Customizing the Features of the M Editor Using Macros and C Extensions

59

Leo A. Notenboom

The Microsoft® Editor is a powerful text editor that runs under MS-DOS® and the OS/2 systems. Its features include the ability to edit more than one file at a time, split the screen into multiple windows, and run tools, including Microsoft compilers, from within the editor.

This article will look at the fundamental concepts and methods for customizing and extending the functionality of the Microsoft Editor. Familiarity with the basic operation of the Microsoft Editor is assumed.

One of the most important features of current text editors is programmability and customization. Microsoft Editor supports three types of programmability: the definition of the meanings of individual keystrokes, the definition of macros, and the creation of editor extension functions. We will examine each of these, with emphasis on and examples of the latter two.

The Keyboard

Programming the keyboard in Microsoft Editor is the process of associating or "assigning" editing functions to keystrokes. When a keystroke is pressed, the function assigned to it is then executed.

Every editing function that Microsoft Editor can perform has a name associated with it, which is used to assign it to one or more keystrokes or execute the function in macros. Any editing function can be assigned to any keystroke that Microsoft Editor recognizes.

The editor assigns a default function to every valid keystroke. It assigns the graphic function to all keys that you use to type in text, such as alphanumeric keys, the unassigned function to all unused keys, and other editing functions to the remaining keys. For example, the exit function is assigned to F8.

Programming the keyboard simply means making new key assignments. For example, you can assign the exit function to Alt-X. To do this you place the following in TOOLS.INI, the initialization file (M is used here to reference the TOOLS.INI section for editor defaults. In practice the section name is that of the actual editor filename and would be [MEP] if MEP.EXE were used. [M MEP] could be used to define a single section for both.):

```
[M]
exit:alt+x
```

Any number of such assignments may be contained in the [M] section of TOOLS.INI.

You can also use the assign function by entering the following commands:

```
arg exit:alt+x assign
```

Using default key assignments, this means you would

ONE OF THE MOST IMPORTANT FEATURES OF CURRENT TEXT EDITORS IS PROGRAMMABILITY. MICROSOFT EDITOR SUPPORTS THREE TYPES OF PROGRAMMABILITY: THE DEFINITION OF INDIVIDUAL KEYSTROKES, THE DEFINITION OF MACROS, AND THE CREATION OF EDITOR EXTENSION FUNCTIONS.

Figure 1: Microsoft Editor 1.01 Functions

Name	Function Performed
arg	Introduce argument to subsequent function
assign	Define keystroke or define macro
backtab	Move to previous tab position
begline	Move to beginning of line
cancel	Cancel operation or argument
cdelete	Character delete
compile	Execute compiler and/or browse results
copy	Copy selected text to clipboard
down	Move down one line
emacscdel	Delete character (joins lines)
emacsnewl	Move to new line (breaks lines)
endline	Move to end of line
execute	Execute other editor functions by name
exit	Exit the editor or the current file
graphic	Place character into file being edited
home	Go to top or top left position on screen
information	Display file history
initialize	Initialize the editor and read portions of TOOLS.INI
insertmode	Toggle insert/typeover mode
lasttext	Redisplay previous text arg for editing
ldelete	Delete lines or boxes
left	Move cursor left
linsert	Insert lines
mark	Place or go to a bookmark
meta	Modify action of subsequent function
mlines	Move backward lines
mpage	Move backward pages
mpara	Move backward paragraphs
msearch	Search backward
mword	Move backward words
newline	Move to next line
paste	Copy contents of clipboard to current cursor location
pbal	Balance unmatched parentheses, braces, or brackets
plines	Move forward lines
ppage	Move forward pages
ppara	Move forward paragraphs
psearch	Search forward
pword	Move forward words
qreplace	Query search and replace
quote	Treat character assigned to other function as text
refresh	Reread or discard current file
replace	Search and replace
restcur	Restore cursor to saved location
right	Move cursor right
savecur	Save cursor location
sdelete	Delete stream of characters
setfile	Move to new file and/or save current file
setwindow	Redisplay or reorient text in window
shell	Execute operating system command
sinsert	Insert stream
tab	Move cursor to next tab stop
unassigned	Print error message
undo	Reverse effects of previous edits
up	Move cursor up
window	Manipulate multiple windows

press Alt-A to execute the arg function, type "exit:alt+x", and press Alt-equal sign to execute the assign function. The Microsoft Editor *User's Guide* has more details on TOOLS.INI and the assign function, and I'll refer to them throughout the article.

The same function can be assigned to more than one keystroke. For example, the sdelete (stream delete) function can be assigned to both Del and Ctrl-G:

```
[M]
sdelete:del
sdelete:ctrl+g
```

Macros and extension functions, as you'll see, are treated just like standard editing functions in that they can also be assigned to any keystroke.

Programming Macros

A macro is a sequence of editing functions and text, which is treated as a single editing function. The editing functions used can be standard functions, extension functions, or previously defined macros. Macros, unlike extension functions, do not add functionality to the editor. Instead, they enable you to combine existing editing functions to perform larger and more complex tasks easily or reduce the number of keystrokes to perform common tasks. **Figure 1** lists Microsoft Editor's standard editing functions.

Macros can be defined "on the fly" in an editing session by using the assign function. It is more common, however, to store macros in TOOLS.INI. You define a macro by stating its name and its function sequence in the following format:

```
name:=function-sequence
```

where each item in "function-sequence" is either a function or macro name or a string of text enclosed in quotes. Note the use of the equal sign, which differentiates a macro definition from a keystroke assignment. A

macro to save the current file without exiting might be defined as:

```
savefile:=arg arg setfile
```

This example defines the macro savefile to be the sequence of editing functions arg arg setfile, which saves the current file without exiting. To assign Shift-F2 to the savefile function, you would make the following assignment:

```
savefile:shift+f2
```

Now whenever Shift-F2 is pressed the current file will be saved to disk. The default keystrokes for this operation (without using the macro) are Alt-A, Alt-A, and F2.

A macro is in fact a new editing function. The name of the macro can be used anywhere that the name of an editing function can be used. It can be used in the function-sequence of another macro, or assigned to a keystroke, as shown above.

Arguments to Macros

Macros have no explicit syntax for accepting arguments. However, any argument that has been selected at the time the macro is executed is passed to the first function of the macro that takes an argument.

One way for a macro to accept an argument is to begin with the copy function, which copies its argument to the clipboard. After using copy, you can then manipulate this text by loading a special file named <clipboard>.

The file <clipboard> is a special type of file known as a pseudo-file. These files function as regular text files, except that they cannot be saved to disk without an explicit filename. Pseudo-files are defined by filenames that are enclosed in angle brackets (<>) and can be employed as temporary scratch buffers during editing. Some pseudo-files are created automatically by the editor and have special meanings. The pseudo-file <clipboard> is written to whenever you use the copy function or delete a block of text.

Figure 2: Arguments Used in the Print Macro

noarg	Print current line
nullarg	Print cursor to end of current line
textarg	Print the text argument
boxarg	Print the selected box
linearg	Print the selected lines
numarg	Print the next num lines
markarg	Print text between cursor and mark

Defining Macros

As an example of defining and using a simple macro, we'll create a print function, which sends a selected region of text to the printer. The copy function will capture the text to be printed so the macro's syntax is as shown in **Figure 2**.

The macro will execute the following steps:

- copy the selected text to the clipboard
- save the clipboard to a temporary file on disk
- send the temporary file to the printer
- delete the temporary file

This macro will be defined in pieces by defining four smaller macros that correspond to the steps above. Writing larger macros as sequences of smaller macros improves readability and makes the parts of one macro available for use in other macros.

The first step is easy; you just execute the copy function to place the selected text into the clipboard. Then the setfile function will change the current file to <clipboard> *(The following and all subsequent macros must be entered on one line. They have been broken here simply to fit in our columns—Ed.)*:

```
print1:=copy arg
    "<clipboard>" setfile
```

To perform the second step, you use setfile again to save the current file, <clipboard>, to a temporary file. Then you use setfile a third time to restore the original current file:

```
print2:=arg arg
       "TEMP.DAT" setfile
       setfile
```

Note that even though you have changed the current file twice in these first two steps, the screen is not updated until input is required of the user. The user does not see the screen switch to the clipboard and back, as he would if the functions were executed individually from the keyboard.

The third step uses the shell function to execute a DOS command to print the file:

```
print3:=arg
    "PRINT TEMP.DAT" shell
```

The last step is to use shell again to delete the file:

```
print4:=arg
    "DEL TEMP.DAT" shell
```

Now you put the pieces together. The completed macro looks like this:

```
print:=print1 print2
       print3 print4
```

Finally, in order to use the macro, you assign it to a key:

```
print:alt+p
```

Now whenever Alt-P is pressed, portions of the current file are printed, based on the type of argument specified beforehand.

Flow Control in Macros

The editor's macro language supports conditional and unconditional branching. Every function in the editor has a TRUE or FALSE return value that can be used for branching.

As a simple example, we'll

AN EXTENSION FUNCTION IS A NEW EDITING FUNCTION WRITTEN IN A PROGRAMMING LANGUAGE SUCH AS C OR ASSEMBLY. EXTENSION FUNCTIONS CALL INTERNAL EDITOR FUNCTIONS TO MANIPULATE TEXT, READ AND WRITE TO FILES, AND CONTROL THE EDITING ENVIRONMENT.

create a new macro, bigprint, which does everything the print macro does and also recognizes the following syntax:

meta bigprint – print the entire file

Since the use of meta does not fit into the method of using the copy function, as we did in the print macro, you'll have to check first for meta and act a little differently if it is on. You can just select the entire file for copy and execute the print macro. Note that once you've selected the entire file, you simply use the print macro to print the file.

Let's assume that we have already defined a macro called selectall, which selects the entire file in preparation for the copy function that starts the print macro. The bigprint macro then looks like this :

```
bigprint:=meta +>nometa
          cancel selectall
          meta :>nometa meta
          print
```

Step by step, here's what this macro does. The meta function toggles between on and off; it has a return value to indicate the new state. Thus you start the macro with meta to test its previous value. Note that this action does not affect the selected argument.

+> is the macro-language "branch if TRUE" statement. This step causes the macro to branch to the label nometa if the previous function returned TRUE. In this example, the previous function, meta, returns TRUE if meta is now on, which would mean meta was off when the macro was entered. If meta was off when we entered, then the user did not specify meta, and we would branch around the next step.

Then, cancel eliminates any argument that the user may have selected. Since we are about to select the entire file, any user-selected argument would be inappropriate.

Next, selectall is reached only if the initial meta function returned FALSE, indicating that meta was on when the macro began. We now execute the selectall macro, defined below, which selects the entire file as input to the copy function.

The second meta turns meta back on. The macro will execute meta one more time near the end of this sequence. Since we will flow through that step, we turn meta on now so that it will be turned off at that time.

:> is the macro-language label-definition statement. This defines the label nometa, which we can branch to from the +>nometa statement above or fall through from the meta function above.

In case the macro was executed with meta off, we will have turned it on in the process of testing it. Here we execute meta once more to turn it back off. Then print executes the previously defined print macro.

By the time the print macro is executed, one of two things has happened: the user turned meta on, so we selected the entire file by executing selectall, or the user left meta off, so we've branched around the selectall and preserved the user's argument. In the latter case, the user's argument is passed to copy at the beginning of the print macro.

Finally, you must define selectall. To select the entire file:

- go to the end of the file
- define a mark there
- go to the top of the file
- select from that mark

Each of these steps is then translated into appropriate macros :

```
select1:=arg ppage
select2:=arg arg
         "endoffile" mark
select3:=arg mpage
select4:=arg "endoffile"
selectall:=select1
          select2 select3
          select4
```

Note that this macro ends with an "open arg," an argument that has not yet been passed to a function. At the end of selectall, arg has been executed and the mark name specified, but no function has been specified to process the argument. When used with the bigprint macro above, the next function executed that processes an argument will be copy. The net result will be arg "endoffile" copy, which copies from the current location (top of file) to the mark location (end of file), placing the entire file in the clipboard.

Now you can assign the bigprint macro to Alt-P. Assuming meta is assigned to F9, pressing F9 followed by Alt-P prints the entire current file.

Programming Extensions

An extension function is a new editing function written in a programming language such as C or assembly. Extension functions call internal editor functions to manipulate text, read and write to files, and control the editing environment.

Extensions are the most powerful and flexible approach to programming the editor. They permit you to add completely new functions to the editor that do not rely on existing functions, can be customized to a much higher degree than macros, and can execute significantly faster since they are compiled code.

The Filter Extension

We will examine the structure of an extension and some common programming techniques by using the filter extension as an example. The source to this extension is shown in **Figure 3** and its make file in **Figure 4**. Note that the extension requires Version 1.01 or later of the Microsoft Editor.

Filter takes as an argument a region within the file being edited and replaces that region

Figure 3: Microsoft Editor Filter Extension

```
/*************************************************************************
*
* Purpose:
*  Provides a new editing function, filter, which replaces its
*  argument with the argument run through an arbitrary operating
*  system filter program.
*
*************************************************************************/
#define ID    " filter ver 1.00 "##__DATE__##" "##__TIME__

#include <stdlib.h>            /* min macro definition */
#include <string.h>            /* prototypes for string fcns */
#include "ext.h"

/
*************************************************************************
*
* Internal function forward declarations
*/
flagType pascal      DoSpawn    (char *);
void     pascal      id         (char *);
void     pascal EXTERNAL SetFilter  (char far *);

/
*************************************************************************
*
* global data
*/
PFILE    pFileFilt      = 0;              /* handle for filter-file*/
char    *szNameFilt     = "<filter-file>"; /* name of filter-file  */
char    *szTemp1        = "filter1.tmp";   /* name of 1st temp file*/
char    *szTemp2        = "filter2.tmp";   /* name of 2nd temp file*/
char     filtcmd[BUFLEN] = "";             /* filter command itself*/


/*** filter - Editor filter extension function
*
* Purpose:
*  Replace selected text with that text run through an arbitrary
*  filter.
*
*  NOARG        - Filter entire current line
*  NULLARG      - Filter current line, from cursor to end of line
*  LINEARG      - Filter range of lines
*  BOXARG       - Filter characters with the selected box
*
*  NUMARG       - Converted to LINEARG before extension is called.
*  MARKARG      - Converted to Appropriate ARG form above before
*                 extension is called.
*
*  STREAMARG    - Treated as BOXARG
*
*  TEXTARG      - Set new filter command
*
* Input:
*  Editor Standard Function Parameters
*
* Output:
*  Returns TRUE on success, file updated, else FALSE.
*
*************************************************************************/
flagType pascal EXTERNAL filter (
unsigned int argData,              /* keystroke invoked with     */
ARG far *pArg,                     /* argument data              */
flagType fMeta                     /* indicates preceded by meta */
) {
char     buf[BUFLEN];              /* buffer for lines        */
int     cbLineMax;           /* max line length in filtered  */
LINE     cLines;             /* count of lines in file       */
LINE     iLineCur;           /* line being read              */
PFILE     pFile;             /* file handle of current file  */
/*
** Initialize: identify ourselves, get handle to current file, and
** discard the contents of the filter-file.
```

CONTINUED

Figure 3: Microsoft Editor Filter Extension CONTINUED

```
*/
id ("");
pFile = FileNameToHandle ("", "");
DelFile (pFileFilt);
/*
** Step 1, based on the argument type, copy the selected region into the
** (upper leftmost position of) filter-file.
** Note that TEXTARG is a special case that allows the user to change
** the name of the filter command to be used.
*/
switch (pArg->argType) {
  case NOARG:              /* filter entire line */
    CopyLine (pFile,
          pFileFilt,
          pArg->arg.noarg.y,
          pArg->arg.noarg.y,
          (LINE) 0);
    break;

  case NULLARG:             /* filter to EOL */
    CopyStream (pFile,
            pFileFilt,
            pArg->arg.nullarg.x,
            pArg->arg.nullarg.y,
            255,
            pArg->arg.nullarg.y,
            (COL) 0,
            (LINE) 0);
    break;

  case LINEARG:             /* filter line range */
    CopyLine (pFile,
          pFileFilt,
          pArg->arg.linearg.yStart,
          pArg->arg.linearg.yEnd,
          (LINE) 0);
    break;

  case BOXARG:              /* filter box */
    CopyBox (pFile,
         pFileFilt,
         pArg->arg.boxarg.xLeft,
         pArg->arg.boxarg.yTop,
         pArg->arg.boxarg.xRight,
         pArg->arg.boxarg.yBottom,
         (COL) 0,
         (LINE) 0);
    break;

    case TEXTARG:
    SetFilter (pArg->arg.textarg.pText);
    return 1;
    }
/*
** Step 2, write the selected text to disk.
*/
if (!FileWrite (szTemp1, pFileFilt)) {
    id ("** Error writing temporary file **");
    return 0;
    }
/*
** Step 3, create the command to be executed:
** user specified filter command + " " + tempname 1 + " >" +
** tempname 2
** Then perform the filter operation on that file, creating a second
** temp file.
*/
strcpy (buf,filtcmd);
strcat (buf," ");
strcat (buf,szTemp1);
strcat (buf," >");
strcat (buf,szTemp2);
if (!DoSpawn (buf)) {
    id ("** Error executing filter **");
    return 0;
    }
```

CONTINUED

with the results of executing an arbitrary filter program on that region. For example, the filter program we'll use as an example will be the DOS SORT program. After selecting a region of lines and executing filter, the lines will be replaced in sorted order.

The filter extension demonstrates how to handle some of the issues encountered when writing extensions, such as dealing with multiple types of arguments, using switches, reading and writing physical files from disk, and executing other editor commands from within an extension.

Filter does its *job by:*

- copying the user-selected data to a pseudo-file
- writing that pseudo-file to a physical file on disk
- executing the filter command on that file and saving its output in a new file
- replacing the contents of the pseudo-file with the saved output of the filter command
- copying the new data in the pseudo-file back to the original file being edited

Before examining filter more closely, we should first look at what exactly comprises an extension.

Editor Extension Structure

An editor extension consists of four sections:

- **extension function and support routines**, the code that implements extension editing functions
- **extension initialization routine**, which is called when the editor first loads the extension
- **editing function definition table**, which defines the functional content of the extension
- **switch definition table**, which defines any switches created by the extension

Extensions also require certain support files and must be compiled and linked with certain options, as outlined in the *User's Guide*.

Extension Initialization

When the editor loads an extension, the initialization routine WhenLoaded, which must be present, is called. It need not do anything and can return immediately. WhenLoaded is useful for making default keystroke assignments and performing any other extension initialization desired.

An extension can be loaded at any time. Normally the user places a load: assignment in TOOLS.INI, which is executed when the editor starts up. For example:

```
[M]
load:filter.mxt
```

The load: assignment, like any other assignment, can be executed at any time by using the assign function.

In the filter extension, WhenLoaded performs three functions:

- outputs an informative "sign-on" message to the editor's dialog line
- gets a file handle for the pseudo-file it will use, creating that file if necessary
- creates a default key assignment by assigning the filter function to the keystroke Alt-F

The second function deserves some explanation. This is the code that implements it:

```
pFileFilt =
  FileNameToHandle
  (szNameFilt,szNameFilt);
if (!pFileFilt) {
  pFileFilt = AddFile
    (szNameFilt);
  FileRead (szNameFilt,
    pFileFilt);
  }
```

The FileNameToHandle call, which is a call to an internal editor function, returns a handle

Figure 3: Microsoft Editor Filter Extension CONTINUED

```
/*
** Step 4, delete the contents of the filter-file, and replace it by
** reading in the contents of that second temp file.
*/
DelFile (pFileFilt);
if (!FileRead (szTemp2, pFileFilt)) {
    id ("** Error reading temporary file **");
    return 0;
    }
/*
** Step 5, calculate the maximum width of the data we got back from
** the filter. Then, based again on the type of region selected by
** the user, DISCARD the users select region, and copy in the
** contents of the filter-file in an equivalent type.
*/
cLines = FileLength (pFileFilt);
cbLineMax = 0;
for (iLineCur = 0; iLineCur < cLines; iLineCur++)
    cbLineMax = max (cbLineMax, GetLine (iLineCur, buf, pFileFilt));

switch (pArg->argType) {
  case NOARG:              /* filter entire line  */
    DelLine  (pFile,
          pArg->arg.noarg.y,
          pArg->arg.noarg.y);
    CopyLine (pFileFilt,
          pFile,
          (LINE) 0,
          (LINE) 0,
          pArg->arg.noarg.y);
    break;

  case NULLARG:              /* filter to EOL  */
    DelStream  (pFile,
            pArg->arg.nullarg.x,
            pArg->arg.nullarg.y,
            255,
            pArg->arg.nullarg.y);
    CopyStream (pFileFilt,
            pFile,
            (COL) 0,
            (LINE) 0,
            cbLineMax,
            (LINE) 0,
            pArg->arg.nullarg.x,
            pArg->arg.nullarg.y);
    break;

  case LINEARG:              /* filter line range */
    DelLine  (pFile,
          pArg->arg.linearg.yStart,
          pArg->arg.linearg.yEnd);
    CopyLine (pFileFilt,
          pFile,
          (LINE) 0,
          cLines-1,
          pArg->arg.linearg.yStart);
    break;

  case BOXARG:             /* filter box */
    DelBox     (pFile,
         pArg->arg.boxarg.xLeft,
         pArg->arg.boxarg.yTop,
         pArg->arg.boxarg.xRight,
         pArg->arg.boxarg.yBottom);
    CopyBox (pFileFilt,
         pFile,
         (COL) 0,
         (LINE) 0,
         cbLineMax-1,
         cLines-1,
         pArg->arg.boxarg.xLeft,
         pArg->arg.boxarg.yTop);
    break;
    }
```

CONTINUED

Figure 3: Microsoft Editor Filter Extension CONTINUED

```
/*
** Clean-up: delete the temporary files we've created
*/
strcpy (buf, "DEL ");
strcat (buf, szTemp1);
DoSpawn (buf);
strcpy (buf+4, szTemp2);
DoSpawn (buf);

return 1;
/* end filter */}



/*** DoSpawn - Execute an OS/2 or DOS command
*
* Purpose:
*  Send the passed string to OS/2 or DOS for execution.
*
* Input:
*  szCmd    = Command to be executed
*
* Output:
*  Returns TRUE if successful, else FALSE.
*
*******************************************************************************/
flagType pascal DoSpawn (
char    *szCmd
) {
char   cmd[BUFLEN];

strcpy (cmd, "arg \"");
strcat (cmd, szCmd);
strcat (cmd, "\" shell");
return fExecute (cmd);

/* end DoSpawn */}

/*** SetFilter - Set filter command to be used
*
* Purpose:
*  Save the passed string parameter as the filter command to be used
*  by the filter function. Called either because the "filtcmd:"
*  switch has been set or because the filter command received a
*  TEXTARG.
*
* Input:
*  szCmd    = Pointer to asciiz filter command
*
* Output:
*  Returns nothing. Command saved.
*
*******************************************************************************/
void pascal EXTERNAL SetFilter (
char far *szCmd
) {
strcpy (filtcmd,szCmd);
/* end SetFilter */}



/*** WhenLoaded - Extension Initialization
*
* Purpose:
* Executed when extension gets loaded. Identify self, create
* <filter-file>, and assign default keystroke.
*
* Input:
*  none
*
* Output:
*  Returns nothing. Initializes various data.
*
```

CONTINUED

to a file that has already been opened. If the file has not been opened, it returns NULL. You do this first, rather than just create the file, in case the user or some other extension or macro has already created a file of this name. If the file is already open, you just use the existing file handle.

If the file isn't already open, you use another internal function, AddFile, to create it. This call creates the internal structures the editor uses to manipulate the file but does not actually read a file from disk. The internal function FileRead is used to load the file with data from disk. Because this is a pseudo-file, the call to FileRead results in no data actually being read. It is necessary here to complete the editor's initialization of the pseudo-file.

Function Definition Table

The function definition table, cmdTable, defines for the editor the functions contained in the editor extension. Each entry contains a pointer to a string of text that contains the function name, a pointer to the function itself, a reserved field that should be set to 0, and a field containing flags. The flags define:

- The types of user-specified arguments this function accepts.
- Whether this function is a cursor-movement function or a text-editing function. Cursor-movement functions are those that, rather than taking an argument, can be used to help define the region of an argument after arg is executed. For *example*, down is a cursor-movement function that does not take an argument but can be used to extend a box or line argument downward.
- Whether this function is sensitive to the meta prefix.

- Whether this function is a window-movement function. Window-movement functions are similar to cursor-movement functions; they do not affect screen highlighting. Window-movement functions differ in that they may take arguments. The editor's ppage function is a window-movement function.

The filter function is defined by the following table entry:

```
{"filter",filter,0,
        KEEPMETA|NOARG|
        BOXARG|NULLARG|
        LINEARG|MARKARG|
        NUMARG|TEXTARG|
    MODIFIES}
```

This entry defines the name of the function as seen by the user to be "filter". The second field contains a pointer to the actual function. The third is reserved and must be null. The fourth parameter contains flags that indicate that the setting of the meta indicator is not to be affected by using this function (KEEPMETA), all the allowed argument types that the user can specify (NOARG through TEXTARG), and the fact that the function may modify text (MODIFIES).

Switch Definition Table

The switch definition table, like the function table, defines for the editor the switches contained in the extension. Each entry contains a pointer to the text containing the name of the switch, a pointer to the location of either the switch itself or a routine to interpret textual switches, and flags defining the characteristics of the switch.

There are three types of switches: numeric, Boolean, and textual. Numeric and Boolean switches are changed directly by the editor when a new assignment is made. Textual switches require a support rou-

Figure 3: Microsoft Editor Filter Extension CONTINUED

```
*****************************************************************************/
void WhenLoaded (void) {

pFileFilt = FileNameToHandle (szNameFilt,szNameFilt);
if (!pFileFilt) {
    pFileFilt = AddFile (szNameFilt);
    FileRead (szNameFilt, pFileFilt);
    }
SetKey ("filter", "alt+f");
id ("text filter extension:");

/* end WhenLoaded */}

/*** id - identify extension
*
* Purpose:
*  identify ourselves, along with any passed informative message.
*
* Input:
*  pszMsg    = Pointer to asciiz message, to which the extension name
*              and version are appended prior to display.
*
* Output:
*  Returns nothing. Message displayed.
*
*****************************************************************************/
void pascal id (
char *pszFcn                    /* function name  */
) {
char    buf[BUFLEN];                /* message buffer  */

strcpy (buf,pszFcn);
strcat (buf,ID);
DoMessage (buf);
/* end id */}




/
*****************************************************************************
**
** Switch communication table to the editor.
*/
struct swiDesc          swiTable[] = {
    {"filtcmd",     (PIF)(long)(void far *)SetFilter,    SWI_SPECIAL},
    {0, 0, 0}
    };

/
*****************************************************************************
**
** Command communication table to the editor.
** Defines the name, location, and acceptable argument types.
*/
struct cmdDesc    cmdTable[] = {
    {"filter",filter,0, KEEPMETA | NOARG | BOXARG | NULLARG |
    LINEARG | MARKARG | NUMARG | TEXTARG | MODIFIES},
    {0, 0, 0}
    };
```

Figure 4: Make File for the Filter Extensions

```
#
# makefile for the filter extensions
#
filter.obj: filter.c
        cl /c /Gs /Asfu filter.c

filter.mxt: filter.obj
        cl /Lr /AC /Fefilter.mxt exthdr.obj filter.obj

filter.dll: filter.obj
        cl /Lp /AC /Fefilter.dll exthdrp.obj filter.obj skel.def
```

Figure 5: Microsoft Editor User Arguments

User Argument	Argument Meaning
no arg	No argument specified. Default action of the function is executed.
nullarg	A null argument. Often used to modify the action of a function or to specify the range over which the function occurs as "cursor to end of line."
textarg	Typed-in textual argument. Used to specify textual information, such as search strings.
numarg	Typed in numeric argument. Often used to specify a specific line number or a number of lines over which a function is to be applied.
markarg	Typed-in argument that references a named mark elsewhere in the current file. Besides moving to a marker position, this is often used to specify the range over which a function is to be executed (current position to marker position).
streamarg	Arg created by moving the cursor, consisting of a range of characters on a single line or a range of characters that spans multiple lines but includes *all* characters between the beginning and end of the range, regardless of position.
boxarg	Arg created by moving the cursor, consisting of a box of characters spanning multiple lines. Only those characters within the box are included. (The distinction between boxarg and multiline streamarg is made by the function invoked.)
linearg	Arg created by moving the cursor, consisting of a range of lines, in their entirety.
meta	The *meta* command prefix is *not* an argument but is used to modify an editing function. In general it is used to either take a command to some extreme (*home* goes to first nonblank character on a line, while *meta home* goes to column one, regardless), or to cause a function to lose some information (*exit* saves the current file, if changed, and exits the editor, while *meta exit* immediately exits the editor without saving any changes).

WHEN THE EDITOR LOADS AN EXTENSION, THE INITIALIZATION ROUTINE WHENLOADED IS CALLED. IT NEED NOT DO ANYTHING AND CAN RETURN IMMEDIATELY. WHENLOADED IS USEFUL FOR DEFAULT KEYSTROKE ASSIGNMENTS.

tine in the extension to interpret the text.

The filter function contains a single textual switch:

```
{"filtcmd", (PIF)(long)
  (void far *)SetFilter,
  SWI_SPECIAL}
```

The string "filtcmd" defines the function's name as seen by the user. SetFilter is a pointer to the routine to be executed when that switch is set. (The elaborate casting performed here avoids compiler warning messages.) SWI_SPECIAL indicates that you have a textual switch, requiring special processing.

The editor calls SetFilter whenever a new assignment to the filtcmd switch is made. It receives as a single parameter a pointer to the new text assigned to the switch, which SetFile can process as desired. In this case, the switch sets the name of the filter program we want filter to use. We copy away the entire text of the switch to a static location for use when filter is executed.

By providing a switch for the filter command, the user can include a default value in TOOLS.INI. For example:

```
[M]
filtcmd:SORT <
```

When the filter function is executed, the SORT command will be employed as the filter program, using redirected input. You can also change filtcmd on the fly.

Extension Functions

When the user executes an extension function, the editor calls the routine defined in the function definition table. The editor passes arguments to the C function that describe the user's arguments to the function, a flag indicating whether the meta prefix was in effect, and the value of the keystroke invoking the function. In our example, they would be:

```
flagType pascal
  EXTERNAL filter (
  unsigned int argData,
  ARG far *pArg,
  flagType fMeta
)
```

The argData parameter is the keystroke that the command was invoked with. For example, if the key A is used, then argData is passed as 65, the ASCII value for A. However, filter doesn't use this parameter. The graphic function, which is the function that handles keystrokes that you use to type in text, simply places the value of argData as a character into the file being edited.

The fMeta parameter is a TRUE/FALSE flag indicating

the state of the meta command prefix at the time the command was executed.

Most of the information normally used by an editing function is present in the pArg parameter, the pointer to the argument structure. This structure is defined as a type indicator followed by a union of six different structures; only one of these structures is used at a time. The type of structure used is based on the type of argument the user passed to the function.

In our example, the filter function begins by taking actions and accessing the appropriate structure elements, based on the type of argument passed:

```
switch (pArg->argType) {
  case NOARG:
            :
  case NULLARG:
            :
  case LINEARG:
            :
  case BOXARG:
            :
  case TEXTARG:
            :
}
```

At this point there is an interesting discrepancy. In the command description table, we specified several different argument types that the actual function does not seem prepared to handle. NUMARG, for example, is included in the cmdDesc table, but not in the function itself.

Some arguments, such as NUMARG and MARKARG, are converted by the editor to another, more basic argument type prior to calling the extension. For example, a numarg, which is a count of the lines specified by the user, is converted to a LINEARG, which explicitly enumerates the range of lines. While the way they are specified by the user is different, they both define a range of lines. Therefore the manner in which they are passed to an extension function is the same; they are both passed as a LINEARG. See **Figure 5** for a list of all user argument types and **Figure 6** for a list of the base types they can be mapped into.

The extension function's access to the files being edited and the editing environment is defined by the internal editor functions, or "call-back" routines, which are listed in **Figure 7**. An extension function generally operates by calling internal editor functions to gain access to the file being edited and to retrieve, add, replace, or modify text therein.

The Filter Function

Up to this point, we've defined everything that goes around our function—the tables, the initialization routines, and the function parameters—and we've mentioned the routines that can be called by the extension function. Let's now take a look at the body of the filter function.

Remember that the purpose of filter is to replace selected text with the results of running that

WHEN THE USER EXECUTES AN EXTENSION FUNCTION, THE EDITOR CALLS THE ROUTINE DEFINED IN THE FUNCTION DEFINITION TABLE. THE EDITOR PASSES ARGUMENTS TO THE C FUNCTION THAT DESCRIBE THE USER'S ARGUMENTS TO THE FUNCTION, A FLAG INDICATING WHETHER THE META PREFIX WAS IN EFFECT, AND THE VALUE OF THE KEYSTROKE INVOKING THE FUNCTION.

Figure 6: User Arguments as Passed to Extension Functions

ARG Received	Possible User Argument Specified
NOARG	No argument was specified.
NULLARG	A null argument was specified.
TEXTARG	A textual argument specified. May be the result of: • having been typed in explicitly, • a one-line boxarg, if the BOXSTR flag is set in the cmdDesc table • a nullarg, if NULEOL flag is set (text is cursor to end of line) • a nullarg, if NULEOW flag is set (text is cursor to end of word)
STREAMARG	An arg was created by moving the cursor. Can also be the result of a mark arg.
BOXARG	An arg was created by moving the cursor, consisting of a box of characters. Can also be the result of a mark arg.
LINEARG	Arg created by moving the cursor, consisting of a range of lines, in their entirety. Can also be the result of a num arg or mark arg.

THE EXTENSION FUNCTION'S ACCESS TO THE FILES BEING EDITED AND THE EDITING ENVIRONMENT IS DEFINED BY THE INTERNAL EDITOR FUNCTIONS, OR "CALL-BACK" ROUTINES.

text through an arbitrary filter program. The arguments to the filter function are shown in Figure 8.

Note that we rely on markarg and numarg being converted to one of the other argument types by the editor before the function is called. Because the editor performs this conversion automatically, any function that takes a linearg and boxarg can easily take numarg and markarg by just setting the appropriate flag in the cmdTable.

We noted earlier that filter operates in five basic steps. The function will begin by calling FileNameToHandle to get a handle to the file currently being edited. This is a common first step in any extension function, as the handle to the current file is required in several internal editor function calls to identify the file that is operated on.

The function will then call DelFile to delete the contents of the pseudo-file, which was created in the initialization routine WhenLoaded. DelFile does not remove the file, since we have to continue to operate on it, but simply deletes the contents of the file.

The first step copies the region selected by the user from the user's file (handle pFile) to our pseudo-file (handle pFileFilt). The type of copy used is based on the type of argument that the user specified. For example:

```
case BOXARG:
 CopyBox (pFile,
  pFileFilt,
  pArg->arg.boxarg.xLeft,
  pArg->arg.boxarg.yTop,
  pArg->arg.boxarg.xRight,
  pArg->arg.boxarg.yBottom,
  (COL) 0,
  (LINE) 0);
 break;
```

This code fragment is executed only if the user argument defines a BOXARG. You use CopyBox to copy the box specified by the user into the pseudo-file. The other argument types are handled similarly, except for TEXTARG, which is treated as a special case, so that by entering:

```
arg text-argument filter
```

the user can specify the actual command to be used as the filter. This is the same command as specified by the filtcmd switch. For example, assuming the default keystroke for arg and Alt-F for filter, to use SORT with redirected input as the filter command, the user would type:

```
Alt-A "SORT <" Alt-F
```

When this type of argument is

Figure 7: Microsoft Editor Internal Functions

Name	Function
AddFile	Add file to editor's file list
BadArg	Display "bad argument" message
CopyBox	Copy BOX from one file to another
CopyLine	Copy LINEs from one file to another
CopyStream	Copy STREAM from one file to another
DelBox	Delete a BOX of characters from a file
DelFile	Delete entire contents of a file
DelLine	Delete a range of LINEs from a file
DelStream	Delete a STREAM of characters from a file
DoMessage	Display message on dialog line
Display	Cause display to be redrawn
fExecute	Execute macro string
FileLength	Return length of requested file
FileNameToHandle	Return handle to current or named file
FileRead	Read disk file into file being edited
FileWrite	Write disk file from file being edited
GetCursor	Return current cursor position
GetLine	Return line of text in file
KbHook	Restore editor's control of keyboard
KbUnHook	Release editor's control of keyboard
MoveCur	Move cursor to new position in current file
pFileToTop	Make specified file the "current" file being displayed
PutLine	Place line of text into file
RemoveFile	Remove file from editor's list.
ReadChar	Read character from keyboard
ReadCmd	Read editing command interpreted from keyboard
Replace	Replace character in file
SetKey	Make keystroke assignment

Figure 8: Arguments Used in the Filter Extension

noarg	Filter the current line
nullarg	Filter text from cursor to end of line
boxarg	Filter the selected box
linearg	Filter the selected lines
numarg	Filter a number of lines
markarg	Filter between cursor and a mark

detected, the filter routine just calls SetFilter to save the command string and returns.

The second step just uses a call to FileWrite to write the contents of the pseudo-file to a disk file named FILTER1.TMP.

The third step begins by building the text of the actual filter command using calls to the C run-time routines strcpy and strcat, which can be used because they fall into the category of acceptable C run-time routines available to extensions. The user-specified filter command is concatenated with the two temporary filenames defined in the data area. The net result of the concatenation, using the SORT command above, is:

```
SORT < filter1.tmp
  >filter2.tmp
```

This sorts the contents of FILTER1.TMP and creates FILTER2.TMP.

The DoSpawn routine, described below, is called to execute the filter command we created, and then the disk file filter2.tmp is created.

In the fourth step, before you read in the results of the filter, you use DelFile again to delete the previous contents of the pseudo-file. Then FileRead, the counterpart to FileWrite, is used to read in the physical file filter2.tmp to the pseudo-file. The net result is that the pseudo-file now contains the results of running its original contents through the filter.

The final step begins by calculating the maximum width of the text that resulted from the filter. In some cases, this width will be different from the width of the original text, and for some argument types, such as boxarg, the correct width should be used when replacing the text.

You determine the maximum width by using the following code:

```
cLines =
  FileLength (pFileFilt);
cbLineMax = 0;
for (iLineCur = 0;
     iLineCur < cLines;
     iLineCur++)
cbLineMax =
  max (cbLineMax,
       GetLine (iLineCur,
       buf, pFileFilt));
```

FileLength returns the current number of lines in the file referenced by the handle pFileFilt. The loop reads each line of the file into a buffer. The text is ignored, but the GetLine routine returns the length of each line it has read.

Then, depending on the argu-

C 5.2 Library Routines and Editor Extensions

Because of the mechanics of building Microsoft Editor extensions, there are certain restrictions on the use of C run-time library routines. These restrictions stem from two issues.

Extensions must be compiled with SS != DS, that is, when executing, the extension function will operate with the same stack as the editor itself but will have access to its own local data segment, which is *not* the same as the stack segment or the editor's own data segment. This restricts C run-time library routines to the compact model set of libraries.

Extensions do not have C start-up support. The C start-up is responsible for initializing a fair number of data items and the C run-time environment. This restricts the use of C run-time library functions to those that do not rely on this initialization.

The following list summarizes functions available to extensions from the standard compact model library. This list uses the function categories from Chapter 4 of the *Run-Time Library Reference*.

Category	Availability
Buffer Manipulation	All
Character Classification and Conversion	All
Data Conversion	All except strtod
Directory Control	All except getcwd
File Handling	All
Graphics	None
Input and Output: Stream	None
Input and Output: Low-level	All except write in binary mode
Input and Output: Console and Port	All except cgets, cprintf, and cscanf
Math	None
Memory Allocation	None
Process Control	None
Searching and Sorting	All except qsort
String Manipulation	All except strdup
System Calls: BIOS	All
System Calls: MS-DOS	All except int86, int86x
Time	All except ctime, gmtime, localtime, and utime
Miscellaneous	All except assert, getenv, perror, putenv, _searchenv

REMEMBER THAT THE PURPOSE OF THE FILTER IS TO REPLACE THE SELECTED TEXT WITH THE RESULTS OF RUNNING THAT TEXT THROUGH AN ARBITRARY FILTER PROGRAM.

WE'VE TAKEN A LOOK AT THE PROGRAMMABILITY THAT THE MICROSOFT EDITOR PROVIDES: KEYSTROKE ASSIGNMENTS, WHICH PERMIT EDITOR FUNCTIONS TO BE TIED TO KEYSTROKES AND EXECUTED; MACROS, WHICH ENABLE EDITOR FUNCTIONS TO BE COMBINED IN COMPLEX SEQUENCES; AND EXTENSIONS, WHICH CREATE ENTIRELY NEW EDITING FUNCTIONS.

ment type again, you delete the original text in the user's file and replace it with the contents of the pseudo-file. Using BOXARG as an example again:

```
case BOXARG:
 DelBox (pFile,
  pArg->arg.boxarg.xLeft,
  pArg->arg.boxarg.yTop,
  pArg->arg.boxarg.xRight,
  pArg->arg.boxarg.yBottom);
 CopyBox (pFileFilt,
  pFile,
  (COL) 0,
  (LINE) 0,
  cbLineMax,
  cLines-1,
  pArg->arg.boxarg.xLeft,
  pArg->arg.boxarg.yTop);
 break;
```

DelBox deletes the box specified by the user, and CopyBox copies in the contents of the pseudo-file as a box.

Finally, before leaving, the function uses DoSpawn again to execute DOS commands to delete the two temporary files that you created along the way.

DoSpawn itself could be implemented in a number of ways; unfortunately, using the C run-time process-control routines is not one of them. An alternative method is to use the intdosx call under DOS or call DosExecProgram under OS/2 to have the operating system execute the program. Our approach is to use the fExecute function instead to execute the editor's shell function.

DoSpawn begins by using strcpy and strcat to create the following string:

```
arg "command" shell
```

where "command" is the text pointed to by the szCmd parameter passed to DoSpawn. By then passing this string to fExecute, the editor shell function causes the command to be executed by the operating system.

Conclusion

We've taken a close look at the three types of programmability that the Microsoft Editor provides: keystroke assignments, which permit editor functions to be tied to keystrokes and executed when the keystrokes are pressed; macros, which enable editor functions to be combined in complex sequences and executed as a single function; and extensions, which create entirely new editing functions. Each is an important and useful tool for customizing and maximizing the power and efficiency of text editing with the Microsoft Editor. ❑