This paper discusses the purpose and design of a program called the System Productivity Facility (SPF). Perspective is provided by means of a brief summary of the earlier Structured Programming Facility (also termed SPF) and the requirements that led to a transformation of the earlier program into a new cross-system dialog manager. The new control facilities are explained to illustrate how the dialog manager supports a wide variety of interactive applications. Ways in which application development is simplified in the areas of data handling and display processing are explored. The purpose of the new table and file tailoring services is explained, and the error recovery philosophy is described.

System Productivity Facility

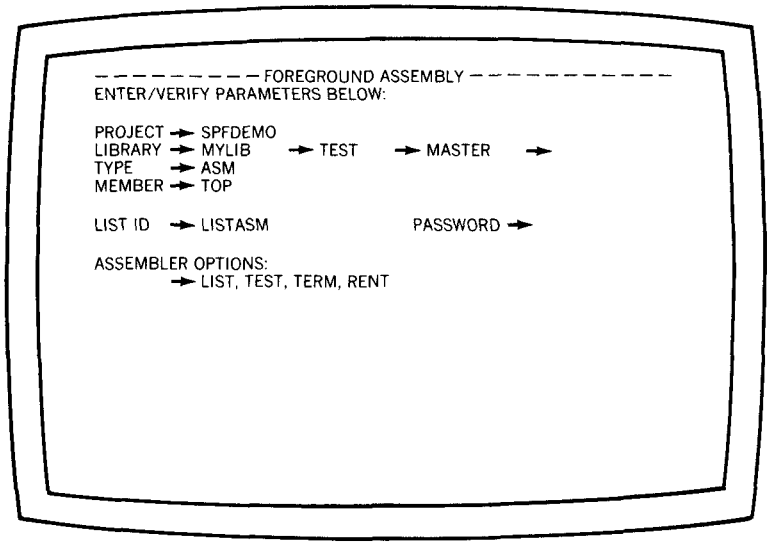
by P. H. Joslin

The System Productivity Facility (SPF) is an IBM program product that is designed to simplify the development of interactive applications. It replaces the earlier Structured Programming Facility programs for TSO and CMS. All functional capabilities of the earlier Structured Programming Facility (hereafter referred to as "old SPF") are included in the new product, which is why the acronym SPF is retained. The new product name, however, signals a significant new dimension and emphasis for SPF. To put that new dimension into perspective and to explain its evolution and significance, we start with a brief look at old SPF.

Structured Programming Facility

The Structured Programming Facility (old SPF) was introduced into the Time-Sharing Option (TSO) environment in June 1975 and into the Conversational Monitor System (CMS) environment in September 1979. It was designed to assist in programming development and to improve programmer productivity through exploitation of display technology. It was one of the first programs, for example, to prompt the user through a sequence of operations via menus. It provided full-screen edit and browse capabilities, with four-way scrolling of data (up, down, left, and

Figure 1 Foreground assembly display example



right), and allowed the screen to be split into separate partitions for parallel operation of different functions. The old SPF also used program function (PF) keys for frequently repeated operations, and provided an on-line tutorial to assist the new or occasional user.

One of the ways old SPF improved productivity was to reduce the need for a user to learn and master complicated command languages and job control statements. As an example, consider the foreground assembly display shown in Figure 1. The user is prompted for information about the source data to be assembled, the sequence of libraries to be searched for macros and copied members, and options that control the assembly processing.

From the user-entered information, the libraries are searched for the member to be assembled, and TSO or CMS commands are then constructed and executed. In the TSO environment, the following command is generated from the user input shown in Figure 1:

```
ASM 'SPFDEMO.TEST.ASM(TOP)'
    LIB('SYS1.MACLIB',
        'SPFDEMO.MYLIB.ASM',
        'SPFDEMO.TEST.ASM',
        'SPFDEMO.MASTER.ASM')
    LOAD('SPFDEMO.MYLIB.OBJ(TOP)')
    PRINT(LISTASM)
    LIST,TEST,TERM,RENT
```

(This example assumes that the member was found in the TEST library.)

In the CMS environment, the generated commands would include commands to link and access the appropriate minidisks and commands to build temporary files (if required), as well as an ASSEMBLE command.

Similar displays are provided for background processing, where the user-entered information causes Job Control Language (JCL) or CMS batch commands to be generated and submitted as a job.

These capabilities not only reduce the user's dependence on command and job control languages, they also reduce keystrokes and minimize the chance of errors. Most information entered by the user is remembered from session to session in a *user profile* and is redisplayed the next time the same function is requested. The user need enter only the information to be changed.

Trends in the late 1970s

among users

When old SPF was developed, it was anticipated that users would want to extend the program in the foreground/background areas and in the tutorial. Provisions were made for an installation to develop new display interfaces to language processors or to any program capable of being executed via commands or JCL. Such extensions required only the addition of display panels and skeleton procedures to the appropriate libraries; no change was required to the distributed code.

Many installations have, indeed, extended the foreground/background options and the tutorial. By the late 1970s, however, installations were attempting to extend SPF far beyond what could be done with display interfaces to commands and JCL. They were beginning to develop totally new functions, even whole applications, requiring user-written programs to be invoked from SPF-style menus. They had discovered that the product included a wealth of internal service routines for generating displays, allocating files and libraries, performing I/O operations, and verifying user inputs. Users were attempting to invoke these facilities to provide new interactive capabilities for their installations.

Unfortunately, use of the internal service routines required access to a very complex set of internal control blocks. Any user-written program that invokes the internal services is tightly bound to the internal control block structure, and that structure has changed with each new release of the product.

This situation gave rise to demands for IBM either to freeze the internal program structure or to provide macros and data declaration statements, in several languages, to isolate installations from

these changes. There were also demands for formal education and better documentation of the internal control block structure, and for step-by-step instructions on the use of the service routines.

Two of the major problems in meeting these requirements were first that it was not possible to freeze the control block structure while allowing flexibility for future enhancements, and second that it would not be productive to educate users on the complexities of the internal structure.

A new approach was needed—one that would simplify the development of new interactive applications without requiring knowledge of (or access to) the internal structure of the product.

within IBM

During the late 1970s, about the same time users were attempting to extend old SPF to new applications, groups in IBM development laboratories were creating more and more display-oriented interactive programs, some for release as products and some for internal use. These programs provided many of the same capabilities, namely the ability to:

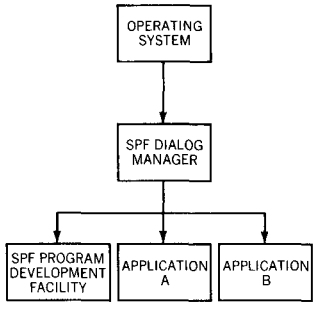
- Display a hierarchy of menus based on user selections.
- Invoke functions (commands and programs) from menus.
- Communicate with the user via data entry displays and messages.
- Provide on-line help and tutorial information.
- Retain user-entered or program-generated data from one session to another.
- Generate sequential output to be passed as input to another process (e.g., JCL to be submitted to the background or SCRIPT text to be formatted for printing).

At that time, a new term, *dialog manager*, came into the language to describe these capabilities. Nearly all interactive programs included a dialog manager of one sort or another. Far too many different, incompatible dialog managers had been developed, or were being developed, resulting in a significant replication of effort and expense. As attention became focused on this situation, the question of having one common dialog manager became an issue. It was recognized that a common dialog manager would be difficult to develop because it would have to be flexible enough to accommodate a variety of different applications. It was also recognized that there would be conversion problems and that many existing programs might never convert due to the costs involved. Despite these problems, a common dialog manager was clearly needed. Of special significance was the need for a cross-system dialog manager, capable of running interactive applications in multiple environments such as MVS/TSO and VM/CMS.

Concepts of the System Productivity Facility

The System Productivity Facility—the new SPF—is designed to address the demands of both system users and IBM program developers. The new SPF was built using old SPF as the base. The dialog control facilities [e.g., menu/tutorial processing, screen management, and program function (PF) key recognition] and the internal service routines are segregated into a new component, the dialog manager. High-level external interfaces to existing services and a variety of new services have been developed.

Figure 2 Relationships of major System Productivity Facility components



The user-visible part of old SPF (browse, edit, utilities, foreground, background, and the contents of the tutorial) has become the other major component of new System Productivity Facility. This component is called the *program development facility*. New dialog testing facilities have been added to this component.

The relationship between the two major components of SPF is shown in Figure 2. The *dialog manager* provides control facilities and services to support execution of interactive applications. Conceptually, it is an environment-independent extension to the operating system. The *program development facility*, which supports development and testing of applications, itself runs as an application under the dialog manager. Installations may continue to extend the program development facility but, more importantly, they may now develop totally new applications that use the services of the dialog manager.

design objectives

Some of our design objectives for SPF were relatively straightforward. Elimination of dependence on internal structure was achieved by designing new external interfaces that do not use the internal control blocks for communication with applications. A cross-system capability was achieved by ensuring that the new external interfaces were the same in all environments, even though the dialog manager itself contains considerable environment-dependent code.

The difficult objectives were the ones that related to flexibility and usability. The internal dialog management capabilities of old SPF were originally designed to support one type of application (now known as the program development facility), coded in one language (PL/S) by one group of highly skilled systems programmers. The transformation of these capabilities into a new general-purpose dialog manager imposed a new set of objectives that required rethinking and redesigning many of the earlier SPF facilities.

The new System Productivity Facility would have to support a wide variety of applications, including applications for which the end user is not skilled in data processing methods. It would

further have to allow applications to be coded in a variety of languages, including the command language of the host system (CLIST for MVS/TSO, or EXEC2 for VM/CMS) and high-level programming languages (such as PL/I or COBOL). The new SPF would also have to accommodate a mixture of languages, with some functions coded in a command language and other functions coded in one or more programming languages.

Another objective was to simplify the specification of display formats and to relieve the application logic from the concerns of cursor placement, initialization of default values, verification of user inputs, and display of error messages and supplementary help information. The new dialog manager also had to provide a convenient mechanism for retaining user-entered and/or program-generated information across sessions, and to allow related sets of information to be managed easily and coherently. It had to facilitate the generation of sequential output in a way that relieves the application logic from the details of output syntax and to simplify error-handling, particularly in cases where the error is probably not recoverable within the framework of the application logic.

The remainder of this paper deals with the way in which these objectives were addressed and satisfied.

Dialog organization and flow

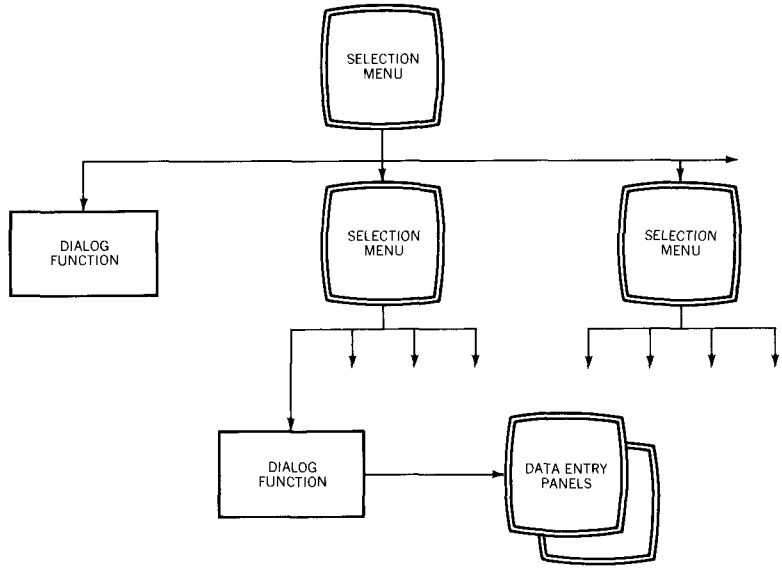
An interactive application comprises the following elements:

- Selection menus, from which the user may select processing options.
- Dialog functions (commands and programs) that perform the requested processing.
- Data entry panels that prompt the user for additional information.

The way in which these elements are organized determines the flow of the dialog, as seen by the end user. SPF must support a wide variety of organizations, so that the dialog flow can be tailored to suit the particular needs of the application, and information can be presented in a natural, user-friendly manner.

A traditional dialog organization is shown in Figure 3. This example starts with the display of a high-level selection menu, which is the primary option menu for the application. User options selected from this menu may result in the invocation of a dialog function or in the display of a lower-level selection menu. Each lower-level menu may also cause functions to receive control or cause still lower-level menus to be displayed. The menu hierarchy may extend as many levels deep as desired.

Figure 3 Traditional dialog organization and flow



Eventually a dialog function receives control. The dialog function may use any of the dialog services provided by SPF. In particular, the function may continue the interaction with the end user by displaying data entry panels to prompt the user for information. When the function completes, the selection menu from which it was invoked is redisplayed.

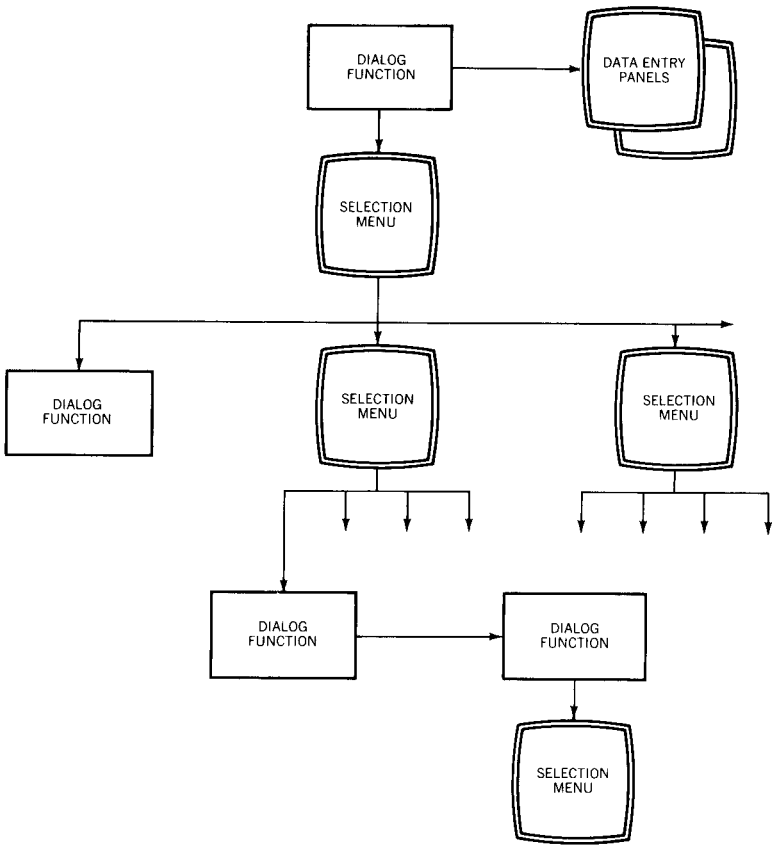
An alternative dialog organization is shown in Figure 4. Here a dialog function receives control before the display of a menu. The dialog function performs application-dependent initialization and displays data entry panels to prompt the user for initial information. It then starts the selection process by invoking the primary option menu for the application.

This example also shows that one function can invoke another without displaying a menu, and that a function may start a lower-level selection process at any point in the hierarchy.

Support for these and other types of dialog organizations requires that either of two "next actions" be allowed at any point in a dialog: Display a selection menu, or invoke a dialog function.

Most dialog managers, including SPF, include a control mechanism to display selection menus, interpret user responses, and take appropriate action (display a lower-level menu or invoke a

Figure 4 An alternative dialog organization

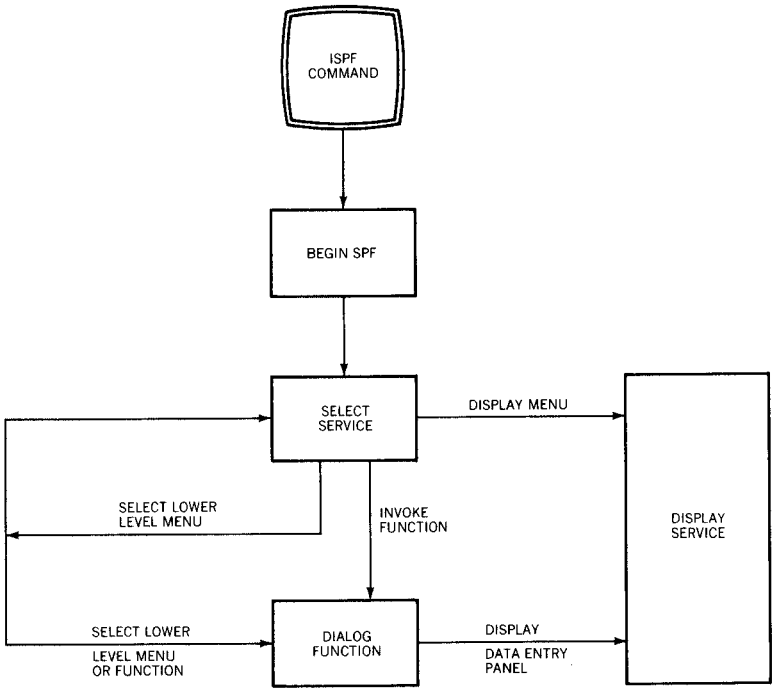


function). To achieve the desired flexibility, we decided to make this control mechanism available as a service that can be invoked from dialog functions as well as from within the dialog manager. It is known as the SELECT service. Selection keywords are passed as input parameters to the SELECT service. They specify the next action as follows:

PANEL (menu-name)
CMD (command)
PGM (program-name) [PARM (parameters)]

The PANEL keyword specifies the name of the next selection menu to be displayed. The CMD and PGM keywords specify a dialog function (coded as a command or program, respectively) to receive control. Input parameters may be passed to the dialog function as part of the command specification or via the optional PARM keyword.

Figure 5 Flow of control for the SELECT service



The flow of control for this process is shown in Figure 5. The process starts via the ISPF command, which is issued outside SPF to invoke the dialog manager. The ISPF command requires the same keyword parameters as the SELECT service. Here, they are used to specify the first selection menu to be displayed or dialog function to receive control.

After the SPF environment has been initialized, the dialog manager invokes the SELECT service, passing as input the initial keywords specified on the ISPF command. If the CMD or PGM keyword is passed to the SELECT service, SELECT simply invokes the dialog function. If the PANEL keyword is passed, SELECT calls the DISPLAY service to display the specified selection menu.

Selection menus contain sufficient information to determine the next action to be taken for any option entered by the user. Specifically, the panel definition specifies a selection keyword (PANEL, CMD, or PGM) corresponding to each user-entered option. The keyword is returned from the DISPLAY service to the SELECT service. If a PANEL keyword is returned, SELECT recursively invokes itself, passing the keyword as input. As a result, SELECT continues to display successively lower levels of menus in the

hierarchy until a dialog function is specified as the next action. Whenever a CMD or PGM keyword is returned as the next action, SELECT invokes the dialog function.

When a dialog function receives control, it may call the DISPLAY service to display data entry panels or other user information. It may also call the SELECT service to start the display of a lower-level menu hierarchy or invoke a lower-level dialog function (without displaying a menu).

When a dialog function completes execution, control is returned to the SELECT service. If the dialog function had been invoked from a menu, that menu is then redisplayed. Alternatively, if the dialog function had been invoked from a higher-level function, the higher-level function resumes execution.

Languages and data communication

program linkages

The objective of supporting dialog functions coded in a variety of languages involves the issues of program linkages and data communication. Conceptually, resolution of the first issue—program linkages—is straightforward. The dialog manager must be able to invoke command-coded functions (CLIST and EXEC2 languages) and program-coded modules. The mechanisms to do this are provided by the host operating systems. Dialog functions, in turn, must be able to invoke SPF services. Two linkage mechanisms have been developed for this purpose:

- ISPEXEC command is used to invoke services from command-coded functions.
- ISPLINK subroutine is used to invoke services from program-coded modules. Standard register conventions, which are supported by most high-level languages, are used.

Function-to-function linkage is provided by the SELECT service, as previously discussed. SELECT provides a convenient mechanism for invoking a command-coded function from a program-coded function or vice versa.

data communication

Data communication has not been as straightforward. Each language has its own conventions for the internal storage of data. A simple mechanism was needed to communicate data between a dialog function and a service, and between two or more dialog functions. Of course, data can be communicated explicitly via calling sequence parameters, but this is cumbersome when many parameters are required. Consider, for example, the need for a dialog function to display a panel containing twenty input fields. A calling sequence (to the DISPLAY service) with twenty or more parameters would be awkward to code and debug.

The CLIST and EXEC2 languages provided the clue to solving this problem. In these languages, variables are defined implicitly; no data declaration statements are required. The variables are treated as character strings that may vary in length, eliminating the need for format transformations as well as concerns for storage allocation and boundary alignment.

This approach was adopted for SPF *dialog variables*, which serve as the primary communication mechanism between functions and services and between two or more functions. A dialog variable is a character string that may vary in length from zero to 32K bytes. It is referenced symbolically, by name.

For functions coded in a command language, the CLIST or EXEC2 variables are automatically treated as dialog variables; no special action is required to define them to SPF. For functions coded in a programming language, internal program variables may be identified to SPF as dialog variables, so that they can be accessed and updated directly by SPF services. An automatic format transformation from character string to fixed binary or bit string may also be specified. Alternatively, a program may copy and replace dialog variables from or to a dynamically generated pool.

Dialog variables are normally associated with the function that is currently in control. Different functions may have variables of the same name with no conflict. Functions may share variables by copying them to or from a shared variable pool or the user profile. Variables in the user profile are automatically retained across sessions.

Dialog variable names appear in panel, message, and skeleton definitions to allow communication with the functions. A variable name in a panel definition, for example, corresponds exactly to the name of a dialog variable that is accessible to a function. The variable may be used to initialize information on the panel (prior to display), and to store input entered by the user.

example

The following example, coded in CLIST language, illustrates the ease with which data is communicated between a dialog function and the DISPLAY service.

```
SET &AAA = 1
ISPEXEC DISPLAY PANEL (XYZ)
SET &CCC = &AAA + &BBB
```

Variable AAA is created by the CLIST simply by setting it to a value. The DISPLAY service is then called (using the ISPEXEC command) to display panel XYZ. The panel name is passed as an explicit calling sequence parameter. Other data, however, is passed by matching the names of dialog variables (in this case CLIST variables) with variable names that appear in the panel definition.

Assume that the panel definition for XYZ contains two symbolic parameters, named AAA and BBB, and that they are defined as input (unprotected) fields. In the panel definition, they might appear as follows:

```
INITIAL VALUE ⇛ _AAA
INCREMENT     ⇛ _BBB
```

where each underscore indicates the start of an input field, followed by the name of the variable.

When the panel is displayed, the first input field is automatically initialized to 1, since it was set by the dialog function before calling the DISPLAY service. The second field is displayed as blank, since it was not set.

Now assume that the user changes the value of the first field to 100 and types 20 in the second field. When the user presses the ENTER key, the values are automatically stored, updating the value of AAA and creating a variable BBB (which was not previously defined by the CLIST). The DISPLAY service then returns control, and the next statement in the CLIST is executed. This statement creates a variable named CCC and sets it to the sum of AAA and BBB, namely 120.

Display formatting

The old SPF display facility was the starting point for designing the new SPF DISPLAY service in which several of the earlier concepts have been retained. Display formats in both facilities are specified via *panel definitions* that include a "picture" of what the end user will see. The picture concept facilitates panel creation and modification. A *message* is treated as supplementary information that may be displayed with a panel or superimposed on the panel that is currently being displayed.

In both facilities, panel and message definitions are maintained in libraries, external to the application code. This allows an installation to easily custom tailor display formats and facilitates translation to other languages. The definitions are created by editing directly into the panel and message libraries; no compile or preprocessing step is required. Panel and message definitions contain a mixture of literal text to be displayed as is, variables for which the current value is substituted dynamically at the time of display, and control information to be interpreted by the DISPLAY service.

display panel processing logic

Certain redesign of these basic concepts has been required in the new SPF. The panel and message definition syntax has been changed to support the substitution of dialog variables that are

specified by name. (In old SPF, variables were specified by their position in a calling sequence.) The major thrust of the new design has been to simplify the panel definition syntax and to allow more of the format-related processing to be moved outboard from the dialog functions (including verification of user input). The goal has been to free dialog functions from the details of end user communication, that is, to decouple panel processing logic from application function logic.

Although old SPF allowed some of the display processing logic to be specified in the panel definitions (outboard from the functions), the specification was static in nature—analogous to data declarations in a program. Initial attempts to extend that approach resulted in syntax that was exceedingly clumsy, without producing the desired degree of flexibility. We then decided to try a procedural approach, in which the panel processing logic would be specified via assignment, IF, and VER (verify) statements. Separate procedural sections were required to specify processing to occur before and after the display of the panel.

The new panel definitions are organized into four sections, of which only the body is required:

- *Attribute section* (optional) defines special characters that are to be used in the body of the panel definition to represent attribute (start of field) bytes. Default attribute characters are provided, which may be overridden.
- *Body* (required) defines the format of the panel as seen by the user (the picture section of the panel) and defines the name of each variable field on the panel.
- *Initialization section* (optional) specifies the initial processing that is to occur prior to displaying the panel. Typically, this section is used to define how variables are to be initialized.
- *Processing section* (optional) specifies processing that is to occur after the panel has been displayed and is typically used to define how variables are to be verified and/or translated.

A sample panel definition is shown in Figure 6. It has no attribute section but simply uses the following default attribute characters:

% (percent sign)	—	text (protected) field, high intensity
+ (plus sign)	—	text (protected) field, low intensity
_ (underscore)	—	input (unprotected) field, high intensity

Each text attribute character is followed by the information to be displayed. Substitutable variables, consisting of a dialog variable name preceded by an ampersand (&), may be included in the text. Each input attribute character is followed immediately by a dialog variable name, with no intervening ampersand.

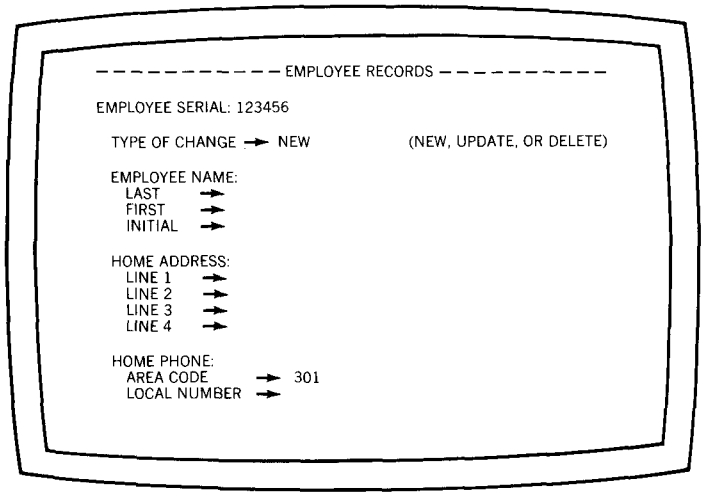
Figure 6 Sample display panel definition

```
%- - - - - - - - - - - - EMPLOYEE RECORDS - - - - - - - - - - -

% EMPLOYEE SERIAL: &EMPSER

+  TYPE OF CHANGE %==> _TYPECHG +     (NEW, UPDATE, OR DELETE)

+  EMPLOYEE NAME
+     LAST    %==> _LNAME              +
+     FIRST   %==> _FNAME              +
+     INITIAL %==> _I+

+  HOME ADDRESS:
+     LINE 1  %==> _ADDR1                                    +
+     LINE 2  %==> _ADDR2                                    +
+     LINE 3  %==> _ADDR3                                    +
+     LINE 4  %==> _ADDR4                                    +

+  HOME PHONE:
+     AREA CODE      %==> _PHA+
+     LOCAL NUMBER   %==> _PHNUM     +

)INIT
  .HELP = PERS032
  .CURSOR = TYPECHG
  IF (&TYPECHG = NEW)
      LNAME    = '   '
      FNAME    = '   '
      I        = '   '
      ADDR1    = '   '
      ADDR2    = '   '
      ADDR3    = '   '
      ADDR4    = '   '
      PHA      = 301
      PHNUM    = '   '

)PROC
  VER (&TYPECHG, LIST, NEW, UPDATE, DELETE, MSG =EMPX210)
  VER (&LNAME, ALPHA)
  VER (&FNAME, ALPHA)
  VER (&I, ALPHA)
  VER (&PHA, NUM)
  VER (&PHNUM, PICT, 'NNN-NNNN')

)END
```

The panel body is terminated with an ")INIT" header statement, which starts the initialization section. The initialization section in this example establishes PERS032 as the name of the related help panel (in the event that the user requests help while viewing the panel) and sets the initial cursor position to the TYPECHG field. It then tests the current value of the TYPECHG variable. If the current value is NEW, all remaining input variables are initialized to blank except for variable PHA, which is initialized to 301.

Figure 7 shows how the panel looks when displayed, assuming that the dialog function has first set variables EMPSER to 123456 and

Figure 7 Sample panel when displayed



TYPECHG to NEW before calling the DISPLAY service. After the user enters information, the data in each panel input field is automatically stored into the corresponding dialog variable. Then the processing section of the panel definition, beginning with a ")PROC" header statement, is executed. In this example, the processing section contains VER statements to verify that information entered by the user meets the following criteria:

- Type of change is NEW, UPDATE, or DELETE.
- Last name, first name, and initial contain all alphabetic characters.
- Area code contains all numeric characters.
- Phone number consists of three numeric characters, followed by a hyphen, followed by four numeric characters.

If any verification check fails, a message is automatically displayed, and the user is given an opportunity to correct the error. The first VER statement explicitly designates the message (EMPX201) to be displayed. For the other VER statements, an appropriate default message is displayed. When the user corrects the error, the variables are again stored, and the processing section is re-executed.

Table services

The objective of supporting a wide variety of applications includes one requirement that had not been addressed by old SPF,

Figure 8 Example table

EMPSER	LNAME	FNAME	I	PHA	PHNUM
598304	ROBERTSON	RICHARD	P	301	840-1224
172397	SMITH	SUSAN	A	301	547-8465
813058	RUSSELL	CHARLES	L	202	338-9557
395733	ADAMS	JOHN	Q	202	477-1776
502774	CARUSO	VINCENT	J	914	294-1168

EMPSER Employee serial number
LNAME Last name
FNAME First name
I Middle initial
PHA Home telephone area code
PHNUM Home telephone local number

namely, to provide a convenient mechanism for generating a data base of user-entered and/or program-generated information, and to support inquiry and update of that data.

To satisfy this requirement, we decided to incorporate a set of services from another dialog manager, the Interactive Productivity Facility (available under VM/370 and VSE). In the process, the services have been enhanced and reimplemented to fit the SPF environment. They are called *table services*. A table is a two-dimensional array of information in which each column corresponds to a dialog variable, and each row contains a set of values for those variables. An example is shown in Figure 8.

The variable names that define the columns of a table are specified when the table is created. At the same time, one or more columns (variable names) may be specified as keys for accessing the table. For the table shown in the figure, EMPSER might be defined as the key variable. Or EMPSER and LNAME might both be defined as keys, in which case a row would be found only if EMPSER and LNAME both match the current values of those variables.

A table may also be accessed by one or more argument variables, which need not be key variables. A complex argument may be constructed to provide powerful search capabilities.

In addition, a table may be accessed by Current Row Pointer (CRP). When a table is opened, the CRP is automatically positioned to TOP, i.e., ahead of the first row. The table may be scanned by moving the CRP forward or back. A row is retrieved each time the CRP is moved.

When a row is retrieved from a table, the contents of the row are stored into the corresponding dialog variables. When a row is stored (updated or added), the current contents of the dialog variables are saved in that row.

A table may be defined as temporary or permanent. A *temporary table* is created in virtual storage and deleted upon completion of processing. A *permanent table* resides on direct access storage (DASD), and it may be opened for update or for read-only access, at which time the entire table is read into virtual storage. All changes to the contents of a table are made in virtual storage. When processing is complete, the entire table may be written back to direct access storage, provided it was originally obtained for update.

Services are provided to create new tables, open existing tables, save tables on direct access storage, and delete tables without saving. Other services retrieve and update rows in the virtual storage copy of a table. In addition, a service is provided to display rows of a table in scrollable format and allow the user to select rows for processing and/or modify selected information in the rows.

File tailoring services

Many interactive applications need to generate sequential output that can be used to drive some other process. Old SPF had such a facility, but it was too closely tied to the generation of Job Control Language statements (JCL) for batch compilation. Although JCL generation is still a requirement, a more flexible mechanism is needed to allow the generation of other types of sequential output, such as control statements to drive a utility or SCRIPT/VS statements to produce a printable report. The objective is not only to provide flexibility, but also to allow the formatting details for the generated output to be specified outboard from the dialog functions.

Again, the solution was found in the Interactive Productivity Facility. Another set of services, called *file tailoring*, was taken from that product and reimplemented for the SPF environment. File tailoring services read skeleton files from a library and write tailored output that may be used to drive other functions. Each record in the skeleton file may contain text that is to be copied verbatim to the output file, intermixed with variable names (preceded by an ampersand). When a variable name is found, the current value of the corresponding dialog variable is substituted in the output file.

Skeleton file records may also contain control statements that provide the following capabilities:

- Set dialog variables.
- Imbed other skeleton files.
- Conditionally include records.

- Iteratively process records in which variables from each row of a table are substituted.

For the iterative processing of records, file tailoring services retrieve all rows from the table and generate one or more output records (as specified in the skeleton) for each row. This provides a convenient way to generate a report or other type of output containing tabular information.

Error handling and recovery

Error handling and recovery is often a major problem in application development. A significant amount of application code is usually required, and this code is of a type that is particularly difficult to debug. One of the major SPF objectives has been to reduce the amount of error handling logic required in an application.

Most SPF services are capable of detecting a variety of errors for which the likely cause is a bug in the dialog function. (For example, a function does not open a table before attempting to process it.) In these cases, the application developer simply wants to find the bug, correct it, and try again without having to code error logic in the dialog function. To permit this, SPF must provide sufficient information to the application developer (at the terminal) rather than returning to the dialog function with an error code.

First, we grouped the return codes from the SPF services into three general categories:

- *Normal completion* code (0).
- *Exception condition* codes (4 and 8) indicate conditions that are not necessarily errors (e.g., no entry found in a table).
- *Error condition* codes (12, 16, and 20) indicate that a service did not complete or only partially completed due to errors.

Then the following two modes of operation were defined to control the action taken in the case of SPF-detected errors (code 12 or higher):

- CANCEL mode. Display and log a message; then terminate the dialog and redisplay the primary option menu.
- RETURN mode. Format an error message (but do not display or log it); then return to the function that invoked the service, passing back the designated return code.

The normal (default) mode is CANCEL. In CANCEL mode, control is not returned to the function that invoked the service. Hence,

the function never sees a return code of 12 or higher, and need not include logic to process this kind of error.

For dialog functions that are intended to process errors, the mode may be set RETURN via a service. Those functions must then have logic to handle return codes of 12 or higher.

Concluding remarks

The basic design objectives for SPF can be summed up in one general objective, namely, to improve application development productivity by simplifying the development process. This is the reason for the new name for SPF, the System Productivity Facility. Realistically, a single program cannot address the requirements for all applications, but SPF has taken a major step in providing a single cross-system capability that supports the development, testing, and execution of interactive applications.

Initial reactions from users have been positive. They have confirmed that the new dialog manager provides many of the services they require and has simplified the development process. At the same time, they have suggested new features and enhancements that are currently under evaluation as future objectives.

GENERAL REFERENCES

SPF Dialog Management Services, SC34-2036 (1981); available through IBM branch offices; provides information about the SPF dialog manager in all operating environments.

SPF-MVS Program Reference, SC34-2038 (1981); available through IBM branch offices; provides information on the use of the SPF program development facility in the MVS/TSO environment.

SPF-VM Program Reference, SC34-2047 (1981); available through IBM branch offices; provides information on the use of the SPF program development facility in the VM/CMS environment.

The author is with the IBM Information Systems Group, 18100 Frederick Pike, Gaithersburg, MD 20760.

Reprint Order No. G321-5155.