop counter, and you can use the VARCHNG array to restrict processing to changed fields.

The order of the fields in the array is not predictable.

**Sample invocation**

```
"MENUEXEC MENU1 ( LISTVARS"
```

## PREFIX—Prefixing a string to EXEC variables

Use PREFIX *string* to tell MENUEXEC to retrieve and store all EXEC variables with the prefix variable *string*.

For example, if you specified `PREFIX A.`, then menu field X would be filled from EXEC variable A.X and, if modified, resaved into A.X.

Special variables are also prefixed. In the example above, the key pressed would be returned as A.PFK.

If you want a period between the prefix string and the variable name, you must specify it in the prefix string. The prefix string is limited to eight characters.

Use PREFIX to separately store and control information from different menus. You can also use PREFIX to simulate longer field names.

**Sample invocation**

```
"MENUEXEC MENU1 ( PREFIX FIRST."
```

## IGNSCFN—Continuing when a command refers to a non-existent field

You might want to create a single MENUCTRL file (containing MENUEXEC subcommands) that can be shared by several menus, even when these menus might not share exactly the same field names. You can avoid having to create several slightly different MENUCTRL files by using IGNSCFN. IGNSCFN tells MENUEXEC to ignore subcommands that refer to non-existent fields instead of causing an error.

**Sample invocation**

```
"MENUEXEC MENU1 ( SUBCMDS IGNSCFN"
```

## IGNORE subcommand—Ignoring the use of specific 327x keys

You can tell MENUEXEC to ignore certain keys that might be pressed by users by using the IGNORE *key1 key2 keyn* subcommand. For example, your application may not assign a function to every PF key. With the IGNORE subcommand, you can tell MENUEXEC to ignore one or more keys that perform no function.

Use IGNORE for any interrupt-generating key, such as a PF key or PA key, ENTER, SYSREQ, CLEAR, etc. You can also have specified interrupt types ignored, such as SMSG, DEVICE, etc.

**Sample invocation**

```
"MENUEXEC MENU1 ( SUBCMDS"
address MENUCMDS
"IGNORE CLEAR PF02 PF12"
"MENUEND"
address CMS
```

## IGNREQ subcommand—Skipping field validation when specific keys are pressed

You can tell MENUEXEC not to validate data entered by a user if certain keys are pressed. For example, a user might enter some data and then decide to exit from the application. You would not want to waste processing time to check that field requirements were met for any data the user may have entered. With IGNREQ *key1 key2 keyn* you can tell MENUEXEC to ignore user data if any of the specified keys are pressed. See "Validating data" on page 21 for more information on setting field requirements.

**Sample invocation**

```
"MENUEXEC MENU1 ( SUBCMDS"
address MENUCMDS
"IGNREQ PF03"
"REQUIRE FIELD1 ALPHA"
"MENUEND"
address CMS
```

## NOUPDPFK subcommand—Skipping file updating when specific keys are pressed

You can tell MENUEXEC not to update any MENUCTRL files with data a user enters or changes if the user presses specified keys. For example, a user might enter some data and then decide to exit from the application. You would not want to waste processing time to update files with any data the user might have changed. With NOUPDPFK *key1 key2 keyn*, MENUEXEC will not update files if any of the specified keys are pressed. For more information on using MENUCTRL files, see "MENUCTRL files" on page 8.

**Sample invocation**

```
"MENUEXEC MENU1 ( SUBCMDS FILE FILE1"
address MENUCMDS
"NOUPDPFK PF03"
"MENUEND"
address CMS
```

## MDT and NOMDTRS—Supplying default values for menu fields

You can simulate user input and force data to be returned to your application as if the user actually entered data by using the MDT option.

Every field on a menu has a Modified Data Tag (MDT) which, by default, is set OFF. The MDT is set ON whenever a user enters data into the field (or when a user clears a field with the ERASE EOF key). MDTs help reduce overhead by returning the minimum amount of screen data to your application.

Protected fields have MDTs, and they—as well as unprotected fields—are set ON by your application with the MDT option. When a user presses ENTER, all fields with the MDT set ON—both protected and unprotected—are transferred to your application as changed fields.

It is sometimes useful to force data back to your application even if the user does not enter any. If you want to do this, follow these steps:

**Using MDTs to supply default values**

1. Transfer data from a MENUCTRL file to EXEC variables that have the same names as the menu fields.
2. Set the MDT to ON for each unprotected field by including the MDT option on the MENUEXEC command line.

   When the user presses ENTER, all fields (including protected ones) having MDTs set are returned to the application as user input.

As we noted earlier, if a menu is called twice from the same EXEC, data in fields that had MDT set ON by the first menu call will be placed into the menu on the second call because of the way data is passed by EXEC variables. So, for the duration of the EXEC, the data entered on the previous display of the menu will be passed to subsequent displays, in effect giving the menu "memory" of the field contents **without** using a MENUCTRL file.

You can have preset application defaults for menu fields tailored for different users.

You can turn MDT OFF from the keyboard with the ERASE INPUT key, which clears all user input including MDTs.

**Sample invocation**

```
"MENUEXEC MENU1 ( MDT"
```

Use NOMDTRS to tell MENUEXEC that fields with MDT ON should **not** be reset (turned OFF) when the menu is redisplayed. This option is only useful when used with the SAVE option.

NOMDTRS lets you leave a previously modified field's MDT ON, so the user input areas will appear "primed", or as though a user just typed into the field. Use NOMDTRS to force reprocessing of all input, even if the user has not changed or entered any data.

**Sample invocation**

```
"MENUEXEC MENU1 ( SAVE NOMDTRS"
```

# Validating data

MENUEXEC has an extensive set of data validation functions that you can use to check data entered into menu fields. You can also write macros for more sophisticated validation requirements (see "REQUIRE macros—Validating user input" on page 27 for an introduction to REQUIRE EXECs, and "REQUIRE EXEC macro facility" on page 68, for details on creating and using REQUIRE EXEC files).

If data fails to fulfill a requirement, the menu is redisplayed and the cursor is placed on the invalid character.

## REQUIRE subcommand—Validating user input

The REQUIRE subcommand lets you define a set of data verification requirements for a field. The data a user enters into that field must meet the requirements; if it does not, MENUEXEC does not return to your EXEC or program. Instead, MENUEXEC redisplays the menu and positions the cursor on the incorrect character or string. If a FIELDMSG exists for this field, it is displayed in the error message field, which is defined by the EMSGFLD or ERRMSG subcommand. (For more details on these two subcommands, see "Displaying error messages and HELP menus" on page 28.)

Before requirement checking begins, the data is logically blank; nulls and blanks are suppressed from both ends. If requests for multiple field validations exist, they are checked in the order they occur in the subcommand stream. If multiple requirements exist for one field, and any of the requirement checks succeed, the field is considered to be validated.

The REQUIRE subcommand has a variety of requirement functions you can use:

- ALPHA
- ANYTHING
- DECIMAL
- ERASEEOF
- EXEC
- FLOAT
- HEX
- INTEGER
- NATIONAL
- NONBLANK
- NOTHING

- PATTERN
- STRING

Each of these requirement functions are described in detail below.

**ALPHA—Requiring a field to contain only alphabetic characters:** The field must only contain the characters "A" through "Z" (upper or lower case) or blanks.

**Sample REQUIRE ALPHA invocation**

```
"MENUEXEC MENU1 ( SUBCMDS"
address MENUCMDS
"REQUIRE FIELD1 ALPHA"
"MENUEND"
address CMS
/*
    Will accept: Meatloaf
                 SODA
                 Ice Cream

        But Not: 2na fish
                 $PROFILE
                 C3PO
*/
```

**ANYTHING—Accepting any user input in a field:** Any user input is accepted (including all blanks but excluding ERASE EOF).

**Sample REQUIRE ANYTHING invocation**

```
"MENUEXEC MENU1 ( SUBCMDS"
address MENUCMDS
"REQUIRE FIELD1 ANYTHING"
"MENUEND"
address CMS
/*
    Will accept: well, $%@#(*
                 anything $%&&*&¬
                 $#A=+ will work!!!!%

        But Not: ERASE EOF key
*/
```

**DECIMAL—Requiring a field to contain only the digits 0 through 9:** The field must only contain the digits "0" through "9". Optionally, specific numeric comparisons can be requested. If these exist, the field must match **any** equality **or all** inequalities. Only non-negative numbers are supported.

**Sample REQUIRE DECIMAL invocation**

```
"MENUEXEC MENU1 ( SUBCMDS"
address MENUCMDS
"REQUIRE FIELD1 DECIMAL > 0 < 101 = 999"
"MENUEND"
address CMS
/*
    Will accept: 1
                 10
                 100
                 999

        But Not: 101
                 900
                 -1 (only positive numbers are supported by DECIMAL)
*/
```

**ERASEEOF—Requiring a field to be cleared with the ERASE EOF key:** You can require that a field must have been cleared by pressing the ERASE EOF key.

This require type is meant to be used in conjunction with other require types. For example, you can specify that the field can be cleared, but if it is not, it must match other criteria.

**Sample REQUIRE ERASEEOF invocation**

```
"MENUEXEC MENU1 ( SUBCMDS"
address MENUCMDS
"REQUIRE FIELD1 ERASEEOF"
"MENUEND"
address CMS
/*
    Will accept: ERASE EOF key
*/
```

**EXEC—Using REQUIRE macros:** You can develop REQUIRE macros to perform specific requirement checking. For an introduction to REQUIRE macros, see "REQUIRE macros—Validating user input" on page 27, for detailed information, see "REQUIRE EXEC macro facility" on page 68, and for examples, see "Sample REQUIRE macro to check for YES or NO input" on page 101, and "Sample REQUIRE macro to check for a valid date" on page 103.

**FLOAT—Requiring a field to contain only the digits 0 through 9 with an optional sign, fractional part, or signed exponent:** The field must only contain the digits "0" through "9" with an optional leading sign, "+" or "-", an optional fractional part, and an optional signed exponent.

Optionally, specific floating point comparisons can be requested. If these exist, the field must match **any** equality **or all** inequalities. Floating point numbers are converted internally and comparisons are performed using System 370 double precision format. Values exceeding this format's mantissa or exponent capacity are considered to fail the REQUIRE criteria.

**Sample REQUIRE FLOAT invocation**

```
"MENUEXEC MENU1 ( SUBCMDS"
address MENUCMDS
"REQUIRE FIELD1 FLOAT  > -50.0 < 50.0 = 999.0"
"MENUEND"
address CMS
/*
    Will accept: -45.1
                 0
                 +999.
                 +40.888

        But Not: 1000.
                 100.
                 -100.00
*/
```

**HEX—Requiring a field to contain only hexadecimal values:** The field must only contain the digits "0" through "9" or the letters "A" through "F" (upper- or lowercase).

Specific numeric comparisons can be requested. If these exist, the field must match **any** equality **or all** inequalities. Only positive numbers are supported. If specific comparisons are included, their values must be given in hexadecimal.

**Sample REQUIRE HEX invocation**

```
"MENUEXEC MENU1 ( SUBCMDS"
address MENUCMDS
"REQUIRE FIELD1 HEX > 10"
"MENUEND"
address CMS
/*
    Will accept: 1F
                 2F
                 FF

        But Not: F
                 0
                 -1 (only positive numbers are supported by HEX)
*/
```

**INTEGER—Requiring a field to contain only the digits 0 through 9 with an optional sign:** The field must only contain the digits "0" through "9" with an optional leading sign, "+" or "-".

Optionally, specific numeric comparisons can be requested. If these exist, the field must match **any** equality **or all** inequalities. Positive and negative numbers are supported.

**Sample REQUIRE INTEGER invocation**

```
"MENUEXEC MENU1 ( SUBCMDS"
address MENUCMDS
"REQUIRE FIELD1 INTEGER > 100"
"MENUEND"
address CMS
/*
    Will accept: 101
                 +1051
                 0900

        But Not: 100
                 55
                 -1
*/
```

**NATIONAL—Requiring a field to contain only alphabetic characters or $, @, or #:** The field must only contain the characters "A" through "Z" (upper case) or the characters "$", "@", or "#". This form can be used to check for valid CMS file names.

**Sample REQUIRE NATIONAL invocation**

```
"MENUEXEC MENU1 ( SUBCMDS"
address MENUCMDS
"REQUIRE FIELD1 NATIONAL"
"MENUEND"
address CMS
/*
    Will accept: $PROFILE
                 XMENU@Relay
                 #OFITEMS

        But Not: *CAVITY
                 &USERID
                 ICE CREAM  (imbedded blanks are not allowed)
                 aa (only UPPERCASE characters are supported)
*/
```

**NONBLANK—Requiring any non-blank user input in a field:** Input into a field is required. Any non-blank, non-null (non-hexadecimal zero) input is required.

**Sample REQUIRE NONBLANK invocation**

```
"MENUEXEC MENU1 ( SUBCMDS"
address MENUCMDS
"REQUIRE FIELD1 NONBLANK"
"MENUEND"
address CMS
/*
    Will accept: Any non-blank, non-null input

        But Not: No input
*/
```

**NOTHING—Requiring a field to contain no user input:** This requirement is only met if **no** user input is entered at all. This form is used in combination with other requirement types to handle cases where no input is necessary, but if input is entered it must be in a specific form (specified by other REQUIRE subcommands for the same field).

**PATTERN—Requiring a field to match a specific pattern:** The field must match a supplied pattern string. The pattern string contains a character type for each character in the field. Possible character types are "A" for alphabetic characters, "C" for any character allowed, "D" for decimal digits, "N" for national characters, and "X" for hexadecimal digits. Any other character found in the pattern string must **exactly** match the character at its position in the field (case is significant).

The length of the field's data must **exactly** match the length of the pattern string (after blank and null suppression). For example, the pattern "'DDD-DD-DDDD'" would match valid United States Social Security numbers.

**Sample REQUIRE PATTERN invocation**

```
"MENUEXEC MENU1 ( SUBCMDS"
address MENUCMDS
"REQUIRE FIELD1 PATTERN '800-DDD-DDDD'"
"MENUEND"
address CMS
/*
    Will accept: 800-555-1212
                 800-233-6686

        But Not: 900-555-1212
                 800-DOC-TORS
                 1-800-555-1212
*/
```

**STRING—Requiring a field to match one of a set of specific strings:** The field must match one of a set of supplied strings. Each string supplied may be followed by a numeric value. This value, if specified, gives the minimum length the field must be to match the given string. For example, the string "'YES' 1" would match fields with "Y", "YE", or "YES", but not "YET" or "YESTERDAY".

**Sample REQUIRE STRING invocation**

```
"MENUEXEC MENU1 ( SUBCMDS"
address MENUCMDS
"REQUIRE FIELD1 STRING 'XMENU' 5 'UNKNOWN' 3 'operator' 4"
"MENUEND"
address CMS
/*
    Will accept: XMENU
                 UNK
                 UNKNOWN
                 operat

        But Not: op
                 OPERATOR
                 XMEN
                 unknown  (case is significant with STRING)
*/
```

## REQUIRE macros—Validating user input

MENUEXEC provides a facility that allows you to generate your own field requirement/validity checking using EXEC 2 or REXX macros. These files are written in REXX or EXEC 2 and have the filetype MENUEXEC. REQUIRE macros are sometimes referred to as REQUIRE EXECs, or REQUIRE EXEC macros.

After the user has entered input and pressed an interrupt-generating key, MENUEXEC calls any macros that you may have specified on the REQUIRE subcommand with the EXEC *execname* requirement:

**Sample REQUIRE EXEC macro invocation**

```
"MENUEXEC MENU1 ( SUBCMDS"
address MENUCMDS
"REQUIRE FIELD1 EXEC DATECHEK"
"MENUEND"
address CMS
/*
    Will process the REQUIRE macro DATECHEK MENUEXEC
    to validate FIELD1's contents.
*/
```

See "Sample REQUIRE macro to check for YES or NO input" on page 101, and "Sample REQUIRE macro to check for a valid date" on page 103 for two examples of REQUIRE macros, the latter of which is DATECHEK MENUEXEC, referred to in the sample above.

These macros work like XEDIT macros—they can call certain MENUEXEC REQUIRE subcommands or other CP and CMS commands (including another level of MENUEXEC). The subcommands provided allow you to retrieve the status of the data in any menu field. You can also write data back to the menu in either the output area or as "simulated" user input.

The REQUIRE macro facility provides the flexibility to perform the following functions:

- Perform customized, complex testing of the contents of a particular field
- Write generalized field validation routines
- Perform multiple field validation (for example, if field one contains "x", field two must contain "y" or "z")
- Create menu hierarchies (display menu "x" if field "y" contains "z")
- Simulate input to a field based on a user's abbreviated input (for example, if the user enters "J", change the input to "January")
- Fill in other menu fields based on input to a particular field (for example, given a social security number entered in field "x", look it up in a file, and fill the menu with other appropriate data)
- Change the attributes of a field
- Position the cursor
- Sound the alarm

The macro signifies the success or failure of the REQUIRE condition by its return code. The return code can also have MENUEXEC rewrite the entire menu if your REQUIRE macro (or something it calls) overwrites the full-screen image. Special return codes are used to cause MENUEXEC to exit immediately or to inform the user of an insufficient storage condition. By using XMENU windowing, REQUIRE macros can also display pop-up menus.

More details on the calling syntax, use of REQUIRE arguments, subcommands, and return codes, are included in "REQUIRE EXEC macro facility" on page 68.

# Displaying error messages and HELP menus

There are several MENUEXEC subcommands and options that allow you to control the display of error messages, explanatory messages, and HELP screens. The MENUEXEC command line options and subcommands we discuss here perform the following functions:

- Display error messages, not just return codes, while debugging
- Specify a field for displaying messages
- Associate an error or explanatory message with a specific field
- Assign a PF key to display explanatory or error messages
- Center error messages in a field
- Assign a PF key to display a HELP screen

## EMSG—Showing error messages and not just return codes

Normally when a MENUEXEC error is detected, only a return code is produced. It can be helpful, particularly during debugging, to receive text messages explaining what happened in addition to the error return codes. Use the EMSG option while developing and debugging an application to show explanatory text. After you have finalized your application, you can remove this option.

**Sample invocation**

```
"MENUEXEC MENU1 ( EMSG"
```

## EMSGFLD|ERRMSG subcommands—Specifying the field in which error messages are shown

You can tell MENUEXEC where on the menu you would like to display error messages for fields. For example, if a user enters incorrect data into a field, you can show an error message anywhere on the menu. Use the EMSGFLD *fieldname* or ERRMSG *fieldname* subcommand to specify the field where you want the message to appear. These two subcommands are synonymous.

You can only define one error message field per menu. If you do not define an error field, the default field name is $ERRMSG.

To specify the message that will be displayed in the error field, use the FIELDMSG subcommand. To specify a key that will display a message in the field where the cursor is positioned, use the FIELDPFK subcommand. For more information, see "FIELDMSG subcommand—Assigning a specific HELP or error message to a field" and "FIELDPFK subcommand—Using a PF key to show a HELP or error message for a field" on page 30.

**Sample invocation**

```
"MENUEXEC MENU1 ( SUBCMDS"
address MENUCMDS
"REQUIRE FIELD1 ALPHA"
"EMSGFLD FIELD"
"MENUEND"
address CMS
```

## FIELDMSG subcommand—Assigning a specific HELP or error message to a field

You can assign an error or explanatory message to a menu field. For example, if a user enters incorrect data into a field, you can display an explanatory message to help the user correctly enter data. The message you assign to a field is shown in the location you specify with EMSGFLD.

The message you assign to a field is shown to the user if the data entered into the field fails a validation test, or if the user presses the key you defined with FIELDPFK to display an explanatory message. For more information see "EMSGFLD|ERRMSG subcommands—Specifying the field in which error messages are shown."

**Sample invocation**

```
"MENUEXEC MENU1 ( SUBCMDS"
address MENUCMDS
"REQUIRE FIELD1 ALPHA"
"EMSGFLD MESSAGE"
"FIELDMSG FIELD1 'Type only alphabetic characters here'"
"MENUEND"
address CMS
```

## FIELDPFK subcommand—Using a PF key to show a HELP or error message for a field

You can specify which PF key will show the user the HELP or error message associated with a menu field. The actual HELP or error message is assigned using FIELDMSG, and the location of the message on the menu is defined with EMSGFLD.

For example, a user might not understand exactly what must be entered into a field. You can assign a brief explanatory message to a PF key to help the user. (See "HELPPFK subcommand—Using a PF key to show a HELP screen" to assign a complete HELP menu to a PF key.)

**Sample invocation**

```
"MENUEXEC MENU1 ( SUBCMDS "
address MENUCMDS
"FIELDPFK PF01"
"REQUIRE FIELD1 ALPHA"
"EMSGFLD MESSAGE"
"FIELDMSG FIELD1 'Type only alphabetic characters here'"
"MENUEND"
address CMS
```

## CENTEMSG—Centering messages in the error message field

Use the CENTEMSG option to center the error message text that is placed in the error message field by either FIELDMSG or FIELDPFK.

**Sample invocation**

```
'MENUEXEC MENU1 ( SUBCMDS CENTEMSG'
address MENUCMDS
"FIELDPFK PF01"
"REQUIRE FIELD1 ALPHA"
"EMSGFLD MESSAGE"
"FIELDMSG FIELD1 'Type only alphabetic characters here'"
"MENUEND"
address CMS
```

The MENUEXEC CENTER subcommand can be used to center text in other fields on a menu.

## HELPPFK subcommand—Using a PF key to show a HELP screen

You can design HELP screens and then specify which PF key will display the HELP screen to the user. Any key the user presses while looking at the HELP screen returns the user to the primary menu.

You can only define one HELP key per menu; if you define more than one, the latest one takes precedence.

**Sample invocation**

```
"MENUEXEC MENU1 ( SUBCMDS "
address MENUCMDS
"HELPPFK PF01 HELPMENU"
"MENUEND"
address CMS
```

# Editing user input from a menu

When a user is finished with a menu and input data has been moved to your application, you can perform a variety of editing functions. The subcommands and options discussed here perform the following functions:

- Convert data to uppercase
- Disallow the changing of blanks in the input fields to nulls
- Remove specific characters, blanks, or nulls from data EXEC variables

## UPCASE—Converting user input to uppercase

Use the UPCASE option to convert data entered into fields to uppercase before being placed in EXEC variables or MENUCTRL files.

You can use UPCASE as an option to convert all character fields. There is also an UPCASE subcommand (UPCASE *fieldname1 fieldname2 fieldnamen*) to convert a specific field or fields.

**Sample invocation**

```
"MENUEXEC MENU1 ( UPCASE"
```

## NONULLS—Not converting blanks to the right of input to nulls

Usually all the blanks to the right of data in an input field are converted to nulls before being passed to the EXEC. This allows the user to use the INSERT key to insert characters into input fields. If you *do not* want MENUEXEC to perform this conversion, use NONULLS.

**Sample invocation**

```
"MENUEXEC MENU1 ( NONULLS"
```

## STRIP subcommand—Removing characters from data returned from a field

Use the STRIP subcommand to remove characters from data entered into fields before being placed in EXEC variables. You can strip leading characters (L), trailing characters (T), or both (B, the default). You can strip blanks and nulls (the default) or a specific character (entered between single quotes).

**Sample invocation**

```
"MENUEXEC MENU1 ( SUBCMDS"
address MENUCMDS
"STRIP T"
"MENUEND"
address CMS
```

# Using windows

XMENU has a **windowing** facility that lets you display multiple, overlapping menus on your 327*x* terminal. The WINDOW option is used to specify the menu to be displayed as a window.

**Sample invocation**

```
"MENUEXEC MENU1 ( WINDOW MENU2"
```

Many variations are possible when using the windowing environment. A table is presented in "Using MENUEXEC options to manipulate windows" on page 34 that details how to use other window-related options and subcommands to meet your requirements. Windows offer the following flexibilities:

- One or more windows of varying size can be displayed on your terminal
- Windows can be smaller than, equal to, or larger than the physical size of your terminal screen
- As many windows can be displayed—at the same time—as can fit in your virtual machine's storage
- Windows can partially or completely overlap each other

## Glossary of XMENU window terms

To understand how to use XMENU windows, you should be familiar with XMENU window terminology. The following definitions explain XMENU window terms:

**Virtual Screen** A rectangular area, some or all of which is displayed on your terminal. You can make a virtual screen any size, but, if you do not specify a size, the default is the largest size your terminal supports. You can make a virtual screen bigger than your terminal screen, and you can specify what part is visible on your terminal. You can only define one virtual screen per terminal.

**Window** A rectangular area holding a character image of character data. Under XMENU, a window is an arbitrarily-sized, virtual 327*x* terminal. 327*x* data is moved to and from an XMENU window.

You can define as many windows as you want. Each window is referenced by a **case-specific**, 1-16 character name.

**Viewport** The rectangular area that displays all or part of a window on the virtual screen. You can think of viewports as the small boxes of data that appear on the screen. Any portion of a window can appear in a viewport, and only one viewport can exist per window.

This viewport can partially or completely overlay any other viewport.

**Borders** Optional frames that surround the viewport on the terminal and make the viewport stand out from the menu it overlays. If you do not specify a character for the borders, the default is an asterisk (*). You can also show the viewport's window name in the top border. If a viewport's data is too close to the edge of the virtual screen, no border is displayed at that edge.

**Display priority** Defines which viewports overlay other viewports. The highest priority viewport is displayed on top of all other viewports and is completely visible. If you do not specify a display priority, new viewports automatically get higher display priority than old viewports.

**Fields** A 327*x* screen consists of fields, which are contiguous strings of characters containing identical attributes or characteristics (such as the same color, brightness, protection, etc.). If you use XMENU or 327*x* programs, you should be familiar with fields.

Because the beginning of each field is defined by an attribute character, which requires one character space on the terminal, you see vertical blank rows between viewports and their borders and between viewports and other viewports. This is an unfortunate 327*x* family restriction which XMENU does its best to work around by only creating walls of attributes when absolutely necessary.

**Protection** A 327*x* field attribute that prevents a user from typing into or changing a 327*x* field. Under some XMENU applications, such as MENUEXEC, the user can only type into the topmost viewport. All lower viewports are protected from user input.

**Menus** Any display created with the XMENU editor XMEDIT, by the MENUEXAM utility, or by the MPGINT subroutine call, is an XMENU menu.

You can display a menu by itself or you can display it in a window. To display more than one menu at once, you must place each into a window or load it with a window.

One or more menus can be displayed in a single window.

## Some basic rules to follow when using windows

To use XMENU windows successfully, you should keep in mind the following points:

- When you use a window to display a menu, the window must be the same size as the menu. If you think of a window as a virtual terminal, this makes sense.
- The virtual screen size must be equal to or bigger than the largest window you plan to display in your application. This ensures that a viewport can always be placed on a virtual screen, even if the window's viewport is resized during an application.
- A viewport can be any size, up to the size of its window. You can place a viewport anywhere on the virtual screen, except for places that would cause a part of it to fall off the virtual screen.

- The only restriction on the number of windows is the size of your virtual machine's storage.
- When a window application returns user input, the window receiving the key pressed and the cursor position is the viewport the cursor was in when the user pressed an interrupt-generating key. Any other window receives key NOAID (that is, nothing specifically pressed), and the last output cursor position.
- The default size for any windowed object is its maximum size, and its default position is the upper left corner. Thus, the default viewport size is the window size, and the viewport's contents are made up of the upper left-hand data of the window in the upper left-hand corner of the virtual screen.

## Using MENUEXEC options to manipulate windows

Using windows in your applications requires very little additional programming. To use windows with EXECs, follow these basic steps:

**Basic steps for using windows**

1. Create the menus you want to display in windows.

   You create small menus using the XMEDIT SIZE option.

   Remember that while you can display several windows simultaneously, the user can enter data into only one window at a time.
2. Use MENUEXEC options and subcommands to define how and where window data is to be displayed.

The following table shows you how to implement some fairly common displays of windows in an EXEC application:

| If you want to: | You do this: |
|---|---|
| Display a menu without a window, but later redisplay it in a window. | Call MENUEXEC as you normally would to display the menu, using the SAVE option so you can reuse the menu later. Then use the WINDOW option to display the menu in a window. |
| Display a menu in a window and save it for later use. | Call MENUEXEC using the WINDOW and SAVE options. |
| Redisplay a menu in a window and delete it after this display. | Call MENUEXEC—with the menu number previously saved—using the WINDOW, SAVE, and PURGE options. |

| If you want to: | You do this: |
|---|---|
| Display a new menu in an old window. | Call MENUEXEC with the new menuname and the old window name. If you want to redisplay the old menu in this window, you need to supply the specific window name again. If you use only one window per menu, you only need to specify the window name the first time you use the menu. |
| Delete a menu and its window without displaying either. | Call XMENUINS—with the menu number previously saved—using the PURGE option. |
| Display a menu in a window only once and overlay it on existing windows (for example, a pop-up HELP window). | Call MENUEXEC using only the WINDOW option. |
| Load one or more menus into windows without displaying them (for example, to initially display several windows simultaneously). | Call MENUEXEC once for each menu you want to display using the WINDOW, SAVE, and NODISP options. |
| Display a portion of a window in a particular place on the screen. | Use the VIEWPORT subcommand to define a viewport's position, contents, and size. |
| Display borders around a viewport. | Use the BORDERS, HBORDERS, or VBORDERS options on the MENUEXEC command line to display a box, horizontal lines, or vertical lines, respectively. The TXTBORDE subcommand is often used with BORDERS to employ 327*x* TEXT borders, such as solid-colored borders. If displayed on terminals that do not support the chosen attribute, the window border will be made up of horizontal dashes and vertical solid lines. |
| Display the window name in the viewport. | Use the TITLE option on the MENUEXEC command line. |
| Display a menu that does not have a window in the middle of displaying other menus and their windows. | Call MENUEXEC as you normally would to display the menu. XMENU keeps track of the transitions between windowed and non-windowed displays. |
| Delete all windows and menus after this display. | Use the RESET option on the MENUEXEC command line. |

This example shows the basic coding for displaying a menu, then displaying a small pop-up menu, then exiting both menus:

**Some basic window coding**

```
/*   This sample REXX program shows how you would display */
/*   a pop-up window on top of another screen             */
/*                                                        */
address command
trace o
menu = 'MAINMENU'
/*                                                        */
/*   Display the main menu                                */
/*                                                        */
"MENUEXEC" menu " ( WINDOW WIND1 CLEAR SUBCMDS SAVE CURSOR FLD1"
/*                                                        */
/*   Do some field validation                             */
/*                                                        */
address MENUCMDS
"REQUIRE FLD1 DECIMAL"
"ERRMSG MESSAGE"
"FIELDMSG FLD1 'Please enter a decimal number'"
"MENUEND"
address CMS
/*                                                        */
/*   If the user presses PF key 3, display the pop-up     */
/*                                                        */
if pfk = 'PF03' then do
   "MENUEXEC POPUP ( WINDOW WIND2 BORDERS SUBCMDS TITLE CURSOR PFIELD"
   address MENUCMDS
   "VIEWPORT 5 20"
   "MENUEND"
   address CMS
end
/*                                                        */
/*   Get rid of saved menu                                */
/*                                                        */
"XMENUINS PURGE" menu
```

There is another sample REXX program in "Use XMENU windows" on page 98, that is a bit more sophisticated and that shows two windows that overlay, one smaller and subordinate to the larger one.

Here are some rules that relate to using the MENUEXEC SAVE option with windows that you may find helpful.

**Basic rules for using windows with SAVE**

- If you want to display a menu only once as a pop-up, use WINDOW but not SAVE.
- If you want to keep the menu for redisplay, use both the WINDOW and SAVE options.
- When you use a window for the last time, add the MENUEXEC PURGE option. It will be purged after it is displayed.
- If you want to purge a window without redisplaying it first, use XMENUINS PURGE. XMENUINS is used to purge MENUEXEC saved windows without having them displayed as they would be with MENUEXEC PURGE. For more information on the XMENUINS utility, see the *XMENU Utilities Reference*.

When your EXEC returns to CMS command level, the virtual screen and all windows and viewports are purged.

Consult Chapter 3, "MENUEXEC" on page 41 for more information about the MENUEXEC window options and subcommands.

## MENUEXEC and SAA Common User Access (CUA)

IBM has stated their commitment to System Application Architecture (SAA) Common User Access (CUA) as their direction for providing diverse applications a common "look and feel".

The original CUA specifications were distributed in 1987. A refinement of CUA was produced in 1989. The CUA rules specify overall screen design and field placement as well as highlighting and color use.

CUA action bars consist of a set of labeled, unnamed, and unprotected fields that are positioned in a window across the top line of a menu. Each choice presented on the action bar must be identified by a single action word, or label. Each field must be unprotected so the user can tab between choices, but must also be unnamed so that any user changes to the action bar are ignored.

With XMENU 2.3, a new MENUEXEC option, ACTION, was added to facilitate the use of CUA action bars with XMENU. The ACTION option instructs MENUEXEC to create the EXEC variable "ACTION", which contains the action word on which the cursor is positioned when the user presses an interrupt-generating key. Here is a sample of code that would implement a CUA action bar:

**Sample code implementing a CUA action bar**

```
"MENUEXEC" menu "( WINDOW wind1 ACTION SUBCMDS"
address MENUCMDS
⋮
"MENUEND"
address CMS
if cursor < 80
   select
      when action = "Help" then call act_help
      when action = "File" then call act_file

⋮
      otherwise nop
   end
```

Pull-down menus, which could be displayed when an appropriate choice is made from the action bar, are individual small menus whose placement could be controlled by the VIEWPORT subcommand.

Creation of SAA-compliant menus is facilitated by the XMENU/CUA Guide Mode component. Talk to your Relay Marketing Representative or refer to the *XMENU/CUA User's Guide* for more information on this XMENU product component.

# Part 2. The XMENU/REXX Interface Utilities Reference

This section provides a complete reference for the XMENU/REXX Interface utilities. Here's a "road map" of the utilities presented in this reference section:

# Chapter 3. MENUEXEC

This section provides a detailed reference description of all of the MENUEXEC commands, options, subcommands, error messages, EXEC variable formats, and REQUIRE EXEC macro facility subcommands and return codes.

The MENUEXEC utility is used to provide REXX, EXEC 2, EXEC, and program support for full-screen menus created by XMENU. MENUEXEC has the ability to automatically use, create, and delete REXX, EXEC 2, and EXEC variables. MENUEXEC accepts commands from the CMS command line, the MENUCMDS subcommand environment, the CMS system stack, and MENUCTRL files. MENUEXEC also has the ability to place data into menu fields from MENUCTRL files, and place user entries to the menu fields into MENUCTRL files. User input to the menu can also be written back to MENUCTRL files.

MENUEXEC runs as a nucleus extension. Optionally, it can also reside in a discontiguous shared segment.

## MENUEXEC command format

To invoke MENUEXEC, use the following command format:

```
MENUEXEC menuname|? [(options] [)]

MENUEXEC ( VERSION
```

### Where

*menuname* Specifies the name of the XMENU menu file to be displayed or, if the LIB option is specified, the XMENULIB member name of the XMENU menu. (See "MENUEXEC options" on page 42 for more information on the LIB option and refer to the *XMENU Editor Manual* for details on the XMENULIB utility.) If a menu with this name was loaded into storage using XMENUINS, it will be used even if the option LIB is specified. (Refer to the *XMENU Utilities Reference* for more information on the XMENUINS utility.)

**?** Types the MENUEXEC HELP file to your terminal.

*options* Specify one or more MENUEXEC options. See "MENUEXEC options" on page 42 for a detailed description of the available options.

**( VERSION** Displays the version of MENUEXEC and the version of the XMENU subroutine library.

# MENUEXEC options

This is an alphabetical listing of all the options that can be specified on the MENUEXEC command line.

| Option | Action |
|---|---|
| **ACTION** | An additional special EXEC variable, "ACTION", is created by MENUEXEC. The variable contains the word on which the cursor was positioned when an interrupt-generating key was pressed. This option is useful for determining the action item selected by a user from an action bar. |
| **ALARM** | The terminal alarm is sounded when the menu is displayed.<br><br>This option can be used, for example, to alert a user to an input error made during the previous display of the same menu. |
| **BORDERS** | Visible horizontal and vertical borders will be drawn around this menu's viewport. If the viewport size is equal to the virtual screen size, or if it is positioned to touch the edge of the virtual screen, one or more borders will not appear.<br><br>If this option is specified without the WINDOW option, it is ignored. Specifying BORDERS is equivalent to specifying HBORDERS and VBORDERS. |
| **CENTEMSG** | Any error message placed into the menu's error message field is centered within that field. |
| **CLEAR** | The screen is cleared before the menu is displayed. Normally this only needs to be done the first time full-screen mode is entered. If CLEAR is used when going from line mode to full-screen mode, the user doesn't need to press the PA2 or CLEAR key to display the menu. If you are already in full-screen mode, for example, upon the second display of a menu from one or a group of EXECs, you can save 327*x* output time by not specifying CLEAR on subsequent displays. |
| **CLOSE** | The virtual printer is closed after any printing is directed to it by MENUEXEC. If this option is not specified, the virtual printer is left open (subsequent virtual printing will be contained in the same spool file). |
| **CTRLVARS** | If specified, this option directs MENUEXEC to only create the special EXEC variables ("PFK", "CURSOR", and "CURFNAME", for example); data in user EXEC variables (input to named menu fields) are neither inspected nor modified. All the special EXEC variables created by MENUEXEC are listed in "MENUEXEC EXEC variables" on page 63. |
| **CURPOINT** | One or two additional stacked or special EXEC variables are created by MENUEXEC. The first, "CURWRD", contains the non-blank character string on which the cursor was positioned when the user completed the input. The second (only generated under REXX and EXEC 2, not EXEC), "CURSTR", contains the contents of the entire field—blank- and null-suppressed from both |

ends—on which the cursor was positioned when the user completed the input.

**CURSOR** *fieldname|offset*

Specifies where the cursor is to be positioned when the menu is displayed. If CURSOR is not specified, the cursor is positioned in the location specified when the menu was created.

You can specify either a symbolic field name or a numeric offset from the upper left corner of the menu (position 0 (zero)). If you wish to position the cursor where it was on the last call to MENUEXEC, use the value returned in the EXEC variable "CURSOR". Additional cursor control is available with the CURSOR subcommand.

**DEV** *xxx*

Returns control to the caller if an interrupt is received from the specified device before any input to the menu is entered by the user. *xxx* is the device address.

**DISK**

Printed output is directed to a file on disk rather than to the virtual printer. See the OUTPUT subcommand for more specific printed output control.

**DROP**

Any EXEC variables associated with fields that are completely cleared by user input (i.e., with the ERASE EOF key) are "dropped" rather than set to the null string. This option only affects EXECs written in REXX.

**EMPTY**

Any field whose corresponding EXEC variable is either empty, not defined, or, under REXX, dropped, is cleared to nulls.

This option can be used together with the SAVE option to allow fields to be cleared by simply dropping their REXX variables. See the description of the REXX option for another way of handling REXX variable input.

**EMSG**

When MENUEXEC errors are detected, error messages are returned; normally, only a return code is given. It is the responsibility of the caller to take corrective action.

This option can be used during debugging to identify errors in a menu or an EXEC.

**FILE** *ctrlname*

Specifies the MENUCTRL file to be processed for MENUEXEC subcommands. When this option is used, the CMS system stack is not checked for MENUEXEC subcommands.

Along with any field data and/or other MENUEXEC subcommands in the file *ctrlname* MENUCTRL, FILE and UPDTFILE subcommands can be contained in this MENUCTRL file, thereby allowing other MENUCNTRL files to be "opened" in addition to *ctrlname* MENUCTRL.

**HBORDERS**

Visible horizontal borders are drawn around this menu's viewport. If the viewport's vertical size is equal to the virtual screen's vertical size, or if it is positioned to touch the top or bottom edge of the virtual screen, one or more borders will not appear.

If this option is specified without the WINDOW option, it is ignored. Specifying BORDERS is equivalent to specifying HBORDERS and VBORDERS.

**IGNSCFN** This option specifies that any MENUEXEC subcommands referring to non-existent fields are ignored; normally, they would cause an error exit. This option is helpful when you want to create a single MENUCTRL file (containing MENUEXEC subcommands) to be shared by several menus.

**IOLOG** Any XMENU terminal I/O is logged to a CMS file. This option is intended for diagnostic purposes only.

**LEAVSMSG** The SMSG environment is left enabled when MENUEXEC returns to your EXEC. Thus, if any SMSGs are sent while MENUEXEC is not in control, they remain pending. The next call to MENUEXEC that uses the SMSG option will receive the oldest pending SMSG.

LEAVSMSG must be specified together with the SMSG option. If SMSG is specified by itself, the SMSG environment is ended when MENUEXEC ends. When your EXEC(s) return to CMS command level, the SMSG environment is automatically ended.

**LIB** *libname* Causes the menu to be loaded from the library file named *libname* XMENULIB. This option is ignored if the menu was loaded from an in-storage copy loaded by the XMENUINS utility or if the SAVE option is specified and the menu name passed is already loaded by a previous MENUEXEC call that also used the SAVE option.

**LISTVARS** The field names of fields changed by user input will be placed into an EXEC variable array named VARCHNG.

VARCHNG.0 will contain the number of changed fields.

VARCHNG.1 will contain the first changed field name. VARCHNG.2 will contain the second changed field name. VARCHNG.3 will contain the third changed field name, and so on.

You can use VARCHNG.0 as a loop counter, and the VARCHNG array to restrict processing to changed fields. The order in which the changed fields appear in the array is not defined.

**MAP** PF keys 13 through 24, and 25 through 36, will be returned as PF keys 01 through 12. This option is helpful when you only need to use twelve or fewer PF keys and want to let the users use the same key position on the keyboard, no matter if their terminal is a 3277, 3279, or whatever. This option also makes it unnecessary for an EXEC to have two or three checks for PF keys that perform the same function.

**MDT** All unprotected fields on this menu will have their Modified Data Tag (MDT) set ON; therefore, the data in each input field will be returned as user input even if the user did not modify it. Specific unprotected fields can be marked with MDT ON by using either the XMEDIT program's attribute selection procedure or, if the field is named, with the MENUEXEC CHANGE subcommand.

MDT can be used to automatically "input" data to a program from a menu filled with default values from a MENUCTRL file.

**NOAPLT** No check is performed for APL or TEXT characters in menu output or input. This option can improve performance if you are certain that no such characters will be found in the I/O data stream. Unpredictable results will occur if APL or TEXT characters are present in the data stream (usually the menu format will appear garbled).

**NOCURSOR** The cursor position is not modified when this menu is written. This option is only useful when used with the SAVE option.

NOCURSOR allows you to write applications, such as performance monitors, that modify the screen while the user is still typing. Because the cursor isn't moved, the user doesn't see it "pop" to a different field on the menu while new data is being written there.

If NOCURSOR is used and the terminal screen needs to be erased (for example, if CLEAR was specified, or if a non-full-screen message came in between displays) the cursor will appear in the upper left corner of the screen.

**NODISP** The menu is not displayed. This option can be used to pre-load menus to be displayed as background windows on the first "real" display. You should use this option together with the WINDOW and SAVE options.

**NOMDTRS** Existing Modified Data Tag (MDT) bits on the terminal will not be reset to zero when the menu is written. This option is only useful when used with the SAVE option.

NOMDTRS leaves a previously modified field's MDT bits on, so it will appear that a user just typed into the field. This option can be used to force reprocessing of all input, even if the user hasn't reentered any.

**NOMRGWCC** Only the topmost window's Write Control Character (WCC) is sent to the terminal. This option can be used, for example, to ensure that only the topmost window's alarm causes the physical terminal's alarm to sound.

This option is only relevent when using windows; it is ignored otherwise.

**NOOPT** No optimization of the output data stream is performed. This option can save processor time at the expense of additional terminal I/O. You might consider using this option in EXECs used in processor-constrained environments having only local, channel-attached (i.e., not remote) terminals.

**NOUNLOCK** The keyboard is not unlocked when this menu is displayed.

NOUNLOCK allows you to write applications that only display data and do not accept user input. This option is most useful when used together with the NOWAIT option.

Note that the RESET key can still be used to unlock the keyboard when the terminal is attached to certain 327*x* controllers.

**NOWAIT** The menu is only displayed; MENUEXEC does not wait for user input. This option is used for displaying output only, or read/only, menus. Any further output to the terminal will remove this menu; therefore, you should ensure in your EXEC that the user has enough time to view the menu before displaying another menu, for example.

**NOXVARS|XVARS**

NOXVARS specifies that even though you are running under an EXEC, you do not want to use or modify EXEC variables. You can use NOXVARS when an EXEC calls a program that calls MENUEXEC, because MENUEXEC operates as if it is running under an EXEC, even if it is not called directly by that EXEC. MENUEXEC modifies and uses variables from the EXEC last entered before being called.

XVARS specifies that variables should be used; this is the default when running under an EXEC.

**NONULLS** Normally, all blanks in the rightmost parts of input fields are converted to nulls before output, thereby allowing the user to use the INSERT key to insert characters into input fields. If NONULLS is specified, this null conversion is not performed.

**PAKDATA** MENUEXEC obtains the cursor position and all modified data if PA1, PA2, PA3, or TEST REQUEST is pressed. If this option is not used, these keys only return limited data; the cursor position and the field changes are not picked up in the EXEC variables.

**PA1** The PA1 key, if pressed, will be trapped on input and returned to the user as "PA1" in the EXEC variable "PFK". If this option is not used, pressing PA1 will put the user into CP READ. Then, entering BEGIN to return to the program results in "CLEAR" being returned as the key pressed.

**PREFIX** *string* All REXX or EXEC 2 variables are fetched and stored using *string* as a variable name prefix. For example, if PREFIX A. is specified, menu field X would be filled from EXEC variable "A.X" and, if modified, resaved into "A.X". Special variables are also prefixed. Continuing with the example above, the key pressed would be returned as "A.PFK".

If you want a period between the prefix string and the variable name, you must include a period in *string*. String is limited to eight characters.

You can use this option to control information from different menus by making the information individually distinct. You can also use this option to simulate longer field names.

Prefixing is also used by Multiple Terminal Facility applications to specify the terminal on which this menu is being used.

**PURGE** At the end of this MENUEXEC call, this SAVEd menu is purged from storage. This option is only effective when used together with SAVE.

If PURGE is specified together with WINDOW, the window is also purged. PURGE and WINDOW should be used together

when SAVE is used and you want to display a one-time-use only menu, such as pop-up HELP or a command confirmation menu.

**RDR** Returns control to the caller if a virtual card reader interrupt occurs before any input to the menu is entered by the user.

**RESATT** All menu field attributes are set to the characteristics defined at the time the menu was created or first loaded.

RESATT gives you a quick way to return fields to their initial attribute states. This can be useful when you are using the SAVE option and have changed fields during the interactive data entry process. For example, if you've protected correct user input or highlighted incorrect user input and want to reset the fields all at once, use RESATT.

Use of RESATT without SAVE consumes computer resources but has no other particular effect (since the newly loaded menu is already in its initial state).

The resetting of attributes occurs before MENUEXEC processes any subcommands, so any CHANGE commands encountered will be honored.

**RESET** At the end of this call to MENUEXEC, all saved menus and windows are purged. This option performs the same function that MENUEXEC performs implicitly when you return to CMS command level.

**RESHOW** The entire menu should be output to the terminal, rather than only those fields changed since the last display. RESHOW should be used if a full-screen write is performed between MENUEXEC calls with the SAVE option. This is necessary because MENUEXEC does not know that a foreign application (this includes a call to another XMENU application or a call to MENUEXEC without the SAVE option) changed the contents of the screen. This option is only effective when used with SAVE.

**REXX** MENUEXEC will distinguish between REXX literal (undefined) variables and empty (zero-length) variables.

Without this option, both undefined and zero-length variables cause the field to retain its currnt contents. With this option, literal variables will cause the field to retain its current contents; empty variables will cause the field to be cleared to nulls (binary zeros).

Note that the EMPTY option causes both cases to clear the field.

**SAVE** The loaded menu should be saved in storage for use by a subsequent MENUEXEC call. This option optimizes the repeated use of a menu (or group of menus) by one EXEC application. With SAVE, an EXEC variable named "MENU" is filled with a unique string for passing this menu to subsequent MENUEXEC calls that also use the SAVE option.

The state of a menu passed between MENUEXEC calls is slightly different from a menu loaded from DASD each time MENUEXEC is called. See "Using the saved menu environment" on page 14 for more details.

**SKIP**

All protected fields on the screen will have their SKIP attribute set. When each input area is filled by the user, the cursor will automatically move over the protected field(s) to the next available input field. Specific protected fields can be marked with SKIP by either the XMEDIT program attribute selection procedure, or, if the field is named, with the MENUEXEC CHANGE subcommand.

SKIP is useful, for example, in optimizing user input to data-entry menus having many fields.

**SMSG**

Returns control to the caller if an SMSG interrupt is received by your virtual machine before any input to the menu is entered by the user. If this occurs, the EXEC variable "SMSG" is set to the special message string, and "SMSGUSR" is set to the userid of the SMSG sender.

**STACK LIFO|FIFO**

Any data in the menu that is modified by a user is stacked either last-in-first-out (LIFO), or first-in-first-out (FIFO). Neither LIFO or FIFO is a default; one or the other must be specified with the STACK option.

Besides stacking modified data fields, if NOXVARS is also specified, STACK can stack information from MENUEXEC created variables, described in "MENUEXEC EXEC variables" on page 63, in the same format as data records. For example, the options STACK LIFO and NOXVARS will result in the MENUEXEC variables and their values (for example, PFKSTR, PFK, CURCOL, etc.) being stacked first, followed by *fieldname 'data'* for each modified field. (*fieldname 'data'* reserves seven character spaces for the *fieldname* plus one blank space followed by single quotes enclosing the changed data.) The menu field positioned at the top of the menu will appear last in the stack.

As another example, STACK FIFO will place the information *fieldname 'data'* in the stack for each changed field, starting with the changed field positioned at the very top of the menu.

**SUBCMDS**

MENUEXEC subcommands are retrieved from the MENUCMDS subcommand environment.

Normally, MENUEXEC checks the CMS system stack for a stacked line containing the word MENUCMDS. If this is found, it and subsequent lines are read and treated as MENUEXEC subcommands.

With the SUBCMDS option, the stack is not checked; rather, MENUEXEC sets up a CMS subcommand environment called MENUCMDS and returns to your EXEC. Any subcommands entered after the record `address MENUCMDS` in REXX, or `&SUBCOMMAND MENUCMDS` in EXEC 2, are treated as MENUEXEC subcommands. With SUBCMDS, you should not pass a leading MENUCMDS subcommand as you would with stack processing.

When the MENUEND subcommand is received, MENUEXEC terminates the MENUCMDS subcommand environment and continues processing. If a return is made to CMS command level

without MENUEXEC first receiving the MENUEND subcommand, MENUEXEC automatically terminates the MENUCMDS subcommand environment without any subsequent display.

A REXX EXEC without the SUBCMDS option might look like this:

```
/* Simple EXEC */
queue 'MENUCMDS'
queue 'CHANGE FIELD1 BLUE REVERSE'
queue 'CHANGE FIELD2 RED BLINK'
queue 'MENUEND'
'MENUEXEC' menuname
```

A REXX EXEC having identical function using the SUBCMDS option would look like this:

```
/* Simple EXEC */
'MENUEXEC' menuname' (SUBCMDS'
address MENUCMDS
'CHANGE FIELD1 BLUE REVERSE'
'CHANGE FIELD2 RED BLINK'
'MENUEND'
address CMS
```

With the SUBCMDS option, each MENUEXEC subcommand is processed individually and its MENUEXEC return code is returned; consequently, you can perform additional EXEC processing between each subcommand.

**TEST** The menu is displayed, but input is ignored and EXEC variables, stacked data, and MENUCTRL file data is neither inspected nor modified. If MENUEXEC is called directly from the CMS command line, TEST is assumed. The TEST option also implies the EMSG and CLEAR options.

You can use TEST (or CMS command line use of the MENUEXEC command) to look at a menu and test its input areas. When a menu is displayed using TEST, any interrupt-generating key causes MENUEXEC to exit.

**TITLE** The name of the window will be displayed in the viewport's top border.

The window name will only appear if it fits in the border, and if the border is displayed. You must use either the BORDERS or HBORDERS option to have a top border appear. If the WINDOW option isn't specified, this option is ignored.

**UPCASE** Any data in the menu that is modified by the user will be converted to uppercase characters before being stacked, filed, or put into EXEC variables. You can specify UPCASE for specific field names by using the UPCASE subcommand. See "MENUEXEC subcommands" on page 51 for more information on this subcommand.

**VBORDERS** This menu's viewport will have visible vertical borders drawn around it. If the viewport's horizontal size is equal to the virtual screen's horizontal size, or if it is positioned to touch the left or

right edge of the virtual screen, one or more vertical borders will not appear.

Specifying BORDERS is equivalent to specifying HBORDERS and VBORDERS. If this option is specified without the WINDOW option, it is ignored.

**VSCREEN** *vsize hsize*

Specifies an explicit size for the virtual screen. If not specified before the first use of windows (by using the WINDOW option), an implicit virtual screen is created that is the size of your real terminal.

*vsize* is the size of the virtual screen in number of rows. *hsize* is the size of the virtual screen in number of columns.

The virtual screen must be equal in size or larger than the largest window to be used by this application. Therefore, if you know that your application will display windows larger than your physical terminal, specify VSCREEN on the first MENUEXEC call where you also use windows.

The virtual screen, all windows, viewports, and menus are implicitly purged when you return to CMS command level.

**WAIT** *time*

Returns control to the caller if a specified amount of time elapses before any user input is entered. *time* is specified as tenths of a second. For example, WAIT 20 causes the menu to be displayed for two seconds before it is cleared from the screen.

If WAIT is used, your virtual machine must have ECMODE set ON (this can be done by the CP command `SET ECMODE ON` or by a specification in your userid directory entry.

**WINDOW** *wname*

Specifies that the named menu should be displayed using the XMENU windowing support.

When you specify WINDOW, the named menu will be displayed on the terminal in a viewport on a virtual screen.

If previous menus were also displayed using WINDOW, this menu's viewport will overlay the previous menus, and their input fields will temporarily be changed to protected; thus, you can only enter data in the topmost viewport.

If you also specify SAVE, this menu and viewport can be reused; otherwise, it and its window and viewport are purged when MENUEXEC exits to your EXEC.

Whenever a window is requested, it implicitly becomes the topmost menu.

The size, position, and contents of the viewport are specified with the MENUEXEC VIEWPORT subcommand.

# MENUEXEC subcommands

The following list presents the MENUEXEC subcommands that can be used in the MENUCMDS subcommand environment, contained in the CMS system stack, or included in MENUCTRL files.

**Note:** See "The SUBCMDS option to enable the SUBCOM interface" on page 8 for instructions on how to pass subcommands using the MENUCMDS subcommand environment. For information on using the CMS stack and MENUCTRL files to pass MENUEXEC subcommands, see "The stack interface" on page 7, and "MENUCTRL files" on page 8.

MENUCMDS is presented first in this list, followed by the format for passing comments and field data from the CMS stack or in MENUCTRL files. The remaining subcommands are presented in alphabetical order.

**MENUCMDS** This line identifies the contents of the CMS system stack, or the records within a MENUCTRL file to be valid MENUEXEC subcommands. If this line is missing from either the top of a MENUCTRL file or the top of the CMS system stack, neither the MENUCTRL file or the stack is read. MENUEXEC does not "unstack" any CMS system stack records unless `MENUCMDS` is found at the top of the stack.

To initiate the MENUCMDS subcommand environment within your EXEC, you must use the MENUEXEC SUBCMDS option and then include any subcommands after the record `address MENUCMDS`, in REXX EXECs, or `&subcommand MENUCMDS`, in EXEC 2 EXECs.

***** *comments* When passing subcommands via the CMS system stack, MENUCTRL files, or in the SUBCMDS environment under EXEC 2, records starting with an asterisk (*) are ignored and may be used for comments. When using the SUBCMDS environment with REXX, comments are enclosed using the normal REXX notation:

```
/* comments */
```

The MENUCMDS record must precede any comment records in the CMS stack or MENUCTRL files. Comment records may appear anywhere in your EXEC when using the SUBCMDS environment.

*fieldname 'data'* Any record starting with a string of seven or less characters that is not recognized as a valid MENUEXEC subcommand is considered to be a data subcommand. This subcommand places the *data* string, contained between the apostrophes ('), into the field named *fieldname* starting in the first position of that field. The field receiving the data can be protected or unprotected, with any attribute combination. Leading or trailing blanks within the apostrophes are significant. Two apostrophes appearing together within the *data* string signify that one apostrophe is to be written to the field, for example, 'Sharkey''s Day'.

= '*more-data*'

If all the required data cannot be contained on a single *fieldname 'data'* record, it may be continued in any number of additional records. These records each start with an equal sign (=) and contain data enclosed in apostrophes, just as in the first data record.

Other MENUEXEC subcommands cannot appear between the first data record and its continuation records.

**ADD** *fieldname*

*offset attributes*
*row column attributes*

A new field is created in the menu at the specified location with the specified attributes. The field name specified as *fieldname* must not already exist in the menu, and it must be alphanumeric with an alphabetic first character. Use either the *offset* (upper-left corner is 0 (zero)) or *row column* (upper-left corner is 1 1 ) format to specify the location of the start of the new field. *attribute* values can be any of those listed in the CHANGE subcommand. Any other MENUEXEC subcommands referring to this field must follow the ADD subcommand.

The ADD subcommand sets the attribute definitions in both the output menu area and the menu's field control blocks. This means that the attribute values specified with ADD become the field's default attribute values. The CHANGE command only changes the attribute definitions in the output area (so that they can be changed back to their defaults if necessary).

**ADDSA|SAADD** *value*

*fieldname field-offset* | *offset* | *row column*

A character attribute is placed in the menu at the specified location, allowing you to change, for example, the color of a field in mid-field. *value* can be a color, an extended highlighting type, or a programmed symbol set number. Valid color or highlighting values are listed under the CHANGE subcommand. The programmed symbol set number can be hexadecimal zero, X'40' through X'EF', or X'F1' ('F1' refers to the APL/TEXT symbol set, if present in the terminal). To restore all three attribute types to their field's default values, enter DEFAULTS as the *value*.

Multiple character attribute values can be assigned to a location by entering multiple ADDSA or SAADD subcommands. If conflicting values exist for a location, the last one encountered takes precedence.

A character attribute overrides any field attributes from (and including) the character at the specified position to the end of the menu, unless another attribute character of the same type is placed in a menu position, using ADDSA or SAADD, following the position of this one.

If you use the *fieldname field-offset* format of this subcommand, the character attribute will be placed within the named field at the specified offset. For example, `fieldname 0 or 1` will place the attribute character at the first position in the field, effectively

changing the whole field until another attribute character is encountered. Entering *fieldname 2* will place the attribute character on the second position in the field, leaving the first position in the field unchanged, and changing all others. If you use *offset* to specify the character location, the character attribute will be placed at the specified offset from the top-left corner of the menu, offset 0. If you use *row column* to specify the character location, the attribute character is placed on the row and column position specified by the two operands. In this form, the upper-left corner is 1 1 (not position 0).

**BORDER** *type char attributes*

**Where**:

*type*:
**ALL|TOPLEFT|TOPRIGHT|BOTLEFT|BOTRIGHT|TOP|BOTTOM|LEFT|RIGHT**

*char*:
*character*|**GE** *character*

*attributes*:
**[BLUE|GREEN|RED|YELLOW|PINK|TURQuoise|WHITE|DEFCOL]**
**[BLINK|UNDERscore|REVerse|NOHIlight]**
**[PSnn]**

The window generated by this call to MENUEXEC will have borders comprised of the characters and attributes specified in the subcommand.

If ALL is specified as *type*, all border positions are changed; otherwise, only the specified portion of the border is modified. TOPLEFT, TOPRIGHT, BOTLEFT, and BOTRIGHT refer to the corners of the window border. TOP, BOTTOM, LEFT, and RIGHT refer to the four sides of the window border.

The *character* specified must be in the range of X'40' to X'FE'. If a graphic escape character is desired, specify GE followed by the desired character. If the character cannot be displayed on your terminal, a dash will appear in its place.

The border will have the extended color, highlighting, and/or symbol set specified. If a terminal doesn't support these attributes, they are ignored.

If a programmed symbol set is specified, it must be a hexadecimal value of zero, or X'40' through X'EF'. A symbol set having the same logical number must be loaded using the LOADPS MENUEXEC subcommand.

If you want a faster way to generate a window border using 327*x* TEXT characters, use the TXTBORDE subcommand.

**CANCEL** This subcommand is used in the MENUCMDS subcommand environment to terminate this call to MENUEXEC without displaying the menu, for example, in the case of error. The

subcommand environment is terminated, and the subcommand returns with a -10 return code.

Either this subcommand or the MENUEND subcommand must be used to end the MENUCMDS subcommand environment when SUBCMDS is specified as a MENUEXEC command line option. If CANCEL is either stacked, or found in a MENUCTRL file, it is ignored.

**CENTER** *fieldname1 [fieldname2 ... fieldnamen]*

Any data moved into the specified field(s) are to be centered within the field(s). The unused areas of the field are filled with nulls (binary zeros), unless the CENTFILL subcommand is also specified.

**CENTFILL** *character*

Specifies the character to be used to fill the unused areas created by the CENTER subcommand. If not specified, nulls (binary zeros) are used as the fill characters.

**CHANGE** *fieldname* **[PROTect|UNPROTect|NUMERIC]**
**[BRIGHT|DIM|DARK|LGHTPEN]**
**[MDT|NOMDT]**
**[SKIP|NOSKIP]**
**[BLUE|GREEN|RED|YELLOW|PINK|TURQuoise|**
**WHITE|DEFCOL]**
**[BLINK|UNDERscore|REVerse|NOHIlight]**
**[PSnn]**

The *fieldname* specified in this subcommand will have its attributes changed before the menu is displayed. Any number of attribute value changes for this field can be placed in the subcommand. If multiple attribute definitions for one attribute type exist, the last one specified is used.

The SKIP and MDT operands of this command are similar to the MENUEXEC options SKIP and MDT. MDT is used to return a field's contents as input, even if the user does not modify the field. This allows you to create menus with default (or MENUCTRL file supplied) data returned as user input, even if the user did not enter data.

NUMERIC assumes unprotected and numeric. LGHTPEN assumes dim and selector pen detectable. BRIGHT is always selector pen detectable, and DARK is never selector pen detectable. If a programmed symbol set is specified, it must be a hexadecimal value of zero, or X'40' through X'EF' (X'F1' cannot be specified as a field attribute). A symbol set having the same logical number must be loaded using the LOADPS MENUEXEC subcommand.

The ADD subcommand sets the attribute definitions in both the output menu area and the menu's field control blocks. This means that the attribute values specified with ADD become the field's default attribute values. The CHANGE subcommand only changes the attribute definitions in the output area (so that they can be changed back to their defaults, if necessary).

**CURSOR** *menu-position*
*fieldname* [*offset*]
*menu-line menu-column*

Specifies the position of the cursor when the menu is displayed. If the *menu-position* operand form is specified, the cursor is placed at that offset from the upper-left corner of the menu (position 0 (zero)). This is the cursor position format returned to the "CURSOR" EXEC variable following the display of a menu. If the  form is specified, the cursor is placed on the field *fieldname* at the start of the field, or, optionally, at the start plus a numeric *offset* (*offset* may be larger than the field size). If *menu-line menu-column* is specified, the cursor is placed on the line and column specified by the two operands. In this form the upper-left corner is 1 1 (not zero).

**DELETE** *fieldname [fill]*

An existing field is deleted from the menu. The data within the field remains in the menu, and the field's attribute character is replaced with a null unless *fill* is specified. If *fill* is specified, this character is used to replace the attribute. Any other MENUEXEC subcommands referring to this field must precede the DELETE subcommand.

**DELETESA|DELSA|SADELETE|SADEL** *value*
*fieldname field-offset* | *offset* | *row column*

A character attribute is removed from the menu at the specified location. *value* can be a color, an extended highlighting type, or a programmed symbol set number. The programmed symbol set number can be hexadecimal zero, X'40' through X'EF', or X'F1'. To remove a character attribute previously defined as DEFAULTS, enter DEFAULTS as the *value*. See ADDSA|SAADD for details on the location-specifying operands, *fieldname field-offset*, *offset*, and *row column*.

This option lets one part of an EXEC remove the character attribute placed by another part of the same EXEC (or removes a character attribute placed during a previous use of this menu with the MENUEXEC SAVE option).

**EMSGFLD|ERRMSG** *fieldname*

Specifies the name of the field into which field error messages are moved. If EMSGFLD is not specified, the default error message field is named $ERRMSG. Only one error message field can be defined for a menu.

**FIELDMSG** *fieldname 'message'*

Associates a HELP or error message with a specific menu field. This message is displayed in the error message field (see the EMSGFLD subcommand) if a REQUIRE validation condition for the field *fieldname* fails, or if the key(s) defined by the FIELDPFK subcommand is pressed. The HELP or error message must be enclosed in apostrophes, and two apostrophes together in the *message* string signify one apostrophe in the data written to the field, for example, Sharkey"s Day. If the *fieldname* designated

to contain the error or HELP message does not exist, the message is not moved to the screen.

**FIELDPFK** *key01 [key02 ... keyn]*

Specifies which key or keys will cause the message defined by the FIELDMSG subcommand to be displayed for the field on which the cursor is positioned. If the key is pressed, and no field message exists for the field, or if there is no error message field (see the EMSGFLD subcommand), this subcommand has the same effect as the IGNORE subcommand.

**FILE** *filename [fieldname1|***ALL***] [fieldname2 ... fieldname20]*

Specifies that the file *filename* MENUCTRL will be opened for subcommand input. If ALL is specified, all data records found in the file will be put into the menu if corresponding field names match. If specific field names are included in this subcommand, only the data records for the specified fields will be taken from the file. If no field names are included with the subcommand, only control records (such as CHANGE subcommand records, for example) will be taken from the file—no data will be loaded into the menu. If the file contains data records for field names not in the menu, they are ignored; however, control records found in the file must match menu field names, or an error will be indicated unless the MENUEXEC option IGNSCFN is also specified.

One MENUCTRL file can be used for several menus and/or several MENUCTRL files can be used for one menu. As many as twenty field names can be specified on each FILE subcommand. Several FILE subcommands can be specified for the same file. Any MENUEXEC subcommands that are found in the file will be executed, as will any additional FILE subcommands. FILE subcommands in MENUCTRL files cannot reference themselves or any other already active file; that is, MENUCTRL file recursion is not permitted.

Each MENUCTRL file must have the MENUEXEC subcommand MENUCMDS starting in the first column in the first record to identify it as a valid MENUCTRL subcommand file. MENUCTRL files must be fixed length, LRECL 80 format.

**HELPPFK PF***nn menuname [libname]*

Causes a HELP menu, named *menuname* MENU, to be displayed if the specified key is pressed. Any key pressed while in the HELP menu causes a return to the primary menu. If *libname* is specified, the HELP menu is loaded from an XMENULIB. Only one HELPPFK subcommand is supported; if more than one is specified, the last one encountered takes precedence. PF keys defined in HELPPFK are subject to your use—or not—of the MENUEXEC option MAP.

**IGNORE** *key1 [key2 ... keyn]*

Flags certain 327*x* keys to be ignored if pressed by the user. Any valid interrupt-generating key may be specified. When the user presses one of the specified keys, MENUEXEC ignores the interrupt. This option saves you the trouble of programming checks for keys that perform no function. Any PF key, PA key,

ENTER, TESTREQ, CLEAR, etc., or any special interrupt (SMSG, DEVICE, etc.) can be specified.

**IGNREQ** *key1 [key2 ... keyn]*
If any of the keys specified are pressed, and REQUIRE subcommands are present, no verification that data match REQUIRE criteria is performed. This subcommand provides a quick way of exiting MENUEXEC if the user data is to be ignored (for example, if the user presses PF1 for HELP, PF3 to QUIT, or some other application exit condition).

**LJFILL** *character* Specifies the character to be used to fill the unused areas created by the LJUST (left justify) subcommand. If not specified, nulls (binary zeros) are used as the fill characters.

**LJUST** *fieldname1 [fieldname2 ... fieldnamen]*
Specifies that data moved into the listed field(s) are to be left-justified within those fields. The unused area of the field is filled with nulls (binary zeros) unless the LJFILL subcommand is also used.

**LOADPS|PSLOAD** *nn symbol-set-name*
Loads a programmed symbol set into the terminal's storage. *symbol-set-name* is the filename of the file *symbol-set-name* PSLOAD. The number assigned as *nn* must be hexadecimal and be between X'40' and X'EF'. Fields within the menu whose attributes contain this symbol set number are displayed using this symbol set. If a symbol set with the same number is already contained in the terminal, it is overlaid by this new set.

**LOADVARS** Causes MENUEXEC to load menu fields with the contents of EXEC variables at the time the LOADVARS subcommand is encountered. If LOADVARS is not specified, EXEC variable loading takes place following the processing of all subcommands.

All user data EXEC variables are loaded when LOADVARS is encountered. The EXEC variables loaded are those in existence at the time the MENUEXEC command was executed. If this subcommand is used more than once per MENUEXEC call, only the first LOADVARS encountered causes variable loading, the other invocations are ignored. Once LOADVARS is specified, no other variable loading takes place following subcommand processing. When the MENUEXEC subcommand NOXVARS or the MENUEXEC option CTRLVARS is specified, LOADVARS is ignored.

**MENUEND** This subcommand is used to mark the last subcommand stacked, the last record entered in the MENUCMDS subcommand environment, or the last subcommand read from a MENUCTRL file.

Either this subcommand or the CANCEL subcommand must be used to end the MENUCMDS subcommand environment, when SUBCMDS is specified as a MENUEXEC option. MENUEND must be used if data appears in the stack or MENUCTRL file following valid MENUEXEC subcommands. Data in the CMS system stack following MENUEND are not read or unstacked.

You can omit MENUEND if the stack or MENUCTRL file only contains MENUEXEC subcommands.

**NOUPDPFK** *key1 [key2 ... keyn]*

If any of the keys specified are pressed and UPDTFILE subcommands are present, no updating of MENUCTRL files takes place, even if the user changed data on the menu. This subcommand provides a quick way of exiting MENUEXEC if the user data is to be ignored (for example, if the user presses PF1 for HELP, PF3 to QUIT, or some other application exit condition).

**NOXVARS** *fieldname1 [fieldname2 ... fieldnamen]*

The fields specified will not be placed in or taken out of EXEC variables.

**OUTPUT** *filename* **[SCRIPT]**

Specifies the filename and, optionally, the format of MENUEXEC print output. This subcommand only affects output if the MENUEXEC option DISK is also specified. *filename* overrides the default filename of the print file (the filename of the menu). SCRIPT specifies that the output should be in DCF (SCRIPT/VS) format rather than ANSI carriage control format.

**PFKSTR PF***nn* '*data*'

Specifies that if PF*nn* is pressed, *data*—the string contained in apostrophes—will be returned in the EXEC variable "PFKSTR". There is no truncation performed when running REXX, however if you are using EXEC 2, only the first 255 characters are returned, and if you are using EXEC only the first eight non-blank characters of the string are returned.

**PRINT** *key1 [key2 ... keyn]*

Creates print output of the screen if any of the specified PF keys are pressed. After the print output is created, if the PF key has also been flagged to be ignored by the IGNORE subcommand, MENUEXEC waits for more user input, otherwise MENUEXEC exits normally. Normally, the print output is sent to your virtual printer in ANSI format with carriage control characters; the OUTPUT subcommand and the DISK option can affect the destination and format of this print output.

**PRTNOH** *key1 [key2 ... keyn]*

Creates print output of the screen if any of the specified PF keys are pressed. After the print output is created, if the PF key has also been flagged to be ignored by the IGNORE subcommand, MENUEXEC waits for more user input, otherwise MENUEXEC exits normally. Normally, the print output is sent to your virtual printer in ANSI format with carriage control characters; the OUTPUT subcommand and the DISK option can affect the destination and format of this printed output.

PRTNOH differs from PRINT in that neither a header line nor a box is printed with the menu.

**REQUIRE** *fieldname requirement*

Defines the data validation requirement(s) for a field that must be met before MENUEXEC will return to your EXEC or program. The data to be checked is logically blank- and null-suppressed

from both ends before requirement checking begins. If the check fails, the menu is redisplayed and the cursor is positioned on the offending data character.

If a FIELDMSG exists for this field, it is displayed in the error message field (defined by the EMSGFLD subcommand). If requirements for several fields within a menu exist, they are checked in the order that they appear in the subcommand stream. If multiple requirements exist for one field, and any of the requirement checks succeed, the entire field is considered to be validated.

Valid *requirement* criteria are shown in the list below. Examples of each are also included in "REQUIRE subcommand—Validating user input" on page 21.

**ALPHA** The field must only contain the characters "A" through "Z" (uppercase or lowercase) or blanks.

**ANYTHING**
Any user input is accepted, including all blanks but excluding the use of the ERASE EOF key.

**DECIMAL** *[=|<|> n1] [=|<|> n2] [...]*
The field must only contain numeric digits, "0" through "9". Optionally, specific numeric comparisons can be requested. If these exist, the field must match any equality or all inequalities. Only positive numbers are supported. For example, to require that a field contain a number from 1 to 10 or 15 or 20, you could enter the following subcommand:

```
REQUIRE field1 DECIMAL > 0 < 11 = 15 = 20
```

**ERASEEOF**
The field must have been cleared by pressing the ERASE EOF key.

This requirement is meant to be used in conjunction with other requirements, for example, to specify that the field can be cleared, but if it isn't, then it must match other criteria.

**EXEC** *execname <'string'>*
An EXEC with a filename of *execname* and a filetype of MENUEXEC is called to determine whether the given field is valid. When this EXEC is called, it is passed the name of the menu, the name of the field and the status of the field. By the use of subcommands you can determine the status of any menu field and data can be read from or written to any field on the menu. A subcommand also exits to retrieve the string passed via *string*. More information on the REQUIRE EXEC subcommand can be found in "REQUIRE macros—Validating user input" on page 27 and "REQUIRE EXEC subcommand format" on page 69.

**FLOATing** *[=|<|> n1] [=|<|> n2] [...]*

The field must only contain the numeric digits "0" through "9" with an optional leading sign ("+" or "-"), an optional fractional part (*nn.nn*), and/or an optional signed exponent (E +¢- *nn*) Specific floating point comparisons can also be requested. If these exist, the field must match anv equality or all inequalities.

Floating point numbers are converted internally and comparisons are performed using System 370 double precision format. Values exceeding this format's mantissa or exponent capacity are considered to fail the requirement criteria. For example, to require that a field contain a number from 1.5 to 6.0221694, your requirement could be coded as follows:

```
REQUIRE field1 FLOAT = 1.5 > 1.5 < 6.0221695
```

The equality and greater than comparisons of 1.5 alleviate the rounding problem inherent in coding > `1.4999...` instead.

**HEX** *[=|<|> n1] [=|<|> n2] [...]*

The field must only contain the numeric digits "0" through "9" or the letters "A" through "F" (upper- or lowercase).

Specific numeric comparisons can be requested. If these exist, the field must match any equality or all inequalities. Only positive numbers are supported. If specific comparisons are included, their values must be given in hexadecimal.

**INTEGER** *[=|<|> n1] [=|<|> n2] [...]*

The field must only contain the numeric digits "0" through "9" with an optional leading sign ("+" or "-"). Specific numeric comparisons can also be requested. If these exist, the field must match any equality or all inequalities. Both positive and negative numbers are supported. For example, to require that a field contain a number from -7 to 6 or 23, the REQUIRE subcommand could be coded as follows:

```
REQUIRE field1 INTEGER > -8 < 7 = 23
```

**NATIONAL**

The field must only contain the characters "A" through "Z" (uppercase only) or the characters "$", "@", or "#". This form can be used to check for valid CMS filenames.

**NONBLANK**

Any non-blank, non-null (non-binary zero) input is required.

**NOTHING** This requirement is met only if NO user input is entered at all.

If the modified data tag (MDT) is set for this field, then this condition can never be satisfied.

This requirement type can be used in combination with other requirement types to allow for cases where no input is necessary, but if input is entered, it must match a certain format (specified by another REQUIRE subcommand for the same field name).

**PATTERN** *'pattern'*

The field must match a supplied pattern string. *pattern* must contain a character type for each character in the field. The character types are "A" for alphabetic characters, "C" for any character allowed, "D" for decimal digits, "N" for national characters, and "X" for hexadecimal digits. Any other character found in the pattern string must EXACTLY match the character at its position in the field. Case is significant.

The length of the field's data must exactly match the length of the pattern string (after blank and null suppression). For example, the pattern "DDD-DD-DDDD" would match valid United States Social Security numbers.

**STRING** *'str1' [lth1] ['str2' [lth2]] [...]*

The field must match a specified string or one of a set of specified strings. Each string must be enclosed in apostrophes and may be followed by an optional numeric value which, if specified, sets the minimum acceptable length for the field string to match the string requirement. Case is significant. For example,

```
REQUIRE field1 STRING 'YES' 2
```

would match fields with "Y", "YE", or "YES", but not "yes" or "YESTERDAY".

**RJFILL** *character* Specifies the character to be used to fill the unused areas created by the RJUST (right justify) subcommand. If not specified, nulls (binary zeros) are used as the fill characters.

**RJUST** *fieldname1 [fieldname2 ... fieldnamen]*

Data moved into the specified fields are right-justified within those fields. The unused area of the field is filled with nulls (binary zeros) unless the RJFILL subcommand is also specified.

**STACK** *fieldname1 [fieldname2 ... fieldnamen]*

If the specified field(s) are modified by the user, their data are placed in the CMS system stack. The data is stacked in the order in which the fields are listed on the STACK subcommand.

**STRIP [L|T|B]** *['char']*

Data returned from a field to EXEC variables is stripped of leading and/or trailing characters. If "L" is specified, leading characters are stripped. "T" specifies that trailing characters are to be stripped. If nothing is specified, the first parameter defaults to "B" and both leading and trailing characters are stripped. The second parameter is also optional, and specifies the character to be stripped. If the second parameter is not specified, blanks and nulls (binary zeros) are stripped. If specified, the character to be stripped must be enclosed by apostrophes.

**TXTBORDE [BLUE|GREEN|RED|YELLOW|PINK|TURQuoise|**
**WHITE|DEFCOL]**
**[BLINK|UNDERscore|REVerse|NOHIlight]**
**[PSnn]**

Windows generated with TXTBORDE will have 327*x* TEXT borders if possible. These text borders form a solid box. If a terminal displaying a window with TXTBORDE doesn't support 327*x* TEXT characters, the window border will be made up of horizontal dashes and vertical solid lines.

With TXTBORDE, the border can optionally have the extended color, highlighting, and symbol set specified in the subcommand. If a terminal doesn't support these attributes, they are ignored.

If a programmed symbol set is specified, it must be a hexadecimal value of zero, or X'40' through X'EF', and a symbol set of the same logical number must be loaded using the LOADPS subcommand.

**UPCASE** *fieldname1 [fieldname2 ... fieldnamen]*

The contents of the field(s) specified with this subcommand are converted to uppercase before being placed in either the CMS system stack or EXEC variables. Data placed into MENUCTRL files are only converted to uppercase if the UPCASE option is used.

**UPDTFILE** *filename [***ALL***|fieldname ... fieldnamen]*

Specifies that the file *filename* MENUCTRL will be updated with the contents of fields changed by user input. Only those fields listed on the UPDTFILE command line will be placed into the MENUCTRL file. If ALL is specified, all fields changed by the user will be moved to the MENUCTRL file. If the file already contains record(s) for a changed field name, the record(s) will be updated with new contents. If the file does not yet have record(s) for a changed field, the new data record(s) will be added to the end of the file. If the file contains data record(s) for fields not defined in the menu or not changed by the menu, they will remain unchanged in the file. If the MENUCTRL file does not exist, or is not on a read/write disk, it will be created.

**VIEWPORT** *vpos hpos [voff hoff [vsiz hsiz]]*

Specifies the position, contents, and size of the window viewport used to display this menu.

*vpos* and *hpos* specify the row and column position of the viewport on the virtual screen. The upper-left corner of the

virtual screen is 0 0—row 0 (zero), column 0 (zero). These parameters must be specified.

*voff* and *hoff* specify the row and column position of the first character of the menu displayed in the viewport, with the upper-left corner of the menu, again, being row 0, column 0. If these parameters are not specified, they default to 0 0.

*vsiz* and *hsiz* specify the row and column size of the viewport. If these parameters are not specified, they default to the remainder of the menu's size, or the remainder of the virtual screen's size.

**WRITE** *menuname* **[REPlace]**

Writes a copy of the current in-storage menu back to a DASD file. If REPlace is specified, and the file *menuname* MENU exists, it is replaced.

The contents of the menu are those at the time the WRITE subcommand is encountered. Several WRITE subcommands can be passed with other intervening subcommands to create several different menus with one MENUEXEC call.

# MENUEXEC EXEC variables

All of the EXEC variables used by MENUEXEC are described in the list below. See "Using EXEC variables with MENUEXEC" on page 5 for a discussion of EXEC variable usage. These variable names are prefixed with an ampersand when used by EXEC 2 or EXEC, for example, "PFK" is coded in EXEC 2 as "&PFK".

Variables are set only in the EXEC level most recently called before calling MENUEXEC.

All of these variables can be prefixed with a string when the MENUEXEC PREFIX option is used.

| VARIABLE | ACTION |
|---|---|
| *fieldname* | Both read and set by MENUEXEC—On input, the contents of the "*fieldname*" EXEC variable replace the contents of the identically named menu field. After user input, each "*fieldname*" variable contains the contents of the identically named menu field that the user modified, or whose Modified Data Tag (MDT) was set ON.<br><br>With REXX, the contents of the field are returned exactly as entered. With EXEC 2, the data is returned exactly as entered but is truncated to its first 255 characters. With EXEC, the data returned to the variable is truncated to its first eight characters, and has its trailing blank or null (binary zero) characters suppressed.<br><br>EXEC variables relating to menu fields are ignored if either the MENUEXEC options NOXVARS or CTRLVARS are used, or if the NOXVARS subcommand was issued for this menu field.<br><br>If an EXEC variable has the same name as a named menu field, the data from that variable is moved into the menu before display. Thus, if the same menu is called more than once from the same |

EXEC, the user input from the first display is placed into EXEC variables, and is re-displayed on following menu displays (assuming the variable was not changed between MENUEXEC calls). This will occur even in two different menus that have the same named field(s). This facility provides user-input memory within an EXEC without using MENUCTRL data files.

**ACTION** Set by MENUEXEC only if the ACTION option is specified on the MENUEXEC command line—Contains the word on which the cursor was positioned when an interrupt-generating key was pressed. This variable is useful in determining the action item chosen by a user from an action bar (see "MENUEXEC and SAA Common User Access (CUA)" on page 37).

**CURCOL** Set by MENUEXEC—Contains the column number on which the cursor was located when the user completed input. Column 1 is the leftmost column of the menu. This variable may be used, for example, as the *menu-column* value of the CURSOR subcommand to reposition the cursor in the same column on a subsequent menu display.

**CURFNAM** Set by MENUEXEC—Contains the name of the field on which the cursor was positioned when the user completed input. This value can be used to position the cursor on subsequent menu displays as an argument to the MENUEXEC option CURSOR. If the field on which the cursor was positioned has no name, this variable is set to a null string.

**CURFOFF** Set by MENUEXEC—Contains the offset from the attribute character in the field on which the cursor was positioned when the user completed input. The value of this variable is returned as 1 if the cursor was in the first data item in the field (the attribute character position is returned as 0). This value can be used to position the cursor on subsequent menu displays as an argument to the MENUEXEC option CURSOR. If the CURSOR subcommand is used, this value must be decremented by 1 because the subcommand treats the field's first data position as 0. If the field on which the cursor was positioned has no name, this variable is set to a null string.

**CURLINE** Set by MENUEXEC—Contains the line on which the cursor was positioned when the user completed input. Line 1 is the top line of the menu. This variable can be used, for example, as the *menu-line* value of the CURSOR subcommand to reposition the cursor in the same column on a subsequent menu display.

**CURSOR** Set by MENUEXEC—Contains the position of the cursor at the last user input, represented as a numeric offset from the upper-left corner of the menu (position 0). This value may be passed as an argument to the MENUEXEC option CURSOR, to reposition the cursor on subsequent menu displays in the same location as at the last user input. "CURSOR" is not changed or set if the last key pressed was "CLEAR".

**CURSTR** Set by MENUEXEC only if the CURPOINT option is specified on the MENUEXEC command line—Contains the character string within the field on which the cursor was positioned when the user

completed the input. This variable is only set when running under REXX or EXEC 2. Leading and trailing nulls and blanks are suppressed from the field before the data is moved to the variable.

**CURWRD** Set by MENUEXEC only if the CURPOINT option is specified on the MENUEXEC command line—Contains the non-blank character string on which the cursor was positioned when the user completed the input. If using EXEC 2, this string is truncated to its first 255 characters; if using EXEC, the string is truncated to its first eight non-blank characters. There is no truncation performed when using REXX.

**LGHTPEN** Set by MENUEXEC only if a selector pen interrupt was detected ("PFK" = LGHTPEN)—Contains the contents of the field selected by the selector pen.

With REXX, the contents of the field are returned exactly as entered. With EXEC 2, the data is returned exactly as entered but is truncated to its first 255 characters. With EXEC, the data returned to the variable is truncated to its first eight non-blank characters.

If the interrupt was not from a selector pen, this variable is neither set nor modified.

**MENU** Set by MENUEXEC only if the SAVE option was specified on the MENUEXEC command line—Contains the XMENU-assigned internal name used to save this menu across MENUEXEC calls. Subsequent calls to MENUEXEC using this menu should also specify SAVE and pass this variable as the *menuname* parameter on the MENUEXEC call. See "Using the saved menu environment" on page 14 for more details and an example of using this variable.

**PFK** Set by MENUEXEC—Contains the name of the key pressed by the user to complete input. The value of "PFK" can be PF01 through PF36, PA1 through PA3, CLEAR, TESTREQ, ENTER, LGHTPEN, CARDRDR, SCANNER, or NOAID. Based on MENUEXEC command line options, certain non-terminal pseudo key values can be returned; these are SMSG, RDR, DEVICE, and TIMER.

When using windows, if the cursor is outside the active viewport when the user presses an interrupt-generating key, NOAID is returned as the "PFK" value.

**PFKSTR** Set by MENUEXEC if PF key strings are set with the PFKSTR subcommand—Contains the string associated with the PF key pressed. If the key pressed has no PFKSTR string associated with it, this variable will not be set by MENUEXEC.

**SMSG** Set by MENUEXEC only if the SMSG option was specified on the MENUEXEC command line and if an SMSG interrupt was received before any user input to the menu—Contains the message sent to you from another user via SMSG.

**SMSGUSR** Set by MENUEXEC only if the SMSG option was specified on the MENUEXEC command line and if an SMSG interrupt was received before any user input to the menu—Contains the userid of the user who sent you a message via SMSG.

**VARCHNG** Set by MENUEXEC—Contains the string "YES" if any data was entered by the user (or if any field was returned because its Modified Data Tag was set). If no data was entered, VARCHNG returns the string "NO". This variable is used to quickly determine whether the user has changed any input fields. If set to "NO" your EXEC may be able to save processing time by not validating previously validated, unchanged data.

If the LISTVARS command line option is specified, VARCHNG.0 contains the number of fields changed by the user, and VARCHNG.1 through VARCHNG.*n* contain the names of the menu fields changed by the user.

# MENUEXEC error messages

MENUEXEC error message codes and formats are listed below in numeric order. The message number is the same as the return code for each error message.

| NUMBER | MESSAGE |
|---|---|
| **8701E** | No menu name specified. |
| **8702E** | Cursor field name invalid or field does not exist. |
| **8703E** | STACK operand is missing or invalid - must be LIFO or FIFO. |
| **8704E** | FILE filename operand is missing or invalid. |
| **8705E** | Insufficient storage available to process MENUEXEC requests. |
| **8706E** | Error reading MENU file record 1 - examine FSREAD return code. |
| **8707E** | MENU file requested is not a properly formatted MENU file. |
| **8708E** | Control file requested would result in infinite FILE request loop. |
| **8709E** | Error reading MENU file record 2 - examine FSREAD return code. |
| **8710E** | Control file requested is not a properly formatted control file. |
| **8711E** | Continuation line encountered which was not preceded by field data. |
| **8712E** | Control file read error - examine FSREAD return code. |
| **8713E** | Field name specified is invalid or field does not exist on the menu. |
| **8714E** | CHANGE command parameter is missing or invalid. |
| **8715E** | Format error encountered in stacked data; possibly un-paired """s. |
| **8716E** | Full screen I/O error occurred - examine FSCWREAD return code. |
| **8717E** | Virtual console device must be a 327x terminal to run MENUEXEC. |
| **8718E** | CURSOR offset specified is invalid or value exceeds the field size. |
| **8719E** | Filename for updating menu control file is missing or invalid. |
| **8720E** | Control file update temporary work file already exists. |
| **8721E** | Menu control file to be updated is not properly formatted. |
| **8722E** | Disk containing old control file and the "A" disk are read/only. |
| **8723E** | Unable to write updated control file; disk is full. |
| **8724E** | MENUEXEC parameter or option is missing or invalid. |
| **8725E** | Error in write of menu control file; check FSWRITE return code. |
| **8726E** | Error in update read of old menu control file; check FSREAD RETCODE. |
| **8727E** | Field name specified in UPDTFILE command is missing or invalid. |
| **8728E** | MENU or XMENULIB file not found. |

**8729E** CURSOR position is invalid or exceeds the screen size.
**8730E** MENU file is too large for the screen.
**8731E** MENU file is old format - convert to new format to use it.
**8732E** Program function key is missing or invalid.
**8733E** Program function key string data is missing or invalid.
**8734E** CURSOR line position is less than "1" or greater than screen length.
**8735E** CURSOR column position is less than "1" or greater than screen width.
**8736E** LIB XMENULIB operand library name not specified.
**8737E** LIB option specified more than once.
**8738E** XMENULIB is not properly formatted.
**8739E** Error reading XMENULIB file - examine FSREAD return code.
**8740E** XMENULIB member MENU not found.
**8741E** Menu name does not match filename; fix with XMENU program.
**8742E** Cannot change attributes of field "zero."
**8743E** CHANGE/ADD command programmed symbol set value missing or invalid.
**8744E** Name of field to be deleted is missing or does not exist.
**8745E** Field "zero" of the menu cannot be deleted.
**8746E** XMENU system error occurred attempting to delete a field.
**8747E** Name of field to be added is missing or already exists on the menu.
**8748E** Offset of field to be added is missing or invalid.
**8749E** Column offset of field to be added is zero or > screen width.
**8750E** Row offset of field to be added is zero or > screen length.
**8751E** Offset of field to be added is zero or past end of the screen.
**8752E** A field already starts at the offset specified on the ADD command.
**8753E** XMENU system error occurred attempting to add a field.
**8754E** FILE control file operand name not specified.
**8755E** FILE option specified more than once.
**8756E** HELPPFK menu name is missing or invalid.
**8757E** HELPPFK library name is missing or invalid.
**8758E** WAIT device address is missing or invalid.
**8759E** WAIT time value is missing or invalid.
**8760E** Unable to establish SMSG environment.
**8761E** SMSG/VMCF interrupt received that was not user-requested.
**8762E** SAADD or SADELETE type is missing or invalid.
**8763E** SAADD or SADELETE location data is missing or invalid.
**8764E** SAADD or SADELETE column value invalid.
**8765E** FIELDMSG message string is missing or invalid.
**8766E** SADELETE requested for character attribute that does not exist.
**8767E** SAADD position requested falls on a field attribute character.
**8768E** Unknown return code received from MADDSA/MDELSA subroutines.
**8769E** SAADD or SADELETE location exceeds the screen size.
**8770E** REQUIRE field name is missing or invalid.
**8771E** REQUIRE type is missing or invalid.
**8772E** REQUIRE specified twice for the same field.
**8773E** FIELDMSG field name is missing or invalid.
**8774E** FIELDMSG specified twice for the same field.
**8775E** REQUIRE PATTERN is missing or invalid.
**8776E** REQUIRE STRING is missing or invalid.
**8777E** REQUIRE DECIMAL|HEX|INT|FLOAT type must be "=," ">," or "<."
**8778E** REQUIRE DECIMAL|HEX|INT|FLOAT value is missing or invalid.
**8779E** REQUIRE STRING minimum length value invalid.
**8780E** EMSGFLD field name is missing or invalid.
**8781E** LOADPS logical symbol set number is missing or invalid.

| | |
|---|---|
| **8782E** | LOADPS symbol set filename is missing or invalid. |
| **8783E** | LOADPS symbol set file does not exist. |
| **8784E** | LOADPS symbol set is not formatted properly. |
| **8785E** | I/O error loading programmable symbol set. |
| **8786E** | Use of the wait option requires "SET ECMODE ON." |
| **8787E** | RJFILL/LJFILL character is missing or invalid. |
| **8788E** | OUTPUT filename operand is missing or invalid. |
| **8789E** | OUTPUT type operand is missing or invalid. |
| **8790E** | Menu name passed with SAVE option is not already loaded. |
| **8791E** | WRITE menu name is missing or invalid. |
| **8792E** | WRITE option not REPlace. |
| **8793E** | WRITE failed, examine MWRITE return code. |
| **8794E** | WRITE failed, menu already exists. Erase or specify "REPlace." |
| **8795E** | REQUIRE EXEC statement is missing the EXEC name. |
| **8796E** | REQUIRE EXEC - EXEC (with file type MENUEXEC) not found. |
| **8797E** | EXEC variable prefix string not specified. |
| **8798E** | Window name is missing. |
| **8799E** | Window name was specified twice. |
| **8800E** | Virtual screen vertical size is missing or invalid. |
| **8801E** | Virtual screen horizontal size is missing or invalid. |
| **8802E** | Window vertical offset on virtual screen is invalid. |
| **8803E** | Window horizontal offset on virtual screen is invalid. |
| **8804E** | Menu vertical offset on viewport is invalid. |
| **8805E** | Menu horizontal offset on viewport is invalid. |
| **8806E** | Viewport vertical size is invalid. |
| **8807E** | Viewport horizontal size is invalid. |
| **8808E** | Virtual screen already exists. |
| **8809E** | Virtual screen size(s) are zero, or too big (>2**32). |
| **8810E** | Insufficient storage to allocate the virtual screen. |
| **8811E** | Window's vertical size is bigger than virtual screen's vsize. |
| **8812E** | Window's horizontal size is bigger than virtual screen's hsize. |
| **8813E** | Error loading menu - check MLOAD/MLOADW return code. |
| **8814E** | Window positioning vertical value/size too large. |
| **8815E** | Window positioning horizontal value/size too large. |
| **8816E** | Invalid value passed to window definition routines. |
| **8817E** | Error positioning menu - check MPOSW return code. |
| **8818E** | BORDER subcommand type is missing. |
| **8819E** | BORDER subcommand type is invalid. |
| **8820E** | BORDER subcommand attribute definitions are missing. |
| **8821E** | BORDER subcommand invalid character data passed. |
| **8822E** | BORDER subcommand remainder of graphic escape is missing. |
| **8823E** | BORDER subcommand attribute value is too large. |
| **8824E** | BORDER subcommand attribute value is missing. |
| **8825E** | BORDER subcommand invalid symbol set value. |

# REQUIRE EXEC macro facility

The REQUIRE EXEC macro facility allows you to generate your own field requirement/validity checking routines using REXX or EXEC 2 macros. These files have the filetype MENUEXEC.

After the user has entered data in the menu and pressed an interrupt-generating key, MENUEXEC calls each of your REQUIRE EXEC macros. These macros function

like XEDIT macros—they can call certain MENUEXEC REQUIRE subcommands or CP and CMS commands (including calls to another level of MENUEXEC). Two examples are included in the Appendix. Refer to "Sample REQUIRE macro to check for YES or NO input" on page 101 and "Sample REQUIRE macro to check for a valid date" on page 103.

## REQUIRE EXEC subcommand format

Because a field can have multiple REQUIRE specifications, REQUIRE EXEC macros are called after other REQUIRE specifications are checked. To call a REQUIRE EXEC macro to perform validity checking, use this form of the MENUEXEC REQUIRE subcommand:

```
REQUIRE fieldname EXEC execname ['string']
```

*execname* The filename of the EXEC to be run (its filetype is MENUEXEC).

*string* An optional string, whose case is significant. *string* is contained in apostrophes. Any apostrophe that should appear within *string* must be represented by two apostrophes, for example, 'Sharky''s Day'.

## Arguments passed to REQUIRE EXEC macros

When a REQUIRE EXEC macro is called, parameters are passed as its argument string. The following arguments are passed:

**1st** Name of the menu
**2nd** Name of the field
**3rd** Status of the field:

| | |
|---|---|
| **NOINPUT** | No user input in the field |
| **INPUT** | Non-zero length user input |
| **EOF** | Zero length user input |

## REQUIRE EXEC macro subcommands

The subcommands provided allow you to retrieve the status of or the data in any menu field. You can also write data back to the menu in either the output area or as simulated user input.

The following subcommands are supported in REQUIRE EXEC macros:

**ALARM** Causes the alarm to sound if this EXEC exits with a match-failed return code. This is the default.

**CHANGE** *fieldname* **PROTect|UNPROTect|NUMERIC]**
**[BRIGHT|DIM|DARK|LGHTPEN]**
**[MDT|NOMDT]**
**[SKIP|NOSKIP]**
**[BLUE|GREEN|RED|YELLOW|PINK|TURQuoise|**
**WHITE|DEFCOL]**
**[BLINK|UNDERscore|REVerse|NOHIlight]**
**[PSnn]**

The *fieldname* specified in the macro subcommand will have its attributes changed the next time the menu is displayed. Any number of attribute value changes for this field can be placed in the subcommand. If multiple attribute definitions for one attribute type exist, the last one specified is used. If a programmed symbol set is specified, it must be a hexadecimal value of zero, or X'40' through X'EF' (X'F1' cannot be specified for a field attribute). A symbol set having the same logical number must be loaded using the LOADPS MENUEXEC subcommand, or unpredictable results may occur.

**CURSOR** *fieldname [offset]*

Causes the cursor to be explicitly positioned within a field. If *offset* isn't specified, the cursor is placed on the first character in the field.

If you position the cursor with this subcommand, the position takes precedence over any automatic positioning by MENUEXEC. If you don't position the cursor, it is placed on the first field to fail a REQUIRE condition.

**GET** *exec-variable-name*

Move the contents of the *string* passed on the REQUIRE EXEC subcommand to the specified EXEC variable. This can be used to generate general-purpose REQUIRE EXEC macros. For example, the string can pass a PL/I-like picture statement.

**NOALARM** Causes the alarm to not sound if this EXEC exits with a match-failed return code.

**PUTIN** *fieldname exec-variable-name*

Move the contents of the specified EXEC variable to the specified menu input field, simulating user input into the field. This can be used to set input based on the contents of a field, for example, to convert an abbreviation to the complete string.

**READ** *fieldname exec-variable-name*

Move the contents of the specified field to the specified EXEC variable. This must be used to get the value of the REQUIREd field, whose name is passed as the second (*fieldname*) argument on the REQUIRE EXEC subcommand, however ANY field's value can be transferred.

**STATE** *fieldname exec-variable-name*

Move the status of *fieldname* to the specified EXEC variable. The possible values of status are:

| | |
|---|---|
| **NOINPUT** | No user input in the field |
| **INPUT** | Non-zero length user input |
| **EOF** | Zero length user input |

**WRITE | PUTOUT** *fieldname exec-variable-name*

Move the contents of the specified EXEC variable to the specified menu output field. This macro subcommand can be used to display error messages.

If you write to the ERRMSG field (by default named $ERRMSG), this message will take precedence over any other MENUEXEC-generated error message.

Subcommands not recognized by MENUEXEC as valid are passed to CMS or CP for execution.

## Return codes from REQUIRE EXEC subcommands

The following codes can be returned from a REQUIRE EXEC subcommand:

| | |
|---|---|
| **0** | Transfer successful. |
| **1** | Field name not passed. |
| **2** | Field name is not in the menu or the name is invalid. |
| **3** | EXEC variable name not passed. |
| **4** | EXEC variable name does not exist or the name is bad. |
| **5** | (GET) No string passed on REQUIRE statement. |
| **8** | Insufficient storage to preform EXEC variable operation. |
| **10** | (CURSOR) Cursor offset not numeric. |
| **11** | (CHANGE) Attribute definition missing. |
| **12** | (CHANGE) Attribute definition invalid. |
| **13** | (CHANGE) Symbol set number missing. |
| **14** | (CHANGE) Symbol set number invalid. |
| **24** | Not running under an EXEC. |
| **28** | Insufficient storage to preform EXEC variable operation. |
| **32** | The EXEC variable was not found. |
| **36** | The EXEC variable name is invalid. |
| **-11** | Insufficient storage to run the EXEC. |

## Return codes you should set when exiting a REQUIRE EXEC macro

The code you return to MENUEXEC is significant. The following is a list of the return codes you should pass back to MENUEXEC from your REQUIRE EXEC macro:

**0** REQUIRE conditions met—Screen NOT changed by the EXEC, or anything the EXEC called. (This pertains to full-screen writes only.)

**1** REQUIRE conditions met—Screen WAS changed by the EXEC, or anything the EXEC called. (This pertains to full-screen writes only.)

**2** REQUIRE conditions NOT met—Screen NOT changed by the EXEC, or anything the EXEC called. (This pertains to full-screen writes only.)

**>2** REQUIRE conditions NOT met—Screen WAS changed by the EXEC, or anything the EXEC called. (This pertains to full-screen writes only.)

**-10** EXIT MENUEXEC immediately (without any output processing whatsoever). This can be used to exit quickly from a REQUIRE EXEC-based hierarchy of menus. Your REQUIRE EXEC(s) should check for this code passed by any subordinate EXEC and should propagate it backward by exiting with the -10 code.

**-11** Insufficient storage to run the EXEC (If you get this from a subcommand, pass it through to MENUEXEC).

# Chapter 4. MENUCTRL

MENUCTRL is a utility that runs from within an EXEC and allows you to load or save EXEC variables to or from MENUCTRL files.

The format of the MENUCTRL command is shown below:

| | |
|---|---|
| MENUCTRL | FILE *filename* ALL\|*name1* [*name2 ... namen*]<br>UPDTFILE *filename name1* [*name2 ... namen*]<br>? |

## Where

**FILE** Requests that a MENUCTRL file be read, and data be moved to EXEC variables.

**UPDTFILE** Requests that data from EXEC variables be moved to a MENUCTRL file.

*filename* Specifies the name of the MENUCTRL file. Files must be in the following format to be used by MENUEXEC, or the MGFILE and MUFILE subroutines: fixed length (RECFM F), LRECL 80, with the MENUEXEC subcommand MENUCMDS entered in the first column of the first record.

**ALL** Specifies that ALL data records found in the MENUCTRL file should be moved to EXEC variables. ALL is only valid for reading MENUCTRL files; that is, it cannot be used with UPDTFILE.

*name1 name2 ... namen*
Specifies the name(s) of EXEC variables. To maintain compatibility with menu field names, these names should not exceed seven characters.

**?** Types a short description of MENUCTRL command options to your terminal.

## Usage notes

MENUCTRL is used to load data from a MENUCTRL file before any menu is displayed. You do not have to be connected to a 3270-type terminal to run MENUCTRL.

An example of MENUCTRL use is in an EXEC where users have the choice of displaying a menu showing available parameters, or choosing to reuse the parameters entered from a previous invocation. To implement such an application, you would do the following:

1. Call MENUCTRL to initialize the appropriate EXEC variables. If the MENUCTRL file does not exist, display the menu (step 4).
2. Determine whether the user wants to use these parameters or wishes to modify them. This decision could be based on a command line parameter passed when the EXEC was called and/or the contents of one of the EXEC variables loaded with MENUCTRL.
3. If the user wishes to reuse the previous variables, continue processing (step 6).
4. If the user wishes to create or modify parameters, call MENUEXEC to display a menu. Needless to say, the names of the EXEC variables in question should match the field names in the displayed menu. The MENUEXEC FILE option or subcommand is not necessarily required, because MENUCTRL has already loaded the EXEC variables.
5. If the user wishes to save the modifications, call MENUCTRL to write the variables back to the MENUCTRL file. For performance reasons, you would do this as infrequently as possible (for example, after the input has been verified, or only prior to exiting the EXEC).
6. Process the user request, then exit (or redisplay the menu for a subsequent request).

MENUCTRL ignores MENUEXEC subcommands found in a MENUCTRL file. Hierarchies of MENUCTRL files can be processed by calling MENUCTRL several times, in increasing order of precedence; for example, if you have a system MENUCTRL file, a group MENUCTRL file, and a user MENUCTRL file, you would load them in this order. Recursion is not possible since MENUEXEC FILE subcommands within the MENUCTRL files are ignored.

## Return codes and messages

| | | |
|---|---|---|
| **1** | 8350E | MENUCTRL command must be FILE or UPDTFILE. |
| **2** | 8362E | No MENUCTRL filename specified. |
| **3** | 8358E | Insufficient storage to run MENUCTRL. |
| **4** | 8360E | No variable names supplied on the command line. |
| **5** | 8359E | "ALL" cannot be specified with "UPDTFILE." |
| **6** | 8361E | Not running under an EXEC. |
| **7** | 8357E | MENUCTRL file has bad format. |
| **9** | 8351E | Some data record in MENUCTRL file has invalid format. |
| **11** | 8354E | Variable name in MENUCTRL file is invalid. |
| **13** | 8355E | Update MENUCTRL temporary file already exists. |
| **28** | 8356E | MENUCTRL file not found. |
| **xx** | 8352E | Error reading file - check return code for FSREAD code. |
| **xx** | 8355E | Error in writing MENUCTRL file - check FSWRITE code. |
| **100** | | (Return code following HELP messages.) |

# Chapter 5. MENUEXAM

The MENUEXAM utility is used from within an EXEC to generate page-mode menus for use by MENUEXEC. Page-mode menus are generated to the size of the terminal on which they are being displayed.

The format of the MENUEXAM command is shown below:

| | |
|---|---|
| MENUEXAM | *fn [ft [fm]]* |

## Where

*fn* The filename of the menu definition file.

*ft* The filetype of the menu definition file. If not specified, it defaults to MENUDEF.

*fm* The filemode of the menu definition file. If not specified, it defaults to *.

## Usage notes

MENUEXAM generates a page-mode menu based on the definition in the menu definition file, and saves it in the MENUEXEC saved menu environment for use by subsequent MENUEXEC calls.

Page-mode menus are dynamically created menus that contain zero or more top title lines and bottom title lines, an optional command line, and some number of fields arranged in rows and columns. Page-mode menus are dynamically generated to fit the size terminal on which they are being displayed.

MENUEXAM can be used effectively in environments having many different terminal screen sizes and types. With MENUEXAM you can create one menu for use by all terminal screens more quickly than you could create a menu for each screen size and save them in an XMENULIB for conditional output. You can even create a MENUEXAM menu whose field's input could be used to generate other page-mode menus, thereby creating an application that generates its own menus.

MENUEXAM lends itself best to row and column-type applications such as order forms, where column headings can be positioned side-by side on a top row to serve as field captions for the data entry fields positioned in the columns below, for example:

```
Quantity   Stock #   Description   @ Price   Total Price
```

# MENUEXAM EXEC variables

MENUEXAM automatically creates the following EXEC variables:

**menu** Set to the MENUEXEC saved environment menu number. This variable is passed in the MENUEXEC command line as the menu name. The MENUEXEC SAVE option must also be specified.

**amcols** Set to the number of columns generated in the menu. This can be used to determine how much data to move to the menu.

**amrows** Set to the number of rows generated in the menu. This can be used to determine how much data to move to the menu.

# MENUEXAM definition file subcommands

The MENUEXAM definition control file contains commands used to define the page-mode menu. Each command is specified on a seperate record in the file. The file must have a record size of 256 or less, and can be fixed or variable format. Commands and operands can be specified starting in any column within a record. Any record starting with an asterisk (*) is treated as a comment.

An example of a MENUEXAM definition file is shown below, followed by a list of the MENUEXAM subcommands.

**SAMPLE MENUDEF A**

```
FLAG MAPGCCOL
TOP 1 'Customer Order Entry Form' WHITE
TOP 2 ' '
TOP 3 ' '
FIELD TOP 3 CUSTNAME 20 TURQ UNDERSCORE
COLUMN 1 20 TURQ UNDERSCORE
COLUMN 2 6 TURQ UNDERSCORE
COLUMN 3 8 PROT
CMDLINE 1 COMMAND===>
BOTTOM 1 'PF1=HELP  PF3=QUIT  PF5=TOTAL  PF12=FILE'
END
```

**BOTTOM** *line ['text'] [attributes]*

Defines a bottom title line. *line* is the number of the bottom line. If *text* is present, it must be enclosed in apostrophes. *attributes* define the field's initial attribute characteristics, for example, blue, underscore, and unprot.

When the menu is created, each bottom line is named BOT.1, BOT.2, BOT.3, and so on. BOT.1 is the topmost bottom line, BOT.2 is below BOT.1, BOT.3 is below BOT.2, and so on. You specify each bottom line's location on the menu with the *line* number parameter of this subcommand.

**CMDLINE** *[number [prefix|suffix [text]]]*

Defines a command line. *number* specifies the number of lines reserved for the command line; the default is 1 (one). *prefix|suffix* specifies whether a command prefix area exists,

and whether it precedes or follows the command line. *text* is the text of the prefix area. If *text* is specified, but *prefix|suffix* isn't, the text precedes the command line.

When the menu is created, the command line is named CMDLINE, the command prefix area is named CMDPREF.

**COLSTART** *number* Specifies a number relating to the first character used for column field names. A number of 0 (zero) creates columns starting with the letter A, 1 (one) starts with B, 2 (two) starts with C, etc.

**COLUMN** *number length [attributes]*

Defines a column. *number* is the number of the column, starting with column 1 (one) on the left. *length* is the length of the column in number of characters. *attributes* define the field's initial attribute characteristics, for example, blue, underscore, and unprot.

Each field in the column area is automatically named with letters for the column and numbers for the row: for example, the topmost left cell is A.1.; B.1 is to the right of A.1; and A.2 is below A.1.

**END** Explicitly ends the menu definition file. No lines are scanned past the END command.

**FIELD TOP|BOTTOM** *number name offset [attributes]*

Defines a field within a top or bottom title. TOP or BOTTOM and *number* specifies the top or bottom title line. *name* is the field name for this field. For unnamed fields, enter an asterisk (*) or a period (.). *offset* is the numeric offset into the line where the beginning of the field is located. *attributes* define the field's initial attribute characteristics, for example, blue, underscore, and unprot.

FIELD records must follow the TOP or BOTTOM record they modify, although they do not have to immediately follow the TOP or BOTTOM record being modified.

**FLAG MPAGCTOP|MPAGCCOL|MPAGSPTL|MPAGDEFS**

Specifies flags used to create the menu. MPAGCTOP specifies that the command line should follow the top title line(s). MPAGCCOL specifies that the column(s) should be centered in the menu. MPAGSPTL specifies that the command line(s) should be placed above the last top title line (to use the last title line as headings). MPAGDEFS specifies that this menu should be created using the default terminal size (rather than the largest size the terminal supports).

**ROWSTART** *number* Specifies a number relating to the first character used for row field names. A *number* of 0 (zero) creates a row field name starting with the number 1, 1 starts with 2, 2 starts with 3, etc.

**TOP** *line ['text'] [attributes]*

Defines a top title line. *line* is the number of the top line. If *text* is present, it must be enclosed in single quotes. *attributes* define the field's initial attribute characteristics, for example, blue, underscore, and unprot.

When the menu is created, each top line is named TOP.1, TOP.2, TOP.3, and so on. TOP.1 is the topmost top line, TOP.2 is below TOP.1, TOP.3 is below TOP.2, and so on. You specify each top line's location on the menu with the *line* number parameter of this subcommand.

## Return codes and messages

Each message below sets the same return code as its message number.

| Number | Message |
|---|---|
| **8375E** | MENUEXAM was unable to use the MENUEXEC saved menu feature. |
| **8376E** | MENUEXAM must be run under an EXEC. |
| **8377E** | An error occurred attempting to write into an EXEC variable. |
| **8378E** | The MENUDEF file name was not passed. |
| **8379E** | The MENUDEF file must have a record size of 256 or less. |
| **8380E** | The MENUDEF file contains an unknown command. |
| **8381E** | An invalid column number was found in a MENUDEF file command. |
| **8382E** | There is insufficient virtual storage available to run MENUEXAM. |
| **8383E** | A bad or missing decimal value was found in a MENUDEF file command. |
| **8384E** | A bad or missing parameter was found in a MENUDEF file command. |
| **8385E** | A bad attribute was found in a MENUDEF file command. |
| **8386E** | A bad flag value was found in a MENUDEF file FLAGS command. |
| **8387E** | No column definitions were found in the MENUDEF file. |
| **8388E** | The MENUDEF command file cannot be found. |
| **8389E** | An error occurred reading the MENUDEF file. Check the CMS FSREAD return code. |
| **8390E** | Your terminal must be a 3270 to run MENUEXAM. |
| **8391E** | There is insufficient virtual storage to generate a menu. |
| **8392E** | The number of title lines, command lines and bottom lines is greater than the number of lines on the terminal. |
| **8393E** | The column length(s) exceed the terminal's line size. |
| **8394E** | A specific row or column starting number is too large. |
| **8395E** | A multiple field title line or bottom line is bigger than the terminal's linesize. |
| **8396E** | An unexpected return code from MPGINT was received. Please notify Relay. |
| **8397E** | Line x was the last line read from the MENUDEF command file. |

# Chapter 6. KOLVSUB

The KOLVSUB utility is used from within an EXEC to read a CMS file, perform symbol substitution based on EXEC variable values, and create a delta file.

The format of the KOLVSUB command is shown below:

| | |
|---|---|
| KOLVSUB | *fn ft fm ofn*\|= [*oft*\|= [*ofm*\|=]] [ ( *options* ]<br>*fn ft fm* PRinter<br>*fn ft fm* PUnch<br><br>**Options:** APPEND<br>CONCHAR *char*<br>DROP<br>MAXLEN *length*<br>REP\|REPLACE<br>VARCHAR *char* |

## Where

*fn* — The filename of the input file.

*ft* — The filetype of the input file.

*fm* — The filemode of the input file.

*ofn* |= — The filename of the output file. If = is specified, the name is the same as the name of the input file.

*oft* |= — The type of the output file. If not specified, or if = is specified, the type is the same as the type of the input file.

*ofm* |= — The mode of the output file. If not specified, or if = is specified, the mode is the same as the mode of the input file.

**PRinter** — Specifies that the output goes to the virtual printer.

**PUnch** — Specifies that the output goes to the virtual punch.

Options:

**APPEND** — Specifies that if the output file already exists, KOLVSUB append to the end of it. The appended file must match the record format and the logical record length of the existing output file.

**CONCHAR** *char* — Specifies the character used for concatenation, that is the character used to concatenate a variable substitution to following text. If not specified, *char* defaults to a dash ("-").

**DROP** — Specifies that if an input line substitutes to all blanks, it should not be written to the output file. This can be used to trim blank lines out of mailing addresses.

**MAXLEN** *length* Specifies the maximum output record length. If not specified, *length* defaults to the logical record length of the input file. *length* cannot be smaller than the input file's logical record length.

**REP|REPLACE** Specifies that if the output file already exists, KOLVSUB replaces it.

**VARCHAR** *char* Specifies the character used to identify symbolic variables. This character is used at the front of each symbol. If not specified, *char* defaults to an ampersand ("&").

## Usage notes

This command allows you to generate one or many files based on a skeleton input file, for example, JCL, mailing letters, mailing labels, forms, and so on.

This command is designed to be used in conjunction with other XMENU utilities and components, such as MENUEXEC (to interactively prompt for substitution data in an EXEC), or XMENU/SQL, (to retrieve data from an SQL/DS database).

Each line of the input file is scanned for symbolic variables, from right to left. When a variable is found, it is substituted, and the line is rescanned. Thus, multiple substitution is supported. A symbolic variable begins with the VARCHAR and ends in either a blank or the CONCHAR.

## Return codes and messages

In general, the KOLVSUB return code is the same as the message numbers listed below.

8200E KOLVSUB must run under an EXEC.
8201E Insufficient KOLVSUB parameters specified.
8202E Too many parameters specified to KOLVSUB.
8203E Invalid KOLVSUB option specified.
8204E The input file cannot be found.
8205E The output file already exists. Use "REP" to replace it.
8206E The output file cannot be the same as the input file.
8207E The VARCHAR option's character is missing.
8208E The CONCHAR option's character is missing.
8209E The MAXLEN value is either missing, not numeric or > 65535.
8210E Substitutions caused a line to exceed maximum size.
8211E There is insufficient virtual storage to run KOLVSUB.
8212E The MAXLEN value passed is smaller than the input file's LRECL.
8213E An invalid variable name was found in the input file.
8214E The appended file must match the input file's RECFM and LRECL.
8215E File read error - check the CMS FSREAD return code.
8216E File punch error - check the CMS PUNCHC return code.
8217E File print error - check the CMS PRINTL return code.
8218E Your disk is full, KOLVSUB output is terminated.
8219E File write error - check the CMS FSWRITE return code.
8220E The input file's minidisk is not accessed.
8221E The output file's minidisk is not accessed.

# Appendix A. Sample XMENU/REXX programs

This section includes several REXX programs that highlight various functions you may wish to incorporate into your own XMENU/REXX programs. These sample programs along with their menus are included on the product tape.

## Basic menu display program

This is a simple example that illustrates the basic structure of that part of a REXX EXEC used to display and process an XMENU screen.

```
/* REXX1 EXEC */
          address command
          clear='CLEAR'
/* 1. */  do forever
/* 2. */  "MENUEXEC REXX1 ( EMSG PA1 MAP" clear
          clear=''
/* 3. */     select
/* 4. */      when PFK="PF03" then leave
              otherwise
             end
          end
          exit rc
/* 1. REXX instruction to process an infinite loop.

   2. First call to MENUEXEC to display the initial screen.

      EMSG  Provides complete error messages for debugging.
      PA1   Makes sure that if the PA1 key is pressed the console
            does not enter CP READ.  This can be confusing to a user.
      MAP   Returns only PF keys in the range 1-12.  Keys 13-24 and
            25-36 are mapped into this lower range.
      CLEAR Avoids the "MORE..." prompt.  This is only required
            the first time full-screen mode is entered.  CLEAR
            is set as a variable prior to the first MENUEXEC call
            and changed to nulls for subsequent calls in the loop.

      Data entered into any symbolically-named fields on the
      menu is returned to the EXEC in a REXX variable of the
      same name.

   3. Process the results of the user's input.  This could be done here
      using simple IF statements; however, select is more generalized
      since one of many values could be returned.

   4. Test the variables returned from MENUEXEC.  The variable PFK is assigned
      a character string corresponding to the key that was pressed.
      Refer to "Using EXEC variables with MENUEXEC" on page 5 to see what other
      possible variables can be returned.
*/
```

## Process a single selection field

This EXEC illustrates how to manipulate a selection menu with a single selection field.

```
/* REXX2 EXEC */
         address command
         $date=DATE('U')
         $time=TIME()
         do forever
            emsg=' '
            clear='CLEAR'
            "MENUEXEC REXX2 ( MAP EMSG PA1" clear
            clear=''
/* 1. */    select
             when select="1" then call PRI
             when select="2" then nop
             when select="3" then nop
             when select="4" then nop
             when select="5" then nop
             when select="6" then call NH
             when select="7" then nop
             when select="8" then nop
             when select="9" then exit 0
             otherwise emsg="Invalid selection."
            end
            $time=TIME()
         end
/* 2. */ PRI:
         return
         NH:
         exit 0


/* 1.  Process the contents of the selection field called "SELECT".
       Call the appropriate subroutine based on its value.
       If the value is not recognized, display an error message.

   2.  Called subroutines.  These could be external EXECs or MODULEs.
*/
```

## Determine the cursor position

This EXEC illustrates how to determine the cursor position.

```
/* REXX3 EXEC */
         address command
/* 1. */ $date=DATE('U')
         $time=TIME()
         clear='CLEAR'
         do forever
         "MENUEXEC REXX3 ( MAP EMSG PA1 CURPOINT" clear
            emsg=' '
            clear=' '
/* 2. */    say 'Cursor word='curwrd
            say 'Cursor string='curstr
            say 'Cursor screen offset='cursor
            say 'Cursor row/column='curline curcol
            say 'Cursor fieldname/offset='curfnam curfoff
/* 3. */    select
             when curfnam="PRI" then call PRI
             when curfnam="NH" then  call NH
             when curfnam="EXIT" then  exit 0
             otherwise emsg="Unrecognized command."
            end
            $time=TIME()
         end
/* 4. */ PRI:
         return
         NH:
         return

/* 1. Display date and time on the screen.

   2. When a selection is made, MENUEXEC returns:

      CURWRD - Character string where the cursor is positioned.
      CURSTR - Field contents where cursor is positioned.
      CURSOR - Numeric offset from top left.
      CURLINE- Menu line number.
      CURCOL - Menu column number.
      CURFNAM- Field name.
      CURFOFF- Offset into field.

   3. Check the field name where the cursor is positioned.
      If it is not in the list, issue a message.

   4. Called subroutines.  These could be external EXECs or MODULEs.

*/
```

## Use cursor selection fields

This EXEC illustrates how to use cursor selection fields.

```
/* REXX4 EXEC */
         address command
/* 1. */ n.1="PRI"
         n.2="NH"
         n.3="NH"
         n.4="NH"
         n.5="NH"
         n.6="NH"
         n.7="EXIT"
         $date=DATE('U')
         $time=TIME()
         s.="?"
         emsg=''
         clear='CLEAR'
         do forever
/* 2. */    j=0
/* 3. */    e=0
            "MENUEXEC REXX4 ( MAP EMSG PA1 " clear
            clear=''
/* 4. */    do i=1 to 7
/* 5. */       if s.i=">" then,
/* 6. */          if j=0 then j=i
               else do
/* 7. */          emsg="More than one selected."
/* 8. */          e=1
               end
            end
/* 9. */    if j=0 then emsg="Please make a selection."
/* 10.*/    else if ¬e then do
/* 11.*/       interpret call n.j
               emsg="Processing successful."
            end
/* 12.*/    s.="?"
            $time=TIME()
         end
/* 13.*/ PRI:
         return
         NH:
         return
         EXIT:
         exit 0


/* 1.  Set an array to the names of the subroutines.

   2.  Initialize an index into the name table.

   3.  Initialize an error flag to false.

   4.  Use an index variable "I" to search each field on the screen.

   5.  Set the table index if it has not already been set.
```

```
   6.  If the table index has already been set this is an error.

   7.  Create the error message.

   8.  Set a flag for later.

   9.  Finished checking.  If "J" is 0, none was selected, so
       display an error message.

   10. Otherwise check to see if too many have been selected.

   11. Index into the name table to make the call to the routine.

   12. Reset the selection character.

   13. Called subroutines.  These could be external EXECs or MODULEs.
*/
```

## Use saved menus

This EXEC illustrates the use of saved menus. Saved menus are preserved in CMS storage across calls to MENUEXEC. Performance is improved because only changed fields are resent to the screen, and no file access is needed or performed to read the menu file every time MENUEXEC is invoked.

Field contents and attributes are *not* reset to initial values.

```
/* REXX5 EXEC */
         address command
         $date=DATE('U')
         $time=TIME()
/* 1. */ menu="REXX5"
         clear='CLEAR'
         do forever
/* 2. */ "MENUEXEC" menu  "( SAVE MAP PA1 EMSG" clear
         clear=' '
            if pfk="PF03" then exit 0
            $time=TIME()
         end


/*
   1.  The variable "MENU" is assigned a value corresponding
       to the name of the menu to be displayed.

       The SAVE option causes MENUEXEC to save the menu in storage
       and assign its own value to the variable "MENU" to be able
       to identify it.

   2.  Subsequent calls to MENUEXEC cause the saved copy to be
       executed.  The screen does not flash.  Menu field values
       are preserved in the saved copy as well as in REXX variables.

       The saved copy is not purged unless the MENUEXEC PURGE
       option is used, or a separate call to XMENUINS is
       made with the PURGE option, or a return is made to the
       CMS command line.

       Be careful when using the SAVE option of MENUEXEC from a sub-
       command environment like XEDIT.  Virtual storage within your virt-
       ual machine can be used up by subsequent reinvocation of this EXEC
       without a PURGE being issued or a return to the CMS command line.

       MENUEXEC cleans up memory occupied by these menus when CMS
       end command processing is performed.  End command processing
       is performed immediately prior to the "Ready;" prompt being
       displayed on your terminal screen.
*/
```

## Add fields to an existing menu, create new menu

This EXEC is a specialized application that can be used to build screen/application generators.

This example shows you how to perform the following functions:

- Add fields and attributes to an existing menu
- Use character attributes for emphasis
- Create a new menu

```
/* REXX6 EXEC */
         trace o
         O1 = 'Name:'
         O2 = 'Address:'
/* 1. */ title = 'A Title Line'

/* 2. */ "MENUEXEC REXX6 ( CLEAR EMSG SKIP SUBCMDS"
/* 3. */ address "MENUCMDS"
/* 4. */ "ADD TITLE 1 20 PROT DIM NOMDT TURQUOISE"
/* 5. */ "ADD TEND  1 48 PROT DIM NOMDT"
/* 6. */ "ADD O1 3 4 PROT DIM NOMDT YELLOW"
         "ADD I1 3 10 UNPROT DIM MDT WHITE UNDERSCORE"
         "ADD E1 3 31 PROT DIM NOMDT YELLOW"
/* 7. */ "ADD O2 4 1 PROT DIM NOMDT YELLOW"
         "ADD I2 4 10 UNPROT DIM MDT WHITE UNDERSCORE"
         "ADD E2 4 36 PROT DIM NOMDT YELLOW"
/* 8. */ "ADDSA REVERSE I1"
/* 9. */ "ADDSA DEFAULT E1"
/* 10.*/ "CENTER TITLE"
/* 11.*/ "CENTFILL -"
/* 12.*/ "CURSOR I1"
/* 13.*/ "WRITE NEWMENU REPL"
/* 14.*/ "MENUEND"
         address command
         exit rc


/*
  1.  Place some data in the TITLE field.

  2.  Call MENUEXEC to read the commands and display the
      screen.  When the user presses any interrupt-producing
      key, MENUEXEC will exit and create the new menu.

  3.  Start the MENUEXEC subcommand environment.

  4.  Make a new field called TITLE at row 1, column 20
      with attributes of PROTected, DIM, NOMDT, and TURQUOISE.

  5.  Add another field called TEND to end the title field.

  6.  Add a captioned field on line 3.

  7.  Add another captioned field on line 4.
```

```
   8. Place a character attribute at the start of the input field
      to change the way it is displayed to reverse video.

   9. Reset the attribute at the end of the input field.

   10. Center the data in the TITLE field.

   11. Fill any unused portions of centered text with hyphens.

   12. Place the cursor in the first input field.

   13. Create a new menu file called NEWMENU MENU A and replace
       a previous one if it existed.

   14. Mark the end of the MENUEXEC subcommand environment.

*/
```

## Print a screen to a SCRIPT file

This is an example using MENUEXEC to print a screen to a SCRIPT file.

```
          /* REXX7 EXEC */
          address command

/*1. */  parse upper arg fn "( LIB" lib

          if fn='' then exit 100

          if lib¬='' then lib="LIB" lib

/* 2. */ "MENUEXEC" fn "( CLEAR MAP PA1 SUBCMDS" lib

          address "MENUCMDS"

/* 3. */ "OUTPUT" fn "SCRIPT"

/* 4. */ "PRINT PF01 PF02 PF03 PF04 PF05 PF06 PF07 PF08",
                "PF09 PF10 PF11 PF12 ENTER"

          "MENUEND"

          address command

          exit rc


/* 1. You must include the name of an XMENU menu as an argument
      when invoking this EXEC.

   2. Display the screen.  When the user presses any function key,
      the menu will be printed to the file call "fn SCRIPT A1".

   3. Direct printed output to a SCRIPT file.

   4. Set every function key to cause a print to be made.

*/
```

## Capture terminal data and select records for display

This EXEC illustrates how to use XMENU to write a simple application that captures data from the terminal, stores it in a file, and selects records for display.

```
/* REXX8 EXEC */
address command
menu="REXX8"                                    /* Name of the menu        */
"CONTYPE"                                       /* Running on a 327x?      */
if rc=0 then do                                 /* Not if the rc is zero   */
   say "Not running on a 327x type terminal."
   exit 200
end
call MENUSQ                                     /* Call the sub for cmds   */
$date=DATE('U')                                 /* Put today's date in menu*/
"MENUEXEC" menu "( CLEAR MAP PA1 SAVE EMSG"     /* Display menu            */
do forever                                      /* Iterate in loop until...*/
   if rc¬=0 then exit rc
   if pfk="PF03" then exit 0                    /* ...PF3/PF15 is pressed  */
   emsg=" "                                     /* Clear out msg line      */
   select                                       /* Process the other keys  */
    when pfk="PF09" then do                     /* Find one by name        */
       call LOOK4
    end
    when pfk="PF08" then do                     /* Get next record         */
       call NEXT
    end
    when pfk="PF07" then do                     /* Get previous record     */
       call PREV
    end
    when pfk="PF10" then do                     /* Add a record            */
       call ADDAREC
    end
    otherwise
   end
   call MENUSQ                                  /* Call the sub for cmds   */
   "MENUEXEC" menu "( MAP PA1 SAVE EMSG"
end
exit 0
/*********************************************************************/
ADDAREC:                                        /* Adds a record           */
queue rest"|"street"|"city"|"direct"|"areacod"|"exch,
      ||"|"phone"|"reserv,
      ||"|"date"|"cost"|"stdserv"|"quafood,
      ||"|"qtyfood"|"winesel"|"comment
"EXECIO 1 DISKW RESTAURA NTS A ( FINIS"
return
/*********************************************************************/
LOOK4:                                          /* Look one by name        */
recno=1                                         /* File record to start    */
sname=translate(rest)                           /* Uppercase XLATE         */
slen=length(sname)                              /* Length for the match    */
incre=1                                         /* File read direction     */
call LOOKON                                     /* Go do the search        */
if rc¬=0 then emsg="No more records satisfy search criteria."
return
```

```
/*************************************************************************/
NEXT:                                       /* Search forward          */
recno=recno+1                               /* Next place to look      */
incre=1                                     /* File search direction   */
call LOOKON                                 /* Go look for it          */
return
/*************************************************************************/
PREV:                                       /* Search back             */
if recno-1<1 then do                        /* Any point in going back */
   emsg="Unable to go back any further."
   return
end
recno=recno-1
incre=-1
call LOOKON
return
/*************************************************************************/
READ_A_REC:
return
/*************************************************************************/
LOOKON:
"EXECIO 1 DISKR RESTAURA NTS A" recno "( FINIS"
do while rc=0
   parse pull rec                           /* Get the record          */
   parse upper var rec rmatch "|" .         /* Get the name field      */
   if abbrev(rmatch,sname,slen) then do     /* A match?                */
      parse var rec rest"|"street"|"city"|"direct"|"areacod"|",
                        exch"|"phone"|"reserv"|"date"|"cost"|",
                        stdserv"|"quafood"|"qtyfood"|"winesel"|",
                        comment             /* Parse out the fields    */
      return 0
   end
   recno=recno+incre
   "EXECIO 1 DISKR RESTAURA NTS A" recno "( FINIS"   /* Read the next  */
end
return 4                                    /* No, find good rec       */
/*************************************************************************/
MENUSQ:                                     /* Load the subcommands    */
queue "IGNREQ PF03 PF07 PF08 PF09"          /* Quit and search bypass  */
queue "EMSGFLD MSGLINE"                     /* Put out error messages  */
queue "REQUIRE  CMDLINE EXEC CMDLINE"       /* Process the cmdline     */
queue "REQUIRE  REST NONBLANK"
queue "FIELDMSG REST 'The name must not be left blank'"
queue "REQUIRE  STREET NONBLANK"
queue "FIELDMSG STREET 'The street must not be left blank'"
queue "REQUIRE  CITY   NONBLANK"
queue "FIELDMSG CITY   'The street must not be left blank'"
queue "REQUIRE  AREACOD ERASEEOF"
queue "REQUIRE  AREACOD NOTHING"
queue "REQUIRE  AREACOD DECIMAL"
queue "FIELDMSG AREACOD 'The areacode must be a number. Press Erase EOF",
                        "to clear it.'"
queue "REQUIRE  EXCH    ERASEEOF"
queue "REQUIRE  EXCH    NOTHING"
queue "REQUIRE  EXCH    DECIMAL"
queue "FIELDMSG EXCH    'The phone exchange",
                        "must be a number. Press Erase EOF",
```

```
                        "to clear it.'"
queue "REQUIRE  PHONE   ERASEEOF"
queue "REQUIRE  PHONE   NOTHING"
queue "REQUIRE  PHONE   INTEGER"
queue "FIELDMSG PHONE   'The phone number",
                        "must be a number. Press Erase EOF",
                        "to clear it.'"
queue "UPCASE   RESERV"
queue "REQUIRE  RESERV STRING 'Y' 1 'N' 1 'y' 1 'n' 1"
queue "FIELDMSG RESERV 'The reservation flag can be only Y or N.'"
queue "REQUIRE  DATE   PATTERN 'DD/DD/DD'"
queue "FIELDMSG DATE   'The date must be a number in the form nn/nn/nn.'"
queue "REQUIRE  COST   DECIMAL"
queue "FIELDMSG COST   'The cost must be a valid decimal number.'"
queue "REQUIRE  STDSERV INTEGER > 0 < 6"
queue "FIELDMSG STDSERV 'The service standard must be a number between",
                        "1 and 5.'"
queue "REQUIRE  QUAFOOD INTEGER > 0 < 6"
queue "FIELDMSG QUAFOOD 'The food quality must be a number between",
                        "1 and 5.'"
queue "REQUIRE  QTYFOOD INTEGER > 0 < 6"
queue "FIELDMSG QTYFOOD 'The food quantity must be a number between",
                        "1 and 5.'"
queue "REQUIRE  WINESEL INTEGER > 0 < 6"
queue "FIELDMSG WINESEL 'The wine selection must be a number between",
                        "1 and 5.'"
queue "MENUEND"                                 /* Bottom of stack marker  */
push  "MENUCMDS"                                /* Top of stack marker     */
return
```

## Validate data from a field using REQUIRE

This example shows the use of the MENUEXEC REQUIRE subcommand to edit data from a field.

```
/* REXX9 EXEC */
address command
menu="REXX9"
call QCMDS
"MENUEXEC" menu "( SAVE CLEAR MAP PA1 EMSG"
do forever
   emsg=' '                                                    /* 1.    */
   if pfk="PF03" then exit 0
   call QCMDS
   "MENUEXEC" menu "( SAVE MAP PA1 EMSG CURSOR" cursor
end
exit 0
/*********************************************************************/
QCMDS:
queue "CHANGE   NAME MDT"                                      /* 2.    */
queue "REQUIRE  NAME NONBLANK"                                 /* 3.    */
queue "FIELDMSG NAME 'Name not alpha - reenter.'"              /* 4.    */
queue "REQUIRE  STREET ALPHA"                                  /* 5.    */
queue "REQUIRE  STREET INTEGER"                                /* 6.    */
queue "FIELDMSG STREET 'Street must be all alpha or all numeric'"
queue "REQUIRE  DATE PATTERN 'NN/NN/NN'"                       /* 7.    */
queue "FIELDMSG DATE 'Date is not in the form nn/nn/nn.'"
queue "REQUIRE  SS#  PATTERN 'NNN-NN-NNNN'"
queue "FIELDMSG SS#  'Social security number incorrect.'"
queue "IGNREQ PF03"                                            /* 8.    */
queue "EMSGFLD  MSGLINE"                                       /* 9.    */
queue "MENUEND"
push  "MENUCMDS"
return


/* 1. Using saved menus means that the message line variable "EMSG"
      is not cleared every time because the menu is not reloaded from
      disk.  Hence it should be cleared by setting it to a blank.


   2. REQUIRE edits data sent from the screen.  The
      requirements are applied to the fields in the order
      in which they are placed in the stack, a MENUCTRL
      file, or appear in the SUBCMDS environment.

      This has the effect of skipping fields that have not
      had their modified data tag set, either by the hardware
      in response to data being entered into a field or by
      software setting the MDT (either by XMEDIT or MENUEXEC
      in this case).

   3. This requirement demands that any data other than blanks or
      nulls be entered into the field called NAME.

   4. If the requirement for field NAME is not met then the message
```

```
        is written to the field on the screen named in the EMSGFLD
        subcommand (see 9).

     5. REQUIREs are "OR'ed" together.  The effect of these two
     6. requirements will be to allow either alpha or numeric
        characters to be entered.  Characters such as % or $ will not
        be accepted.

        To form more complex editing requirements than just "OR",
        you can write a MENUEXEC REQUIRE macro.

     7. This is the pattern matching requirement.  It requires data
        entered in the field called DATE to be in the following form:
        two decimal digits, a slash, two digits, slash, two digits.

     8. The IGNREQ subcommand causes the requirements to be ignored
        if this PF key is pressed.  This permits the user to "QUIT"
        out of a menu without having to complete the field according
        to the requirement specifications.

     9. The EMSGFLD subcommand names the field on the menu where
        MENUEXEC should place FILEDMSGs generated from the data
        entered into fields not meeting the requirement option.
*/
```

## Edit data from a field and EXEC procedural code

This example shows the use of MENUEXEC REQUIRE macros to edit data from a field, and procedural code in the EXEC.

```
/* REXX10 EXEC */
address command
menu="REXX10"
call QCMDS
"MENUEXEC" menu "( SAVE CLEAR MAP PA1 EMSG"
do forever
   if pfk="PF03" then exit 0
   msgline=' '
   alarm=''
   sint="DIM"                                                    /* 1.    */
   call STCHK                                                    /* 2.    */
   queue "CHANGE STATE" sint                                     /* 3.    */
   call QCMDS
   "MENUEXEC" menu "( SAVE MAP PA1 EMSG CURSOR" cursor alarm     /* 4.    */
end
exit 0
/*********************************************************************/
STCHK:                                                           /* 5.    */
select
 when state="CA" then nop
 when state="MA" then nop
 when state="NY" then nop
 otherwise
    cursor="STATE"                                               /* 6.    */
    alarm="ALARM"
    sint="BRIGHT"
    msgline="The state must be CA, MA or NY"
end
return
/*********************************************************************/
QCMDS:
queue "CHANGE   NAME MDT"
queue "REQUIRE  NAME NONBLANK"                                   /* 7.    */
queue "FIELDMSG NAME 'You must complete the name field.'"
queue "IGNREQ PF03"
queue "EMSGFLD  MSGLINE"
queue "MENUEND"
push  "MENUCMDS"                                                 /* 8.    */
return

/* 1. A string is set to the value on the STATE field default
      attribute.  Any field that could potentially have its attribute
      changed needs to be explicitly reset to its default value
      because saved menus are being used.

   2. The call is made to the subroutine that checks the validity
      of the STATE field.  This code is executed after all
      the REQUIRE subcommands have been met.  To make sure
      procedural validation code is executed in sequence with
      REQUIRE subcommands it might be necessary to write them as
      macros.
```

3. The instruction to set/reset the attribute for the
   STATE field is placed on the stack. It does not matter
   that the other subcommands have not yet been stacked.

4. The MENUEXEC call has two variables strung on the end:
   alarm and cursor. The variable "CURSOR" is returned by
   MENUEXEC after execution. It is fed back into MENUEXEC
   as an option unchanged if the STATE field is valid.
   If the field is invalid it is set to the name of the STATE
   field.

   The "ALARM" variable name has no significance. It is reset every
   time at the top of the processing loop to a null string.
   If the STATE field is in error it is given the MENUEXEC option
   value of "ALARM". The ALARM option causes MENUEXEC to sound
   the terminal alarm if one is present.

5. This is the STATE checking code.

6. If the value of STATE does not match any of the values in the
   SELECT statement, then the field is intensified, the cursor
   placed on the field, and a message placed in the "MSGLINE"
   variable.

7. This requirement causes MENUEXEC to demand input into this field.

8. By "PUSHing" or stacking FIFO the MENUCMDS eyecatcher,
   all subcommands can be placed on the stack until
   the stack is complete.
*/

## Use XMENU windows

This example shows how to set up two menus that overlay, the smaller window being subordinate to the larger one.

```
/*
   XMEXAMP EXEC - This example shows how to set up two menus that
                  overlay and where the smaller window is subordinate
                  to the larger window.

   To run this example...

   1. Create a MENU by issuing 'XMEDIT LARGMENU'.
      (If you need help with this, refer to XMENU in
      minutes: An end-user's guide to creating menus.)

      Include an UNPROTECTED field named FIELD1 on it somewhere.
      Also include a PROTECTED field named $ERRMSG on it (length 72).

   2. Create another menu.  Issue 'XMEDIT SMALMENU ( SIZE 20 20'.
      It requires no fields, but put some text in to be displayed.

   3. Create the file XMEXAMP EXEC, as shown below, then
      enter 'XMEXAMP'.  This should bring up the LARGMENU.

   4. Press PF02(PF14).  This will bring up the SMALMENU at
      position R3,C5 on the current menu.

   5. Press PF03(PF15).  This will remove the SMALMENU.
      Pressing any other key will not remove this menu.

   6. Enter anything in FIELD1 and press ENTER or press PF03(PF15) to
      exit XMEXAMP.


*/
   trace

   first_time1 = 1
   first_time2 = 1

   menu1       = 'LARGMENU'
   menu2       = 'SMALMENU'


   do until pfk = 'PF03' /* PF03/15 is QUIT from the larger screen.   */


/* Display the menu, read in the above commands and begin windows!    */

      'MENUEXEC 'menu1'( MAP EMSG WINDOW X1 SAVE IGNSCFN EMPTY SUBCMDS' clear

/*    MENUEXEC Options Definition...

      MAP       - MAPs the PFkeys so that PF13 = PF01, PF14 = PF02...
      EMSG      - Causes MENUEXEC error msgs to be displayed.
      WINDOW X1 - defines the window to be used.  When using MENUEXEC,
```

```
                only one menu per window is allowed.
        SAVE      - causes the SAVEd MENU ENVIRONMENT to be active,
                    allowing menus to be used from virtual storage
                    rather than DISK.
        IGNSCFN   - Errors caused by field names specified in control
                    statements will not cause MENUEXEC to give non-zero
                    return code (execution continues).
        EMPTY     - Any field that is empty, not defined, or dropped from
                    REXX will clear that field to nulls when displayed.
        SUBCMDS   _ Use the CMS Subcom interface to accept subcommands.
*/

        address 'MENUCMDS'          /* Shows the start of the
                                       MENUEXEC cmd stack                */
        'IGNREQ PF03 PF02'          /* Tells MENUEXEC to ignore
                                       field requirements when PF02/14/03/
                                       or 15 is pressed.                 */
        'REQUIRE FIELD1 ANYTHING' /* Return to the exec only if there
                                       is something typed in FIELD1 (except
                                       PF03 per the IGNREQ statement).   */
        "FIELDMSG FIELD1 'Please enter anything!'"
                                  /* Will display in $ERRMSG if REQUIRE
                                       specs are not met in FIELD1       */
        'MENUEND'                 /* Shows the bottom of the MENUEXEC
                                       cmd stack                         */
        address command           /* go back to normal environment     */

        if rc <> 0 then exit rc

        if first_time1 then do
           menu1 = menu      /* save the in-storage menu name           */
           first_time1 = 0   /* reset the in-storage flag               */
           clear = ' '       /* Only CLEAR once                         */
           end

        if PFK = 'PF02' then Call SMALWNDW

        DROP field1

    end

/* The next two lines remove the menu from virtual storage              */

    if first_time1 = 0 then 'XMENUINS PURGE' menu1
    if first_time2 = 0 then 'XMENUINS PURGE' menu2

    exit


SMALWNDW: PROCEDURE EXPOSE menu2 first_time2

    do until pfk = 'PF03' /* PF03/18 is QUIT from the larger screen.   */

/* Display the menu, read in the above commands.                        */

       'MENUEXEC' menu2 '( MAP EMSG WINDOW X2 SAVE SUBCMDS'

/*     MENUEXEC Options Definition...
```

```
      MAP       - MAPs the PFkeys so that PF13 = PF01, PF14 = PF02...
      EMSG      - Causes MENUEXEC error msgs to be displayed.
      WINDOW X1 - defines the window to be used.  When using MENUEXEC,
                  only one menu per window is allowed.
      SAVE      - causes the SAVEd MENU ENVIRONMENT to be active,
                  allowing menus to be used from virtual storage
                  rather than DISK.
      IGNSCFN   - Errors caused by field names specified in control
                  statements will not cause MENUEXEC to give non-zero
                  return code (execution continues).
      EMPTY     - Any field that is empty, not defined, or dropped from
                  REXX will clear that field to nulls when displayed.
      SUBCMDS   _ Use the CMS Subcom interface to accept subcommands.
*/

      address 'MENUCMDS'        /* Shows the start of the
                                   MENUEXEC cmd stack                 */

      'VIEWPORT 3 5 0 0 20 20' /* The VIEWPORT cmd defines where
                                   the menu will overlay the current
                                   screen.

                                   '3 5' says viewport should start
                                   at row 4 column 6 of the current
                                   screen.

                                   '0 0' says menu displayed starting
                                   with row 1 column 1 of the menu.

                                   '20 20' says that the menu
                                   viewport should display all 20 rows
                                   and 20 columns of the menu.        */

      'MENUEND'                 /* Shows the bottom of the
                                   MENUEXEC cmd stack                 */

      address command           /* Return EXEC processing to normal   */

      if rc <> 0 then exit rc

      if first_time2 then do
         menu2 = menu     /* save the in-storage menu name            */
         first_time2 = 0  /* reset the in-storage flag                */
         end

   end
   RETURN
```

## Sample REQUIRE macro to check for YES or NO input

The following example illustrates how to write a macro to validate a field. It checks to see if "YES" or "NO" was entered into a field.

```
/* A 'REQUIRE EXEC' to determine if a YES|NO has been entered correctly.

      Call form:  REQUIRE fieldname EXEC YESNO

      Called  by XMENU to determine if field is valid.

      Using the different types of exits (custom tailoring) you can
      use this macro to manipulate the data in any way you wish and
      transfer the data back to the XMENU screen.

*/
   trace o

/* Get the arguments and read the field into 'localvar' */

   arg menuname fieldname status

/* status can be NOINPUT, INPUT or EOF;

      NOINPUT - No user input in the field
      INPUT   - Non-zero length user input
      EOF     - zero length user input       */

   'READ' fieldname 'LOCALVAR'

/* Check the field for validity   */

   IF status = 'EOF' | status = 'NOINPUT' then signal exit2

/* Require conditions met.  Screen NOT changed by this exec. */
/* Therefore EXIT 0                                          */

   localvar strip (translate (localvar))
   IF ABBREV('YES',localvar,1) THEN exit 0
   IF ABBREV('NO',localvar,1)  THEN exit 0

/* Field is invalid, Therefore exit3 */

EXIT3:

/* ERROR EXIT.  Require exec conditions not met. Screen IS changed
   by this exec.   */

   BLANKS = '   '
   'PUTIN' fieldname 'BLANKS'

   exit 3

/*  The following exits are not used in this example, however
    they are included here to show what other exit conditions
    can be used during require exec processing                */

EXIT1:

/* Replace the variable with the upper case version.  Use this exit
   if this is to be done... or replace with the full version of 'YES'
   or 'NO' if desired. */
```

```
   'PUTIN' fieldname 'localvar'

   exit 1

EXIT2:

/* ERROR EXIT.  Require exec conditions not met. Screen not changed
   by this exec.   */

   exit 2

EXITMENU:

/* Use this exit to tell MENUEXEC to exec the menu entirely  */

   exit -10
```

## Sample REQUIRE macro to check for a valid date

The following example illustrates how to write a macro to validate a field. It checks to see if the date field is valid.

```
/* A 'REQUIRE EXEC' to determine if a date has been entered correctly.

      Call form:  REQUIRE fieldname EXEC DATECHEK

      Must be called by XMENU to determine if a date field is valid.

*/
   trace o
   arg menuname fieldname status
   'READ' fieldname 'localvar'
/*
   00/00/00 & 99/99/99 return '0'
*/
   if localvar = '00/00/00' then signal exit0
   if localvar = '99/99/99' then signal exit0
/*
   Parse...
*/
   PARSE VALUE localvar WITH month '/' day '/' year .
   localvar = strip(localvar)
   month = strip(month)
   day   = strip(day)
   year  = strip(year)
   l1 = length(localvar)
   if l1 < 6 then signal exit2
   if month < 1 & month > 12 then signal exit2
   if length(year) ¬= 2 then signal exit2
   daymap   = '31 28 31 30 31 30 31 31 30 31 30 31'
/*
   Handle leapyear
*/
   if year ¬= '00' & month = '02' then do
        mod4 = year // 4
        if mod4 = 0 then daymap=overlay('29',daymap,4)
        end
   maxday = subword(daymap,month,1)
   if day > maxday | day < 1 then signal exit2
   if length(day) = 1 then day = '0' || day
   if length(month) = 1 then month = '0' || month
/*
   Rebuild date buffer and return
*/
   finaldate = month || '/' || day || '/' || year
   if finaldate = localvar then signal exit0
        else do
           'PUTIN' fieldname 'finaldate'
            signal exit1
        end
EXIT2:
   exit 2
EXIT1:
   exit 1
EXIT0:
   exit 0
```

# Index

## Special Characters



## Numerics



## A



## B



## C



## D

## E



## F



## H

## N



## O



## P



## R



## S

## T



## U



## V



## W



## X