Chapter R1
User Interface

Revised: Jan. 21, 2011

R1.1 INTRODUCTION

The first module of the MPX project requires you to construct a user interface. The user interface is the central mechanism by which you will access and control the evolving elements of the complete MPX. Therefore it is a natural starting point for the project. The designation R1 indicates that this is the first *required* module. The software you implement here will be needed for most other modules, including the final module. If your instructor does not choose to assign you to develop your own version of this module, then you will be provided a model version for use in future projects.

The user interface presented in Module R1 has the form of a simple command interface, called COMHAN (short for command handler). Its behavior will be similar (though not identical) to the basic command handler of a Linux shell or Windows command line.

A command line interface is chosen in part because it is the simplest model to implement. Other user interface paradigms could be used for your version of MPX, such as a menu interface or GUI. These possibilities are discussed in Section R1.7, Optional Features, at the end of the chapter.

In a sense, COMHAN must be considered as something separate from the rest of MPX. Its role is closer to that of an application program, and it is a user rather than a provider of system services. However, it is only through COMHAN that you will be able to test and demonstrate the system services you develop. Therefore COMHAN is of central importance throughout the MPX project.

After initialization, this procedure will execute a continuous loop with three principal tasks: to *read* a command entered via the system console keyboard, to *analyze* the command, and to *execute* the command. Before accepting each command, the handler should provide a prompt to inform the user that a new command may be entered. Thus COMHAN will have the following basic form:

```
initialize
display opening message
while not done
```

```
        display prompt
        accept command
        analyze command
        execute command
    end while
    display closing message
    cleanup
    return to host system
```

In some operating systems, most commands cause a separate executable program file to be loaded and executed. Moreover, often any executable file may be invoked as a command by typing its file name. In MPX, for simplicity, a fixed set of commands is "built in," and all commands invoke procedures that are part of MPX itself.

Module R-1 requires the implementation of both the command handler itself and a basic set of commands. Additional commands will be added when later modules are implemented. The commands you include in this module should support, as a *minimum*, the following operations:

- Display help information.
- Display the current MPX version number.
- Display or set the date.
- Display a directory of the available MPX process files.
- Terminate execution of MPX.

Each of these command operations is described in more detail below. Additional commands may be specified by your instructor; some examples are given in the Optional Features section of this chapter.

R1.2 KEY CONCEPTS

Command Handling

This section will discuss some of the issues to be considered in command handling, the principal concept to be studied in this module. The principles of good user interface design are discussed in some detail in the text. We will assume throughout this chapter that you will be using a command-line model for your handler. If not, then a different set of issues will have to be considered.

As discussed in the text, there are many possible forms for the detailed structure of a command line interface. Some of the important issues are:

- Will arguments be included on a command line, or should parameter information be obtained by another means such as explicit queries?

- Should a keyword or positional syntax be used? If keywords are used, what is their form?

-

- What characters are used to separate commands and arguments? Are multiple blanks permitted?
- How long can individual commands or arguments be? Are there any restrictions on their form?
- Will you provide a means for enclosing arguments in quotation marks, so they may contain special characters such as spaces?
- Can commands be abbreviated? Should command recognition be case-sensitive?
- What "wild cards," if any, will be permitted in command arguments? Will these be handled by the command analyzer or by the individual command procedures?
- Should command names generally be short but cryptic or long and descriptive? How will you ensure consistent naming for similar operations, such as commands that display various information on the screen?
- Will you allow commands consisting of more than one word (*e.g.* set date, show version, etc.)
- What type of a help facility will be provided? Will there be a standard format for system messages?

Your instructor may specify answers for some of these questions. Other decisions you must make yourself.

Additional design issues might relate to terminal handling operations, such as line editing, echoing, and typeahead. In Module R1, however, COMHAN will rely on the host operating system for terminal handling, obtaining input lines using the sys_req support procedure described below. During a later stage of the project, you may design your own terminal handler.

More sophisticated features found in some command handlers include the ability to repeat previous commands, process commands from a script, define aliases for commands, etc. These could be considered as optional enhancements for COMHAN, and some of them are discussed in the Optional Features section.

Remember that commands will be added to COMHAN in the future. COMHAN should be carefully structured to allow *easy* addition of new commands.

This is your chance to develop the "ideal" user interface. Remember all those complaints you have had about command interfaces you have used? See if you can design a command handler that addresses those complaints.

R1.3 DETAILED DESCRIPTION

The Command Handler

A general outline for the operation of COMHAN was given in Section R1.1, and design issues to be considered were covered in Section R1.2. To provide you with some opportunity to develop your own design according to these principles, we will *not* specify the structure and operation for COMHAN in complete detail. Instead, we will provide some general guidelines, and discuss some of the design issues you should consider. Of course, your instructor may provide a more detailed specification.

In general, the heart of COMHAN will be a single procedure that executes the algorithm shown in Section R1-1. We recommend that the mechanism for processing commands be separated from all procedures which execute specific commands. Therefore the COMHAN main loop, with any related supporting procedures and data structures, should form one distinct source file within your project.

Although MPX is a continually evolving system, it is certainly desirable to anticipate the needs of future modules, so that completed elements such as COMHAN will require few if any changes. Initially you may think of COMHAN as the main procedure for MPX, a basic processing loop that executes a small, fixed set of commands. However, to accommodate the changing needs of future modules, we recommend the following steps:

- Do not make COMHAN itself the main procedure. Instead, provide a separate main procedure, in a separate source file, responsible for performing initialization, invoking COMHAN, and final termination.

- Provide a special closing procedure in the main file, which should be called by COMHAN *just before* it terminates. In Module R1 this procedure will do nothing, but it will have an important role in later modules.

- Remember that, although COMHAN will always retain the basic commands included in Module R1, many additional commands will be added in future modules. All information about specific commands should be obtained from a table or list rather than being embedded in the COMHAN source code.

Basic Commands

This section describes the required command operations to be implemented in Module R1. Each description includes *suggestions* for command and argument definitions. You may modify these suggestions as you develop your own command format. However, all of the specified functionality should be included in your system.

Display Help Information

A command operation should be included to display "help" information for MPX and for each command. The minimum help information available should include a summary list of all

-

commands available, and a more detailed description of each specific command. The actual help information should be stored in one or more files so it may be easily edited.

The obvious name for the help command is `help`. Alternate names such as `?` (a question mark) may also be accepted. A natural approach is to allow one optional argument. A help command with no arguments produces the summary list. If an argument is given, it must be a command name, and help for that command is displayed.

As far as possible, each command description should be short enough to be displayed in its entirety on a single screen. If longer help files are allowed, your program must include provisions for displaying their contents one screen at a time. For example, you may display up to a limited number of lines, then wait for the input of either a space to continue, or the enter key to terminate the help display. This "screen-at-a-time" display is similar to the *more* facility of many Unix environments.

Display the MPX version

A command should be provided to display a brief description of the version of MPX that is currently running. As a minimum, this information should specify the date that the system was last modified. You may decide to include a time display, authors' names, a specific version number, etc. This information should be kept up to date as MPX evolves.

Display or Change the System Date

One or more commands should be provided to display and to change the system date. Our suggested structure for this facility is a date command with optional arguments. The command without arguments displays the current date. If arguments are given, they should represent a valid day, month, and year in a suitable format. In this case the command should change the system date to the specified new value.

Your procedure should check all date arguments for validity, and not accept meaningless dates. Remember to consider the differing numbers of days in each month, and to make allowances for leapyears. You may allow as wide a range of years as you wish.

Two support procedures are provided to set and change the system dates. These procedures are `sys_get_date` and `sys_set_date`; they are described in detail in Section R1.4. These procedures do *not* directly access the actual system date. Instead, they read and write a global variable maintained by the support software. This variable is initialized from the real system date only when MPX is initialized.

Display the MPX Process Directory

The purpose of this operation is to display information about all files present in a specific directory that (apparently) contains executable MPX processes. These files are recognized by their filename extension: ".MPX". Although actual process files are not used until later modules of the project, you should create or copy some files with the MPX extension to enable testing of the directory display operation.

-

The minimum information to be displayed is the name and size (in bytes) of each file. A useful and attractive display format should be used, including a suitable heading.

An appropriate form for this command would be a single word without arguments such as directory. Optionally, you may use arguments to provide variations in the listing format, such as inclusion or exclusion of the file sizes. An appropriate directory should be chosen to be searched by this command; this could be either your current directory (the one containing your executable MPX program) or a special subdirectory with a name such as MPXFILES. Optionally, you could allow the user to specify a directory by giving its pathname as a command argument.

This command makes use of three support procedures: `sys_open_dir`, `sys_get_entry`, and `sys_close_dir`. The `sys_open_dir` procedure opens the appropriate directory for searching. It will return an error code if the directory cannot be found. Each time the procedure `sys_get_entry` is called, it returns information about the *next* file in the chosen directory, if any, that is recognized as an MPX file. This information consists of the file name and file size. Finally, `sys_close_dir` should be used to close the directory when the search is complete. Each of these procedures is described in more detail in Section R1.4.

Terminate MPX

This operation stops execution of MPX and returns to the host operating system. A single command such as quit with no arguments should be used. You should require confirmation from the user before performing this operation. For example, your program could display a message such as:

```
Are you sure you want to terminate MPX?
```

And the user may be required to type an appropriate response.

R1.4 SUPPORT SOFTWARE

Overview

This section describes the support software provided for use with Module R1. As explained in Chapter I1, this software is intended to provide a consistent interface between your command handler and basic command procedures, on one side, and the resource management facilities of the operating system on the other. In Module R1, these routines obtain services directly from the host environment. During later modules you will replace some of these support routines with your own software. However, the interface should not change, so the replacement should require no modifications to software you have already implemented.

Ten support routines are provided for the Module R1 commands. They are:

`sys_init`	Initialize the MPX support software
`sys_exit`	Terminate MPX
`sys_req`	Perform terminal input or output

-

sys_alloc_mem	Allocate a block of memory
sys_free_mem	Free an allocated memory block
sys_get_date	Obtain the current system date
sys_set_date	Change the current system date
sys_open_dir	Open an MPX file directory
sys_get_entry	Get an MPX file entry from the directory
sys_close_dir	Close an open directory

These support procedures are found in the supplied object file MPX_SUPT.OBJ. Their defining prototypes are in the file MPX_SUPT.H. Each procedure is explained in detail below.

Most support procedures can return one or more error codes to indicate that a problem has occurred. All error codes will be negative integers in the range -101 through -199. Each of these error codes is assigned a symbolic name of the form ERR_SUP_abcdef. These names are defined in the header file MPX_SUPT.H. The specific error codes returned are listed below with each procedure.

sys_init

The sys_init procedure performs system-dependent initialization for MPX. This procedure should be called at the beginning of your main program. It will perform different actions depending on the modules currently included in MPX. For Module R1 this routine performs no action.

To advise the support procedures which modules are presently included in your system, you must provide sys_init with a code specifying the included modules. This code can be constructed from constants of the form MODULE_Rn and MODULE_On, which are defined in MPX_SUPT.H. For example, in Module R1 you would call sys_init with an argument of MODULE_R1. If you have completed the required modules and are also including optional modules O2 and O5, your code would be MODULE_R4 + MODULE_O2 + MODULE_O5. The final integration phase is designated by the code MODULE_F (plus any included options).

The prototype for sys_init is:

```
int sys_init (int modules);
```

The returned value will be zero if no problem occurred; otherwise it will be an error code. The error can occur if the specified module code is invalid. The returned error code in this case is:

ERR_SUP_INVMOD	invalid module code

The symbol for this code is defined in MPX_SUPT.H.

sys_exit

The sys_exit procedure performs system-dependent cleanup operations and returns to the host operating system using the C *exit* function. It should be called when you are ready to

exit MPX; it does not return to the calling program. Like `sys_init`, its actions are dependent on the modules included, and it has no special action for Module R1. The prototype for `sys_exit` is:

```
void sys_exit (void);
```

`sys_req`

The `sys_req` function is used in Module R1 primarily to read a line from the keyboard. In later modules this function will be used to process a number of other service requests generated by MPX processes. Functions from the standard C library `stdio` could also be used for keyboard input. However, consistent use of `sys_req` ensures a simple transition in later modules when you will begin using your own terminal device driver.

Keyboard input via `sys_req` is specified by the device code `TERMINAL` and the operation code `READ`. Each of these symbols is defined in `MPX_SUPT.H`. `sys_req` reads characters typed at the keyboard into the buffer until an *enter* key is typed or until `buf_size-1` characters have been entered. The string will always be ended with a null character. The routine places a *newline* code in the buffer if *enter* is typed.

`sys_req` can also be used for output to the terminal screen. An operation code of `WRITE` is used for this case (along with the `TERMINAL` device code). However, it is permissible and probably more convenient to use the *printf* function in the standard `stdio` library. Output calls to *printf* and `sys_req` can be freely mixed. The support software will ensure that both procedures use the same low-level mechanism, even in later modules when your own terminal driver is included.

Two important screen control functions are also supported by `sys_req`: clearing the screen and explicit cursor positioning. These operations are invoked by specifying the operation codes `CLEAR` or `GOTOXY`. For the clear operation, the content of the buffer is ignored. For the cursor positioning operation, the first two character positions in the buffer should contain non-negative integers specifying the desired horizontal and vertical positions, in that order. The upper left corner of the display area is considered to be position (0,0).

The prototype for `sys_req` is:

```
int sys_req (int op_code, int device_id,
     char buffer[], int *buf_size_p);
```

The meaning of the parameters is as follows:

op_code: specifies the operation performed. For Module R1 the permitted operations are specified by the symbols READ,WRITE, CLEAR, and GOTOXY, which are all defined in MPX_SUPT.H.

device_id: specifies a code for the device for which a data transfer is requested. For Module R1 the only device used is the terminal. You should use the symbol TERMINAL for this parameter, which is defined in MPX_SUPT.H.

-

buffer: for input, specifies the address of a character array of sufficient size to hold the longest line you will accept. A size of at least 80 is recommended. Remember to allow extra positions for *newline* codes and the trailing null. For output, the buffer contains the text string to be output. You must define this array in your program.

buf_size_p: For input, a pointer to an integer value giving the total size of the buffer. For output, this value should specify the exact number of characters to be output (not counting the trailing null, if any).

If there was no error, the return value for `sys_req` will be a nonnegative integer giving the length of the string that was input or output. If there was an error, a negative value representing an error code will be returned. Possible error codes are:

ERR_SUP_INVDEV	invalid device id
ERR_SUP_INVOPC	invalid operation code
ERR_SUP_INVPOS	invalid character position
ERR_SUP_RDFAIL	read failed
ERR_SUP_WRFAIL	write failed (also used for clear and goto)

Symbols for these codes are defined in `MPX_SUPT.H`.

Here is an example of the use of this function to place the cursor at the home position (0,0) on the screen:

```
char buf[80];
int size;
int err;
     ...
buf[0] = 0;
buf[1] = 0;
size = 2;
err = sys_req(GOTOXY, TERMINAL, buf, &size);
if (err != OK) ...
```

sys_alloc_mem

The `sys_alloc_mem` procedure is provided to dynamically allocate blocks of memory. This facility may be used by COMHAN if a dynamic structure is chosen for the command table, as well as by selected command procedures both in this module and later modules.

The ANSI C functions `malloc` or `calloc` are also available for allocating memory blocks. However, in later phases of this project you may develop your own memory manager as a substitute for those provided by the system. To enable a clean transition to the use of your own procedures, `sys_alloc_mem` should be used exclusively for memory allocation.

The prototype for `sys_alloc_mem` is:

```
void *sys_alloc_mem (size_t size);
```

The behavior of sys_alloc_mem is equivalent to the ANSI C calloc function. The single parameter size specifies the size in bytes of the block to be allocated; this is typically obtained by the C sizeof operator. sys_alloc_mem returns a value of type pointer to void, which must be explicitly cast to your desired type. A null pointer is returned if the allocation failed. The allocated region comes from the "far heap"; it is always paragraph-aligned; and it is initialized to zero.

A typical sequence to create a new object of type mytype would be:

```
typedef ... mytype;
   ...
mytype    *newobj_p;
   ...
newobj_p = (mytype*) sys_alloc_mem((size_t)
          sizeof(mytype));
if (newobj_p == NULL) handle error;
```

sys_free_mem

In a similar fashion, sys_free_mem is provided to free blocks of memory previously allocated using sys_alloc_mem. This procedure should be used in preference to the equivalent functions of ANSI C and TURBO C, for the same reasons given above. Note that blocks allocated by sys_alloc_mem *cannot* be deallocated properly by any other procedure.

The prototype for sys_free_mem is:

```
int sys_free_mem (void *ptr);
```

The argument ptr is a pointer to the block to be freed; this should be a block previously allocated using sys_alloc_mem. If the free is successful, a value of zero is returned. Otherwise, a negative error code is returned; possible error codes are:

ERR_SUP_INVMEM	invalid memory block pointer
ERR_SUP_FRFAIL	free failed

Symbols for these codes are defined in MPX_SUPT.H.

sys_get_date

The sys_get_date function obtains the value of the MPX system date. The date is returned in a structure of type date_rec which must be provided by your program. Type date_rec is defined in MPX_SUPT.H. It has the following form:

```
typedef struct {
     int month;
     int day;
     int year;
     } date_rec;
```


-

The prototype for `sys_get_date` is:

```
void sys_get_date (date_rec *date_p);
```

The single parameter `date_p` is a pointer to the structure you have defined. After calling this function, the date will be stored in your structure. Element `month` will contain an integer in the range 1-12 representing the month. Element `day` will contain an integer in the range 1-31 representing the day of the month. Finally, element `year` will contain an integer representing the year number (A.D.) Note that this will be the *complete* year number; 1999 will be stored in its entirety, not simply as the value 99 (avoiding Y2K problems).

sys_set_date

The `sys_set_date` function stores a new value in the MPX system date. Note that changing this value does not change the real MS-DOS system date.

This function makes use of a structure of type `date_rec`, as described above. You must store the desired new date value in this structure before calling this function. The prototype for `sys_set_date` is:

```
int sys_set_date (date_rec *date_p);
```

If the date is successfully set, a value of zero is returned. Otherwise, a negative error code is returned; possible error codes are:

ERR_SUP_INVDAT	invalid date
ERR_SUP_DATNCH	date not properly changed

Symbols for these codes are defined in `MPX_SUPT.H`.

sys_open_dir

The `sys_open_dir` function initializes access to a directory to be searched for MPX files. Any previously open directory is automatically closed. This function accepts one argument, which is a character string specifying a pathname (absolute or relative) for the desired directory. The pathname may include a drive letter. If the argument string is null, the current directory will be searched. The prototype for this function is:

```
int sys_open_dir (char dir_name[]);
```

If the directory is successfully opened, a value of zero is returned. Otherwise, a negative error code is returned; possible error codes are:

ERR_SUP_INVDIR	invalid name or no such directory
ERR_SUP_DIROPN	error opening directory

Symbols for these codes are defined in `MPX_SUPT.H`.

sys_get_entry

The function sys_get_entry is used to extract MPX file entries from a directory which has been previously opened by the sys_open_dir function. Any file with the type extension .MPX is considered to be an MPX file. The first call to this procedure returns information about the *first* MPX file found, if any. Each subsequent call returns information about the *next* file found. If there are no more MPX files, an error code is returned The information returned for each file consists of the filename (*without* the .MPX extension), and the size of the file in bytes.

The prototype for sys_get_entry is:

```
int sys_get_entry (char name_buf[], int buf_size,
                   long *file_size_p);
```

The meaning of the parameters is as follows:

name_buf: specifies the address of a buffer to receive the filename, which you must define in your program. The size of the array should be at least 9 to hold a simple MS-DOS filename without the type extension. The function will place the filename in this buffer, terminated with a null character.

buf_size: specifies the size of the buffer name_buf. The function will store no more than buf_size - 1 characters, followed by a null.

file_size_p: A pointer to a variable of type *long* which you must define in your program to receive the file size. This is typically specified using address notation, *e.g.*, &file_size. The function will store the file size in bytes in this variable.

If the operation succeeds, sys_get_entry returns a value of zero. Otherwise, a negative error code is returned; possible error codes are given below. Note that normal use of this function consists of invoking it repeatedly until the error code ERR_NOENTR is returned.

ERR_SUP_DIRNOP	No directory is open
ERR_SUP_NOENTR	No (more) entries found
ERR_SUP_RDFAIL	The directory read failed
ERR_SUP_NAMLNG	The name was too long for the buffer

Symbols for these codes are defined in MPX_SUPT.H.

sys_close_dir

This function should be used to close the open directory after searching is completed. There are no arguments. The prototype for this function is:

```
int sys_close_dir (void);
```

If the directory is successfully closed, a value of zero is returned. Otherwise, a negative error code is returned; the possible error code is:

`ERR_SUP_DIRCLS`	error closing directory

The symbol for this code is defined in `MPX_SUPT.H`.

R1.5 TESTING AND DEMONSTRATION

It is important to test your project thoroughly and systematically to gain a high level of confidence in its correct operation. All general command handling features should first be tested with simple commands. Following this, each command you have implemented should be carefully exercised according to its specifications (which should be part of your documentation, explained below). It is especially important to test for boundary values and invalid input, as far as possible. For example, does the *set date* operation properly accept the *smallest and largest* values that should be valid? Does each command deal properly with invalid arguments?

Since each COMHAN may have a different specification, we cannot provide a detailed test plan applicable to your specific version. However, some of the key elements that should be part of your test *if they are a part of your design* include the following:

- Are all variations of command syntax accepted?
- Are invalid commands (including the empty command) handled properly?
- Are all valid abbreviations accepted?
- Is every command supported by the help operation? Are long helpfiles paged properly?
- Are invalid command names reasonably rejected?
- Is correct version information supplied by the version command?
- Does the date command display the correct date? Are valid changes accepted? Are all valid dates accepted, and all invalid dates rejected?
- Does the directory command correctly list all files with the .MPX type extension? Are the names and sizes correct? Is the display format neat and attractive? Are alternate directory names handled properly, including alternate drives, if you have implemented this feature?
- Does the termination command work properly? Can the user escape from this command if desired?

Using the considerations that we have listed, plus any additional ones that apply to your system, you should prepare a comprehensive *written* test plan which can be followed at any time to ensure thorough testing. A systematic, *repeatable* testing procedure can be an invaluable tool not only to ensure the correct working of your project on one occasion, but also to ensure that later changes and enhancements do not cause trouble with earlier components. Moreover, the test procedure serves as an effective demonstration to convince your most important skeptic (your instructor) that the project is indeed correct.

-

R1.6 DOCUMENTATION

General documentation requirements for MPX modules have been described in Chapter I1. Remember to maintain meaningful, correct comments throughout your code, and to provide appropriate comment headers for each file and for all procedures. Update file dates and change logs *every* time you modify a file!

At this time you should create an initial version of the MPX *User's Manual*. This manual should contain at least the following elements:

- Overview of MPX
- General features of the COMHAN user interface
- How to input commands
- Detailed description of *each* command currently included
- Summary of error messages

Remember to write your *User's Manual* so that *you* would be happy with it!

An initial *Programmer's Manual* should also be created, as described in Chapter I1. This manual should carefully specify all of the files, functions, and data structures that are part of your Module R1 software.

Remember that this manual is intended for future programmers who must maintain the MPX software. Ask yourself the question, "would I be happy if I were given the responsibility to maintain this program using this manual?"

We will build on these two manuals as the implementation of MPX-PC progresses. You will add command and procedure documentation to both manuals in the future. We strongly recommend that you prepare these manuals in such a way that they can be easily updated.

R1.7 OPTIONAL FEATURES

This section briefly discusses some optional extensions and variations that you or your instructor may choose to include as part of the Module R1 project. Many of these options have been mentioned earlier in this chapter. They may be suitable for inclusion in longer courses, upper-level courses, or as optional extra-credit assignments.

The most ambitious option would be a change in the basic user interface paradigm. A **menu system** could be used to replace the command-line syntax. In this case all valid commands could be presented as a numbered list. A command is specified by typing a number. When the command is selected, additional menus or prompts would be needed to obtain needed arguments.

Your computer is probably equipped with a pointing device such as a mouse or touchpad. You may wish to make use of these input devices to enhance the menu model by allowing "point-and-click" item selection. Mouse support is not provided with the standard MPX software, so you will need to develop your own support routines for this purpose.

-

The use of **graphics** may further enhance your interface. A full graphical user interface (GUI) would include commands represented as icons or buttons that are selected by a pointing device. In addition, prompts or information display could make use of multiple windows and other GUI devices. A simpler use of graphics might involve selected use of color or special fonts, or display of a fancy logo when your program is initialized.

Graphics programming can be difficult and is well beyond the scope of this manual. In addition, if you choose to rely on graphics, you should realize that many different graphic models are used on different PC versions. You may be limiting the operation of your project to specific hardware or software configurations that can support the type of graphics you are using.

The remainder of this discussion will continue to assume that a command-line model is being used. In this case a few optional features you might consider are:

- Remember the most recent command, and allow the user to repeat it in a simple way. This could be extended to a **history** concept as provided by some UNIX shells.

- Allow a series of commands to be read from a **command file** or **script**. You will need a command to invoke such a script such as RUN or EXEC, and you will need a method for keeping the script open while other commands are executing. You must consider whether your script facility will allow arguments, and whether nested scripts will be allowed.

- Allow the standard MPX prompt to be changed. A simple command could be used to specify a new prompt or return to the default one.

- Allow **aliases** to be defined for simple commands or, more ambitiously, for complete command lines. A full alias facility should allow the user to display aliases, delete aliases, etc. Consider whether aliases can be abbreviated like commands, whether commands can be abbreviated *during* alias definition, and whether aliases will be accepted as arguments by the help command.

R1.8 HINTS AND SUGGESTIONS

The command handler project is a core element of MPX-PC. It will be enhanced in future modules. Eventually, it becomes a system process within MPX-PC. It is important that it be structured so that it can be easily maintained and enhanced.

Remember to consider the need to add new commands, and make this process as easy as possible. In general the routines for scanning and interpreting commands should be separated from the routines for executing them. Commands and aliases (if supported) may be stored in a dynamic list structure rather than a fixed-size array to allow easy expansion.

MPX is an evolving software system that will eventually grow fairly large. A modular structure is essential. Your text discusses various considerations to be used in modularizing operating system software. It is clearly not a good idea to place all your software in a single source file, but it is equally imprudent to use a separate file for every C function. In Module R1,

-

for example, the main program and command processing routines should clearly be separated from the actual command procedures. However, all of the command processing routines could be grouped in a single file, while the basic commands could form one or more additional files.

Although you may be tempted to hope that your program's operation will be error-free, the truth is that errors *always* occur, and a well-designed software system will devote a substantial portion of its code to dealing with errors. Follow a defensive programming strategy! We recommend the following steps:

- With rare exceptions, every function should check the validity of all of its input parameters and values.
- Most functions should respond to error conditions by returning an appropriate code to their caller. Only a few designated high-level functions should take direct action such as displaying an error message or aborting or restarting a major activity.
- Examine the error code system used by the support routines described above. Develop a similar system for your own software.
- Plan to devote at least a quarter of your design effort to error handling.

In addition to the specific commands assigned by your instructor, you might consider the implementation of additional commands that would aid in verifying or debugging the correct behavior of your software. For example, a command to display the content of your command table and other key data structures may be very helpful.

Good luck!

-