Synergy Programmer's Manual

Order No. AA-EU61A-TH
Order No. AD-EU61A-T1

December 1985

This manual describes the tools and procedures that you use to build an application that can be installed and run in the Synergy environment.

REQUIRED SOFTWARE: Professional Host Tool Kit V3.0, PRO/Tool Kit, 3.0 or later, Synergy V2.0 or later

OPERATING SYSTEM: P/OS V3.0 or later

digital™

DIGITAL EQUIPMENT CORPORATION
Maynard, Massachusetts 01754-2571

First Printing, February, 1985
Updated, December, 1985

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

The specifications and drawings, herein, are the property of Digital Equipment Corporation and shall not be reproduced or copied or used in whole or in part as the basis for the manufacture or sale of items without written permission.

The following are trademarks of Digital Equipment Corporation:

CTI BUS	MASSBUS	Rainbow
DEC	PDP	RSTS
DECmate	P/OS	RSX
DECsystem-10	PRO/BASIC	Tool Kit
DECSYSTEM-20	PRO/Communications	UNIBUS
DECUS	Professional	VAX
DECwriter	PRO/FMS	VMS
DIBOL	PRO/RMS	VT
digital™	PROSE	Work Processor
	PROSE PLUS	

CONTENTS

FIGURES

TABLES

PREFACE

MANUAL OBJECTIVES

This manual tells you how to build an application that can be installed and executed in the Synergy environment. Synergy software tools are also described.

INTENDED AUDIENCE

You should have some experience developing applications for the Professional under P/OS. In particular, you should be familiar with the Tool Kit, P/OS, PRO/GIDIS, and Synergy software for the Professional.

SYSTEM REQUIREMENTS

You should have the following software:

- Professional Host Tool Kit V2.0, or later,
  or PRO/Tool Kit V2.0, or later
- P/OS V2.0, or later
- Synergy V1.0, or later

STRUCTURE OF THIS DOCUMENT

The manual has eleven chapters, two appendices, and a glossary:

- **Synergy Overview** introduces the Synergy environment, as seen by an application developer.
- **Designing a New Application** describes all aspects of application design specific to the Synergy environment.
- **Adapting a P/OS Application** describes the modifications needed to move an application into the Synergy environment.
- **The Synergy Interface** provides a general description of the call interface between the application and Synergy services.

- **Task Control Services** describes each of the Synergy services that are used to control the execution of the application.
- **Window Services** describes each of the Synergy services that are used to create and manipulate the application windows.
- **Menu Services** describes each of the Synergy services that are used to display menus and solicit input from the end user.
- **The Frame Compiler** describes the software tool that is used to prepare menu, HELP and message frames.
- **Debugging the Application's Windows** describes additional software tools that are used to check the output of the Frame Compiler and to take screen dumps on a printer.
- **The Clipboard** describes the clipboard files and the rules for their use.
- **Synergy Conventions** describes the conventions that are used in Synergy applications.
- **Baton Twirler** provides listings of the files needed to build a sample window application.
- **Table of Synergy Services** is an alphabetized list of all Synergy Services.
- **Glossary** defines special terms used in the Synergy context.

ASSOCIATED DOCUMENTS

- *Tool Kit User's Guide*
- *Tool Kit Reference Manual*
- *P/OS System Reference Manual*
- *PRO/GIDIS Manual*
- *Synergy User's Guide*

CONVENTIONS USED IN THIS MANUAL

Convention or Term	Meaning
[optional]	In an FCT command line format, square brackets indicate that the enclosed item is optional. In a file specification, square brackets are part of the required syntax.
UPPERCASE	Uppercase words and letters, used in examples, indicate that you should type the word or letter exactly as shown.
<MixedCase>	Mixedcase words in angle brackets, used in FCT command line formats, indicate that you should substitute a word or value of your own. Usually the mixedcase word identifies the type of substitution required.
...	A horizontal ellipsis indicates that you can repeat the preceding item one or more times. For example: parameter [,parameter...]
Tool Kit	This general term refers to the software you use to develop applications to run on a Professional computer.
Host Tool Kit	The Host Tool Kit is Tool Kit software that runs on a host computer, rather than on the Professional itself.
PRO/Tool Kit	The PRO/Tool Kit is the Tool Kit software that runs on the Professional computer.
User	The word "user" always refers to the person utilizing the Synergy application that you are building. You, as the application builder, are never referred to as the user.

Synergy services are described in a standard format:

```
ABCD - Sample Service Call

Status                  2 words (output)
StringLength            word    (input)
InputString()           n bytes (input)
```

The uppercase symbol **ABCD** is the global symbol defined by

Synergy. The call shown above expects three parameters called Status, StringLength, and InputString. Parameter names are chosen only for their mnemonic content. An additional explanation is provided if the intention of the parameter is not clear from its name. An array parameter is shown with a () after the name.

The parameter's data type is shown to the right of each parameter name. The use of the parameter as input or output is indicated.

The parameters must be supplied in the listed order.

CHAPTER 1

CHAPTER 1

SYNERGY OVERVIEW

1.1 INTRODUCTION TO SYNERGY

Synergy is an application that runs under the P/OS hard disk operating system. Synergy provides an environment for the execution of applications. An application that is built to run in the Synergy environment can use Synergy services to:

- Create and manipulate windows on the screen
- Solicit user input through fields in its windows
- Solicit user input through menus in special windows
- Provide message and HELP information in special windows
- Suspend its execution, enabling other Synergy applications and P/OS applications to run
- Exchange data with other Synergy applications through a common data exchange file called a "clipboard"

An application that is built to run in the Synergy environment is designed and implemented with the Tool Kit, using any of the Tool Kit languages and run-time support facilities. (Existing applications that run under the P/OS hard disk operating system can run in the Synergy environment, with some changes.)

Applications that take full advantage of the Synergy screen management techniques will use most of the facilities described in this manual.

1.2 APPLICATION CONTROL

Synergy consists of a collection of tasks that provide the environment for the Synergy applications. These tasks can be thought of as providing two major functions:

- The *window manager* presents the Synergy Main Menu and provides the user interface for controlling windows and tasks. In the *Synergy User's Guide* all the functions that are not performed by applications are discussed as though they were being performed by the window manager, in order to simplify the terminology and present Synergy as a single entity.

- The *window server* does the actual work involved in moving windows and provides menu and HELP services. Almost all service calls from the application go to the window server. (Even the window manager calls the window server to do its screen manipulation.)

In this manual, we will speak of all the Synergy services being provided by the window server.

1.2.1 The Active Application

Several Synergy applications can be running at the same time, but only one application can alter information on the screen. This is the *active application*.

The front window is the only window for which a GIDIS viewport is defined, and thus it is the only window whose display can change. The active application owns the front window and is the only application that should write in this window.

The active application also receives all keyboard input.

An application may consist of one task or several tasks. The application can run with the terminal attached and can use an AST routine to read the keyboard. When an application gives up control to the window server by calling the Suspend service, it must detach the terminal and must ensure that none of its tasks that continue to run do any I/O through the terminal.

The window server can service only one application and one request at a time. You must ensure that requests appear serially, which means that any two tasks of your application must not request a window service at the same time, and all tasks must ensure that requests are sent in the proper order.

1.2.2 Installing and Removing Applications

Synergy applications are installed on both P/OS and Synergy application menus, using the "Install application" option of the P/OS Disk/Diskette Services Menu. After the user completes the Application/Group Name Change Form, a window appears near the bottom of the screen, requesting that the user take an additional action to select a Synergy group in which the application name is to appear.

This request for a Synergy application group is necessary to insert the application's name on the Synergy Main Menu. You must place a special command in your application's install file to trigger this action during installation.

The user removes a Synergy application with the "Remove application" option of the P/OS Disk/Diskette Services Menu. A special command that you place in your application's install file causes the application's name to be removed from the Synergy Main Menu. No additional action is required by the user during removal of a Synergy application.

1.2.3 Starting and Exiting the Application

The user starts the Synergy application from either the P/OS Application Menu or the Synergy Main Menu. When the application is suspended, control returns to the Synergy Main Menu.

If the user suspends the application and then suspends the Synergy Window Manager (in order to do some work at the P/OS level), he can resume the application by simply starting it again. Again, he has the choice of starting from either the P/OS Application Menu or the Synergy Main Menu.

If the application exits without ever having been suspended, it returns to the point from which it was started. Thus, an application that is started from a P/OS Application Group menu will return to that menu on exit, provided it has not been suspended. However, once an application has been suspended and resumed, it returns to the Synergy Main Menu on exit.

1.2.4 Task Control Services Overview

There are three major services that control the execution of a Synergy application:

- *Initialize* - This service is called when the application starts. It establishes a handshake with the Synergy system and retrieves any application-specific data that was saved the last time the application was run.

- *Suspend* - This service is called when the application is suspending its execution, usually in response to a press of the F5 key. Control is returned to the application when the user tells the window manager to resume execution of the application.

- *Done* - This service is called when the application is about to exit. It passes application-specific data back to Synergy so that it can be saved on behalf of the application.

1.3 WINDOWS IN SYNERGY

1.3.1 Window Description

A window is a rectangular area of the screen which serves to focus the user's attention. It usually has a dark border, called the windowframe, and a light background. It contains dark letters or graphic images.

NOTE

A special window type is available, called a *VT window*, to ease migration of an application from the P/OS hard disk environment to the Synergy environment. A VT window always occupies the full screen, with no windowframe. It usually displays light letters and graphic images against a dark background. VT windows are discussed only in Chapter 3. Applications that take full advantage of Synergy window facilities do not use VT windows.

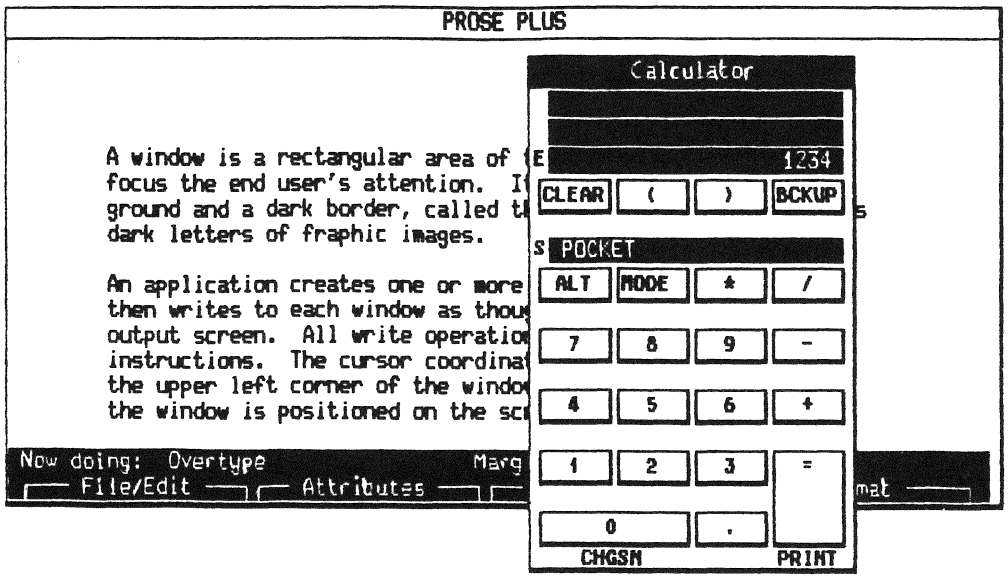


Figure 1-1: Windows on the Screen

An application creates one or more windows on the screen and then writes to each window as though it were a separate output screen. All write operations are done with GIDIS instructions. The cursor coordinates are given relative to the upper left corner of the window, regardless of where the window is positioned on the screen. GIDIS automatically translates the window-relative coordinates to screen-relative coordinates, so the application can be unaware of where the window actually is on the screen.

Windows are often smaller than a full screen. Window positions may intersect, so that windows may obscure part or all of other windows. Each window exists at some level, exactly analogous to pieces of paper lying on a desk: The top paper covers the parts of all papers it overlaps; the bottom paper is covered by the parts of all other papers that overlap it.

We use the terms "top" and "bottom" to describe the stacked pieces of paper on the desk. We use the terms "front" and "rear" to describe the stacked windows on the screen.

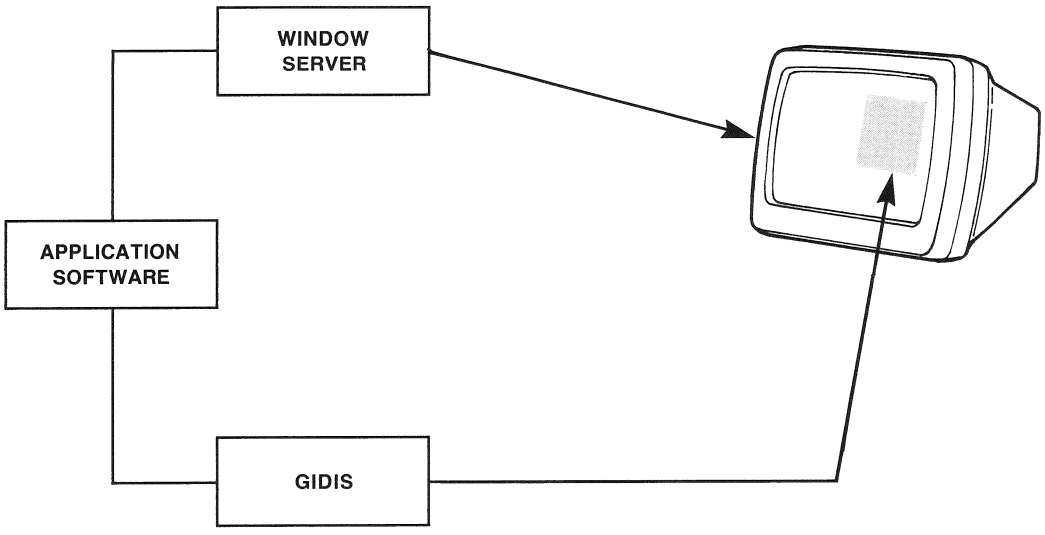
Each window is independent of all other windows, so that the application need not be concerned with whether the windows overlap. There is no need to "tile" the windows on the screen. The user may want to refer to two or more windows simultaneously and thus may want to change the position of the windows in order

to tile them. However, most applications can ignore this window positioning activity.

Even when an application has more than one window, it can write only to its front window. The application calls a Select Window service to select any of its windows as the front window before writing to it. The Select Window service moves the window in front of all other windows.

In order to guarantee that an application writes only to its own windows, the application is required to create the windows through calls on the Synergy interface and then restrict its writing to GIDIS instructions. Synergy adjusts the GIDIS state so that the application is always addressing the front window. The application avoids doing text-mode QIOs to the screen in order to guarantee that the cursor position stays within the front window and to guarantee that the entire screen will not scroll.

Creation and display of special windows for menus and HELP are handled entirely through the Synergy window server. These windows, which usually have a short life on the screen, require very little development effort and very little space within the application's address space.



1.3.2 Window Attributes

All windows have a position and a size. Windows may be positioned anywhere on the screen, as long as they fit entirely within the screen. They may be as large as the screen, or as small as one character. The initial size, position, and attributes of a window are defined when the window is created with the Create Window service.

Windows consist of two parts: a windowframe and a writable area. The windowframe is a black border surrounding the writable area of the window. The position of the windowframe is defined by the X and Y coordinates of its upper left hand corner. The writable area of the window is specified by a width and height.

There are nine window attributes: stackable, titled, hidden, color, white border, clear on change, VT, invisible, and three-plane.

Giving a window the *stackable* attribute means that the application promises to abide by some restrictions and that the window server can take advantage of those restrictions and gain some efficiencies in managing the window. Stackable windows are treated as a stack, that is, on a first-in, last-out basis. A stackable window can be created in front of a nonstackable window, but once a stackable window exists on the screen, only stackable windows can be created in front of it. Furthermore, the stackable windows are destroyed in reverse order of their creation. The creation and destruction of these windows is not interrupted by any other window operations, such as changes in window size or reordering of the stack of windows. The stackable attribute is used largely by the window server when it is creating menu and HELP windows in response to calls from the application. It can be used by applications, provided the applications abide by the same rules. A maximum of four stackable windows can exist on the screen at one time. Certain menu services can create up to three stackable windows, so you should exercise care in calling menu services when you have created more than one stackable window.

In a *titled window* the top of the windowframe is thicker and contains title text. When the titled window is the front window, its title is highlighted (light letters on a dark background).

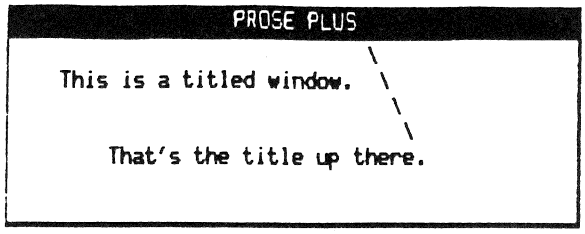


Figure 1-3: A Titled Window

A *hidden window* is not visible at all on the screen, unless it is the front window. (Windows that are **not** hidden are always visible unless they are totally obscured by the windows in front of them.)

A *color window* can display color graphics. A noncolor window displays only black-and-white, i.e., monochrome. Because the window server must manipulate three times as much information when dealing with a color window, manipulation of a color window is slower than manipulation of a monochrome window. Normal drawing speed is the same in monochrome and color windows, however.

A window may have a *white border* between the windowframe and the writable portion of the window. The white border attribute is optional, since some applications may need to write to the edge of the window.

A window with the *clear on change* attribute is blanked by the window server after the user has changed the window size. If clear on change is not requested, the window contents are redrawn after the change. Redrawing can take several seconds for large color windows. An application that refreshs the entire window following any size change should request the clear on change attribute, so that time is not wasted.

A *VT* window is a special, full-screen window that permits the full range of terminal subsystem instructions, both text mode and graphics mode. See Chapter 3.

An *invisible* window is one for which Synergy does **not** do any of the normal video drawing operations. That is, ordinarily when a window is created, Synergy fills it in with white, and draws the windowframe around it. For a window with the invisible attribute, this is NOT done (no drawing whatsoever is done when an invisible window is created -- the video display is unaffected). Also, ordinarily when a Synergy window is deleted, the portion of the display "underneath" the window (other windows, etc.) is restored automatically. When an invisible window is deleted, this is **not** done. The video display is unaffected so that whatever was drawn into the invisible window REMAINS after the invisible window is deleted. Note that this is a dangerous thing to do, in that you can affect the contents of other windows, or even of the gray Synergy background.

There are situations where it is advantageous to use an invisible window. You might want to use an invisible window to guarantee that a certain sequence of PRO/GIDIS drawing instructions will be restricted to a portion of the application's normal drawing window, especially when the application does not have full control over what that sequence of PRO/GIDIS instructions is (for example .GID files that reset global addressing parameters). You could create a normal window and display the contents in it, but deleting the window will make the drawing disappear (which you may not want). Using an invisible window, you can do the above, with the result that the drawing will appear in the portion of the main window and will remain until that main window itself is deleted.

The *three-plane* attribute is similar to the COLOR attribute, except that a three-plane window requires that only EBO hardware be present. (Unlike the color attribute which requires a color monitor and end-user authorization using the Synergy setup feature). For example, the three-plane attribute makes gray scale windows possible on a monochrome display.

1.3.3 Video Protocols

Synergy supports both text and graphics in non-VT windows by requiring the use of the GIDIS protocol. Notice that GIDIS, although designed primarily as a graphics protocol, is quite capable of displaying text as well.

In order to avoid interference between windows, Synergy maintains a private copy of the state of GIDIS for each window. If an application creates more than one window, it switches between them by selecting the desired window with a call on the Select Window service. Synergy saves the GIDIS state of the old window and establishes the GIDIS state of the new window.

There is no "virtual window" larger than the actual window. Data scrolled off a window is lost, just as data scrolled off the top or bottom of a VT102 screen is lost. An attempt to write with coordinates that are outside the actual size of the current window results in clipping and loss of the data that is outside the writable area of the window.

GIDIS provides the following character renditions: italic (both forward and backward) and reverse video. Dim, bold and underline renditions can each be emulated by defining a font; Synergy defines special fonts that provide these character renditions. Blink is the only VT102 rendition that is not available, although the GIDIS block cursor can be used to blink a single rectangle of any size.

Nearly all GIDIS operations are available, but applications must observe certain restrictions (see Section 4.10).

1.3.4 Resources

Synergy copies a window's part of the video bitmap to disk in order to save the contents of the window for later restoration.

Each full-screen monochrome window requires 64 blocks (32 KB) of disk memory to hold the bitmap, plus approximately two blocks to hold the GIDIS state information. A color window has three planes of bitmap memory, so the requirements for storing the color window's bitmap memory are tripled. (The user must have a color monitor, and must choose the color option on the Synergy Set-Up Menu, before the application can create a color window.)

Synergy allocates a *raster file* on the hard disk for use as a storage area for application windows. Demands on the raster file increase as the user suspends applications and starts additional applications. If a peak demand exceeds the available raster space, Synergy extends the raster file. The raster file shrinks back to a minimum size when Synergy exits. (Notice that Synergy can exit only when all the Synergy applications have exited. Suspending an application and suspending Synergy in order to return to P/OS level does not constitute an exit.)

If disk space is exhausted it may be impossible to extend the raster file. This condition can arise when the Synergy window server is creating a new window in response to a service call from your application. The "Raster error" condition is returned to your application. The application must detect this error return and alert the user. The procedure is outlined in Section 2.1.5.

When an application starts, it usually creates at least one window. Although applications can create additional windows, they should destroy any windows that are no longer needed. This frees space in the raster file and also keeps the screen from being cluttered. (All application windows are destroyed automatically when the application exits.)

There is a limit of 16 simultaneous windows. Since most suspended applications have only one or two windows on the screen at the time of suspension, this limit is rarely reached.

When a window is created (or removed) in front of a color window, the color window is saved (or restored). The time required to save or restore the color window is three times the time for an eqeivalent monochrome window. This tripling of time applies only to the operadions on the color window, however. The time to save or restore a monochrome window i30oot affected.

Copy time is approximately one second per full plana copied. Therefore, changing from one full-screen color window to another full-screen color window requires about six seconds.

NOTE

A window with the VT attribute is always treated as a full-screen color window. All three planes of video bitmap are saved and restored.

1.3.5 Changing the Window Size

An application can change its window size by calling a window service. In addition, the user can change an application window's size while the application is suspended. The user changes the window size by using a Synergy Main Menu option. Applications must therefore be able to adjust to a new window size when control is returned from the Suspend call (WIINT). The window server returns a signal that the window size has changed and also returns the new width and height. The application may or may not need to repaint the window to conform to the new size (depending on what is being shown).

Applications can restrict size changes by setting upper and lower bounds on the window dimensions, and can create windows whose size may not be changed at all.

The application never receives notification that the user has moved the window to a new location on the screen, and there is no way for the application to restrict such movement. An application that is sensitive to the screen position of its windows can call the Get Window Parameters service, after each Suspend call to determine the window position.

1.3.6 Coordinate Systems

The video hardware consists of an array of pixels, 1008 wide and 240 high. (The video hardware is 1024 pixels wide, but Synergy uses only the leftmost 1008 pixels.) Hardware coordinates are not used to specify screen positions, however.

1.3.6.1 GIDIS Coordinates - GIDIS requires that coordinate systems be isotropic. A horizontal movement of N units must cover the same physical distance on the screen as a vertical movement of N units, so that geometric figures (such as circles) have the correct proportions (e.g., round circles, not ovals).

A coordinate system based on hardware pixels is not isotropic because the pixels on a Professional screen are not square -- they have an aspect ratio of 2:5. They are two and a half times higher than they are wide.

Synergy defines a matrix of "logical" pixels that is mapped to the hardware pixels. The logical pixels are isotropic and smaller than hardware pixels. Specifically, a logical pixel is half as wide and one fifth as high as a hardware pixel.

This gives Synergy a coordinate system with horizontal positions ranging from 0 on the left to 2015 on the right, and vertical positions ranging from 0 at the top to 1199 at the bottom. This defines a screen which is 2016 logical pixels horizontally by 1200 logical pixels vertically.

Notice that since the standard character cell is 12 hardware pixels wide (24 logical pixels), the Synergy screen holds 2016/24, or 84 full characters, rather than the usual 80 characters. In a window that has a windowframe and a white border, the writable area is reduced to 2000 GOS units, or 83 1/3 characters.

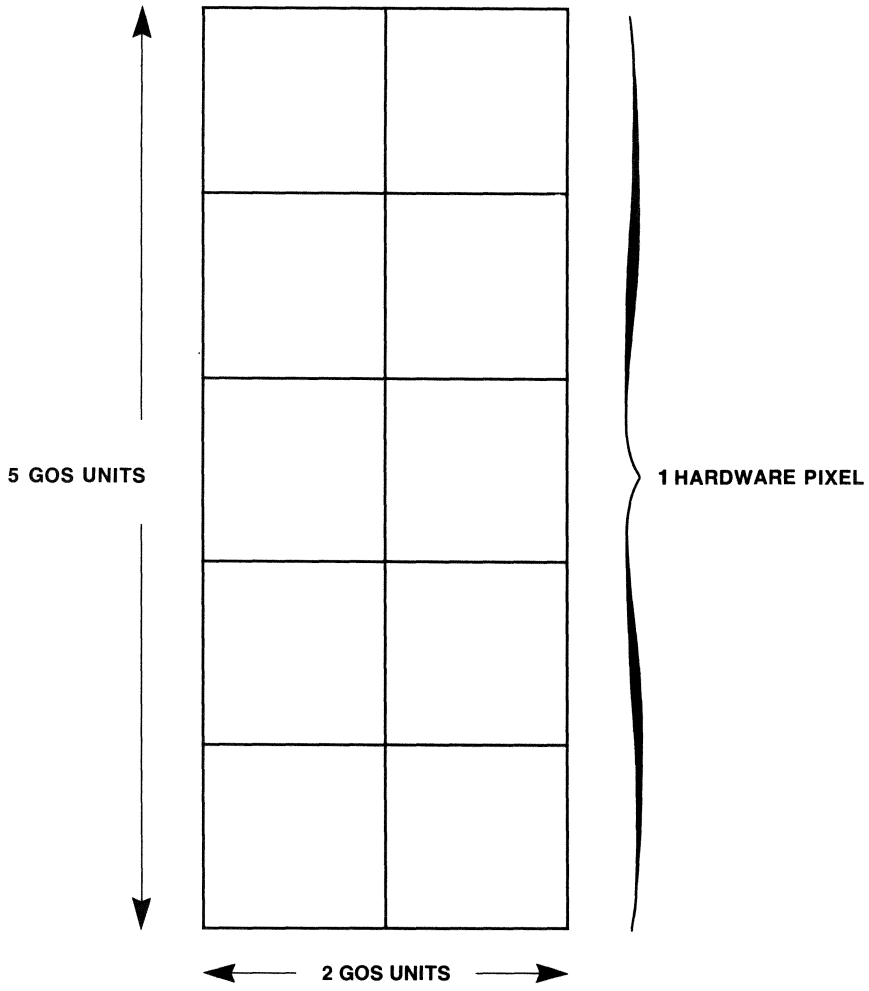


Figure 1-4: Logical Pixel Mapping (GOS Units)

Logical pixels are defined by Synergy and are known as GOS units (GIDIS Output Space units). (An application can define its own GOS units since they are part of the state information that is saved and restored for each of the application's windows.) Further discussions of coordinates in this document refer to the Synergy-defined GOS units.

1.3.6.2 Window Dimensions - Synergy defines the windowframe to be 2 GOS units wide on the left, 6 units on the right, and 10 units on the bottom. If there is a title, the top of the

windowframe is 65 units high. If there is no title, the top of the windowframe is 5 units high.

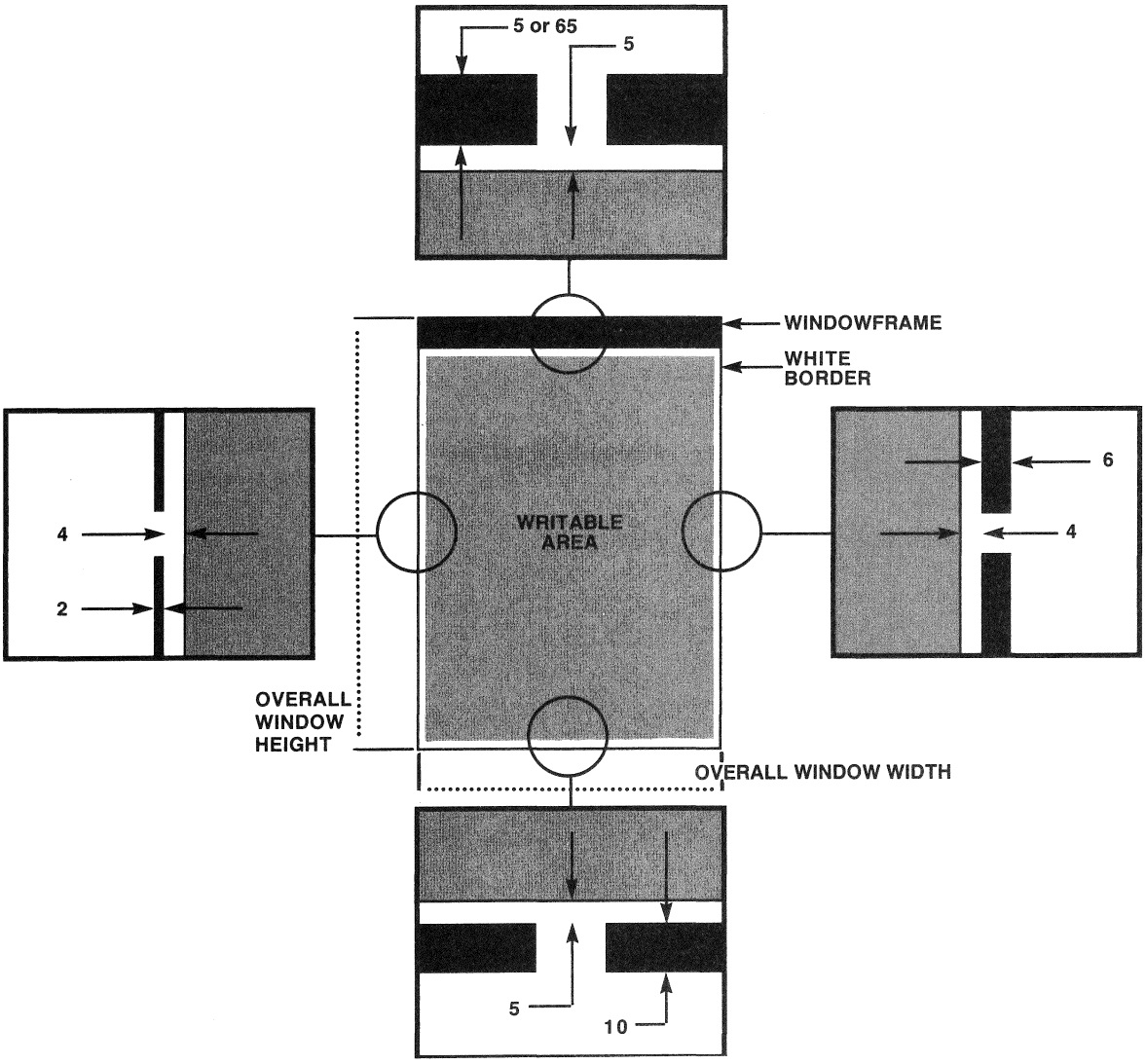


Figure 1-5: Window Dimensions in GOS Units

The windowframe is thicker on the right and at the bottom to give a shadow effect.

The optional white border between the windowframe and the writable area is 5 GOS units above and below the writable area, and 4 GOS units to the left and right of the writable area. Most applications request the white border so that the information that they place in the writable area cannot touch the windowframe.

The outside dimensions of the windowframe must be a multiple of 32 GOS units in width and a multiple of 5 GOS units in height. You request the window size by specifying the dimensions of the writable area, however. The window server will scale your requested size upward, if necessary, to guarantee that when the optional white border and windowframe are added, the total window size satisfies these multiples.

Normally an application is not concerned with exactly how large the window is, although there is a window service (WIGEW) that returns all of the exact sizes to your application.

An attempt to create a window with a writable area greater than the width or height of the screen returns an error. If the width and height for the writable area can be accommodated, but the frame and white border cannot, the window server reduces the writable area to accommodate the full window. Thus, a request for a window with a writable area that is 2010 units wide and 1190 units high would create a window with a writable area that is 2000 units wide and 1170 units high.

Creating a window of width and height equal to zero results in a full screen window with no white border or window frame.

1.3.7 Window Positions

You can request that a window be placed at any horizontal or vertical position, but the window server always adjusts the coordinates that you supply by rounding them down to the nearest *positioning unit*. Synergy positions every window horizontally on units of 16 hardware pixels, or 32 GOS units. Synergy positions every window vertically on a hardware pixel, or 5 GOS units. This means that if you specify a window position that is anywhere between 0,0 and 31,4, the window server adjusts the position down to 0,0. Likewise a requested position that is between 32,5 and 63,9 is adjusted to 32,5.

1.3.8 Window Services Overview

There are numerous window services, but the primary service is

the call to create a window. This service must be called to create the application's window. If the application uses additional windows, the call is repeated to create each such window.

The remaining window services are used to modify the window's size, position, or title. There is a service that requests an update of the window information from the window server, in case the user has modified the window's size or position while the application is suspended.

1.4 MENUS

A menu is a special window that is used to solicit input from the user. The input can be in the form of a selection from the menu choices, or entry of a string of characters or numbers.

It is important that menu operations be uniform for all applications, so that the user need not learn a new human interface for each application.

Menu services are defined at two levels:

- The *high-level* services typically create a window, display information in it, solicit a response from the user, destroy the window, and return the response to the application, all in a single call.

- The low-level services, called *primitives*, can be used to perform the same action over a sequence of calls. You can use primitives when you want to alter the system's behavior in its interaction with the user.

1.4.1 High-level Menu Services

High-level services are provided for your convenience and to foster a consistent human interface among different applications. (All high-level menu functions are actually implemented within the window server by calls on primitives.)

Services are provided for single and multiple choice menus, message frames, HELP frames and HELP menus, and set-up menus.

Many services are available in two forms, static or dynamic. A *static call* passes a *frame ID* and relies on the window server to fetch most of the window description from a frame file. A *dynamic call* passes all window data directly from the application at run time.

A *frame file* is a file that accompanies the application's task image (or images). It contains frame descriptions, which can be specified using a frame ID. Each frame description includes a frame type, positioning information, and text that is to appear in the frame. The application can have only one frame file open at any one time. The frame file contains all types of frames (menu, HELP, etc.).

1.4.2 Primitive Menu Services

The primitive menu services provide a means for creating a menu or an editing window with one call, then manipulating the contents of the window with additional calls. The window must be destroyed with yet another call when interaction with the user is completed. This requires more work on the part of the application developer, but lends flexibility and control to the behavior of the menu.

CHAPTER 2

CHAPTER 2

DESIGNING A NEW APPLICATION

This chapter presents guidelines for designing a new application. Perhaps no application can truly be called a new application, since most embody some aspects of an existing application, if not the actual source code. "New" in this context simply means that the developer has the inclination (and the time!) to consider the visible, interactive part of the application and to design or redesign it so that it is consistent with the existing Synergy models of the human interface.

2.1 THINKING ABOUT THE HUMAN INTERFACE

An excellent discussion of the human interface appears in the Digital Press Book, *The Human Factor*, by Richard Rubinstein and Harry Hersh. This book develops over 80 guidelines for good human interface design.

Although the Synergy tools and services make it convenient to build an application that has a well-designed human interface, good design does not happen automatically. There are a large number of decisions that must be made at every level of design to ensure a consistently good human interface.

The Human Factor urges the reader to test an application with representative users before committing it to distribution. The experience of the Synergy developers enforces this message. Even a small amount of such testing can reveal important flaws in the design. Often, design is based on assumptions about the user's experience or ability to cope with mistakes. Testing can reveal whether these assumptions are true and can suggest minor changes that may make a large difference in the user's success with the application.

THINKING ABOUT THE HUMAN INTERFACE

The Synergy applications are "integrated." Integration involves two things:

- The movement of data between applications
- The human interface of the applications

The clipboard method of moving data between applications employs an easy-to-use data file. The clipboard uses no new programming technique. It is simply a standardized file format and file naming convention. The clipboard provides the user with a conceptual model of the data flow between applications that parallels the passing of a clipboard containing written information between two people. Furthermore, each application that uses the clipboard names it, discusses it, and displays the options for using it, in the same way. The utility of the clipboard relies on adherence by all applications to the conventions that create it as a model.

Nothing prevents the design of Synergy applications that can share data using techniques other than the clipboard. The user's expectation, however, is that the clipboard is the medium for data sharing; and users will expect to see a Synergy application use the clipboard. Alternate methods of data sharing may puzzle users and require additional learning on their part.

The remainder of Synergy's integration relies entirely on the window and menu interface and the user's manipulation of it through the keyboard.

Each application appears to the user through one or more windows on the screen. Each application solicits input from the user through standard pop-up windows that contain menus or forms. The use of the keyboard to respond to these menus and forms is uniform across all the applications.

The Synergy system provides a large number of service calls that make it convenient for you to present this human interface in your application. Although the text in your menus and forms is unique to your application, the user is already familiar with the look and feel of this interface, since all Synergy applications use it.

Chapter 11 presents the conventions that are recommended for designing a fully integrated Synergy application. You may encounter a conflict between the model established by the Synergy applications and an alternative model that may be suggested by your application. You must decide on the tradeoffs between conflicting models. Within the applications that make up the Version 1.0 Synergy system, there is evidence of these tradeoffs. In certain cases, the developer either felt that the Synergy

conventions were too restrictive or that an alternative model was already established in the user's mind and so chose an alternative to the Synergy model. Chapter 11 contains a discussion of some tradeoffs that were made in the Version 1.0 Synergy applications.

2.1.1 The Type of Interaction

The Synergy conceptual model is to put information on the screen in such a way that the user sees as much as possible of his immediate memory portrayed in front of him. The intention is to reduce the need for the user to remember things, over either the short term or the long term.

This is the point of a menu-driven system versus a command-driven system. Instead of remembering the syntax and spelling of a command line, the user sees the relevant information on the screen and chooses from it. The menu of relevant choices is portrayed in a window that is just big enough to contain it. This tends to focus the user's attention to the smallest amount of information required for the next action and also leaves the most recent events in view, represented by other windows behind the menu.

If the user asks for HELP, the HELP text appears in another window, which again focuses attention and leaves the recent context in view behind the window.

The same type of interaction can carry over into the application's use of windows. The application can show the user what the current information is, and can invite the user to interact directly with that information in the window. The application can switch between two or more windows, if the information takes different shapes (the Graph application puts data in one window and the picture of the data in another window), or if the information comes from different locations that must be shown each in its own context.

2.1.2 The Screen Contents

The Synergy screen consists of overlapping windows. One of the windows is always the front window, the window that commands the user's most immediate attention. The windows behind the front window present a context for the user. They can be ignored if they are not needed in order to deal with the front window, or they can be consulted. The user is given a standard interface for moving the application windows about on the screen, so that

windows that are moderate in size can be located so that they remain in view while the front window is addressed.

In planning your application's use of windows, keep in mind that the windows behind the front window may be useful to the user, either because he actually wants to consult them for their information, or because they provide a reminder of the most recent actions.

The front window automatically provides an area of greatest attention, but within the window there should always be a point of attention. This is usually a blinking cursor or cursor bar. The movement of this point of attention provides clues to the user concerning the action of the program. Confirmation of the user's actions is often shown by a simple change in the shape or location of the cursor.

2.1.3 The Keyboard

Synergy uses the keyboard in essentially the same way as P/OS, but adds specific meaning to more of the function keys. All Synergy applications assign the same meaning to the F5 key, and most Synergy applications assign a common meaning to the F11, F12, F13, and ADDTNL OPTIONS keys, which is to display the application's top level menu, called the flow control menu.

Synergy provides a menu and HELP interface similar to the P/OS menu and HELP interface. Users can make menu choices by using the ARROW keys to move the cursor or by typing the leading characters of the menu option. The DO, RETURN, and HELP keys are used in the same way.

The Synergy interface presents a model in which the keyboard is attached only to the front window. When menus appear on the screen, keyboard actions are taken as responses to the menu window. When the menu is removed, the keyboard actions are taken as responses to the new front window. In addition, the Synergy Window Manager permits the user to define certain keys as strings of keystrokes. When the user presses one of these user-defined keys (UDK), the Synergy Window Manager substitutes the string of keystrokes.

This model of keyboard use requires that all Synergy applications buffer their keyboard input through a character-passing buffer. The character-passing buffer is an implied parameter of many Synergy service calls. A detailed description of the character-passing buffer and its use is given in Section 4.7.

2.1.4 The Format of HELP

Synergy HELP is always invoked by the HELP key and always appears in a window. Each HELP window contains a HELP message and menu options that lead to more HELP.

Since the HELP window is in front of the most recent window, the context in which the user requested the HELP is usually visible. The HELP services that are provided and the conventions that are recommended in Chapter 11 make it possible to provide extensive on-line HELP that most users can use without feeling lost.

2.1.5 The Handling of Errors

Error conditions are detected at various levels during the execution of your application:

- User errors
- Programming errors
- Resource errors
- Application abort

The methods for handling these errors are discussed in the following sections.

2.1.5.1 User Errors - When the user makes an error responding to your application, you may want to inform the user by displaying a message in your application window or by displaying a message in a special message window. If you choose to show the message in a special window, the Synergy convention is to request that the user press the RESUME key in order to proceed.

You may want to keep the message window short, assuming that the user's mistake is one of carelessness rather than ignorance. The Synergy service that displays your message frame has an option for linking a HELP frame to the message frame. If the user presses the HELP key while the short message frame is on the screen, he sees another window with the HELP message in it. You can place the longer explanation of the error condition (and how to correct or avoid it) in the HELP window. The HELP window can even lead into a tree of additional HELP information. You can design and program much of this user assistance in a way that keeps it outside your application task. Thus, the application code merely detects the error condition and makes a single call

on the Synergy service to start the user assistance.

2.1.5.2 Programming Errors - These are the errors that you expect will never happen, such as hardware faults or bounds checks on array accessing.

Many of the Tool Kit languages supply run-time systems that attempt to report these kinds of errors on the terminal, either by writing error messages directly to the screen (in text mode), or by calling a service in the POSRES cluster library. You must short-circuit these potential text-mode outputs to the screen, since such a message would likely be written outside your application window. You should request that all error conditions be returned to your application code, so that you can report the error without affecting the remaining application windows. (You may be able to sever the requirement that the language run-time system makes on the POSRES cluster library, and remove the POSRES library name from the command file that you use to build the application task.)

When your application reports this type of error to the user, it should tell the user that the error is not his error. It should also give the user some information that will help you to pinpoint the problem when the error is reported.

2.1.5.3 Resource Errors - When you create a window or execute a menu service that creates a window, the window server may need to extend the raster file. If the disk is full the extend request fails and the service returns an error status. You may encounter the resource problem again if you try to display an error message in a normal message window, since this will also try to create a new window. Synergy always reserves the resources necessary to display a special window called the *error window*. If your application detects the status return that signals a resource error, you should call the Error Window service, then exit the application.

2.1.5.4 Application Abort - The user can press INTERRUPT DO (or CTRL/C) while your application is running. If you have requested that the signal be returned to your application, it is returned. If your application does not make this request, the Synergy window manager gets the signal and terminates your application, with a message to the user.

2.2 FITTING INTO THE SYNERGY MODEL

Most Synergy applications have a sequence of interactions with the user that follow this pattern:

1. On starting, the application creates its titled window on the screen.

2. If the application normally deals with data in a file, the application puts a file selection or file creation window on the screen in front of its application window. (The application can supply copyright information or welcoming information in the application window or in the header area of the file selection window.)

3. The user indicates what file is desired.

4. The application begins its work in the application window.

5. The user reacts with data in the application window and calls up menus by pressing any of the F11, F12, F13 or ADDTNL OPTIONS keys.

6. The user asks for HELP at any time by pressing the HELP key.

7. The user suspends the application to do work in other applications, or to manipulate windows on the screen, by pressing the F5 key. The user resumes the application by selecting the application again.

8. The user leaves the application by pressing the MAIN SCREEN key or the EXIT key. MAIN SCREEN causes the application to save any work that has been done. EXIT causes the application to give the user a choice of saving new work or quitting without saving it. Both keys return the user to the Synergy Main Menu.

2.3 BUILDING THE APPLICATION

Your application consists of one or more task images and an install file that tells P/OS how to install and remove the application and how to start it when the user selects it from a P/OS Application Group Menu.

An application task image is constructed with the Professional Application Builder (PAB), using your object modules and the object library supplied with the assembler or compiler that you are using. The files and procedures to be followed are described

in the *Tool Kit User's Guide* and the *Tool Kit Reference Manual* and in the documentation that accompanies the assembler or compiler that you are using.

A Synergy application uses Synergy services to manipulate windows and to display menu and HELP frames. The Synergy services are supplied in separate task images and an object library. Your application task interfaces to these Synergy services through routines that are linked as part of your task image. The interface routines are drawn from the Synergy Interface Library, which you supply during task build.

In addition to the install file and the task images, your application contains an object frame file. The object frame file contains the menus, HELP frames and message frames that are displayed by the tasks during their execution. You create the source frame file by writing it in a frame language. The source frame file is then compiled into an object frame file by the Frame Compiler Tool, FCT.

2.3.1 Task Names

All Synergy applications run as spawned tasks from the Synergy window manager task. Hence all Synergy tasks must have unique task names.

The following names are already in use by Synergy Version 1.0 tasks:

```
CETSK  - PROSE PLUS          SPSRES - Spreadsheet
CHESS  - Chess               WIAG   - Graph
GEDF01 - PROSE PLUS          WICAF  - Calculator
GEDSYN - PROSE PLUS          WICAT  - Calculator
MXPRO  - Communications      WICNV  - Datamanager Convert
PRSSK  - PROSE PLUS          WIFSV  - File Services
PVUSYN - File Services       WIRG   - Datamanager
SPLCHK - PROSE PLUS          WIRS   - Datamanager
SPSHEE - Spreadsheet
```

All task names used in future releases of Synergy will have the WI prefix, so you can avoid conflicts by not using the above names or any names beginning with WI.

2.3.2 The Synergy Interface Library

The name of the Synergy Interface Library is LB:[1,5]WINLIB.OLB. The application in Appendix A makes reference to this library (see Page A-70).

2.4 INSTALLING THE APPLICATION

A Synergy application is installed on a P/OS Application Group menu and also on the Synergy Main Menu. The Synergy application can be started from either menu.

Some special commands are required in the install file of a Synergy application. There are additional rules for the install file if you wish to create a "shared" application to be run from P/OS V3.0. Shared applications in P/OS V3.0 require an .INB file in addition to the .INS file required for P/OS V2.0 (see *PRO/Toolkit Manual*). The following sections describe modifications needed for both the .INB file and the .INS file.

2.4.1 SYNERGY INSTALL FILE (.INS)

The application installation file (the .INS file) must begin with a special comment line:

```
!SYNERGY/I2
```

Do not insert a space between the exclamation point and the following S. This line indicates to Synergy that this is a valid Synergy application install file. There are additional switches that can be applied to this line and these will be described later.

The next modification to the .INS file is immediately after the 'Name' command line. Insert these three lines after the 'Name' command line:

```
FILE [ZZPROVUE]SYNCHK2.TSK/DELETE
FILE [ZZPROVUE]SYNERR.HLP/KEEP
EXECUTE [ZZPROVUE]SYNCHK2.TSK/INS
```

These commands copy two files to the hard disk upon installation and execute the SYNCHK2 task. The purpose of the SYNCHK2 task is to verify that correct version of the Synergy Window Manager (V2.0) is already installed on the user's system. If the correct version of the Synergy Window Manager is not on the user's system

an appropriate error message will be displayed.

The two file lines imply that you must have these two files (SYNCHK2.TSK and SYNERR.HLP) on your application diskette in directory [ZZPROVUE]. Therefore, before completing your application diskette you must copy these files from the Synergy Tool Kit diskette (SYNTK1) to your first application diskette in directory [ZZPROVUE].

The next modifications to your .INS file allows Synergy to update its Main Menu whenever the application is installed or removed from the P/OS application menus.

- Application installation:

  ```
  EXECUTE [ZZPROVUE]INSAPP.TSK/INS
  ```

  This command is placed in the install file immediately before the first "Install" command line. When the user installs the application, INSAPP.TSK ensures that the application name is added to the Synergy Main Menu.

- Application removal:

  ```
  EXECUTE [ZZPROVUE]REMEXE.TSK/REM
  ```

  This command is placed in the install file immediately before the first "File" command line. If the user removes the application, REMEXE.TSK ensures that the application name is also removed from the Synergy Main Menu.

After installation, the Synergy application can be started from either the P/OS Application Menu or the Synergy Main Menu. (If the application is suspended with the F5 key, control returns to the Synergy Main Menu.) In order to ensure that the Synergy Window Manager is in control when the application is started from the P/OS menu, the install file does not call for the running of the application, but instead directs P/OS to run the Synergy Window Manager. The commands that would normally be in the install file for running the application are made into comment lines by placing an exclamation mark in front of the command.

When the window manager begins to run, it reads the install file and executes the commented commands; and then it starts the application by spawning it.

If the application were not a Synergy application, its .INS file might have the following commands: (Assume the application task name is APLNAM in the file APPLFILE.TSK.)

```
INSTALL [ZZSYS]PBFSML.TSK/LIBRARY
INSTALL APPLFILE.TSK/TASK
RUN     APLNAM
```

As a Synergy application, the INSTALL and RUN commands are made into comments and two new commands are inserted, causing the Synergy Window Manager to be started. The result is:

```
!INSTALL [ZZSYS]PBFSML.TSK/LIBRARY
!INSTALL APPLFILE.TSK
!RUN     APLNAM
INSTALL  [ZZPROVUE]SYNRUN.TSK/TASK
RUN      WI$MGR
```

If the .INS file contains ASSIGN commands, these too must be made into comments. Thus, an ASSIGN MENU MYMENU.MNU line becomes !ASSIGN MENU MYMENU.MNU.

Figure 2-1 shows a side-by-side comparison of an application's .INS file as it would be for a non-Synergy application and as it is after Synergy modifications. Assume that the original application uses the P/OS menu and HELP services available through the POSRES cluster library. Assume that the application is rewritten to include some Synergy menu services that use a Synergy frame file, but that it continues to use the P/OS menu and HELP services as well.

BEFORE	AFTER
	!SYNERGY/I2
NAME "Sample"	NAME "Sample"
	FILE [ZZPROVUE]SYNCHK2.TSK/DELETE
	FILE [ZZPROVUE]SYNERR.HLP/KEEP
	EXECUTE [ZZPROVUE]SYNCHK2.TSK/INS
	EXECUTE [ZZPROVUE]REMEXE.TSK/REM
	FILE SAMPLEFRM.OFF/DELETE
FILE SAMPLEV1.TSK/DELETE	FILE SAMPLEV1.TSK/DELETE
ASSIGN MENU MYMENU.MNU	!ASSIGN MENU MYMENU.MNU
ASSIGN HELP MYHELP.HLP	!ASSIGN HELP MYHELP.HLP
	EXECUTE [ZZPROVUE]INSAPP.TSK/INS
INSTALL [ZZSYS]PBFSML.TSK/LIBRARY	!INSTALL [ZZSYS]PBFSML.TSK/LIBRARY
INSTALL SAMPLEV1.TSK/TASK	!INSTALL SAMPLEV1.TSK/TASK
RUN SAMPLE	!RUN SAMPLE
	INSTALL [ZZPROVUE]SYNRUN.TSK/TASK
	RUN WI$MGR

Figure 2-1: Sample (.INS) Install File

2.4.2 SYNERGY INSTALL FILE (.INB) FOR SHARED APPLICATIONS

As mentioned in a previous section, the .INB file allows an application to be shared on the P/OS V3.0 application environment. The .INB file is similar to the .INS file except that it contains additional information on the specific placement of the application files. Refer to the *Tool Kit Reference Manual* for more information on shared applications.

To create a shared application, you will need both a .INS file and a .INB file on your application diskette. If you do not wish to create a shared application you will not need the .INB file which is described in this section.

NOTE

The .INB file can only be tested on a P/OS V3.0 system.

As with the .INS file, the shared application installation file (the .INB file) must begin with a special comment line:

```
!SYNERGY/I2
```

The next modification to the .INB file is immediately after the 'Name' command line. Insert these four lines after the 'Name' command line:

```
FILE [ZZPROVUE]SYNCHK2.TSK/DELETE
FILE [ZZPROVUE]SYNERR.HLP/KEEP
EXECUTE [ZZPROVUE]SYNCHK2.TSK/INS/USR
EXECUTE [ZZPROVUE]SYNCHK2.TSK/INS
```

The next modifications to your .INB file allows Synergy to update its Main Menu whenever the application is installed or removed from the P/OS application menus.

Insert the following two command lines before the first "Install" command line:

```
EXECUTE [ZZPROVUE]INSAPP.TSK/INS/USR
EXECUTE [ZZPROVUE]INSAPP.TSK/INS
```

Then insert the following two command lines before the first "File" command line:

```
EXECUTE [ZZPROVUE]REMEXE.TSK/INS/USR
EXECUTE [ZZPROVUE]REMEXE.TSK/INS
```

After installation, the Synergy application can be started from

either the P/OS Application Menu or the Synergy Main Menu. If the shared application were not a Synergy application, its .INB file might have the following commands:

```
INSTALL [ZZSYS]PBFSML.TSK/LIBRARY/CLUSTER
INSTALL APPLFILE.TSK/TASK/NETWORK
RUN APPLNAM
```

As a Synergy application, the INSTALL and RUN commands are made into comments and two new commands are inserted causing the Synergy Window Manager to be started. The result is:

```
!INSTALL [ZZSYS]PBFSML.TSK/LIBRARY/CLUSTER
!INSTALL APPLFILE.TSK/TASK/NETWORK
!RUN APLNAM
INSTALL [ZZPROVUE]SYNRUN.TSK/TASK/CLUSTER
RUN WI$MGR
```

If the .INB file contains ASSIGN commands, these too must be made into comments. For example, an ASSIGN MENU MYMENU.MNU line becomes !ASSIGN MENU MYMENU.MNU.

Figure 2-2 shows a side-by-side comparison of an applications's .INB file as it would be for a non-Synergy application and as it is after Synergy modifications.

BEFORE

```
NAME "Sample"



FILE SAMPLEV1.TSK/DELETE/NETWORK
ASSIGN MENU MYMENU.MNU
ASSIGN HELP MYHELP.HLP


INSTALL [ZZSYS]PBFSML.TSK/LIBRARY/CLUSTER
INSTALL SAMPLEV1.TSK/TASK/NETWORK
RUN SAMPLE
```

AFTER

```
!SYNERGY/I2
NAME "Sample"
FILE [ZZPROVUE]SYNCHK2.TSK/DELETE
FILE [ZZPROVUE]SYNERR.HLP/KEEP
EXECUTE [ZZPROVUE]SYNCHK2.TSK/INS/USR
EXECUTE [ZZPROVUE]SYNCHK2.TSK/INS
EXECUTE [ZZPROVUE]REMEXE.TSK/REM/USR
EXECUTE [ZZPROVUE]REMEXE.TSK/REM
FILE SAMPLEFRM.OFF/DELETE
FILE SAMPLEV1.TSK/DELETE/NETWORK
!ASSIGN MENU MYMENU.MNU
!ASSIGN HELP MYHELP.HLP
EXECUTE [ZZPROVUE]INSAPP.TSK/INS/USR
EXECUTE [ZZPROVUE]INSAPP.TSK/INS
!INSTALL [ZZSYS]PBFSML.TSK/LIBRARY/CLUSTER
!INSTALL SAMPLEV1.TSK/TASK/NETWORK
!RUN SAMPLE
INSTALL [ZZPROVUE]SYNRUN.TSK/TASK/CLUSTER
RUN WI$MGR
```

Figure 2-2: Sample (.INB) Install File

2.4.3 Installing a standard P/OS application

It is possible to install a standard (non-Synergy) application into the Synergy environment. The main advantage of this is to allow a non-Synergy application to be run from the Synergy Main Menu as well as the P/OS application menus.

To install a non-Synergy application, modify the non-Synergy application's install file as specified in the previous sections (2.4.1 and 2.4.2). Then place a "VT" switch on the "!Synergy/I2" command line. The Synergy command line should appear as follows in your install file:

```
!SYNERGY/I2/VT
```

Once you have made these modifications to your non-Synergy application's install file and have installed the application, it will appear in both the Synergy Main Menu and the P/OS Application Menu.

You can now run this appliction from the Synergy Main Menu. The "VT" switch that appears in the install file tells Synergy to create a full screen VT style window before starting the application. This preserves the rest of the Synergy environment while you are running in Synergy. When you exit the application the VT style window is deleted and the Synergy environment is restored.

NOTE

> This modification does not provide any additional functionality to the application. It simply allows you to run a non-Synergy application from the Synergy environment. If you wish additional 'Synergy-type' functionality in the application (such as Suspend or Window menus) the application itself must be modified.

2.5 RUNNING FROM THE TOOL KIT AND OTHER APPLICATIONS

When you are developing an application from the PRO/Tool Kit, it is convenient to be able to start the application from the Tool Kit, rather than from P/OS Main Menu level. In addition, it may be desirable to start a Synergy application by spawning it from a non-Synergy application.

NOTE

> The execution of a Synergy application from the Tool Kit may place a heavy demand on system resources.

The Synergy Window Manager accepts a command line at start-up and uses the command line to determine the context under which it is being started.

- The application can be started by selecting it from either the P/OS Application Group menu or the Synergy Main Menu. This method of starting uses the application's install file. The commands in the install file call for installation of the SYNRUN.TSK file and execution of the window manager task, as described in the preceding section. In this case, there is no command line being passed to the window manager.

- The window manager can be started from DCL by executing the following three commands:

```
$ INSTALL LB:[ZZPROVUE]SYNRUN.TSK
$ RUN WI$MGR/COMMAND="MANAGER"
$ REMOVE WI$MGR
```

  This method of starting passes the command "MANAGER" to the window manager. The window manager displays the Synergy Main Menu. You can then select any application that has been installed on the Synergy Main Menu. This is equivalent to starting the Synergy Window Manager from a P/OS Application Group menu.

- A Synergy application can be started directly from DCL by executing the following three commands:

```
$ INSTALL LB:[ZZPROVUE]SYNRUN.TSK
$ RUN WI$MGR/COMMAND="START [appldir]"
$ REMOVE WI$MGR
```

  You replace "appldir" with the directory name that contains the application's installation file. If the application under development is called FOO and it has been fast-installed from the directory [FOO], the DCL line would be

```
$ RUN WI$MGR/COMMAND="START [FOO]"
```

  If the application has been installed with P/OS Disk/Diskette Services and installation placed it in [ZZAP00143], the command would be

```
$ RUN WI$MGR/COMMAND="START [ZZAP00143]"
```

  Notice that the application must have a properly constructed (Synergy) INS file (see Section 2.4.1).

- Either of the command lines may be passed to the PROTSK routine with the install/run/remove option. This enables a running application to start the Synergy window manager at its Main Menu level, or to start a Synergy application. (See the PROTSK routine, described in the *P/OS System Reference*

Manual.)

In either case, the Synergy application's exit status is not returned by the PROTSK routine until the Synergy window manager exits. The window manager exits with either success (1) or failure (greater than 1), which it derives from the application's exit status.

When a Synergy application is started using the "START []" command, the application is considered to belong to the application that calls the PROTSK routine. As long as the Synergy application does not suspend itself (using the Suspend service), the window manager remembers its owner, so that when the Synergy application exits, the window manager exits as well, returning status as described above. The application that called the PROTSK routine then receives control.

However, if the Synergy application suspends itself (in response to the F5 key), the window manager displays the Synergy Main Menu. At this point, the user can start other Synergy applications and can even suspend the Synergy environment. Since all these possibilities exist, the window manager assumes ownership of the application. If the user suspends the window manager, the window manager exits by issuing success status to the task that called the PROTSK routine. Notice that the Synergy application (the callee) may still be executing. If the task that called the PROTSK routine (the caller) requires that the Synergy application complete its execution before the caller can proceed, the caller and callee must establish some other method of communicating exit status.

CHAPTER 3

ADAPTING A P/OS APPLICATION

This chapter supplies guidelines for the developer who is modifying a P/OS hard disk application so that it will run in the Synergy environment. All necessary modifications are described in general terms. The details are provided in other chapters.

Use this chapter to determine the scope of your work and to organize and plan the modifications.

The application's source code must be modified so that it communicates with the Synergy services. There are three areas in which the source code must be modified:

- Reading the keyboard and using the character-passing buffer
- Suspending the application
- Using the screen

The files that control the application's task build and installation must be modified also.

You should also consider whether the application can profit from use of the clipboard as an input or an output medium, or both.

3.1 KEYBOARD USE

Synergy provides a buffer for keyboard input called the *character-passing buffer*. The character-passing buffer is passed to an application when Synergy starts or resumes the application. The application must use any bytes in the character-passing buffer before doing any QIOs to read the keyboard directly. Likewise, when the application calls a Synergy service, all keystrokes that have been read (but not used) must be placed in the character-passing buffer for use by the Synergy service. The character-passing buffer gets passed back and forth between the

application and Synergy.

In effect, all keyboard input flows through the character-passing buffer. A detailed description of the character-passing buffer and its use is given in Section 4.7.

Since all Synergy applications use the Synergy character-passing buffer, they are all required to read the auxiliary keypad in the same mode; namely, 8-bit, application keypad mode. This distinguishes the numeric and punctuation keys on the keypad from the same keys on the main array of the keyboard.

You can assign the same meaning to these keys as to their counterparts on the main array of the keyboard, so that the user is not aware of the distinction in the way that they are read. However, in order to unambiguously pass any type-ahead on to other applications, all applications must read the keys in the mode that distinguishes the actual key that has been pressed.

3.2 SUSPENDING THE APPLICATION

All Synergy applications recognize the F5 key in their keyboard input and call a task control service which suspends the application. The F5 key should be recognized at all times, and the task should never require additional keystrokes before it suspends its execution.

If the application spawns additional tasks, the developer need not stop all the tasks before calling the Suspend service, but must ensure that any tasks that continue to run execute no input or output to the terminal while the calling task is suspended.

3.3 SCREEN USE

You have three choices of how to use the screen. Each of the choices involves the creation of a window, but two of the choices mean fewer changes to the application code.

- You create the window with the VT attribute, and you do all remaining screen operations exactly as in the P/OS hard disk environment, including using the menu and HELP services of P/OS available in the POSRES cluster library.

- You create the window with the VT attribute so that you can continue to do the same terminal output that you did in the P/OS environment, but you replace the calls on POSRES by calls on the Synergy menu and HELP services. This means

creating a Synergy frame file from the frame files produced with the Frame Development Tool (FDT).

- You create the window without the VT attribute -- as a standard Synergy window, probably smaller than the full screen. This means you must do all screen output with GIDIS QIOs and that you must **not** use the menu and HELP services of P/OS.

3.3.1 Retaining the VT Window Type

A Synergy application must avoid any screen output until it calls a service that creates a window. The window may be created with a special attribute called the VT attribute. This creates a full-screen window.

It is possible to use either the P/OS menu and HELP services (in POSRES) or the Synergy menu and HELP services, or even a mixture of the two.

The application can suspend itself by calling the Suspend service, since Synergy saves the screen contents and the GIDIS state of the terminal subsystem. When the application is restarted after the Suspend service, Synergy guarantees that the screen is correctly restored and that the GIDIS state is restored. The application must restore the text-mode state of the terminal subsystem; and if it has altered the color map, it must repeat the color map set-up.

Notice that Synergy restores the video bitmap so that the screen looks right to the user. However, the terminal subsystem maintains additional screen information, which is not restored by Synergy. If the user presses the PRINT SCREEN key after the application resumes, the printed result might not be an accurate representation of the screen. To guarantee that PRINT SCREEN will work correctly after suspending and resuming, you must repaint the entire text-mode contents of your VT window (with any character attributes that were used initially).

Keep in mind that the window server always saves three planes of video bitmap for a VT window (if they are present), regardless of whether the user has asked the Synergy Window Manager to allow the use of color windows, and regardless of whether the application requests that the window be created with the color attribute.

3.4 MODIFICATIONS TO OTHER FILES

3.4.1 Task Build Files

You must modify any ODL file that you submit to the Professional Application Builder to build a task that references a Synergy service. The ODL file must include a reference to the Synergy Interface Library. The library routines add about 2000 (decimal) bytes to your task image. If you are replacing calls on the POSRES cluster library, you may be able to remove the reference to the POSRES library in the command file and regain an equivalent amount of space.

3.4.2 Install File

In addition to telling Disk/Diskette services how to install and remove your application, the install file tells P/OS how to start your application when the user selects it from a P/OS application menu. When your application becomes a Synergy application, its main task image is no longer started by P/OS. You must modify the install file in such a way that the install file tells P/OS to start the Synergy Window Manager. The Synergy Window Manager then reads the install file and spawns your application's main task image as a subtask.

3.5 USING THE CLIPBOARD

The clipboard consists of two files that are used to pass user data between Synergy applications. The files have fixed names and are always stored in a system directory. Applications follow a set of simple rules in writing and reading these files.

The advantage to the user is that he need not name the files or remember where they are.

If you decide to modify your application to read from or write to the clipboard, you should follow these rules. You should also follow the conventions for describing the clipboard actions in menus and in your user documentation. For example, it would be a mistake to tell the user that your application uses the clipboard, and then require him to type the directory name and filename of the clipboard file every time that he wants your application to use it!

CHAPTER 4

CHAPTER 4

THE SYNERGY INTERFACE

4.1 INITIAL STATE

Your application calls Synergy services to perform various actions on its behalf. Many of the services read the keyboard or alter the screen contents. One of the services, Suspend, even gives the user a chance to start another application.

These actions that take place outside the application's code may alter the states of the terminal. Synergy sets the terminal back to a known state on return from each call. These initial states are described in the following sections.

4.1.1 At Synergy Start-Up

At Synergy start-up, the server performs the following actions:

```
Text mode set-up:
        Text cursor home
        Text cursor off
Keyboard set-up:
        Set ANSI cursor key mode
        Set application keypad mode
        Set 8-bit codes only (CSI, not ESC [)
GIDIS set-up:
        Initialize (-1 - Everything)
        Load Synergy alphabets
        Set output cursor (No cursor)
        Set writing mode (6 - Replace)
        Set alphabet (0)
        Set cell display size (24,50)
        Set cell unit size (24,50)
        Set Synergy colors in the color map
        Create gray background
```

4.1.2 At Window Creation

When a window is created, the following actions occur:

```
GIDIS set-up:
        Set IDS to size of writable area
        Initialize (2!4!8 -Reset Global attributes,
             Text, and Cursor)
        Set output cursor (No cursor)
        Set writing mode (6 - Replace)
        Set alphabet (0)
        Set cell display size (24,50)
        Set cell unit size (24,50)
        If it's a color window then
                Set plane access (7)
        else if this system has EBO then
                Set plane access (4)
        else
                Set plane access (1)
        Set primary color (0)
        Set secondary color (4)
        Fill window with white
        (GIDIS active position is at 0,0 in the window)
```

4.1.3 On Return from Suspend

After a Suspend, which may include a change in window size:

```
GIDIS set-up:
        Set Synergy colors in the color map
        Set IDS to size of writable area
        Set alphabet (0)
        Set cell display size (24,50)
        Set cell unit size (24,50)
```

4.1.4 After Other Window Operations

After other window operations:

```
GIDIS set-up:
        Set IDS to size of writable area
        Set alphabet (0)
        Set cell display size (24,50)
        Set cell unit size (24,50)
```

The keyboard is in application keypad mode: The keypad keys do not return the characters "1", "2", etc., but return CSI sequences. The keyboard is in 8-bit mode. Function keys return 8-bit sequences instead of the longer 7-bit sequences.

For VT windows, no GIDIS set-up is done, except for setting the color map (see Section 4.2).

4.2 COLOR MAP

Synergy establishes the settings of the color map during start-up with the following values:

Color Index	% of Red	% of Green	% of Blue	Result
0	0	0	0	black
1	100	0	0	red
2	0	100	0	green
3	0	0	100	blue
4	100	100	100	white
5	0	100	100	cyan
6	100	0	100	magenta
7	100	100	0	yellow

You can change these settings by altering percentages, but you should be aware of the effect of your changes and the rules that must be followed:

- Changes that you make to the color map will be applied to all windows on the screen, since there is no way to restrict the effect of the color map to your window only.

- When you call a Synergy menu service, the menu will be displayed using your color map settings. Since menus are displayed with color index 0 providing the primary color and color index 4 providing the secondary color, you could make a menu very hard to read by setting these color indices to noncontrasting colors. Try to avoid modifying the 0 and 4 settings. If you must modify them, you may have to precede each call on menu services with an adjustment that makes the menu readable.

- When your application suspends itself or exits, Synergy resets the color map to the Synergy settings. Thus, whenever your application resumes after a suspend, you must reset the color map to your own settings.

The application in Appendix A alters the color map (see Pages

A-5, A-7, A-14, A-15, and A-17).

4.2.1 WIZPSC - Zap Primary/Secondary Colors

Status	2 words (output)
PrimaryColor	1 word (input)
SecondaryColor	1 word (input)

This call alters the color indices that the server uses when drawing window frames and such things. The new color values should be in the range 0-7. The server simply passes these values on to PRO/GIDIS, in SetPrimaryColor and SetSecondaryColor instructions. Once zapped, the specified indices will be used for ALL future window operations (even other Synergy applications). The primary and secondary colors are not reset until the entire Synergy environment is exited (not suspended -- exited). In other words, the NEXT TIME Synergy is run, the primary/secondary colors will be back to 0 and 4. If you want the new primary/secondary colors to be in effect only until your application exits, the application must zap them back to 0 and 4 when it exits.

Note that zapping either the primary or secondary to be values other than 0 or 4 forces all future windows to become color windows (because all three planes of bitmap must be saved/restored). This includes stackable windows, and further includes stackable windows that Menu Services creates.

In fact, you should not issue this call if there are ANY windows that have already been created (even windows from other applications). Synergy will not crash or become corrupted if you do so, but the on-screen appearance of the old windows (and the old gray background) can be wrong.

4.2.2 WIZCMP - Zap Color Map Entry

Status	2 words (output)
Map	1 word (input)
Index	1 word (input)
Red	1 word (input)
Green	1 word (input)
Blue	1 word (input)
Mono	1 word (input)

This call zaps the color map entries that Synergy enforces. The six input parameters are the same parameters that the PRO/GIDIS SetColorMapEntry instruction takes.

Similar to the WIZPSC call, any changes to the color map made using this call remain in effect until the entire Synergy environment is exited.

Note that if all you want is to change the on-screen colors while your application is running, you should **not** use this call. Instead you should simply issue GIDIS SetColorMapEntry instructions from your application; then Synergy will reset the color map to the Synergy defaults when the application exits or suspends. The WIZCMP call is provided to change the defaults that Synergy uses, so they are permanent for the duration of the Synergy environment.

4.2.3 WIRCMP - Reload Color Map

The following call causes Synergy to reset the PRO/GIDIS color map back to the default Synergy colors.

Status	2 words (output)

This call is useful if your application changes the color map using the PRO/GIDIS SetColorMapEntry instruction, and you wish to return to the standard Synergy color map settings.

Synergy implicitly calls this routine whenever an application exits (WIDON) or suspends (WIINT).

If the Zap Color Map (WIZCMP) routine has been used to change the default Synergy color map, WIRCMP reloads the color map with those changed default values -- **not** the original power-up color pallete.

4.3 FONTS AND ALPHABETS

GIDIS defines alphabet 0 as the DEC Multinational Character Set. The font style (character shape) is essentially the same as the font style defined by the text mode of the terminal subsystem.

Synergy defines additional fonts for displaying characters in menus, and for some special effects that it requires. At Synergy start-up, the fonts are installed in common regions and then loaded by name into GIDIS alphabets as follows:

Alphabet Index	Font Name	Font Description
7	WI$F0	Special
8		(reserved)
9	WI$F1	Dim
10	WI$F2	Normal
11	WI$F3	Bold
12		(reserved)
13	WI$F5	Normal underlined
14	WI$F6	Bold underlined
15	WI$F8	Boxed

To display characters from one of the Synergy fonts in your application window, you must use a GIDIS SET_ALPHABET instruction to select the desired alphabet before issuing any DRAW_CHARACTERS or DRAW_PACKED_CHARACTERS instructions. All Synergy fonts except the Special font use the standard cell unit size and cell display size of 24 wide by 50 high (GOS units), which is the same as the terminal subsystem's text-mode character that is 12 hardware pixels wide and 10 hardware pixels high.

The application in Appendix A uses these fonts (see Page A-45).

Synergy defines these special fonts for various reasons:

- Synergy needs to provide various renditions of the standard characters. The renditions provided are dim, bold, and underlined versions of the normal character.
- Synergy needs to draw a box around certain text to make it look like a key caption.
- Synergy needs to combine some of the characters from the DEC Special Graphics character set with the characters that form the DEC Multinational character set.
- Synergy needs a few special characters that are not available elsewhere.

4.3.1 User-Defined Fonts

You can load fonts that you design into any of the alphabets 1 to 6. However, when you call a Synergy window or menu service, or when you suspend your application, you lose all your font definitions. Thus, you must reload your fonts on return from the Synergy services. Notice that this means that fonts must be

loaded after a window is created, not before. If you create two windows, your fonts must be loaded each time you select a new front window.

You should avoid loading your own font into any of the alphabets used or reserved by Synergy. If you must load your font into an alphabet that is used or reserved by Synergy, you must use the Restore Fonts service (see below) that requests Synergy to reload its font into that alphabet. This prevents the display of menus, HELP, etc., using your fonts instead of the Synergy fonts. This request must be made prior to any call on Synergy menu services, prior to a call on the Suspend service, and prior to exiting your application.

Synergy reloads its own fonts before returning control to your application after a suspend.

4.3.2 WIRFNT - Restore Fonts

Status	2 words (output)
BitMask	1 word (input)

Bits in the BitMask correspond to alphabet numbers that Synergy should restore. Thus, to instruct Synergy to reload its font into alphabet 7, you would supply a BitMask with the value 128 (2 ^ 7).

Synergy ignores bits 0 to 6 of the BitMask, so that you can supply a mask of -1, which causes Synergy to reload all of its fonts.

4.3.3 Special Font

The Special font uses a larger cell display size (16 by 16 hardware pixels) and has only two characters defined in it. These are the arrow that the window manager uses to choose a new front window, and a pattern that is used to create the gray background.

4.3.4 Text Fonts

The Dim, Normal and Bold fonts, and the underlined versions of Normal and Bold, all have exactly the same character shapes, with the variation being in the number of pixels that are turned on and whether or not the bottom row of pixels is turned on for

underlining. This is a full 256-character alphabet, which contains the DEC Multinational printing characters and is augmented with additional characters from the DEC Special Graphics character set that are placed in the nonprinting positions of the DEC Multinational set. This is the Synergy Character Set, shown in Table 4-1.

Notice that characters 134 and 135 (decimal) when placed together form the clock icon that is used in the wait message of the title line of windows. Characters 136 and 137 (decimal) are reserved. Characters 156 to 159 (decimal) are the multiplication and division signs, the centered dot and the checkmark. Character 160 (decimal) is the ellipsis used in various ways by PROSE PLUS, Graph, and Spreadsheet. The remaining special characters in the 128 to 155 (decimal) positions are various characters from the DEC Special Graphics character set, including the characters known as the line-drawing characters.

Notice that these characters can be placed in a frame file, providing that you edit the frame file with an editor that handles nonprinting 8-bit characters. To use the text fonts in a menu, HELP or message frame, see Section 7.2.1. Be sure to observe the conventions that are established for these text fonts (see Chapter 11).

4.3.5 Printing the Synergy Character Set

If these fonts are displayed in your application window, and the window is printed on a dot-matrix printer such as the LA50 with the PRINT SCREEN key, they will be printed correctly on the paper. This is because the PRINT SCREEN key sends the actual video bitmap to the printer (as sixels).

However, if your application tries to store these characters in a file, and the user prints the file using Print Services, the characters will be sent to the printer for interpretation as DEC Multinational characters. Thus, all the special Synergy characters will print as blanks or reserved symbols, since they are placed into the nonprinting area of the DEC Multinational Character Set.

4.3.6 Boxed Font

The Boxed font is the same as the Synergy Character Set, with the addition of a dim line above each character (the top of the box) and a normal line below each character (the bottom of the box). All the characters of the Synergy Character Set are available

except the following:

- The ASCII codes for curly braces (decimal 123 and 125) are used to select the character shapes that form the left and right ends of the box.

- The ASCII codes for the lowercase letters, u, d, l, and r (decimal 117, 100, 108, and 114) are used to select the character shapes for the up arrow, down arrow, left arrow and right arrow, respectively.

The boxed font is intended solely for displaying key captions, and all key captions are displayed in uppercase, by convention (see Chapter 11). To use the boxed font in a menu, HELP, or message frame, see Section 7.2.1.

Table 4-1: Synergy Character Set

ROW	b4 b3 b2 b1	0 (b8 b7 b6 b5 = 0 0 0 0)	1 (0 0 0 1)	2 (0 0 1 0)	3 (0 0 1 1)	4 (0 1 0 0)	5 (0 1 0 1)	6 (0 1 1 0)	7 (0 1 1 1)
0	0 0 0 0	0 0 0	20 16 10	SP 40 32 20	0 60 48 30	@ 100 64 40	P 120 80 50	` 140 96 60	p 160 112 70
1	0 0 0 1	1 1 1	21 17 11	! 41 33 21	1 61 49 31	A 101 65 41	Q 121 81 51	a 141 97 61	q 161 113 71
2	0 0 1 0	2 2 2	22 18 12	" 42 34 22	2 62 50 32	B 102 66 42	R 122 82 52	b 142 98 62	r 162 114 72
3	0 0 1 1	3 3 3	23 19 13	# 43 35 23	3 63 51 33	C 103 67 43	S 123 83 53	c 143 99 63	s 163 115 73
4	0 1 0 0	4 4 4	24 20 14	$ 44 36 24	4 64 52 34	D 104 68 44	T 124 84 54	d 144 100 64	t 164 116 74
5	0 1 0 1	5 5 5	25 21 15	% 45 37 25	5 65 53 35	E 105 69 45	U 125 85 55	e 145 101 65	u 165 117 75
6	0 1 1 0	6 6 6	26 22 16	& 46 38 26	6 66 54 36	F 106 70 46	V 126 86 56	f 146 102 66	v 166 118 76
7	0 1 1 1	7 7 7	27 23 17	' 47 39 27	7 67 55 37	G 107 71 47	W 127 87 57	g 147 103 67	w 167 119 77
8	1 0 0 0	10 8 8	30 24 18	(50 40 28	8 70 56 38	H 110 72 48	X 130 88 58	h 150 104 68	x 170 120 78
9	1 0 0 1	11 9 9	31 25 19	) 51 41 29	9 71 57 39	I 111 73 49	Y 131 89 59	i 151 105 69	y 171 121 79
10	1 0 1 0	12 10 A	32 26 1A	* 52 42 2A	: 72 58 3A	J 112 74 4A	Z 132 90 5A	j 152 106 6A	z 172 122 7A
11	1 0 1 1	13 11 B	33 27 1B	+ 53 43 2B	; 73 59 3B	K 113 75 4B	[133 91 5B	k 153 107 6B	{ 173 123 7B
12	1 1 0 0	14 12 C	34 28 1C	, 54 44 2C	< 74 60 3C	L 114 76 4C	\ 134 92 5C	l 154 108 6C	| 174 124 7C
13	1 1 0 1	15 13 D	35 29 1D	- 55 45 2D	= 75 61 3D	M 115 77 4D	] 135 93 5D	m 155 109 6D	} 175 125 7D
14	1 1 1 0	16 14 E	36 30 1E	. 56 46 2E	> 76 62 3E	N 116 78 4E	^ 136 94 5E	n 156 110 6E	~ 176 126 7E
15	1 1 1 1	17 15 F	37 31 1F	/ 57 47 2F	? 77 63 3F	O 117 79 4F	_ 137 95 5F	o 157 111 6F	177 127 7F

KEY

CHARACTER **SP** 40 OCTAL / 32 DECIMAL / 20 HEX

Table 4-1 (continued)

8		9		10		11		12		13		14		15		COLUMN b8 b7 BITS b6 b5 b4 b3 b2 b1	ROW
1 0 0 0		1 0 0 1		1 0 1 0		1 0 1 1		1 1 0 0		1 1 0 1		1 1 1 0		1 1 1 1			
♦	140 96 60	– SCAN 3	160 112 70	▪▪▪	240 160 A0	°	260 176 B0	À	300 192 C0		320 208 D0	à	340 224 E0		360 240 F0	0 0 0 0	0
▒	141 97 61	– SCAN 5	161 113 71	¡	241 161 A1	±	261 177 B1	Á	301 193 C1	Ñ	321 209 D1	á	341 225 E1	ñ	361 241 F1	0 0 0 1	1
␉	142 98 62	– SCAN 7	162 114 72	¢	242 162 A2	2	262 178 B2	Â	302 194 C2	Ò	322 210 D2	â	342 226 E2	ò	362 242 F2	0 0 1 0	2
␌	143 99 63	– SCAN 9	163 115 73	£	243 163 A3	3	263 179 B3	Ã	303 195 C3	Ó	323 211 D3	ã	343 227 E3	ó	363 243 F3	0 0 1 1	3
␍	144 100 64	├	164 116 74		244 164 A4		264 180 B4	Ä	304 196 C4	Ô	324 212 D4	ä	344 228 E4	ô	364 244 F4	0 1 0 0	4
␎	145 101 65	┤	165 117 75	¥	245 165 A5	μ	265 181 B5	Å	305 197 C5	Õ	325 213 D5	å	345 229 E5	õ	365 245 F5	0 1 0 1	5
␏	146 102 66	┴	166 118 76		246 166 A6	¶	266 182 B6	Æ	306 198 C6	Ö	326 214 D6	æ	346 230 E6	ö	366 246 F6	0 1 1 0	6
␐	147 103 67	┬	167 119 77	§	247 167 A7	·	267 183 B7	Ç	307 199 C7	Œ	327 215 D7	ç	347 231 E7	œ	367 247 F7	0 1 1 1	7
	150 104 68	│	170 120 78	¤	250 168 A8		270 184 B8	È	310 200 C8	Ø	330 216 D8	è	350 232 E8	ø	370 248 F8	1 0 0 0	8
	151 105 69	≤	171 121 79	©	251 169 A9	1	271 185 B9	É	311 201 C9	Ù	331 217 D9	é	351 233 E9	ù	371 249 F9	1 0 0 1	9
␊	152 106 6A	≥	172 122 7A	a	252 170 AA	o	272 186 BA	Ê	312 202 CA	Ú	332 218 DA	ê	352 234 EA	ú	372 250 FA	1 0 1 0	10
␋	153 107 6B	╖	173 123 7B	≪	253 171 AB	≫	273 187 BB	Ë	313 203 CB	Û	333 219 DB	ë	353 235 EB	û	373 251 FB	1 0 1 1	11
␌	154 108 6C	╳	174 124 7C		254 172 AC	¼	274 188 BC	Ì	314 204 CC	Ü	334 220 DC	ì	354 236 EC	ü	374 252 FC	1 1 0 0	12
␍	155 109 6D	÷	175 125 7D		255 173 AD	½	275 189 BD	Í	315 205 CD	Ý	335 221 DD	í	355 237 ED	ý	375 253 FD	1 1 0 1	13
␎	156 110 6E	▪	176 126 7E		256 174 AE		276 190 BE	Î	316 206 CE		336 222 DE	î	356 238 EE		376 254 FE	1 1 1 0	14
– SCAN 1	157 111 6F	√	177 127 7F		257 175 AF	¿	277 191 BF	Ï	317 207 CF	ß	337 223 DF	ï	357 239 EF		377 255 FF	1 1 1 1	15

KEY

CHARACTER Æ

306 OCTAL
198 DECIMAL
C6 HEX

4.4 IMPOSED DEVICE SPACE

When your application receives control from Synergy, the GIDIS Imposed Device Space has been set to the writable area of your front window. This prevents you from writing outside your window, since the clipping region is the same as the IDS.

You can issue the GIDIS SET_CLIPPING_REGION instruction, however, in order to modify the clipping region. You should be careful to keep the new clipping region within your window's writable area. GIDIS does not prevent you from setting the clipping region beyond the IDS boundaries. If you do this, it is then possible to issue GIDIS instructions that write outside your window's writable area.

4.5 INTERTASK COMMUNICATION METHOD

If your application consists of more than one task, you should be aware of the task operations that occur within Synergy. You should not interfere with these operations.

If your application desires to receive data through ASTs, you will need to use a special interface described below.

4.5.1 Synergy Task Communication

The applications and the window manager communicate only with the window server. Communication is totally synchronous and can be likened to an interprocess coroutine call. This is implemented through a VARIABLE SEND DATA/VARIABLE RECEIVE DATA OR STOP pair of directives in each direction, to the window server (input parameters), and from the window server (output parameters). Before the packet is sent, the window server is stopped (a state in P/OS in which a task does not execute, and does not compete for memory). Once the packet is sent, the caller will stop, and the window server is allowed to execute. On the return path, the roles are reversed.

VARIABLE SEND DATA restricts data packets to be smaller than 512 bytes. This is too small for many of the parameter packets. Therefore, multiple VARIABLE SEND DATA directives are used to implement packets of up to 2048 bytes. Additional SEND directives are used only if the packet is larger than 512 bytes.

The interprocess calls are accomplished by subroutines in the Synergy Interface Library that are linked into each application task that uses Synergy services. These modules, including a 512-byte buffer and the Synergy character-passing buffer, occupy about 2000 decimal bytes of your task's address space.

You can overlay all or part of this area, by calling the modules out explicitly in the ODL file that is used to link your task. The module name is the same as the global symbol name for each service.

When an application receives the F5 key as input, it calls the Suspend service (WIINT). The interprocess call that follows does not return until some later time. Instead of returning to the calling task, the server "returns" to the window manager. The window manager eventually calls the TRANSFER CONTROL service in the window server, passing the task that is to be activated. Instead of the window server returning control directly to the window manager, it "returns" to the task that is to be activated.

NOTE

When you call a Synergy service, the interface routine stops the task while it waits to receive the data packet that is returned by the window server. A stop is not a legal operation if the task is at AST state. Therefore, you must not call Synergy services from AST state.

4.5.2 Receiving Data Packets

If your application task executes Receive Data directives in order to communicate with other tasks, your use of this facility can conflict with Synergy's use.

No conflict arises if you execute your Receive Data directives from user state (as opposed to AST state). Since Synergy's use of the Send and Receive Data directives is strictly synchronous, the Synergy interface routine stops your task until the window server sends back the result of the service call. Furthermore, the interface routine specifies that it should receive only those packets that are sent by the Synergy window server.

However, if your task is using ASTs (perhaps an unsolicited-input-character AST), and you execute a Receive Data directive while at AST state, you could potentially receive a data packet that is intended for the Synergy interface routine. You can guard against this by always specifying the sender task

name from which you are expecting data in any Receive Data directive done from AST state.

4.6 CALL INTERFACE TO SYNERGY SERVICES

The Synergy services are implemented as interface routines in the Synergy Interface Library. The library is referenced in the .ODL file used to build the Synergy application.

In the following sections, each service is described individually with the global symbol that is defined in the Synergy Interface Library. The parameters that are passed on the call are listed in the expected order, with an indication of the data type and whether the parameter's value is supplied as input to the call or output from the call, or both input and output. Examples are given for most calls.

Synergy interface routines conform to the calling conventions for other P/OS library routines (the PDP-11 R5 sequence), with the additional feature that no registers are modified by the call.

Notice that the interface routines lie between your application code and the window server task. The interface routines pack your input parameters into a data packet. If you supply a null entry (-1 in the pointer of the parameter list) for an input parameter, the interface routine supplies a 0 or null string in the packet. This means the window server does not see a missing integer parameter; it sees a 0-valued parameter.

The window server task sends back all the output parameters in a data packet. The interface routine unpacks the returned values into the output parameters that you requested. If you supply a null entry for an output parameter, the interface routine just ignores the value returned in the data packet.

The interface routines and the window server do very little checking on the validity of input parameters. It is possible to pass faulty input, or no input, and get back a status value that indicates the call was successful. If your application is calculating parameter values dynamically, you may want to build in your own checking code to ensure that the calculated values are acceptable before using them in the service call. Such code could be made conditional, so that it can be easily removed when debugging is completed.

The interface routines pass the call to the window server task, and wait for its return (see Section 4.5.1).

4.6.1 Parameters

Words are 16-bit integers, unless otherwise noted.

Boolean values are stored as one-word integers. The value "true" is represented by the integer -1, and the value "false" is represented by the integer 0. No other values should be used.

Strings are sequences of bytes whose values may include any graphic character, including multinational characters. Unless specifically stated, string parameters should not include control characters (less than ASCII space) or escape sequences. Most strings are straight text, and include no formatting information. In particular, horizontal tab has no meaning in a window.

Make sure that paramters which are unused or are documented as reserved are passed zero values in the call.

All services pass back a status code in the first parameter. The status parameter is a two-word integer array that indicates the results of the requested operation. The first word indicates the general result, while the second word may contain additional information. See Table 4-2.

4.6.2 WICAL -- Call Window Service

Unlike all other Synergy calls, the following routine does not use the PDP-11 R5 Calling Sequence format. Rather, it expects its two parameters to be pushed onto the stack by the caller (followed by the normal JSR PC, WICAL).

Routine address	1 word	(input)
Parameter block address	1 word	(input)

This call is an optional means of indirectly calling Synergy services. This routine saves R5, then loads R5 with the second parameter (parameter block address). Then WICAL (JSR) calls to the routine specified by the routine address. Upon return from the called routine, WICAL restores the original value of R5, cleans the two parameters off of the stack, and then returns to the application. (In fact WICAL may be used to call any PDP-11 R5 sequence routine, not just Synergy services.)

WICAL is useful in two cases:

- When the parameter block is constructed at run-time (rather than statically at compile-time by the language compiler) so that you cannot call the routine directly from the source code using a normal CALL type statement.

For example, you can allocate the parameter block as an array of integers (do not forget to include the parameter count word, as defined by the R5 Calling Sequence Standard); assign the addresses of the individual parameters into elements of the array; then pass the routine you want to call and the address of the array to WICAL.

- When calling Synergy routines with static argument lists from PRO/Pascal, you may find that large parameter lists exceed the limits of the compiler. In this case, you can allocate and construct the parameter block yourself in Pascal, and then use WICAL. For example:

```
PROCEDURE DFLOW; SEQ11;  { Routine with lots of params }

PROCEDURE WICAL( PROCEDURE SEQ11Procedure;
                 VAR ParamBlock: [Unsafe] Integer ); EXTERNAL;

PROCEDURE ThisCallsDFLOW;

CONST
  ParamCount = 100;
VAR
  FlowPB: ARRAY [ 0..ParamCount ] OF Integer;
  StatusBlock: ARRAY [ 1..2 ] OF Integer;

BEGIN { of procedure ThisCallsDFLOW }
  FlowPB[ 0 ] := ParamCount;
  FlowPB[ 1 ] := IAddress( StatusBlock );  { 1st parameter }
  .
  .

  WICAL( DFLOW, FlowPB );  { Call the Synergy routine }
  .
  .
END;  { of procedure ThisCallsDFLOW }
```

Table 4-2: Returned Status Values

WORD 1 VALUE	WORD 1 MEANING	WORD 2 MEANING
1	Success	Not specified
-1	Directive error (in window server)	DSW (Directive Status Word)
-2	RMS error (in window server)	RMS I/O error

WORD 1 VALUE	WORD 1 MEANING	WORD 2 MEANING
-3	Bad value (in application)	Parameter number in error
-4	Receive error (in application)	DSW (Directive Status Word)
-5	Send error (in application)	DSW (Directive Status Word)
-6	Interpreter error	0 - Unknown 1 - Invalid function 2 - Not implemented
-7	Protocol error	0 - Unknown 1 - Invalid function 2 - Not implemented
-8	Bad CurrentValue for class 2 option in set-up menu call	undefined
-9	Frame type incompatible with menu call	Frame found
-10	Network error	DECnet error codes
-11	No more names to return from Old File selections	Undefined
-12	Mismatch of option classes in set-up menu call	Undefined
-13	String size too large in set-up menu call	Undefined
-14	Wrong number of options in set-up menu call	Undefined
-16	WIRMS message too big	Undefined

WORD 1 VALUE	WORD 1 MEANING	WORD 2 MEANING
-17	Window error	1. Too many windows 2. Invalid position 3. Invalid size 4. No active window 5. Window has no title 6. Window must be front window 7. Invalid window ID 8. Invalid operation
-18	Block I/O error	1. Cannot create file 2. End of file 3. Device full 4. No such file 5. File not open on specified channel 6. Memory unavailable 7. No channel is available 8. Invalid file specification 9. Invalid channel number 10. System directive error 11. File is locked 12. Illegal operation 13. Not at end of file 14. Privilege violation 15. Line too long 16. File already exists 17. Not a sequential file 18. Invalid record address 19. Invalid record format 20. System I/O error
-19	Menu primitive error	1. No menu or string editing window exists 2. Front window is not a menu window 3. Front window is not a string editing window 4. Too many menus and string editing windows 5. Too many headers 6. Header too wide 7. Window too wide 8. Too many entries 9. Entry too wide 10. Nonprinting character 11. Invalid entry position

WORD 1 VALUE	WORD 1 MEANING	WORD 2 MEANING
-20	Terminal error	1. Buffer length invalid 2. Initial length invalid 3. Initial position invalid 4. Directive failure
-21	Raster error	1. Bad parameter value 2. No rasters available 3. Insufficient file space
-22	Internal error	1. Stack pointer corrupted 2. Packet protocol 3. Directive failed 4. Bad packet type 5. New task interrupted packet stream 6. New function interrupted packet stream 7. Invalid length for task context block
-23	Memory error	1. Internal error 2. Attempt to DISPOSE with an invalid pointer 3. NEW received a negative size 4. Zero or negative size block 5. Memory not available 6. Free memory list has invalid pointer 7. Memory block larger than 1024 8. Free memory list has loop 9. Memory block overruns end of pool

4.7 PASSING TYPE-AHEAD TO SYNERGY ROUTINES

The terminal subsystem automatically collects keyboard input in a *type-ahead buffer*. Characters are released from this buffer in response to QIOs that are executed by the application code.

4.7.1 MGTCB - Expand Call-Back Code

Status	2 words (output)

Before the service is called, the application must ensure that the character-passing buffer is in the correct format. Specifically, the call-back code must be removed from the buffer, any additional characters left-justified, and the buffer length field set to the correct value.

NOTE

> Do not assume that the character-passing buffer is empty beyond the call-back code. There is **always** at least one character in the buffer beyond the call-back code, and if there are applications running that use an AST routine to read the keyboard, they may be appending characters to the end of the character-passing buffer while the call-back code is in the buffer.

The application in Appendix A uses the character-passing buffer (see Pages A-10 and A-47).

4.8 FILE USAGE

Your application can be suspended by the user while other applications are run. This puts a demand on system resources. The following suggestions are offered:

- Before you call the Suspend service, you should free as many system resources as possible. One way to do this is to close data files. You can close the data file before the WIINT call and reopen it on return from the WIINT call. (Notice that the window server automatically closes the frame file for you and then reopens it before returning control to your application.)

- If you close files before suspending, you may want to do a fast reopen of the file using the device and file identifier in the NAM block that was supplied on the first open, rather than doing a reopen with the file specification. However, you should be aware that if the file is on another node of a network, the device and file identifiers are meaningless and the open will fail.

- Your application is more susceptible to an abnormal termination while it is suspended, since the user may forget that your application is running and may turn off the computer. If you leave a file open during the suspend, your application should anticipate a locked file error return when it tries to open the file on start-up. Alternatively, you may want to open the file with a request that RMS not lock the file if it is closed abnormally.

- Open your data files with the minimum required access in order to reduce the resource requirements.

- When opening data files in the user's default directory, address them with the pseudo device name, SY:[], rather than making explicit reference to a device name.

4.9 SPECIFYING KEY CODES

Many of the menu service routines read the keyboard (through the character-passing buffer, of course) and return the keystroke that terminates the service (often the DO key) to the application. Many of the terminating keystrokes are multi-byte CSI sequences, which would require returning a variable-length string of bytes as the terminating code. Instead, the window server returns a key code as a 16-bit integer.

Normal keys, like "A" or space or "?", are represented by their ASCII codes. For example, "A" is represented by decimal 65. Multinational keys are represented by the appropriate values -- the copyright key is represented by decimal 169. Any key value less than decimal 255 is a printable, or graphic, character.

Control keys (normally represented by decimal 0 to 31) are specified by decimal 256 plus their normal value. For example, the normal value of CTRL/C is 3, but CTRL/C is represented in this scheme by 256 + 3, or decimal 259.

Invalid keys (invalid escape sequences) are specified by decimal 1024.

Remember that the HOLD SCREEN and PRINT SCREEN keys are never accessible to applications in P/OS.

The other keys -- function keys, ARROW keys, keypad keys -- are represented by values between 512 and 1024 (see Table 4-3). For a discussion of the use of the keys in termination key lists, see Section 8.2.5.

The application in Appendix A defines the key codes (see Page A-45).

Table 4-3: Key Encodings

KEY CAPTION	ENCODING	KEY CAPTION	ENCODING
BREAK	512 + 13 = 525	F5	512 + 15 = 527
SETUP	512 + 14 = 526		
INTERRUPT	512 + 17 = 529	MAIN SCREEN	512 + 20 = 532
RESUME	512 + 18 = 530	EXIT	512 + 21 = 533
CANCEL	512 + 19 = 531		
F11	512 + 23 = 535	F13	512 + 25 = 537
F12	512 + 24 = 536	ADDTNL OPTIONS	512 + 26 = 538
HELP	512 + 28 = 540	DO	512 + 29 = 541
F17	512 + 31 = 543	F19	512 + 33 = 545
F18	512 + 32 = 544	F20	512 + 34 = 546
PF1	512 + 35 = 547	PF3	512 + 37 = 549
PF2	512 + 36 = 548	PF4	512 + 38 = 550
FIND	512 + 1 = 513	SELECT	512 + 4 = 516
INSERT	512 + 2 = 514	PREV SCREEN	512 + 5 = 517
REMOVE	512 + 3 = 515	NEXT SCREEN	512 + 6 = 518
UP ARROW	512 + 39 = 551	RIGHT ARROW	512 + 41 = 553
DOWN ARROW	512 + 40 = 552	LEFT ARROW	512 + 42 = 554
Keypad ,	512 + 43 = 555	Keypad 3	512 + 50 = 562
Keypad -	512 + 44 = 556	Keypad 4	512 + 51 = 563
Keypad .	512 + 45 = 557	Keypad 5	512 + 52 = 564
Keypad Enter	512 + 46 = 558	Keypad 6	512 + 53 = 565
Keypad 0	512 + 47 = 559	Keypad 7	512 + 54 = 566
Keypad 1	512 + 48 = 560	Keypad 8	512 + 55 = 567
Keypad 2	512 + 49 = 561	Keypad 9	512 + 56 = 568
<X] (delete)	512 + 57 = 569		

4.10 RESTRICTIONS

Synergy does not fully protect applications from one another, and cannot protect itself from abuse by applications. In order for Synergy and its applications to all work properly, each application must abide by certain restrictions.

- An application must not use text-mode QIOs to the terminal for screen output, and it must be careful not to use any system service that would issue such output QIOs (such as standard P/OS menu services). The only exception is the use of instructions that change keyboard characteristics. These may be used, but the application must reset their state on return from each Suspend (WIINT) call, since the keyboard state will have been reset by the window server. If the application is using a VT window, this reset does not occur.

- An application may change terminal driver characteristics (by sending a SF.SMC QIO to the terminal susbsystem), but it must reestablish the desired characteristics on return from each WIINT call, since the terminal driver state will have been reset by the window server.

- An application may attach the terminal (to do unsolicited input character ASTs), but must detach the terminal before calling any window service.

- An application must check **all** keyboard input and call the Suspend service, whenever the F5 key is pressed. The application may or may not choose to process any characters that precede the F5 key, but should not wait for more characters before suspending. An application in the middle of some noninteractive operation, such as a database update, may choose to complete the operation before suspending, or may choose to abort the operation. Since the user's next action is not predictable, application files and other context should be in known states before suspending.

- An application should be able to refresh its window after a Suspend service if it allows the window size to be changed by the user and if the display would not look right in the newly sized window.

- All stackable windows are destroyed by the Suspend service.

- Exercise care in changing the color map, since that action changes the colors in other windows as well. Since the color map is not saved, and is reset whenever the application is suspended, it remains the application's responsibility to reestablish its own color map when the application resumes after each such suspension.

RESTRICTIONS

- An application should not change the GIDIS imposed device space. The clipping region and the GOS units can be changed. The window server always sets these for a newly created window, and after a return from a suspension, the window server reestablishes these values for the front window.

- An application may use GIDIS named fonts, and may define additional named fonts beyond those provided by Synergy. An application can define implicitly named fonts (those defined at run time, one character at a time), but they will have to be reestablished after a call to any Synergy service.

CHAPTER 5

CHAPTER 5

TASK CONTROL SERVICES

5.1 TASK CONTROL SERVICES

These operations are used to initialize the interface between the application task and the Synergy services, and to pass control and information back to the services. They are described here in alphabetical order.

5.1.1 WIDON - Application Done

Status	2 words	(output)
ContextBlockLength	word	(input)
ContextBlock()	n bytes	(input)

This service is called when the application is about to exit. The application can pass up to 32 bytes of context data, which the window server saves on its behalf. The context block is returned by the Initialize service (WIINI) the next time the application runs. The window server does not attempt to apply any meaning to the context block. Each application can use the context block in its own way. (Do not confuse this data with the window descriptor block.) The application in Appendix A uses the context block (see Pages A-6, A-14, and A-15).

Like all other services, this service returns to the application. It does not cause the application to exit.

Once the Done service has been called, the application must not make any additional calls on the window server and must not do any more terminal I/O.

The application in Appendix A uses the WIDON service (see Pages A-15 and A-48).

5.1.2 WIINI - Application Initialization

Status	2 words	(output)
ExpectVersion	word	(input)
ActualVersion	word	(output)
ContextBlockLength	word	(input and output)
ContextBlock()	n bytes	(output)
ScreenWidth	word	(output)
ScreenHeight	word	(output)
CharacterWidth	word	(output)
CharacterHeight	word	(output)
PixelWidth	word	(output)
PixelHeight	word	(output)
Color	word	(output)

This service initiates communication between the application and Synergy.

NOTE

> This service can be called only once per execution of the application, and must be called before any other service or any terminal I/O.

ExpectVersion is the version of the window server expected by the application; it should be set to 2 with this release. ActualVersion is the actual version of the window server.

ContextBlock is a block of up to 32 bytes. An application uses this area to retrieve information stored by the Done service the last time the application executed. The length of the context block is both input and output. As input it specifies the maximum number of bytes to be returned. As output it specifies the number of bytes actually returned. If the returned length is 0, it signifies that this is the first time the application was run. (It is recommended that this area be used to save the X and Y coordinates of the windowframe and the width and height of the writable area of the window. This information can then be used to create a window of the position and size the user last wanted.)

The widths and heights are the sizes of the screen, of the default character, and of the actual hardware pixel in the Synergy coordinate system (GOS units). The values can be used as a basis for graphics calculation if you want to write an application that is independent of the current hardware.

The Color parameter is boolean; it is true if color (monitor and three planes of video bitmap) is being used; it is false if color is not being used (only one plane in use). The user selects the use of color from the Synergy Set-Up Menu. This output parameter

is not simply an indication of whether the hardware is present to do color images on the screen; it indicates that the hardware is present **and** that the user wants to use it. (To find out whether the color hardware is present, see the EBO parameter to the WISYP call.) Applications should not create windows that use color (see Section 6.1.5) unless the Color parameter returned by the Initialize service is true, or unless the application is creating a window with the VT attribute.

The application in Appendix A uses the WIINI service (see Pages A-14 and A-48).

5.1.3 WIINT - Suspend the Application

Status	2 words	(output)
WhyReturn	word	(output)
WindowID	word	(output)
Width of writable area	word	(output)
Height of writable area	word	(output)

This service suspends execution of the application and gives control to the window manager. This service must be called when the application sees the F5 key in its own keyboard input, and when the application sees the F5 key returned by any service call.

If the application has created any stackable windows, they must be destroyed before this service is called.

The window manager can return control from the Suspend service to the application under two different conditions:

- When the user tells the window manager to resume execution of the application, the window manager returns from the Suspend service with the WhyReturn parameter set to 0. This tells the application to continue its execution. If the application has created only one window, it can ignore the WindowID, Width and Height parameters. If the application has created more than one window, it can use the WindowID parameter that is returned to learn which of its windows is in front.

- If the application has allowed its window(s) to be changed in size, and the user has requested a size change, the window manager returns immediately after completing the size change action with the WhyReturn parameter set to 1. This tells the application to adjust the window whose size has been changed, but **does not give the application permission to continue execution.** The application is required to adjust the window

and **immediately** call the Suspend service again, since the user still thinks he is manipulating the windows with the window manager.

The application is told which window was changed (WindowID) and the new dimensions (Width and Height). When a window size changes, the contents of the window depend on the window attribute "clear on change." If this attribute is true, the window manager blanks the entire writable area of the window, and the application must refresh it. If "clear on change" is false, the window manager merely retains whatever was in the writable area before the size changed. When a window is made larger, the new portion (bottom or right side) is cleared (with white). When a window is made smaller, the writable area is restored with the previous contents (upper left corner). Data outside the new writable area is lost.

When the user moves the front window to a new location on the screen, no indication is given to the application. Therefore, your application should not assume that the screen position of any window is unchanged over a Suspend service call. If the application depends on the screen position of its window, you must call the Get Window Parameters service (WIGEW) on return from WIINT to update the window descriptor block with the current screen location.

The application in Appendix A uses the WIINT service (see Pages A-20 and A-48).

5.1.4 WISYP - Get System Parameters

Status	2 words	(output)
ExpectVersion	word	(input)
ActualVersion	word	(output)
ScreenWidth	word	(output)
ScreenHeight	word	(output)
CharacterWidth	word	(output)
CharacterHeight	word	(output)
PixelWidth	word	(output)
PixelHeight	word	(output)
Color	word	(output)
EBO	word	(output)
GuideMode	word	(output)

This service supplies the information normally returned by the Initialize service (WIINI), but without the implication that the application is just starting. (See Section 5.1.2.) In addition, two optional extra parameters exist that are not available with the WIINI call.

The EBO parameter is boolean; it is true if the Extended Bitmap Option Module (three planes of video bitmap) is present in the system.

The GuideMode parameter is boolean; it is true if the Guide Mode setting on the Synergy Setup Menu is true. If you wanted your application to have facilities for doing things differently for new users than for experienced users, you might use this flag to determine whether to treat the user as new or experienced.

5.2 SYNERGY MESSAGE BOARD

The Synergy Message Board calls are intended for applications that wish to send messages to the Synergy Message Board. For example, you might want a mail program to use the Message Board services to notify users when they have new mail messages. To view any messages, users must return to the Synergy Main Menu.

The Synergy Message Board is a very limited resource and should be used with restraint. The message board can contain a maximum of 5 (five) messages at one time. Each message can be up to 40 characters in length.

If the message board receives more than five messages at one time, the oldest message is automatically deleted.

5.2.1 MGMSG - Send Message to Synergy Message Board

Status	2 words (output)
GroupID	word (input)
MessageLength	word (input)
MessageText	n bytes (input)

The MGMSG call sends one message to the Synergy Message Board. Message length must be less than 40 characters. The Group ID is a 16-bit integer choosen by the caller to indicate a group or category for the message.

In practice, each application should use the same group ID for all of its messages. The caller should also make sure that the group ID is significantly random so that it does not conflict with a group ID of another application.

5.2.2 MGDMS - Delete Message from Message Board

Status	2 words (output)
GroupID	word

The MGDMS call deletes all messages with the same ID as the one specified in the Group ID parameter. This call allows an application to delete all of its messages (or specific group of messages) with a single call.

CHAPTER 6

Table 6-1: Window Descriptor Block

WORD	MEANING
1	Window ID
2	X coordinate of upper left corner of windowframe
3	Y coordinate of upper left corner of windowframe
4	Width of the writable area of the window
5	Height of the writable area of the window
6	Flag Word (1 = true, 0 = false) Bit 0 Stackable Bit 1 Titled Bit 2 Hidden Bit 3 Color Bit 4 White border Bit 5 Clear on change size Bit 6 (Reserved, must be 0) Bit 7 VT Bit 8 Invisible Bit 9 (Reserved, must be 0) Bit 10 3 planes Bit 11 to 15 (Reserved, must be 0)
7	Minimum width of writable area of the window
8	Minimum height of writable area of the window
9	Maximum width of writable area of the window
10	Maximum height of the writable area of the window
11	X offset from windowframe to writable area of window
12	Y offset from windowframe to writable area of window
13	Overall width of the window
14	Overall height of the window
15	Owner task, word 1 of RAD50 name
16	Owner task, word 2 of RAD50 name

6.1.2 Specifying Window Coordinates

When a window is created, its position can be specified in coordinates expressed in GOS units. The position can also be specified with *pseudo coordinates*, that indicate a general location on the screen. The window server interprets the pseudo coordinates and determines the exact positioning. By using pseudo coordinates, you can avoid a great deal of computation in your application.

Positions in both X (horizontal) and Y (vertical) may be specified using the coordinates shown in Table 6-2.

Table 6-2: Window Coordinates

VALUE	MEANING
-32767	Don't care
-32766	Off window
-32765	Screen minimum
-32764	Screen maximum
-32763	Screen centered
-32762	Window minimum
-32761	Window maximum
-32760	Window centered
-4095..-1	Window-relative position (The upper left-hand corner of a window can be positioned to the left or right of the upper left corner of an existing window. When positioning to the left of the existing window, negative coordinates are used, so a position 300 pixels to the left is -300. In order to encode window-relative positions, subtract 2048 from the relative coordinate. Therefore -300 pixels to the left is encoded as -2348, and 2047 pixels to the right is encoded as -1. -2048 is the upper left corner of the window.)
0..2015	Screen (or absolute) horizontal position (1008 is middle of screen)
0..1199	Screen (or absolute) vertical position (600 is middle of screen)

Any value not specified in Table 6-2 is an invalid position.

When there is no window on the screen, the window-oriented positions are handled as though a window existed that is exactly the size of the screen.

When a window is created or moved and its position is specified in such a way as to make part of the window fall off the screen, the window server adjusts the coordinates automatically to bring all of the window onto the screen. The coordinates that you specify and the size of the window must be valid before this adjustment can take place, however. The window server never shrinks or truncates an oversized window, and it never corrects an invalid coordinate.

For example, a horizontal position of 2016 is not a valid starting position. However, a horizontal position of 2015 is valid and is equivalent to specifying -32764. Either 2015 or -32764 guarantees that the window will be on the far right of the screen.

(Notice that video hardware may have larger screens in the future. On a wider screen, -32764 will still mean "right side," but 2015 may not have that meaning.)

6.1.3 Specifying Window IDs

Each window has a unique ID that is assigned by the window server when the window is created, and is returned to the application. The application uses the window ID to identify the window when an operation is requested. Four special IDs can be used as input parameters to some of the operations.

An ID with the value -1 can be used in the Select operation (WISLW) to get access to the entire screen. This is not strictly a window; the technique is used primarily by the window manager. Any application using this pseudo window must "undo" its modifications to the screen and thus restore the screen to its state before the application used the pseudo window. (The window manager uses this technique when it draws window corners and blinking bars during window operations. It draws them in complement mode, then draws them again in complement mode to erase them.)

An ID with a value of -2 can be used as input to any of the window operations as a reference to the front window. An application can always refer to its front window with an ID of -2, instead of using the window ID that is returned in the window descriptor block. An application that creates more than one window must be certain that it knows which window is the front window, however. On return from a suspension, it can check the WindowID parameter that is returned to see that it matches the desired window's ID. If it does not match, the application can call the Select Window service (WISLW) using the desired window's ID to bring that window to the front.

An ID with a value of -3 is used to indicate the next window. This is accepted as input by the Select (WISLW) operation. This is a window server operation, and should not be used by applications, lest they obtain a window belonging to another application.

An ID of -4 is used to indicate the rear window. This is accepted as input by the Select (WISLW) operation. This is a window server operation, and should not be used by applications, lest they obtain a window belonging to another application.

6.1.4 WICHW - Change the Size and Position of a Window

Status	2 words	(output)
DescriptorLength	word	(input)
WindowDescriptor()	n bytes	(input)

This service is used to change the size and position of the front window. It is an error to attempt to change a stackable window or a window that is not the front window. When a window changes size, any text in the title is centered in the new title area, and is truncated if necessary (on the right side only).

The input from the window descriptor block is the window ID, X and Y coordinates of the windowframe, and the width and height of the writable part of the window.

6.1.5 WICRW - Create a Window

Status	2 words	(output)
DescriptorLength	word	(input)
WindowDescriptor()	n bytes	(input and output)

This service saves the front window (if any), and creates a new window. The new window becomes the front window. It is an error to create a nonstackable window when the front window is stackable.

The input from the window descriptor block is the X and Y coordinates of the windowframe, of the window, and the window attributes: stackable, titled, color, white border, clear on change, VT, invisible, and three-plane. The output is the window ID.

Creating a window of width and height equal to zero results in a full screen window with no white border or windowframe.

If you request a color window but the user has not told the Synergy Window Manager to permit color windows, your request for a color window is ignored and you get a monochrome window. This means that the window server will save only one plane of the video bitmap.

You must be certain that the window server returned a TRUE setting for the Color parameter on the Initialize (WIINI) call before you assume that your request for a color window was granted. Do not write to planes two or three of the video bitmap unless all three planes are being saved.

This caution does not apply to windows with the VT attribute, since these windows are always considered to be three-plane windows.

Also, the three-plane attribute will always give you a three-plane window, so long as the EBO option is present on the system.

Notice that you cannot specify the hidden attribute in the WICRW call. You must first create the window and then ask that it be hidden by calling the WIHDW service.

Notice also that only the window ID field of the window descriptor block is returned by this service. To update the other entries of the window descriptor block, you must call the Get Window Parameters (WIGEW) service.

When a window is created, the maximum and mininum sizes default to the actual window size, so the size of the newly created window cannot be changed by the user. You must call the Set Window Parameters (WISWP) service to change the maximum and minimum sizes if you want to allow the user to modify the window's size.

If you request a title, but the window's writable area is too large to permit a title line on the screen, the title is omitted.

The application in Appendix A uses the WICRW service (see Pages A-16, A-26, A-29, A-33, and A-48).

6.1.6 WIDSW - Destroy a Window

Status	2 words (output)
WindowID	word (input)

When a window is no longer needed, it must be destroyed. Destroying a window removes it from the screen, uncovering any windows occluded by it. All storage allocated to the window is freed.

The window being destroyed must be the front window. The next window becomes the front window, whether or not it is owned by the same application. If the application has another window, it must call the Select Window (WISLW) service to ensure that its next GIDIS output actually goes to its own window.

The application in Appendix A uses the WIDSW service (see Pages A-15, A-28, A-31, A-36, and A-48).

6.1.7 WIERW - Display Error Window

```
Status                    2 words (output)
  Up to 5 strings:
     TextLength           word    (input)
     Text()               n bytes (input)
```

The window server maintains a hidden window, called the *error window*. It is used only when an application calls the WIERW service (typically, when the application cannot create a window of its own due to a lack of resources). The Display Error Window service allows the application to display information to the user. Since the error window is created at Synergy start-up, it is always available, even when the raster file is full.

The application supplies up to five lines of text. The maximum length of each text line is 40 printing characters, although you can include additional characters to control renditions (see Section 7.2.1). The window server selects the error window and displays the text in it. Control returns to the application only when the user presses RESUME.

NOTE

> The WIERW service can be called at any time, even after the failure of a WIINI call, or before the WIINI call is attempted. The WIERW service destroys any windows that the application has already created. The WIERW service can be followed only by a WIDON call, and then an exit from the application.

The application in Appendix A uses the WIERW service (see Pages A-9 and A-50).

6.1.8 WIEWT - End Wait Message

```
Status                    2 words (output)
```

This service erases the message and clock icon created by the WISWT or WIXSWT service. The window manager redisplays the previous title -- centered or right-truncated -- in the title line of the front window.

6.1.9 WIGEW - Get Window Parameters

Status	2 words	(output)
DescriptorLength	word	(input)
WindowDescriptor()	n bytes	(input and output)

This service is used to retrieve a full description of a window.

The input from the window descriptor block is the window ID. The window ID may be set to -2, in which case the descriptor block is filled with information about the front window. The entire descriptor block is output, including the actual window ID of the window being described.

Notice that five services use the window descriptor block as a parameter, and may return output information in the window descriptor block. Only the Get Window Parameters service updates all fields of the window descriptor block, however. Before using values in the window descriptor block (such as location or dimension) your application may need to call the WIGEW service.

The application in Appendix A uses the WIGEW service (see Pages A-15, A-20, and A-48).

6.1.10 WIHDW - Hide a Window

Status	2 words	(output)
WindowID	word	(input)

This service is used to remove a window from the display. It is not destroyed -- Select a window (WISLW) will bring it back -- but it is not visible. A hidden window is not in the stack of windows; thus it cannot be selected by the user.

The specified window must be the front window. The next window becomes the front window, whether or not it belongs to your application.

An error occurs when an attempt is made to hide a stackable window.

6.1.11 WIIDA - ID of a Window at a Point

Status	2 words	(output)
X	word	(input)
Y	word	(input)
Present	word	(output)
WindowID	word	(output)

This service is used to determine whether a visible window exists at point X,Y on the screen. If it does, Present is returned as true and WindowID contains the window ID of the visible window at that point. If no visible window exists there, Present is returned as false.

6.1.12 WIPOW - Change Position of a Window

Status	2 words	(output)
DescriptorLength	word	(input)
WindowDescriptor()	n bytes	(input and output)

This service is used to move a window to a new location. The window must be the front window.

The input from the window descriptor block is the window ID, and the X and Y coordinates of the windowframe. The X and Y coordinates may be pseudo values (such as -32763 for screen center). Be sure to call the Get Window Parameters (WIGEW) service if you want the window descriptor block updated after the WIPOW call.

6.1.13 WIPSW - Push a Window

Status	2 words	(output)
WindowID	word	(input)

This service moves the specified window to the rear of the display (behind all other windows).

When the specified window is the front window, the next window becomes the front window, whether or not it belongs to your application. The highlighting of titles on the former front window and the new front window is adjusted automatically.

An error occurs when an attempt is made to push a stackable window, since stackable windows cannot be reordered.

6.1.14 WISLW - Select a Window

Status	2 words	(output)
WindowID	word	(input and output)

This service is used to bring a specified window to the front of the display, whether or not it is a hidden window. The current front window is saved (as the second window in the stack) and the specified window becomes the front window.

GIDIS output is directed only to the front window. If your application uses more than one window, use this service to ensure that the correct window is in front before you issue GIDIS instructions.

An error occurs when an attempt is made to select a window when the current front window is stackable, since stackable windows cannot be reordered or covered by a nonstackable window.

This routine accepts two special pseudo values in WindowID. Value -3 refers to the second window, the window immediately behind the front window. Value -4 refers to the rear window. If two windows exist on the screen, both -3 and -4 select the rear one. If only one window exists, both -3 and -4 select it. If no window exists, both -3 and -4 select the pseudo-window that is the entire screen.

The highlighting of titles on the former front window and the new front window is adjusted automatically.

6.1.15 WISWP - Set Window Parameters

Status	2 words	(output)
DescriptorLength	word	(input)
WindowDescriptor()	n bytes	(input)

This service sets the maximum and minimum width and height of the window. These values control how much the user may change the size of a window. When a window is created (with WICRW) these values are defaulted to the actual window size. So, call the Set Window Parameters service if you want to allow the user to be able to modify the window size.

This routine will not set a limit that is more restrictive than the current size. An attempt to do so will set the appropriate limit to the current size.

The input from the descriptor block is the window ID, minimum and maximum widths, minimum and maximum heights and the window attribute, "clear on change" and "invisible." There is no output from this service.

The application in Appendix A uses a window descriptor block (see Pages A-16 and A-48).

6.1.16 WISWT - Start Wait Message

Status	2 words (output)
MessageLength	word (input)
MessageText()	n bytes (input)

This service displays a clock icon and a message in the title area of the front window. The current title is temporarily erased. An error occurs if the front window does not have a title. This service is used to inform the user that a time-consuming operation is in progress. The clock icon is automatically added to the front of the message, so you should not include it in the message text.

You must call the End Wait Message (WIEWT) service to restore the original title of the window when the time-consuming operation is completed.

If you permit the user to adjust the window size, plan your wait message so that it makes sense in the narrowest window that you allow.

See Section 6.1.18 for a variation on this service.

6.1.17 WITTL - Change Title of Front Window

Status	2 words (output)
TitleLength	word (input)
TitleText()	n bytes (input)

This service erases the current title of the front window and displays the specified title in its place. Do not call this service to change the title of a window that is displaying the wait message; use the End Wait Message service, WIEWT.

The application in Appendix A uses the WITTL service (see Pages A-16 and A-48).

6.1.18 WIXSWT - Start Wait with Message Frame

Status	2 words	(output)
FrameID	word	(input)

This service combines the action of two other services, WIRMS (Read Message Frame) and WISWT (Start Wait Message). The message frame specified by FrameID is read into the window server's buffer, and its first line is taken as the wait message. The wait message is displayed with the clock icon in the title area of the front window. The message text is not made available to the application. The clock icon is added automatically to the front of the message, so you should not include it in the message frame.

CHAPTER 7

CHAPTER 7

MENU SERVICES

This chapter includes descriptions of all services that are referred to generally as menu services. This includes:

- Frame file services
- High-level menu services
- Filename services
- Directory name services
- Primitive menu and string editing services

7.1 FRAME FILE SERVICES

Many of the services described in this chapter use a FrameID parameter to refer to a menu in the application's frame file. Before the frame file can be referenced, it must be explicitly opened. The frame file should be closed before the application exits. When cooperating tasks use different frame files, each task must close its frame file before relinquishing control to the other task.

7.1.1 OPENME - Open Frame File

Status	2 words	(output)
FrameFileSynchNumber ($FCTV$)	word	(input)
FilenameLength	word	(input)
FilenameText()	n bytes	(input)

Opens the frame file for use by the application.

The FrameFileSynchNumber is the number that the Frame Compiler Tool equates to the global symbol, $FCTV$. The OPENME service checks the number against a number stored in the frame file that it opens. If the numbers match, the application was built with the global symbols that FCT defined for the actual frame file that is being opened. If the numbers do not match, an error message is displayed by the OPENME service, although execution is allowed to proceed. (This is a debugging error and should never occur in a production application.)

Only one frame file can be open at any point during the application's execution. Notice, however, that there is no error return if you try to open a second frame file. The OPENME service assumes that you are trying to open the same file that is already open and simply ignores the call.

The window server closes the frame file when the application calls the Suspend service and then automatically reopens the correct frame file before it reactivates the application and returns to it from the Suspend call.

There is no restriction against using two or more frame files in an application, but since each frame file defines the $FCTV$ symbol, you will get a multiple-symbol definition error during the task build if two files are used by the same task. You must alter the MACRO file produced by FCT for the second frame file by changing the $FCTV$ symbol to, say, $FCT2$. Thus, if you wanted two frame files in the same task, you could open the first file using the symbol $FCTV$. Then after closing it, you could open the second frame file using the $FCT2$ symbol.

There doesn't seem to be any advantage to using two frame files in one task, unless you want to try to put the majority of your HELP frames in a second frame file and locate it on a separate diskette. Then the user who wants HELP would insert the diskette while running your application. Notice that it is not possible to read menus from one frame file and have menu services automatically read HELP frames from the other frame file. Switching between two frame files must be done by calling CLOSEM and OPENME services from the application code.

The default file specification is APPL$DIR:. Therefore, if you omit the device and directory name from the file specification for the frame file, it will automatically be opened from the [ZZAPnnnnn] directory.

The application in Appendix A uses the OPENME service (see Pages A-15 and A-50).

7.1.2 CLOSEM - Close Frame File

Status	2 words (output)

Closes the frame file.

7.1.3 WIRMS - Read Message Frame

Status	2 words (output)
CountOfLinesReturned	word (output)
Offsets()	n words (output)
MessageBuffer()	n bytes (output)
FrameID	word (input)
MaxBufferLength	word (input)

Reads a message frame from the frame file and returns it to the application.

The message text is returned in the MessageBuffer as a single string of characters (no separators, no CRLFs). The Offsets array contains byte offsets into the MessageBuffer for each line of the message. (The first offset is always 0.) The Offsets array must have at least one more entry than the number of lines expected in the message, since the last offset points to the byte beyond the last line. The length of each line can be computed by subtracting the line's offset from the offset of the next line.

An error is returned if the message frame's text exceeds the buffer size declared by MaxBufferLength. Notice that CountOfLinesReturned is not an input parameter; there is no bounds check on the Offsets array.

A message frame can also be displayed directly on the screen in a message window by calling a menu service (see Section 7.2.17).

The application in Appendix A uses the WIRMS service (see Pages A-8 and A-50).

7.2 HIGH-LEVEL MENU SERVICES

High-level menu services display menus, solicit input from the user, and return the user's input to the application. High-level menu services are provided as a convenience to the application developer, and to foster consistency in the user interface.

An application requests a high-level menu service through a call on the window server. Many of the services offer a static or dynamic form of call. The static call retrieves the text of the menu from a frame file using a frame ID which is passed as a parameter. The dynamic call passes all the text directly from the application.

A high-level menu service creates a window for the menu, determining the window size automatically. The window's position on the screen is controlled by parameters that have been stored in the frame file or that are passed on the call. The service then reads the keyboard (through the character-passing buffer), and responds to the user's actions. When an appropriate action has been taken by the user -- that signals the end of the menu interaction -- the menu service destroys the menu window and returns appropriate values to the application.

A menu consists of headers and options. Headers are displayed at the top of the menu. The text of each header is left-justified. Options are displayed on lines below the last header. Options may be stacked in a single column, spread across a single row, or arranged in a matrix of rows and columns. The number of options supplied must be equal to the row count times the column count. (One or more options may be blank, however. See the SKIP and NOCHOOSE attributes, described below.)

One of the actions that the user can take is to press the HELP key. Menu services recognize the HELP key and automatically retrieve a HELP frame from the frame file using the appropriate frame ID which either was passed from the application or was stored in the menu frame in the frame file. The HELP frame is displayed in another window, and the menu service permits the user to move around a HELP tree if one has been provided in the frame file. Eventually, the user presses the RESUME key to leave HELP. The menu service then resumes menu processing. This processing of the HELP key is totally automatic. When the menu service eventually returns the user's response to the menu, it gives no notification to the application that HELP was used.

Each option on a menu can have an associated HELPframeID. The HELPframeID contains the frame ID of the HELP frame to be used, should the user ask for HELP when the cursor bar is positioned on the option. If the user asks for HELP on an option that has a 0 HELPframeID, the window server displays a window that informs him

that no HELP is available. The user must then press RESUME to continue without HELP. This is annoying to most users, especially in the Synergy environment, where HELP always seems to be present. On the rare occasion when you do not want to supply HELP for a user, it may be less annoying to him if you code the HELPframeID with a -1 value (NOHELP). When the window server is asked to provide HELP on an option whose HELPframeID is -1, it simply beeps. The same message is conveyed to the user, but he need not read a new window, and he need not press a key to continue.

Each option can have an associated OptionValue. The OptionValue is a number that identifies the option. Rather than return the text of the option that was chosen by the user, menu services return the OptionValue.

OptionValues must range between 0 and 255. OptionValues 254 and 255 have a special meaning:

- OptionValues 254 (NOCHOOSE) are displayed in dim rendition and cannot be chosen by the user. NOCHOOSE options are useful on menus which are displayed often, but have options which are sometimes invalid. The user always sees the menu in a familiar form -- with the same options -- and moves the cursor bar the same way. Options in dim rendition signal to the user that they will simply beep when selected.

- OptionValues 255 (SKIP) are displayed in bold rendition, and the cursor bar never stops on them. SKIP options can be used to provide blank lines or extra text on the menu.

When options are arranged in a matrix and blank options are used to group the options into logical subgroups, you should use the NOCHOOSE attribute rather than the SKIP attribute. A blank line with the NOCHOOSE attribute lets the cursor move smoothly over it, whereas the SKIP attribute prevents the cursor bar from moving sideways onto it.

Single-choice and flow control menus also associate a nextframeID with each option, in addition to the HELPframeID and the OptionValue. The nextframeID points to another frame in the frame file. When an option is chosen by the user, the associated nextframeID is **not** used by the menu service; it is merely returned to the application as an output parameter. The application can use it to select the menu to be displayed on the next call to menu services.

NextframeIDs provide a mechanism for storing the structure of a menu tree in the frame file. Their use is not required, although the actual frameID is always present in the frame file or in the dynamic call, probably set to 0.

The rendition of options is controlled by menu services. Options are usually displayed in normal rendition. When selections are being made on multiple-choice menus, the currently selected options are redisplayed in dim italics.

7.2.1 Menu Renditions

A *rendition* is a variation in the way a character is displayed on the screen. For example, a character -- say an "A" -- might be displayed at one of three different levels of brightness; dim, normal or bold. Each of these brightness levels is considered a rendition. Giving the character a slant (italic rendition) or drawing a line under it (underline rendition) are other examples.

Text in menus is generally in normal rendition, but you can vary the rendition by inserting special nonprinting character sequences within the headers and the options in menus and string editing windows, and in the prompt strings in string editing windows.

The nonprinting sequence starts with a character with the value 28 decimal. One, two, or three digits follow, and the sequence ends with either a "+" or a "-". The digits are characters "0" through "9". FCT will accept character 28 in the source frame file, but it will also translate the two-character sequence "\$" into character 28 in headers and options (see Section 8.2.6).

- The digits form numbers in the range 0 to 999.
- These numbers represent a binary value whose bits stand for individual attributes.
- The last character determines whether these values are set or cleared; "+" sets attributes (i.e., the value is ORed with the current rendition), "-" clears attributes (i.e., the value is NOT ANDed with the current rendition).

Bit	Value	Attribute
0	1	Intensity
1	2	Intensity
2	4	Italic
3	8	Underline
4	16	Reverse
5	32	Boxed

Intensities are determined by using bits 1 and 0 in combination. There are three levels of intensity -- dim, normal, and bold.

Bits 1 and 0	Intensity
00	Dim
01	Normal
11	Bold

The combination of dim and underline is not supported. The default renditions are:

Headers:	Normal
Options:	Normal
Prompt string:	Normal
SKIP option:	Bold
NOCHOOSE option:	Dim

To underline the word ABC in a line, you must precede it with the special character, 28, followed by "8+" and then follow the word with the special character 28 and "8-". In a text line for FCT, the sequence would be:

```
Here is the \$8+ABC\$8- word, underlined.
```

The string "Press \$32+{RESUME}\$32- to leave HELP." will set the font to be the boxed font at the start of the string "{RESUME}" and set it back after it (see Section 4.3.6).

The application in Appendix A displays strings in its own window and does its own interpretation of the embedded control sequences (see Pages A-11 through A-13).

7.2.2 KEY USAGE

Each high-level menu service predefines a set of function keys.

For all menus except HELP menus, the MAIN SCREEN, EXIT, and F5 keys are returned to the application with success status, with all other output parameters returned as zeros. User selections for the set-up menu are ignored. You should always check the KeyPressed parameter on a success return to see if the user has pressed one of these keys.

The CANCEL key undoes all selections, returns the cursor bar to the first item on a menu, and waits for further input from the user.

The SELECT key is recognized on multiple-choice menus. It changes the rendition of an option, updates the count of selected options, and waits for further input from the user.

The DO and RETURN keys mean an action is to be performed. On single-choice, flow control, and multiple-choice menus, the option the cursor bar is on is selected, the menu is destroyed, and control returns to the application with successful status and all output parameters. Set-up menus do not return to the application when these keys are pressed. These keys are used to perform the operation for the option and the menu remains displayed with the cursor bar moved to the next set-up option. A set-up menu is successfully terminated only when the user presses EXIT. See Chapter 11 for the conventions used to display set-up menus.

The HELP key displays the HELP frame associated with the option the cursor bar is on. During the display of HELP frames, the NEXT SCREEN and PREV SCREEN keys are used to move forward and backward within the HELP tree. The RESUME key terminates the display of HELP frames, and returns to the menu.

If a HELP frame has options, a cursor bar is displayed on one of the options, and the DO and RETURN keys are also enabled. If the user presses DO, RETURN, or NEXT SCREEN, the current HELP frame is destroyed, a new HELP frame is read from the frame file using the NextFrameID of the current option, and the user's response to the new HELP frame is solicited.

7.2.2.1 Termination Key List - You can enhance the definition of keys by supplying a termination key list. If a key that is in the termination key list is pressed, the menu service treats it as a success, destroys the window, and returns all selections.

For example, you may want to treat the ADDTNL OPTIONS key as a special key on one of your menus. You put the ADDTNL OPTIONS key on the termination key list. When the menu service detects an ADDTNL OPTIONS key while displaying that menu, it returns the key to your application instead of just beeping.

Menu services consults the termination key list **before** it responds with the usual processing of the key. Thus, if you put the SELECT or CANCEL keys on the termination key list, you are telling menu services that it should not process that key but should return it to the application as a "success" response.

7.2.3 Single-Choice Menus

A single-choice menu allows the user to make one choice before control returns to the application.

7.2.4 EXSING - Static Single-Choice Menu

Status	2 words	(output)
KeyPressed	word	(output)
NextFrameIDChosen	word	(output)
OptionValueChosen	word	(output)
FrameID	word	(input)

7.2.5 DSINGL - Dynamic Single-Choice Menu

Status	2 words	(output)
KeyPressed	word	(output)
NextFrameIDChosen	word	(output)
OptionValueChosen	word	(output)
TerminationKeyCount	word	(input)
TerminationKeyList()	n words	(input)
X	word	(input)
Y	word	(input)
HeaderCount	word	(input)
For each header:		
HeaderLength	word	(input)
HeaderText()	n bytes	(input)
RowCount	word	(input)
ColumnCount	word	(input)
For each option:		
OptionHelpFrameID	word	(input)
OptionNextFrameID	word	(input)
OptionLength	word	(input)
OptionText()	n bytes	(input)
OptionValue	word	(input)

7.2.6 HELP Menu

The application can call the window server to display a HELP frame from the frame file. There is no dynamic version of this call.

See the description of HELP key processing in Section 7.2 and the discussion of HELP frames in Chapter 11.

7.2.7 EXHELP - Static HELP Menu

Status	2 words	(output)
FrameID	word	(input)

The application in Appendix A uses the EXHELP service (see Pages A-38 and A-50).

7.2.8 Multiple-Choice Menus

Multiple-choice menus allow the user to make multiple selections before returning to the application. The menu service inserts a header just above the options that indicates the maximum number of selections allowed and the number of selections that have been made. This header is updated automatically by the menu service as the user moves around the menu, making and cancelling selections with the SELECT key. The last selection is made and the menu is terminated when the user presses the DO or RETURN key.

7.2.9 EXMULT - Static Multiple-Choice Menu

Status	2 word	(output)
KeyPressed	word	(output)
CountOfOptionsChosen	word	(output)
OptionValuesChosen()	n bytes	(output)
FrameID	word	(input)

7.2.10 DMULTI - Dynamic Multiple-Choice Menu

Status	2 words	(output)
KeyPressed	word	(output)
CoutOfOptionsChosen	word	(output)
OptionValuesChosen()	n bytes	(output)

TerminationKeyCount	word	(input)
TerminationKeyList()	n words	(input)
X	word	(input)
Y	word	(input)
MaxChoices	word	(input)
HeaderCount	word	(input)
For each header:		
HeaderLength	word	(input)
HeaderText()	n bytes	(input)
RowCount	word	(input)
ColumnCount	word	(input)
For each option:		
OptionHelpFrameID	word	(input)
OptionLength	word	(input)
OptionText()	n bytes	(input)
OptionValue	word	(input)

7.2.11 Flow Control Menus

The flow control menu services display a two-level menu, get a choice from the user, and return the choice to the application. A flow control menu provides an easy way to compress two levels of a menu hierarchy.

A flow control menu is a list of titles across the top of the screen, with a single-choice, single-column submenu "hanging" from one of the titles. The call specifies the InitialSubmenu to be displayed (numbered from 0, left-to-right). As the left and right ARROW keys are pressed, the original submenu disappears and a different submenu appears below the title to the left or right. The function keys F11, F12, F13 and ADDTNL OPTIONS correspond to the first four submenus.

NOTE

> Although more than four submenus are permitted, you should restrict your flow control menu to four submenus since there is a convention that maps these four function keys onto these four submenus.

Once the flow control menu is displayed by a flow control menu call, these keys are recognized by menu services; and they cause the corresponding submenu to be displayed, without displaying the submenus in between. If there are less then four submenus, the rightmost of these function keys display the rightmost menu.

DO or RETURN signifies selection and successful return to the application.

A flow control menu must have at least one title; each title must have a submenu with at least one option.

7.2.12 EXFLOW - Static Flow Control Menu

Status	2 words	(output)
KeyPressed	word	(output)
NextFrameIDChosen	word	(output)
OptionValueChosen	word	(output)
FrameID	word	(input)
InitialSubMenu	word	(input)

The application in Appendix A uses the EXFLOW service (see Pages A-37 and A-50).

7.2.13 DFLOW - Dynamic Flow Control Menu

Status	2 words	(output)
KeyPressed	word	(output)
NextFrameIDChosen	word	(output)
OptionValueChosen	word	(output)
TerminationKeyCount	word	(input)
TerminationKeyList()	n words	(input)
InitialSubMenu	word	(input)
X	word	(input)
Y	word	(input)
TitleCount	word	(input)
For each title:		
TitleLength	word	(input)
TitleText()	n bytes	(input)
SubmenuOptionCount	word	(input)
For each submenu option:		
OptionHelpFrameID	word	(input)
OptionNextFrameID	word	(input)
OptionLength	word	(input)
OptionText()	n bytes	(input)
OptionValue	word	(input)

7.2.14 Set-Up Menu

Set-up menus are special-purpose menus that display the current settings for a number of diverse characteristics. The user can

confirm the current setting by not changing it, or can change the setting to a new value. Control returns to the application when the user presses the EXIT key or any other key in the termination key list. There is no validation of new values by menu services.

Each option has a *class* associated with it. There are six classes, numbered as follows:

1. A *Binary class* option has two text strings associated with it, supplied by the application. The user toggles between them, by pressing the DO key.

2. A *Menu class* option has a single-choice menu associated with it that supplies the text string appropriate to the characteristic. The user chooses a value for the characteristic by pressing the DO key to see the menu, then making a selection on the menu.

3. An *Alphastring class* option has a text string as a value. The user changes it by editing the text string.

4. A *Numericstring class* option has a number string as a value. The user changes it by editing the number string. Numbers are unsigned integers that range between 0 and 65535.

5. An *Alphastring/NOECHO class* option has a text string as a value. The user changes it by editing the text string. The string is not echoed to the screen, except as a checkerboard character in each character position. This option class is used to permit users to enter text strings, such as passwords, without echoing the text on the screen.

6. A *Numericstring/NOECHO class* option has a number string as a value. The user changes it by editing the number string. Numbers are unsigned integers that range between 0 and 65535. The digits are not echoed to the screen, except as a checkerboard character in each character position. This option class is used to permit users to enter numbers without echoing them on the screen.

7.2.15 WIXPS - Static Set-Up Menu

Status	2 words	(output)
KeyPressed	word	(output)
CountOfOptionsChanged	word	(output)
OptionValuesChanged()	n bytes	(output)
FrameID	word	(input)
RowCount	word	(input)

For each option:		
OptionClass	word	(input)
If Binary (1):		
CurrentFlag (0=first, 1=second)	word	(input & output)
If Menu (2):		
CurrentValue	word	(input & output)
If Alphastring (3) or Alphastring/NOECHO (5):		
MaxStringLength	word	(input)
CurrentStringLength	word	(input & output)
CurrentStringText()	n bytes	(input & output)
If Numericstring (4) or Numericstring/NOECHO (6):		
CurrentNumber	word	(input & output)

The set-up menu in the frame file supplies the position, headers, termination key list, and options. The input parameters supply the initial setting of each option. You supply the initial setting for each option in the order that the options appear in the frame file. The window server checks that the OptionClass parameter matches the class that FCT provided when it compiled the frame.

The set-up menu in the frame file may have options with the SKIP attribute. These have an option class that tells menu services to ignore them. Do not include parameters for these options.

See the description of output parameters under Section 7.2.16.

The application in Appendix A uses the WIXPS service (see Pages A-21 and A-50).

7.2.16 WIPS - Dynamic Set-Up Menu

Status	2 words	(output)
KeyPressed	word	(output)
CountOfOptionsChanged	word	(output)
OptionValuesChanged()	n bytes	(output)
TerminationKeyCount	word	(input)
TerminationKeyList()	n words	(input)
X	word	(input)
Y	word	(input)
HeaderCount	word	(input)
For each header line:		
HeaderLength	word	(input)
HeaderText()	n bytes	(input)
OptionCount	word	(input)
For each option:		
OptionHelpFrameID	word	(input)
OptionLength	word	(input)

```
OptionText()                          n bytes (input)
OptionValue                           word    (input)
OptionClass                           word    (input)
If SKIP (0): nothing else
If Binary (1):
  CurrentFlag
   (0=first, 1=second)                word    (input & output)
  FirstTextLength                     word    (input)
  FirstText()                         n bytes (input)
  SecondTextLength                    word    (input)
  SecondText()                        n bytes (input)
If Menu (2):
  NextFrameID                         word    (input)
  CurrentValue                        word    (input & output)
If Alphastring (3) or Alphastring/NOECHO (5):
  MaxStringLength                     word    (input)
  CurrentStringLength                 word    (input & output)
  CurrentStringText()                 n bytes (input & output)
If Numericstring (4) or Numericstring/NOECHO (6):
  CurrentNumber                       word    (input & output)
```

The output parameters from WIXPS and WIPS services are as follows:

- CountOfOptionsChanged - This is 0 if the user has not changed any of the set-up options. If the user changes one or more set-up options, the count is returned in this field.

- OptionValuesChanged - Notice that this is a byte array. These bytes have unspecified values if CountOfOptionsChanged is 0. Otherwise, the first n bytes in this string supply the OptionValue of the n options that have been changed. An OptionValue is a number less than 254, so it fits in a byte. Be sure there are as many bytes in this string as there are options on the set-up menu. The OptionValues of the changed options are not returned in any order.

- CurrentFlag - For each Binary class option, you supply the initial setting of the option in this field on input, and you receive the final setting of the option in this field on output.

- CurrentValue - For each Menu class option, you supply the OptionValue of the option on the pop-up menu that should be used as the initial setting. The OptionValue of the final option chosen from the pop-up menu is returned in this parameter.

- CurrentStringLength and CurrentStringText - For each Alphastring and Alphastring/NOECHO class option, you supply the initial string's length and text in these fields on input, and you receive the final string's length and text in these fields on output.

- CurrentNumber - For each Numericstring and Numericstring/NOECHO class, you supply the initial number in this field on input, and you receive the final number in this field on output.

There is some redundancy in the output parameters. You can, if you want, simply ignore the CountOfOptionsChanged and the OptionValuesChanged. In this case, you would look at each of the set-up options to see what values were returned.

You can use CountOfOptionsChanged to bypass this scan of all the set-up options. If the CountOfOptionsChanged shows that no options were changed, you need not check the other outputs. If the CountOfOptionsChanged shows that some options were changed, the first byte in OptionValuesChanged identifies the OptionValue of the first option that was changed, the second byte identifies the OptionValue of the second option that was changed, etc.

7.2.17 Messages

Message menus are menus without any options. They consist only of headers. The menu service displays the headers and waits until the user presses a termination key. The EXIT, MAIN SCREEN, and F5 keys are always recognized as terminators by the service. You should supply other keys as termination keys. Be sure to check the KeyPressed parameter on return from the service, to see what action to take (see Section 11.2.3).

Message frames are also used to store text strings in the frame file. You can read the text of the message frame into memory without putting it on the screen by calling the WIRMS service (see Section 7.1.3).

7.2.18 EXMESS - Static Message Frame

Status	2 words	(output)
KeyPressed	word	(output)
FrameID	word	(input)

The application in Appendix A uses the EXMESS service (see Pages A-37 and A-50).

7.2.19 DMESSA - Dynamic Message Frame

Status	2 words	(output)
KeyPressed	word	(output)
TerminationKeyCount	word	(input)
TerminationKeyList()	n words	(input)
X	word	(input)
Y	word	(input)
HelpFrameID	word	(input)
HeaderCount	word	(input)
For each header:		
HeaderLength	word	(input)
HeaderText()	n bytes	(input)

7.3 STRING EDITING

The string editing menu services provide all the functionality of the string editing primitives in a single call. They create a string editing window, read and echo the user's keystrokes, and then destroy the window. The string editing services require a frame ID to supply the header and prompt. There are no dynamic versions of these calls. To vary the interactive behavior, you must use the primitives.

Notice that there is a string editing primitive, Edit String Field (WIEF), that allows you to create a string editing field in your application window, without creating a special window (see Section 7.6.4).

When reading keystrokes into a string field, control characters are accepted and echo in the dim font. Also, the ENTER key on the numeric keypad inserts a control-M (carriage return) character into the string.

7.3.1 WIXSTR - Alphanumeric String Editing

Status	2 words	(output)
KeyPressed	word	(output)
ReturnStringLength	word	(output)
ReturnString()	n bytes	(output)
FrameID	word	(input)
InitialStringLength	word	(input)

InitialString()	n bytes (input)

The frame-type of the frame specified in the call must be "Alphastring."

The frame file supplies two options. The text of the first option is taken as the prompt; the text of the second option as the initial value of the string to be edited.

If InitialStringLength is 0, InitialString is ignored. If InitialStringLength is nonzero, the InitialString overrides the default string stored in the frame file.

The frame file's default string (second option) gives the maximum length of the field, however. Be sure to allocate this many bytes for the ReturnString parameter.

The two options in the frame file are displayed on a single line in the window, so their combined length cannot exceed the allowable window width of 78 characters.

7.3.2 WIXNUM - Numeric String Editing

Status	2 words	(output)
KeyPressed	word	(output)
ReturnNumber	word	(output)
FrameID	word	(input)
InitialNumber	word	(input)

The frame-type of the frame specified in the call must be "NumericString."

The frame file supplies two options. The text of the first is taken as the prompt, the second is ignored. You supply the starting number in the InitialNumber parameter. A five-byte field in the window is provided for the number.

The user can edit or enter an integer value from 0 to 65535. The number is converted to binary and returned in ReturnNumber. The user cannot enter a plus or minus sign, or a decimal point. You must call the Alphanumeric String Editing service (WIXSTR) and do your own parsing of the input if you want to allow signed numbers or decimal points.

7.4 FILENAME SERVICES

These services are used to retrieve the names of files. A full

file specification is returned to the application. The full file specification has the usual format:

NODE::DEVICE:[DIRECTORY]FILENAME.TYP;VERSION

In a DECnet or cluster environment the file specification can be up to 70 characters long. You should plan ahead for these environments and reserve 70 bytes in your parameters so that these longer file specifications can be returned.

7.4.1 Old File

This service displays a list of existing files in one or more directories and allows the user to make selections.

Your application supplies a wildcard specification as the last header. The service prepares a menu window using all the headers and a matrix of options that is always six rows by three columns.

The initial display of options contains all filenames that match the wildcard specification. If more than 18 filenames match the wildcard specification, the service displays a down arrow on the menu to indicate that the user can scroll the menu window down onto additional matching filenames. As soon as scrolling is started, an up arrow is displayed to show the user that he can scroll back up over the filenames. Up to 256 filenames may be presented at a time.

If the user's directory contains more than 256 files, or if the user wants to look at multiple directories and the combined number of filenames exceeds 256, then the user must extract some subset of files by using a new wildcard specification. The user modifies the wildcard specification by pressing the FIND key and entering a new specification. The current and new specifications are then merged and redisplayed in the last header. The matrix of options is updated with files matching the new wildcard specification. The service permits the user to enter a node name, in case the system is tied into DECnet.

If the user presses FIND, then supplies a new wildcard specification, then presses SELECT instead of DO, all filenames that match the new specification (up to the maximum allowed) are selected automatically and redisplayed in dim italic.

If the user presses ADDTNL OPTIONS, the window server displays a second menu showing a matrix of options that contains volume and user directory names. (System directories are not displayed.) The matrix is four rows by three columns, and it also scrolls. If the user chooses a new directory name, the new volume and

directory name replace the volume/directory names in the wildcard specification, and the Old File menu is updated with a new wildcard specification and a new matrix of filenames.

The matrix of filenames shows the filename, type and version number for each file. Only the highest version number is shown for each file, unless the wildcard specification has an asterisk in the version number field.

If the wildcard specification has an asterisk in the directory name field, then all directories on the volume are shown, and the filenames are grouped by directory name.

Applications that use a particular file type should probably supply that file type in the wildcard specification, which will serve to reduce the number of files that the user has to view. (See Section 11.5.)

The service interprets the SELECT, DO, and CANCEL keys in the same manner as they are interpreted on the multiple-choice menu.

On return from OLDFLE or WIXOLD, the first filename is returned to the application with a count of the number of filenames (options) selected. To retrieve the remaining filenames, WICOLD must be called, before calling any other services.

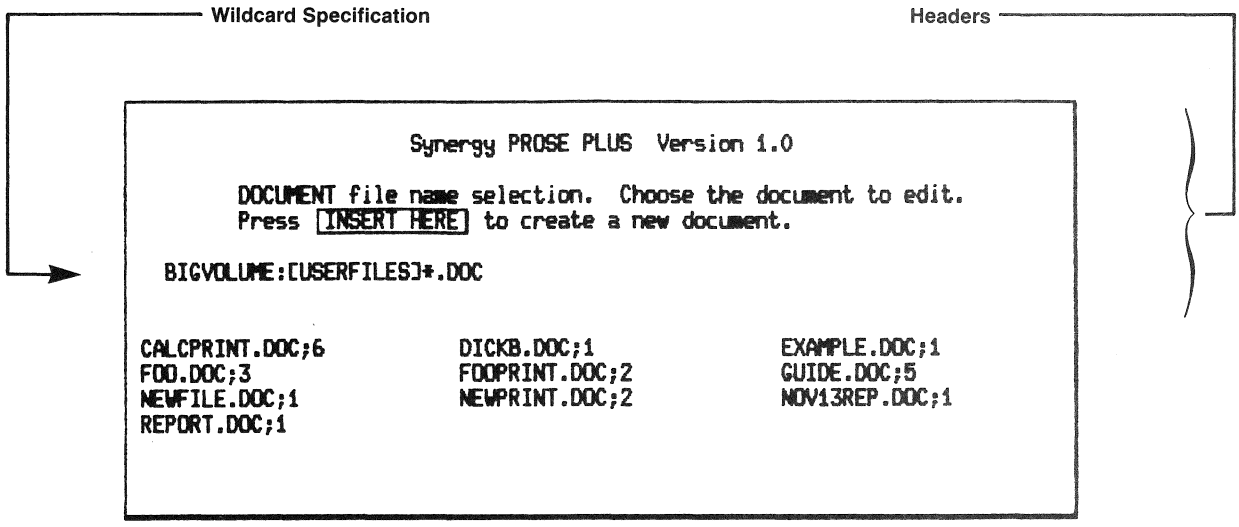


Figure 7-1: Old File Menu

7.4.2 WIXOLD - Static Old File

Status	2 words	(output)
KeyPressed	word	(output)
CountOfNamesChosen	word	(output)
FilenameLength	word	(output)
FilenameText()	n bytes	(output)
FrameID	word	(input)
MaxChoices	word	(input)

FrameID must select a message frame. The message frame supplies the positioning information for the window, the termination key list, the HELPframeID, and the headers including the wildcard specification. The wildcard specification is supplied by the last header in the message frame. (See Section 8.4.5 for a description of message frames.)

7.4.3 OLDFLE - Dynamic Old File

Status	2 words	(output)
KeyPressed	word	(output)
CountOfNamesChosen	word	(output)
FilenameLength	word	(output)
FilenameText()	n bytes	(output)
TerminationKeyCount	word	(input)
TerminationKeyList()	n words	(input)
MaxChoices	word	(input)
X	word	(input)
Y	word	(input)
HelpFrameID	word	(input)
HeaderCount	word	(input)
For each header:		
HeaderLength	word	(input)
HeaderText()	n bytes	(input)

The last line of header text supplies the initial wildcard specification.

7.4.4 WICOLD - Get Selected Filename

Status	2 words	(output)
FilenameLength	word	(output)
FilenameText()	n bytes	(output)

You use WICOLD when more than one filename was selected by a call to OLDFLE or WIXOLD. Each time WICOLD is called, one more filename is returned to the application. An error is returned after all filenames have been retrieved. Be sure to retrieve all the filenames before calling any other services.

7.4.5 New File

These services display a string editing window and accept file specifications as input.

The application supplies an initial file specification in InitialString which is merged with system defaults and displayed as the default file specification. The user can edit any portion of the default file specification. The final string is returned to the application in FilenameText.

The user can press the ADDTNL OPTIONS key to display a second menu showing a matrix of options containing volume and directory names. The matrix is four rows by three columns, and it scrolls. If the user chooses a new directory name, the new volume and directory name replace the volume/directory names in the file specification, but editing continues.

The last header line is used as a prompt. If a prompt is not desired, the last header should have length zero. The maximum string size is 78 bytes. The prompt and default file specifications are displayed on a single line in the window, so their combined length cannot exceed the allowable window width of 78 characters. See Section 11.5 for conventions to be followed when using these Filename services.

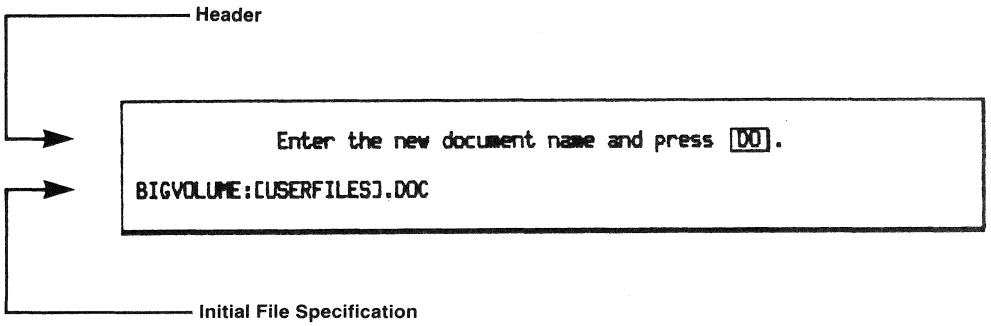


Figure 7-2: New File Menu

7.4.6 WIXNEW - Static New File

Status	2 words	(output)
KeyPressed	word	(output)
FilenameLength	word	(output)
FilenameText()	n bytes	(output)
FrameID	word	(input)
InitialStringLength	word	(input)
InitialString()	n bytes	(input)

FrameID must select a message frame. The message frame supplies the positioning information for the window, the termination key list, the HELPframeID, the headers, and the prompt as the last header (See Section 8.4.5 for a description of message frames.)

7.4.7 NEWFLE - Dynamic New File

Status	2 words	(output)
KeyPressed	word	(output)
FilenameLength	word	(output)
FilenameText()	n bytes	(output)
TerminationKeyCount	word	(input)
TerminationKeyList()	n words	(input)
InitialStringLength	word	(input)
InitialString()	n bytes	(input)
X	word	(input)
Y	word	(input)
HelpFrameID	word	(input)
HeaderCount	word	(input)
For each header:		
HeaderLength	word	(input)
HeaderText()	n bytes	(input)

7.4.8 Any File

An application can combine the actions of the static services WIXOLD and WIXNEW by calling the Static Any File service, WIXANY. Two message frames must be supplied, one for the Old File part and one for the New File part of WIXANY. The window server starts by displaying the Old File menu, but reacts to the INSERT HERE key. If the user presses INSERT HERE, the window server destroys the Old File menu and automatically displays the New File menu. (Be sure to avoid putting the INSERT HERE key on either menu's termination list!)

7.4.9 WIXANY - Static Any File

Status	2 words	(output)
KeyPressed	word	(output)
CountOfNamesChosen	word	(output)
FilenameLength	word	(output)
FilenameText()	n bytes	(output)
OldfileFrameID	word	(input)
MaxChoices	word	(input)
NewfileFrameID	word	(input)
InitialFilenameLength	word	(input)
InitialFilenameText()	n bytes	(input)

OldfileFrameID and NewfileFrameID must select message frames. The message frames supply the positioning information for the windows, the termination key lists, the HELPframeIDs, and the headers -- including the wildcard specifications in the Old File menu and the prompt in the New File menu. (See Section 8.4.5 for a description of message frames.)

Your application determines whether the Old File menu or the New File menu supplied the filename by examining both the Status and the CountOfNamesChosen parameters after return from the service:

- If Status shows success and CountOfNamesChosen is nonzero, the Old File menu was used to select one or more filenames. The first filename is in FilenameText. Use WICOLD to retrieve the additional filenames.

- If Status shows success, CountOfNamesChosen is zero, and KeyPressed is the DO key, then the New File menu was used to supply a name, which is in FilenameText.

- If Status shows success, CountOfNamesChosen is zero, and Keypressed is other than the DO key, the user terminated the menu without supplying any filename. Check for MAIN SCREEN, EXIT, F5 or a key on one of your termination lists.

7.5 DIRECTORY NAME SERVICES

These services are used to show a list of directories to the user and to permit the user to choose a directory name from the list. Only directories that are on the local network node are shown.

You pass the frame ID of a message frame. The message frame supplies positioning information for the window, the HELPframeID, and headers.

7.5.1 WIXCHD - Get Directory Name

Status	2 words	(output)
KeyPressed	word	(output)
DirectoryNameLength	word	(output)
DirectoryNameText()	n bytes	(output)
FrameID	word	(input)

The user's choice of directory name is returned in the DirectoryNameLength and DirectoryNameText parameters. Be sure to examine the KeyPressed parameter to determine if any selection was made. Interpret the EXIT key to mean "no selection," rather than "exit from application."

7.5.2 WIXSHD - Show Directory Names

Status	2 words	(output)
KeyPressed	word	(output)
FrameID	word	(input)

The user can press RESUME or EXIT to terminate the display of the directory names.

7.6 PRIMITIVE MENU AND EDITING SERVICES

The *primitive services* that are described in this section are used by the Synergy high-level menu services to manipulate the screen and keyboard. These primitive services are available for application use in situations where the high-level services do not provide the desired effect.

These primitive services provide a fine degree of control over the appearance of menus and the interactions between the user and the system. However, for most situations the high-level services may be more than adequate. You should gain a good understanding of the capabilities of the high-level menu services before deciding that you must use these primitive services.

The primitive services are generally used in a sequence:

1. Create the window by calling a Create service.

2. Execute a loop, reading keys by calling a Get Key service and examining the output of the call until a satisfactory termination key is pressed.

3. Destroy the window by calling a Destroy service.

Notice that the high-level menu services automatically do certain user-friendly operations that are not performed by the primitive services, although you can program them yourself. Specifically, the high-level services look at the type-ahead buffer before creating the window. If the type-ahead buffer supplies an acceptable response, the display of the window is completely bypassed. Also, the high-level menu services match the user's typed responses against the menu options as a method of positioning the cursor bar, as well as responding to the ARROW keys.

There are two categories of primitive services:

- String editing (soliciting typed input from the user)
- Menus (soliciting menu choices from the user)

7.6.1 String Editing Primitives

String editing primitives solicit typed input from the user and return it as a string to the application. You can create and use a special window to do the editing, or you can request that the editing be done in the application window.

When reading keystrokes into a string field, control characters are accepted and echo in the dim font. Also, the ENTER key on the numeric keypad inserts a control-M (carriage return) character into the string.

7.6.2 WICRS - Create String Editing Window

```
Status                  2 words (output)
X                       word    (input)
Y                       word    (input)
HeaderCount             word    (input)
For each header:
  HeaderLength          word    (input)
  HeaderText()          n bytes (input)
PromptLength            word    (input)
PromptText()            n bytes (input)
DefaultLength           word    (input)
DefaultText()           n bytes (input)
CursorPosition          word    (input)
InputType               word    (input)
```

MaxLength word (input)

This service creates a string editing window. String editing windows are stackable.

The X and Y coordinates position the upper left-hand corner of the windowframe. From 0 to 20 headers may be supplied, and are displayed at the top of the window, left-justified. HeaderLength must not exceed 78. The prompt text is displayed to the left and on the same line as the default text. PromptLength and/or DefaultLength may be 0. DefaultLength must not exceed MaxLength. The sum of PromptLength and MaxLength must not exceed 78. CursorPosition specifies the initial setting of the editing cursor in the type-in area. It is customary to place it at the right end of the default text; just set CursorPosition equal to DefaultLength. If there is no default text, a setting of 0 or 1 positions the cursor on the leftmost character.

InputType specifies either Alpha (0) or Numeric (1). Numeric string editing windows permit only the numeric keystrokes and are thus limited to positive integers. If you want to permit the user to type signed or fractional numbers, you must use the Alpha setting and do your own check of the keystrokes for correctness.

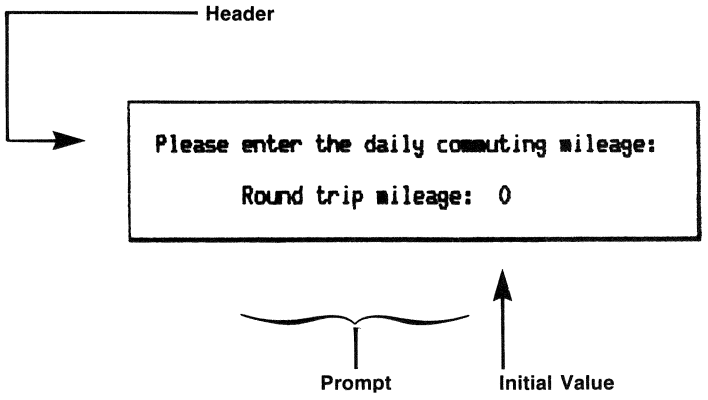


Figure 7-3: String Editing Window

Remember that text displayed in a string editing window may appear in different renditions. (See Sections 7.2.1 and 8.2.7.)

Notice that this service is limited to creation of the string editing window and display of its initial contents. The actual editing does not begin until you call the Get Key from String Editing Window service (WIGKS).

7.6.3 WIDES - Destroy String Editing Window

Status	2 words (output)

This service is used to destroy the string editing window. A string editing window is a stackable window and must be explicitly destroyed before a nonstackable window is created or the Suspend service is called.

7.6.4 WIEF - Edit String Field

Status	2 words	(output)
KeyPressed	word	(output)
CursorPosition	word	(input and output)
InputType	word	(input)
MaxLength	word	(input)
StringLength	word	(input and output)
StringText()	n bytes	(input and output)

This service provides the string editing features of a string editing window, without creating a special window. The service is used to provide string editing directly in the application's front window. You must position the GIDIS cursor at the point in the front window where you want the user's keystrokes to be echoed. You then call the WIEF service, and on return you retrieve the data from the StringText parameter. The WIEF service provides string editing in insert mode with use of the left and right ARROW keys and the delete key.

Since there are no X and Y parameters, this primitive relies on the current GIDIS cursor position as the leftmost character position of the editing area. MaxLength specifies how long the editing area is; 78 is the maximum value, assuming you have put the GIDIS cursor on the left edge of a screen-wide window. You must be sure that the StringText parameter has the same number of bytes as the MaxLength value. You can supply default text for the string by passing the default text in the StringText parameter and setting the StringLength parameter to the length of your default text. Set StringLength to 0 if you have no default text. The WIEF service displays the default text, left-justified and space filled, in the type-in area. It then places an editing (blinking) cursor on the character position specified by CursorPosition (use 0 or 1 for the leftmost character), and begins to read the keyboard through the character-passing buffer. InputType specifies the type of keystroke that is accepted, as in the WICRS service (see Section 7.6.2).

When the user presses a function key other than a left or right ARROW key, or the delete key, the editing is terminated and WIEF returns to the application. You get back in the StringText parameter all characters as they appear on the screen, that is, as the user sees them. StringLength counts the rightmost character that the user sees, even though the user may have moved the cursor (with left arrow) back to the beginning of the string at the time of termination. StringLength includes any spaces that the user explicitly entered on the end of the string, but does not include spaces that were supplied by WIEF during the expansion of the default text. The final position of the editing cursor is returned in CursorPosition. The terminating key is returned in KeyPressed.

7.6.5 WIGKS - Get Key from String Editing Window

Status	2 words	(output)
InitialCursorPosition	word	(input)
KeyPressed	word	(output)
FinalCursorPosition	word	(output)
StringLength	word	(input and output)
StringText()	n bytes	(input and output)

This service begins editing the field in the string editing window created by WICRS. Keystrokes are echoed in insert mode in the editing area, and the cursor is moved appropriately. Left and right ARROW keys are recognized, and the delete key deletes the character to the left of the cursor. Control returns to the application when any other key is pressed.

Although the WICRS call can supply a default string and a starting position for the cursor, the WIGKS call can also supply these values. This allows the WIGKS service to be used repeatedly in the same string editing window. For example, the application may determine that the string supplied by the user in the first WIGKS call is in error. The application may offer the user an error message in a new window. When the user presses RESUME and the error window is removed, the application may call the WIGKS service again with a corrected version of the string as the new default value, and allow the user to continue to edit the string.

The application supplies an initial string using the StringLength and StringText parameters, and locates the cursor within the editing area using the InitialCursorPosition parameter (0 or 1 for the leftmost character). The cursor position must be between zero and MaxLength (set on the WICRS call). WIGKS returns the KeyPressed, the FinalCursorPosition, and the resultant StringLength and StringText. Notice that the initial string and

the resultant string share the same application buffer.

An error occurs if the active window is not a string editing window.

7.6.6 WIHDR - Change header

This service changes the text of a header line in a string editing window. It is described in Section 7.6.12, since it also can be applied to a menu.

7.6.7 Menu Primitives

Menu primitives create and manipulate the contents of menus, and in so doing solicit menu choices from the user. The Synergy window server does most of the work of formatting the menu from the text that you supply and then moving the blinking cursor bar in response to the user's keystrokes. The user generally makes a menu choice by pressing the DO or RETURN key when the cursor bar is on the desired option.

The actual text of a menu option is never returned; rather an OptionValue associated with the option is returned. OptionValues are integers in the range 0 to 255. OptionValues 254 and 255 are used for options that appear on the menu but are not selectable by the user. Options that have OptionValue 254 are called NOCHOOSE options and are automatically displayed in dim rendition. The cursor bar can be moved onto these options; but if the user attempts to select them the window manager beeps the keyboard. Options that have OptionValue 255 are displayed in bold font and are called SKIP options because the cursor bar skips over them. Options with OptionValue 254 or 255 may have blank text.

Option parameters are listed column by column.

7.6.8 WICRM - Create Menu Window

```
|       Status                  2 words (output)
|       X                       word    (input)
|       Y                       word    (input)
|       HeaderCount             word    (input)
|       For each header:
|         HeaderLength          word    (input)
|         HeaderText()          n bytes (input)
```

```
RowCount                word      (input)
ColumnCount             word      (input)
For each option:
  OptionLength          word      (input)
  OptionText()          n bytes   (input)
  Rendition             word      (input)
  OptionValue           word      (input)
```

This service creates a menu as a stackable window. The X and Y coordinates position the upper left corner of the windowframe. The lines of header text are displayed at the top of the window, and the options are displayed below them. The width of the window is determined by the longest header line, or by the longest line containing options. The height of the window is determined by the number of headers and the number of rows of options. There are 20 text lines available and each line can be up to 78 bytes long. When more than one option is placed on a line, each option requires enough bytes to contain the longest option plus 2. Thus, each column has the same width. If two columns of options are intended, no OptionLength can exceed 37 bytes.

The product of the RowCount and ColumnCount parameters determines the total number of options that must be present. Options may be arranged in as many rows and columns as desired, subject to the maximum of 20 text lines and 78 bytes per line. A menu with no options is allowed (and is used to implement the high-level HELP service).

Each option is a text string with an integer OptionValue that will be returned when the option is selected. If there is more than one column of options, the parameters supply all options for the first column, then all options for the second column, etc.

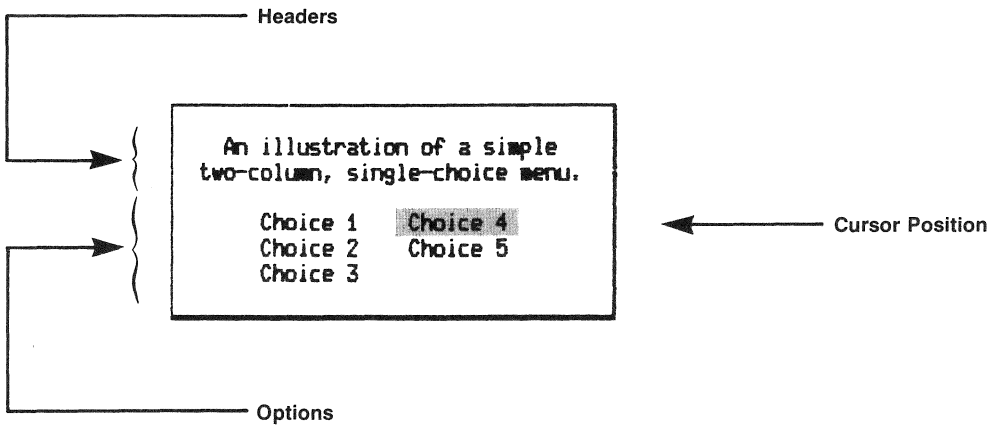


Figure 7-4: Single-Choice Menu

Notice that this service only creates the menu window and displays the text in it. The actual solicitation of the menu choice must be done by calling the Get Key from a Menu service

(WIGKM).

The cursor bar is initially positioned on the first menu option (row 0, column 0), even if this option has an OptionValue of 255 (SKIP). If necessary, you should call the WIPPS service to position the cursor bar onto a non-SKIP option. The cursor bar is automatically turned on.

7.6.9 WIDEM - Destroy Menu Window

Status	2 words	(output)

This service destroys the menu window. It must be called after WICRM and before a nonstackable window is created or the Suspend service is called.

7.6.10 WIENM - Change Option in a Menu

Status	2 words	(output)
OptionRow	word	(input)
OptionColumn	word	(input)
OptionLength	word	(input)
OptionText()	n bytes	(input)
Rendition	word	(input)
OptionValue	word	(input)

This service changes the OptionText and/or OptionValue for a menu option. The old text and value are discarded and replaced with the new ones.

OptionRow and OptionColumn are counted from zero.

An error occurs if the menu is not the front window.

7.6.11 WIGKM - Get Key from a Menu

Status	2 words	(output)
OptionValueChosen	word	(output)
RowChosen	word	(output)
ColumnChosen	word	(output)
KeyPressed	word	(output)

CHAPTER 8

THE FRAME COMPILER, FCT

8.1 INTRODUCTION TO FCT

FCT is a program that converts a source frame file into an object frame file.

A source frame file is an ASCII text file, created with EDT or a similar editor, that contains menu frames, HELP frames, and messages that have been described with the FCT language.

An object frame file is an RMS sequential file that contains menu frames, HELP frames, and messages in a format that is optimized for use by the window server.

FCT compiles the source frame file, producing the object frame file, optionally a listing file, and some symbol definition files.

FCT diagnoses various errors in the source file with a message to the terminal. The message shows the offending line from the source file, the line number of the offending line, and an error message stating the nature of the error. FCT continues to compile by recovering either on the next token of the source line, or on the next line of the source file.

FCT produces a listing file that shows all lines of the source file (numbered) and a cross reference listing that shows all frame names and the references to them from within the frame file. Diagnostic messages are also embedded in the listing file.

FCT produces a MACRO source file that defines each frame name from the source frame file as a global symbol, equated to a frame identifier (16-bit integer). This MACRO file must be assembled and linked with the application code that expects to refer to these frames in calls on menu services. The task builder (PAB) resolves references to the global symbols, replacing them with the frame identifiers.

For PASCAL programmers, FCT produces a PASCAL source file that defines the frame names as named constants. This file may be included in any PASCAL source that expects to refer to the frames.

FCT recognizes references to frame names from within the source frame file itself, and resolves these references automatically. Unresolved references are diagnosed as missing frames.

8.2 FCT LANGUAGE

The purpose of the language is to define *frames*, that is, blocks of text and the attributes that are needed to use them as menus, HELP, and messages in Synergy applications.

Most of what appears in a source frame file is the text of the frame. The language is designed to make that text easy to edit and easy to visualize in a window.

The first character of each line designates what that line contains. Since all commands begin with a period (.), lines that begin with a period are called *command lines*. Lines that do not begin with a period are either *blank lines* or *text lines*. A line that begins with two adjacent periods is treated as a text line; one of the periods is removed by FCT.

Command lines contain tokens such as keywords, integers, punctuation characters, or quoted strings. The tokens may be separated by any combination of spaces, commas, and horizontal tabs. Any command line may end with a comment; the comment is introduced by an exclamation mark. FCT ignores all text to the right of the exclamation mark.

Command lines are short. There is no need to continue a command on a second line.

All keywords must be spelled out in full, or abbreviated to the first three characters. No other abbreviations are allowed. Keywords and their abbreviations are not reserved words, hence they can be used as frame names. Keywords are case-insensitive.

A command line may have no command, which is useful for inserting long comments.

```
.! *****
.! *** NOTE TO INTERNATIONALIZERS ***
.! The text lines of this frame must not exceed
.! 30 characters.  Here is a handy ruler.
.!       1         2         3
.!3456789012345678901234567890
```

Example 8-1: Comments in a Frame File

Integers may be signed plus or minus. The default is plus, if no sign character is supplied. An integer may be followed by a "b" (or "B"), which indicates that the integer value must be stored in a single byte in the object frame file. The byte-sized integer is needed only within binary message frames (see Section 8.4.6).

Quoted strings use double-quote characters as delimiters. You can embed a double-quote character in the string by using two adjacent double-quote characters. Thus "A""B" forms the string A"B. Quoted strings must not contain control characters such as TAB.

In the following line formats, the brackets ([and]) indicate optional portions of the line. The <xyz> format indicates the metasymbol xyz, which must be replaced according to the explanatory text that follows the format. The ellipsis, ..., indicates repetition of the preceding construct. There are eight line types:

```
.TABLE
.FRAME command line
.HOME command line
.OPTIONS command line
.KEYS command line
Text line
Binary message line
Blank line
```

Formats for the eight line types are described below, then the rules for constructing frames are given in Section 8.4.

8.2.1 .TABLE

```
.TABLE
```

This line must be the FIRST non-blank and non-comment line in the source frame file (i.e. before any .FRAME lines).

The .TABLE line starts the definition of a vector table that creates an index of the frames in the frame file. This vectored index is used by Synergy at run-time to locate frames in the object frame file, given a Frame ID.

8.2.2 .FRAME Command Line

```
.FRAME <FrameName> <FrameType>
```

This line terminates any previous frame and introduces a new frame.

A FrameName is required and must be from one to six characters long, consisting of RAD50 characters (A to Z, 0 to 9, dollar sign and period). The leading character must not be numeric. The FrameName becomes a global symbol for the newly introduced frame. It is the name by which the frame may be referenced from within other frames, and from the application code. Keywords and their abbreviations are acceptable as FrameNames. The FrameType is required and must be one of the following:

```
FLOW
SINGLE
MULTI [ <MaxChoices> ]
HELP
MESSAGE [ BINARY ]
ALPHASTRING
NUMERICSTRING
SETUP
```

For the frame type MULTI, MaxChoices can be supplied. This is an integer that indicates the maximum number of choices that the application is prepared to receive on a multiple-choice menu. If MaxChoices is omitted, it defaults to the number of options that are supplied in the frame. If MaxChoices is supplied, it must not be greater than the number of options supplied.

For the frame type MESSAGE, the attribute BINARY can be supplied. This frame type, known as a *binary message frame*, is distinct from the standard MESSAGE frame type (see Section 8.4.6).

Notice that the frame type SETUP is not spelled with a hyphen. The word PROPERTY can be used in place of the word SETUP.

8.2.3 .HOME Command Line

```
.HOME  [<HorizontalSpec>] [<VerticalSpec>]
```

This line provides screen positioning information for the frame. The frame may be positioned along the vertical axis and along the horizontal axis, independently. Furthermore, either positioning is either relative to the screen or relative to the application's current window (the front window). The positioning information is approximate; that is, the window server may adjust any coordinates so that the frame can be properly displayed.

Although the format shows the HorizontalSpec first, the order of the two specs does not matter. There are two formats for the horizontal spec:

```
                          [ LEFT     ]
              [ SCREEN ]  [ CENTER   ]
COLUMN :      [ WINDOW ] : [ RIGHT    ]
                          [<Integer>]

COLUMN : OFF
```

For either the SCREEN or the WINDOW, LEFT specifies that the frame should be located farthest left; CENTER specifies that the frame should be centered horizontally; RIGHT specifies that the frame should be located farthest right. An integer value may be supplied as an approximation of the horizontal coordinate of the left edge of the windowframe. If the SCREEN keyword has been used, the integer relates to the coordinates of the screen. If the WINDOW keyword has been used, the integer is relative to the left edge of the window. In this case a negative horizontal coordinate may be given, which means that the frame is to be positioned n units to the left of the front window (see Section 6.1.2).

OFF means that the frame should be located so as not to overlap the front window, if possible. This means that the frame may be placed to the left or right of the window, as determined by the window server.

There are two formats for the vertical spec:

```
                          [ TOP     ]
          [ SCREEN ]      [ CENTER  ]
ROW :     [ WINDOW ] :    [ BOTTOM  ]
                          [<Integer>]
```

```
ROW : OFF
```

For either the SCREEN or the WINDOW, TOP specifies that the frame should be located farthest toward the top; CENTER specifies that the frame should be centered vertically; BOTTOM specifies that the frame should be located farthest toward the bottom. An integer value may be supplied as an approximation of the vertical coordinate of the top edge of the windowframe. If the SCREEN keyword has been used, the integer relates to the coordinates of the screen. If the WINDOW keyword has been used, the integer is relative to the top edge of the front window. In this case a negative vertical coordinate may be given, which means that the frame is to be positioned n units above the front window (see Section 6.1.2).

OFF means that the frame should be located so as not to overlap the front window, if possible. This means that the frame may be placed above or below the window, as determined by the window server.

Either the VerticalSpec or the HorizontalSpec may be omitted from the .HOME command line. If the spec is not present, that specification is determined by the window server at run time. It can be regarded as a "don't care" specification.

If the entire .HOME command line is omitted, both vertical and horizontal positioning fall into the "don't care" category.

Some examples follow.

```
.HOME COLUMN:WINDOW:LEFT ROW:WINDOW:TOP
```

This puts the frame in the upper left corner of the front window.

```
.HOME COLUMN:OFF ROW:WINDOW:CENTER
```

This puts the center of the frame on the same vertical center as the front window, but moves the frame horizontally, to the left or right of the front window, so as to obscure it as little as possible.

```
.HOME ROW:SCREEN:BOTTOM
```

This puts the frame at the bottom of the screen, and leaves its horizontal placement up to the window server.

```
.HOME COLUMN:SCREEN:LEFT
```

This puts the frame along the left edge of the screen, with the vertical placement determined by the window server.

```
.HOME COLUMN:WINDOW:20 ROW:WINDOW:20
```

This puts the upper left corner of the windowframe about 20 coordinates in and down from the upper left corner of the front window.

Although the .HOME command line is generally placed immediately after the .FRAME command line, it can occur anywhere within the frame definition.

8.2.4 .OPTIONS Command Line

```
.OPTIONS [ COLUMNS:<ColumnCount> ] [ ROWS:<RowCount> ]
```

This line introduces a set of options for the frame and determines how they are to be displayed within the frame. To display a single column of options, the line should read,

```
.OPTIONS COLUMNS:1
```

that is, the ROWS parameter may be omitted. To display a single row of options, the line should read,

```
.OPTIONS ROWS:1
```

that is, the COLUMNS parameter may be omitted. To display a matrix of options, both the COLUMNS and ROWS parameters must be supplied. Note that if both parameters are omitted, that is, only the command itself is given, a default of COL:1 is assumed.

The text of the options appears on lines that follow the .OPTIONS command line. They are text lines; the text of each option appears on a separate line. The option lines are entered in column-major order, that is, top-to-bottom, left-to-right order. Thus, twelve options arranged in a three-column grid would require twelve lines, representing the twelve options in the grid as shown below.

```
1    5    9
2    6   10
3    7   11
4    8   12
```

If fewer than twelve options are supplied, FCT supplies any blank options (called SKIP lines) that are needed to complete the grid. If more than twelve options are supplied, FCT produces a diagnostic message.

8.2.5 .KEYS Command Line

```
.KEYS <KeyCode> [ <KeyCode> ]...
```

This line introduces one or more termination keys for the frame. (Termination keys tell the window server to terminate menu processing when the key is pressed and to return the key value to the application.) The KeyCode is any of:

- An integer in the range 1 to 569
- A quoted character, such as "a"
- A keyword, such as RESUME (or its abbreviation, RES)

The integer values and keywords used for key codes are supplied in Table 4-2. When the key caption is two words, only the left word should be used. Thus, MAIN is sufficient for MAIN SCREEN. Keys on the auxiliary keypad must be coded as their numeric values. Thus, Keypad Enter is coded as 558.

A maximum of 30 keys may be supplied via .KEYS command lines. Although the .KEYS command lines are usually placed immediately after the the .HOME or .FRAME command lines, they can occur anywhere in the frame definition.

8.2.6 Blank Line.

A blank line with no characters at all, or only space characters is totally ignored by FCT. It may be used to make the source frame file easier to read. A blank line does not produce a blank header or blank option line in the frame.

See the next section for a method of specifying blank header or option lines.

8.2.7 Text Line

```
<Textstring> [\ [ <attribute list> ] ]
```

The text line is used to supply a text string that appears in the frame.

If the leading character of Textstring is a period, two adjacent periods must be supplied. FCT removes one period and treats the remainder of Textstring as a normal text line. A double period is equated to a single period only in the leading character position of Textstring.

Leading or trailing space characters within Textstring are not removed. Textstring must not contain control characters less than ASCII 28, such as a TAB character.

Textstring is terminated by a backslash if there are attributes to follow. In order to embed a backslash in Textstring, two adjacent backslashes are required. Thus,

```
ABC\\DEF   \
```

yields a ten-character string of text, that looks like

```
ABC\DEFbbb
```

where b's represent spaces.

The ASCII 28 character is used to introduce a special control sequence used by the window server. The ASCII 28 may be inserted in Textstring with an editor, but since it is a nonprinting character it is difficult to deal with. FCT provides an alternative method of embedding ASCII 28 in Textstring. The 2-byte sequence "\$" (backslash and dollar sign) can be used in place of the ASCII 28. This 2-byte sequence is converted into a single ASCII 28 character in the object frame file. The \$ sequence may appear anywhere in the Textstring. The control sequences are described in Section 7.2.1.

The attribute list, if present, consists of one or more attributes, supplied in any order. Attributes are tokens, and follow the same separation rules as tokens on a command line; namely, spaces, commas, and tabs in any combination.

Attributes generally accompany the text lines that serve as options on menus. When the window server displays a menu, it uses the attributes associated with each option line to control the action that takes place if the user chooses that option.

The attributes are:

```
<OptionValue>
? <HelpName>
> <NextName>
< <PrevName>
SKIP
BINARY <0-String> <1-String>
ALPHASTRING [ NOECHO ]
NUMERICSTRING [ NOECHO ]
```

An OptionValue, if present, is a numeric quantity in the range 0 to 255. The OptionValue defaults to 0 if it is omitted. The OptionValue is returned to the application by the window server when the user chooses this option.

A HelpName is signaled by the question mark that precedes it. If present, it is the name of a HELP frame that is associated with this text. FCT converts the HelpName into a HELPframeID. (The window server uses the HELPframeID to display the HELP frame if the user presses HELP while the cursor bar is on this option.) The word NOHELP is reserved (in this context only) to mean that the HELPframeID should be coded as -1 for interpretation by the window server. Normally, the window server provides a "no HELP available" message if it finds a 0-valued HELPframeID when the user presses HELP. There are times when even this message should be suppressed, and ?NOHELP achieves that effect. When the window server finds a -1 for a HELPframeID, it simply beeps and takes no other action.

A NextName is signaled by the right angle bracket that precedes it. If present, it is the name of a frame that is to be associated with this text. FCT converts the NextName into a nextframeID. (This is the name of the next frame that should be used if the user presses DO while the cursor bar is on this option. In the case of a HELP frame, the window server acts on this name when the NEXT SCREEN key is pressed, and automatically displays the next HELP frame. In other instances the window server merely returns the nextframeID to the application.) The NOHELP name may be used in this context with the same meaning that it has as a HELP name (see above).

A PrevName is signaled by the left angle bracket that precedes it. If present, it is the name of a HELP frame that is to be associated with this frame. A PrevName can be supplied only on a HELP frame; it is used to provide a backward link to another HELP frame. The window server acts on this name when the PREV SCREEN key is pressed, and automatically displays the previous HELP frame. The NOHELP name may be used in this context with the same meaning that it has as a HELP name (see above).

The SKIP keyword signals that the line is to be skipped by the cursor bar as it moves over the options of a menu. The Textstring of such a line need not be null. The SKIP attribute is always mutually exclusive with the other attributes. (Although FCT translates a SKIP attribute into an OptionValue of 255, you should not supply an OptionValue of 255. Use the SKIP keyword instead.)

The BINARY attribute supplies two strings that are used in a special case of an option on a set-up menu. The 0-string and the 1-string are coded as quoted strings. They are the two "settings" of a set-up characteristic that can be toggled. For example, the entire option line might be coded as

```
Scroll:  \ 4 BINARY "Smooth" "Jump" ? HELPXY
```

The ALPHASTRING and NUMERICSTRING attributes appear only within set-up menus. The optional word NOECHO tells the window server that the characters that the user enters are not to be echoed to the screen. Instead, the window server echoes the checkerboard character to the screen. Use NOECHO when coding a set-up option that requests private information such as a password.

The use of attributes is dependent on where the text line appears. A text line that appears ahead of the .OPTIONS command line is a header line and, except for certain HELP frames, does not have any attributes. A text line appearing after an .OPTIONS command line is an "option" and generally does require attributes. The rules for use of attributes depend on the frame type (see Section 8.4).

There are instances in which the text of a header line or an option line is to be blank. For example, the message of a HELP frame consists of header lines that might be grouped into paragraphs and separated by blank lines. Also, the "rest position" of the cursor bar on a menu may be a blank option.

In no case should blank text be coded as a totally blank line in the source frame file, since blank lines are ignored by FCT.

A blank header or option is coded as a line with a backslash as the first character (thus the Textstring is null). If this is a header line, it contains only the backslash:

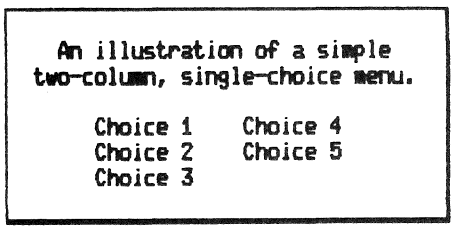
```
\
```

If this is an option line and the line is to be skipped by the cursor bar, the attribute "SKIP" appears in lieu of any other attributes. The line appears as:

```
\SKIP
```

- Each option can have a HelpName attribute.
- No option can have the BINARY, ALPHASTRING, or NUMERICSTRING attribute.
- .KEYS command lines can appear anywhere in the frame definition.

```
.! Two-column, single-choice menu
.! All keywords are abbreviated.
.FRA FRAME2 SIN
.HOM ROW:SCR:CEN COL:SCR:CEN
  An illustration of a simple
two-column, single-choice menu.
.OPT COL:2 ROW:3
Choice 1\ 1 ?HLPSI1
Choice 2\ 2 ?HLPSI2
Choice 3\ 3 ?HLPSI3
Choice 4\ 4 ?HLPSI4
Choice 5\ 5 ?HLPSI5
```



Example 8-3: A Single-Choice Menu

```
.! A multiple-choice menu, that
.! allows three choices.  The program must
.! check that they are a legal combination.
.FRAME MULTI3 MULTI 3
.HOME ROW:WINDOW:100 COL:WINDOW:100
.KEYS NEXT
Please choose one entry in each column.

Press \$32+(NEXT SCREEN)\$32- if you do not want
    ice cream at this time.
\
.! (The menu service inserts a line here.)
.OPTIONS COLUMNS:3 ROWS:5
Type\SKIP
\SKIP
Sugar cone\ 11 ?HLPM11
Plain cone\12 ?HLPM12
Dish\       13 ?HLPM13

Flavor\SKIP
\SKIP
Chocolate\  21 ?HLPM21
Vanilla\    22 ?HLPM22
Strawberry\ 23 ?HLPM23

Scoops\SKIP
\SKIP
Triple\     33 ?HLPM33
Double\     32 ?HLPM32
Single\     31 ?HLPM31
```

Please choose one entry in each column.
Press [NEXT SCREEN] if you do not want
ice cream at this time.

Choose up to 3 items, 0 chosen

Type	Flavor	Scoops
Sugar cone	Chocolate	Triple
Plain cone	Vanilla	Double
Dish	Strawberry	Single

Example 8-4: A Multiple-Choice Menu

8.4.3 Set-Up Menu

(Set-up menus were previously called property sheets.)

- Text lines that precede the first .OPTIONS command line are regarded as header lines for the frame. They cannot have attributes.
- There can be only one .OPTIONS command line, which is followed by the text lines that supply the text of the options.
- The .OPTIONS command line must implicitly or explicitly specify that the options are to be arranged in a single column, that is, .OPTIONS COL:1.

- Each option must have one of the attributes, BINARY, ALPHASTRING, NUMERICSTRING, SKIP, or NextName. If NextName is given, the name must be the name of a single-choice menu. The window server uses the NextName attribute to display the single-choice menu for setting the characteristic. The window server ignores the .HOME command line in this menu, and attempts to position it to the right of the set-up menu option.)

- Each option can have a HelpName attribute.

- .KEYS command lines can appear anywhere in the frame definition. **Do not** put the EXIT key on the termination key list. You can put RIGHT ARROW or LEFT ARROW on the termination key list. The window server then recognizes these keys as terminators, but only when the cursor is on options with the BINARY, Nextname or SKIP attribute. When the cursor is on an option with the Alphastring or Numericstring attribute, the window server responds to the ARROW key by moving the cursor within the string being edited.

```
.! Setup menu
.FRAME SETUP1 PRO
.HOME COL:WINDOW:CENTER ROW:WINDOW:CENTER
Indicate your choices for
  the doorbell switch.
.OPTIONS COL:1
Lighted:   \ 1 BINARY "YES" "  NO" ?HLPSU1
Style:     \ 2 >STYLEM            ?HLPSU2
Name:      \ 3 ALPHASTRING        ?HLPSU3
Quantity: \ 4 NUMERICSTRING      ?HLPSU4
\SKIP
Press \$32+<EXIT>\$32- to accept values.\SKIP

.FRAME STYLEM SINGLE
.OPTIONS COL:1
Ranch\     1 ?HLPSM1
Colonial\  2 ?HLPSM2
Revival\   3 ?HLPSM3
Modern\    4 ?HLPSM4
```

```
Indicate your choices for
  the doorbell switch.

Lighted:   YES
Style:     Colonial
Name:      _____________________
Quantity:  2___

Press [EXIT] to accept values.
```

Example 8-5: A Set-Up Menu

8.4.4 HELP Frame

A HELP frame can consist of a block of text; it may also have options like a single-choice menu. If it has options, the options are introduced by a .OPTIONS command line, and the options have a NextName that points to another HELP frame. If

there are no options -- that is, no .OPTIONS command line -- a NextName may be given on the first text line of the frame. This NextName points to another HELP frame that is displayed when the user presses the NEXT SCREEN key.

All HELP frames, regardless of whether they have a .OPTIONS command line, can specify a PrevName on the first text line. The PrevName, if present, must point to another HELP frame, which is the frame displayed when the user presses the PREV SCREEN key.

- Text lines that precede the .OPTIONS command line are regarded as header lines for the frame. These lines provide the HELP text for the frame. These lines cannot have attributes, except for the first text line, which can have a PrevName and a NextName.

- There can be only one .OPTIONS command line, which is followed by the text lines that supply the text of the options. The .OPTIONS command line can be omitted if there are no options.

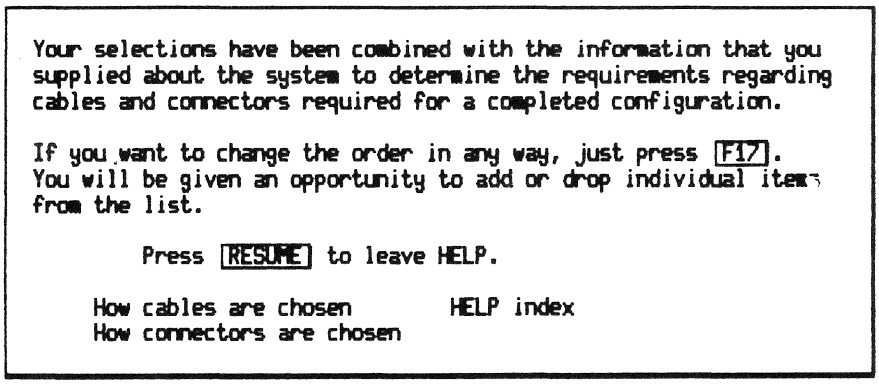
- Option lines cannot have OptionValue, HelpName, BINARY, ALPHASTRING, or NUMERICSTRING attributes.

- Option lines that have text of nonzero length must have either a SKIP or NextName attribute. The NextName attribute must reference a HELP frame.

- Option lines that have no text can have either a SKIP attribute or no attribute at all. When no attribute is present, the line is assumed to be a "rest line."

- .KEYS command lines are not allowed.

```
.! A context-sensitive help frame.
.! PREV SCREEN takes user to the HELP Index.
.FRAME HLPMSG HELP
.HOME COL:WIN:CEN ROW:OFF
Your selections have been combined with the information that you\ <HLPIDX
supplied about the system to determine the requirements regarding
cables and connectors required for a completed configuration.
\
If you want to change the order in any way, just press \$32+<F17>\$32-.
You will be given an opportunity to add or drop individual items
from the list.
\
        Press \$32+<RESUME>\$32- to leave HELP.
.OPTIONS ROWS:2
How cables are chosen\          >HLPMS1
How connectors are chosen\      >HLPMS2
HELP index\                     >HLPIDX
```

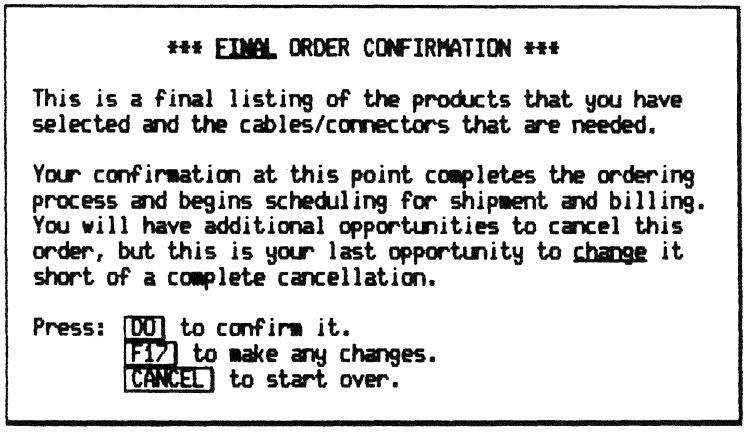


Example 8-6: A HELP Frame

8.4.5 Message Frame

- All text lines are regarded as header lines for the frame. These lines cannot have attributes, except for the first text line, which can have a HelpName.
- .OPTIONS command lines are not allowed.
- One or more .KEYS command lines are required to specify the termination keys that can be used by the user to proceed from this menu. The order of the .KEYS command lines is not significant.

```
.! Sample message frame
.FRAME FINAL1 MESSAGE
.HOME COL:WIN:CEN ROW:WIN:BOT
.KEYS CANCEL, F17, DO
          *** \$10+FINAL\$10- ORDER CONFIRMATION ***\ ?HLPMSG
\
This is a final listing of the products that you have
selected and the cables/connectors that are needed.
\
Your confirmation at this point completes the ordering
process and begins scheduling for shipment and billing.
You will have additional opportunities to cancel this
order, but this is your last opportunity to \$8+change\$8- it
short of a complete cancellation.
\
Press: \$32+{DO}\$32- to confirm it.
       \$32+{F17}\$32- to make any changes.
       \$32+{CANCEL}\$32- to start over.
```



Example 8-7: A Message Frame

8.4.6 Binary Message Frame

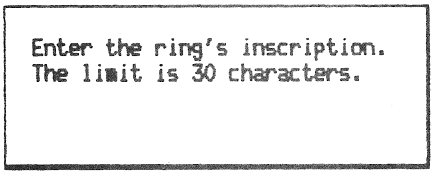
- Only binary message lines, blank lines, and comment lines may appear in a binary message frame. Blank lines and comment lines do not terminate a continued binary message line.

8.4.7 Alphastring and Numericstring Menu

- Any text lines that precede the .OPTIONS command line are regarded as header lines. These lines cannot have attributes.

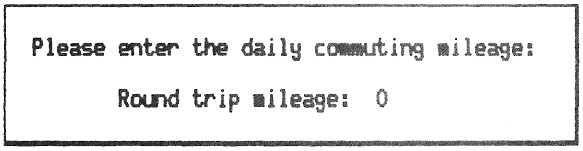
- One .OPTIONS ROWS:1 command line is required, followed by two options (text lines).
- The first option line supplies the prompt text. Null prompt text is coded as a line that begins with a backslash. The line cannot have attributes.
- The second option line supplies the default text for the string; the length of its Textstring supplies the MaxLength of the allowed string. Be sure to pad it with spaces to the length of the longest string that the user is allowed to enter. This line can have a HelpName attribute, but other attributes are not allowed.
- .KEYS command lines are not allowed.

```
.! Alphastring menu
.FRAME ASTRGF ALPHASTRING
Enter the ring's inscription.
The limit is 30 characters.
.OPTIONS ROW:1
\
                                \ ?HLPALF
```



Example 8-8: An Alphastring Menu

```
.! Numeric string menu
.FRAME NSTRGF NUMERICSTRING
Please enter the daily commuting mileage:
.OPTIONS ROW:1
Round trip mileage:  \
\ ?HLPNUM
```



Example 8-9: A Numericstring Menu

8.4.8 VECTOR TABLE

On lines following the .TABLE line (which must be the first source line in the frame file), list the names of all of the Frame IDs that are ever referenced in the source code of your application.

For example, suppose you have a frame file that contains ten frames. Of those ten, six are referenced by the source code of the application (passed as FrameID parameters in some static Menu Service call such as EXFLOW). Assume the names of these six frames are FLOW1, SINGLX, DIALUP, CANTGO, ADDOPT, and NOTYET. The remaining four frames in the file are not directly referenced

from the source, but are referenced from other frames in the frame file (for example, help frames). Assume their names are HELPAD, HELPX, HLPFLO, and STRNG5. In this example, the sequence of lines for the vector table would look like:

```
.! Table of frame IDs used from the application source code.
.TABLE
FLOW
SINGLX
NOTYET
DIALUP
CANTGO
ADDOPT
```

The order of the frames is arbitrary, but there are some cautions relating to changing the order. The order of the frame IDs in the table has no correspondence with the order of the frames themselves in the frame file.

The above example creates a table (which is stored at the beginning of the object frame file created by FCT) with six entries in it. These six frames are the only ones which your application source code can refer to. (You can also include frames in the table which are never passed at run-time to Synergy, but those frames simply take up unnecessary space in the object frame file's vector table.)

Each frame ID in the vector table is assigned an ordinal number; succeeding table entries are assigned increasing numbers. Thus, the frame's number depends on its position in the table (frames not included in the table are not assigned numbers and thus cannot be passed in Menu Service calls). If you change the order of the frames in the table, the numbers assigned to the frames will change, which will require re-task-building the application (and for PRO/Pascal sources, recompilation).

At run-time, the application passes a FrameID to Synergy. The actual binary number that is passed is the frame's ordinal number. Synergy uses this ordinal as an index into the vector table. The indexed entries in the table contain sufficient information (internal to Synergy) to allow Synergy to find and retrieve the actual contents of the specified frame in the object frame file.

The application does not have to explicitly know what ordinal number is assigned to what frame ID. FCT produces a .MAC file that contains global definitions equating FrameID symbol names to their ordinal values. Once you assemble this WHATEVER.MAC file (using the command "$ MACRO WHATEVER") and link the resulting WHATEVER.OBJ module in with your application task(s), the relationship between FrameID symbol and ordinal value will be taken care of by the linker. For PRO/Pascal programs, FCT also produces a .PAS file which you can include in your source files

using %INCLUDE. The include file contains CONST definitions equating FrameID identifiers with their ordinal numbers.

Once the frame file is set up with the frames in the table, you can make changes to the bulk of the frame file, without ever having to recompile or relink the application again. Specifically, you can add or delete information in the frames, and you can even move the frames around in the frame file. (For example, if the frame file is translated into a different language, the application tasks do not need to be changed.)

If you ever change the table itself, then the application will need to be relinked. In addition, for Pascal the sources need to be recompiled so that the new .PAS include file is used. But note that as long as you simply add new frames to the end of the table, then only those modules that refer to those new frames need to be recompiled, since the ordinal numbers for the original frames are unaffected by subsequent table entries.

8.5 FCT OPERATING INSTRUCTIONS

FCT operates as a native VMS program or as a PRO/Tool Kit program.

8.5.1 FCT on VMS

FCT is started from VMS command level with a RUN command addressing the FCT.EXE file.

FCT prompts for a source filename. You enter the name, and press RETURN. Append /LIST to the source filename if you want a listing file.

If the source file is not in your default directory, you can give a logical name as part of the file specification. Use the VMS ASSIGN command to equate the logical name to a desired device and directory.

If you supply a file type, FCT removes it and substitutes "SFF" as the file type, so be sure to name source files with the .SFF file type.

All output files go into the default directory. All output filenames match the input filename, and are distinguished by their file types:

OFF is the object frame file.
LST is the listing file.
MAC is the MACRO symbol file.
PAS is the PASCAL symbol file.

8.5.2 FCT on PRO/Tool Kit

To run FCT in the PRO/Tool Kit, enter the Tool Kit and type

```
RUN $FCT
```

In order to run, FCT requires that the BASIC-PLUS-2 Resident Library be installed. If it is not installed, type the following command:

```
INSTALL LB:[ZZSYS]BP2RES
```

FCT prompts for a source filename. You enter the name, and press RETURN. Append /LIST to the source filename if you want a listing file. The source file must be in the current user directory. All output files go to the same directory.

CHAPTER 9

CHAPTER 9

DEBUGGING THE APPLICATION'S WINDOW

This chapter describes additional tools that may be useful during application development.

9.1 VUE APPLICATION

VUE is a Synergy application that can be used during development to look at the frames in a frame file without executing the application that owns the frame file.

Suppose you are working on a source frame file for the XYZ application. You are trying to imagine what the frames will look like when displayed by the application. VUE lets you see most frames on the screen. In the case of frames that contain references to other frames, such as a HELP tree, you can walk around the tree just as you would when the XYZ application is running.

You should use VUE to examine every frame in a frame file before releasing the application. It is possible to create a frame, such as a HELP frame, that is too large to be displayed on the screen. Normal quality assurance procedures applied to your application may not discover this problem. A thorough examination of every frame in the frame file, using VUE, guarantees that all frames can be displayed. You will also discover that viewing a frame on the screen is a far better way to evaluate it for correctness and effectiveness than looking at it in a listing.

Set-up menus cannot be examined with VUE, since VUE cannot provide all the parameters required for the service call to display a set-up menu.

Also, only those frames included in the frame file's vector table may be examined (you can add any frames you want to the table).

9.1.1 Installing VUE

VUE is installed like any Synergy application. After installation is complete, you must find the ZZAPnnnnn directory in which VUE was installed. Copy down the number nnnnn.

If you are using a Professional Host Tool Kit, copy the file named VUE.COM to the directory on your host machine where you expect to work on the frame file. (If you are using the PRO/Tool Kit, this step is unnecessary.)

You are now ready to use VUE.

9.1.2 Using VUE

When you are ready to view your source frame file do the following:

1. Run FCT on the source frame file to obtain an object frame file. Assuming this operation is error-free, continue below.

2. FCT produces a .MAC file that has the same name as the source frame file. Print this on your local printer.

3. If you are using a Professional Host Tool Kit, execute the VUE.COM file with two additional parameters as follows:

   ```
   @VUE nnnnn ffffff
   ```

   "nnnnn" is the ZZAPnnnnn directory number where you installed VUE. "ffffff" is the filename of your frame file (no file type). Example:

   ```
   @VUE 78 MYFRAMES
   ```

4. If you are using the PRO/Tool Kit, copy the object frame file produced by FCT to the VUE application directory, renaming it VUE.OFF. If the object frame file is named MYFRAMES.OFF, this copy command might resemble:

   ```
   $ COPY MYFRAMES.OFF [ZZAP00078]VUE.OFF
   ```

5. Start the VUE application.

6. VUE requests that you supply the synchronization number placed in the frame file by Synergy. This is the symbol $FCTV$ which is defined on the eighth line of the MAC file that you printed out. Just enter the number that you see there. (The number may be negative.) Assuming that you do not get an error message from the window server, continue below.

7. VUE requests a frame ID. Frame IDs are the numbers to which the frame names are equated in the MAC file. Type any of those numbers in order to see the corresponding frame. If you have a frame called HLPDIR, you will find its name equated to a number, such as HLPDIR=732. Enter 732 and a RETURN in order to see the HLPDIR frame.

   If the frame is a HELP frame or a menu frame that references HELP frames in its options, you can use the keyboard to travel around the HELP tree, just as you would when the application is running.

   Various error conditions are shown on the screen, so if you have created a frame that is too big, etc., you will see the error message instead of the frame. Consult the list of error codes in Table 4-2.

8. To stop viewing a frame, do the following:

   - If it is a HELP frame, press RESUME.
   - If it is some other kind of frame, press EXIT.

   VUE prompts you for a new frame ID.

9. To leave the VUE application, press EXIT when you are being prompted for a frame ID. (VUE cannot be suspended; it ignores F5.)

9.2 MAKE SCREEN WHITE APPLICATION

This application provides a white background on which your application's windows can be displayed. (The normal background for Synergy is gray.) A white background may be useful when making printouts of the screen with the PRINT SCREEN key.

The application is installed like any Synergy application.

When run, the application creates a full-screen window and then automatically suspends itself. When you tell the window manager to resume its execution, it exits.

9.3 PRINTING THE SYNERGY SCREEN

The action of the PRINT SCREEN key reverses the black and white areas of the screen. The assumption is that you are running the Professional in a mode that displays light lettering on a dark background, and that the printout is more useful if it shows dark lettering on a white background.

However, the normal window in Synergy displays dark lettering on a light background. When this is reversed by the action of the PRINT SCREEN key, the printout is not very useful.

A patch command file, SYNREVERS, is included in the Tool Kit, to enable you to reverse the Synergy displays. Once the patch has been made, Synergy will display dark windows (with a light windowframe), and the lettering within the window will be light. When the windows are displayed in this manner, the PRINT SCREEN key produces a printout that looks like the normal Synergy screen. Screen prints can be very useful when documenting your application for end users.

The patch can be applied from DCL by running the ZAP program and supplying an indirect reference to the command file, SYNREVERS.CMD. The patch should be applied when Synergy is not running, since it will not take effect until Synergy is started-up.

After you have obtained the desired screen prints, you can reverse the action of the SYNREVERS patch by running the ZAP program again and supplying an indirect reference to the command file, SYNNORMAL.CMD.

Again, the patch does not take effect until Synergy is started-up.

9.4 FDT TO FCT CONVERSION

The Frame Development Tool, FDT, has been augmented with a WINDOW command. This command converts a frame file produced with FDT into a source frame file for FCT use. Additional editing is required, but the major portion of the text is converted. The

details of the conversion are described in the *Tool Kit User's Guide*.

CHAPTER 10

THE CLIPBOARD

The clipboard is a method of moving data between Synergy applications. The clipboard relieves the user of the need to supply a filename when the data is written and then select a filename when it is read back.

10.1 INTRODUCTION TO THE CLIPBOARD

The clipboard consists of two files. The files have the following names and locations:

- SYSDISK:[ZZPROVUE]CLIPBOARD.TAB -- a table file
- SYSDISK:[ZZPROVUE]CLIPBOARD.DOC -- a text file

When your application receives a request to "Write to clipboard," the application should delete all existing clipboard files. The application should then create CLIPBOARD.TAB and/or CLIPBOARD.DOC.

When an application writes data to the clipboard, it is not known where that data will be read. For example, Spreadsheet data may be included as text in a PROSE PLUS document and as input to Graph. In the first case only ASCII text is needed, but for the graph a table file is required. Thus, Spreadsheet creates both CLIPBOARD.DOC and CLIPBOARD.TAB, in order to permit both paths for the data. If your application can write data in its tabular format as well as a report or text format, it should write both formats.

There are some applications that do not understand table files (such as editors). These need not create CLIPBOARD.TAB when they create CLIPBOARD.DOC, but they should ensure that any existing CLIPBOARD.TAB files are deleted, so that if both files of the clipboard exist, they are synchronized.

When an application receives a request to "Read from clipboard," the appropriate clipboard file should be retrieved. If the file is not there, the application should inform the user that it cannot use the data in the clipboard.

NOTE

> It is possible for there to be data in the clipboard, even if the desired file does not exist, so the application should NOT say "no data exists" unless it has verified that neither CLIPBOARD.DOC nor CLIPBOARD.TAB exist.

Do not delete the clipboard files after a read operation, since they may be read into another application.

10.2 THE TEXT FILE

CLIPBOARD.DOC contains only ASCII text. When a GIDIS file is being copied to the clipboard, CLIPBOARD.DOC is created and should contain a VDM reference to the GIDIS file. The application should use the file service NEWFLE or WIXNEW to request a name for the GIDIS file. The actual GIDIS file is not considered part of the clipboard and should be stored in the SY:[] directory, if the user does not supply a device or directory with the filename.

If the clipboard file is present, but the .GID file that it refers to is not in the user's directory, you should inform the user that the graphics file cannot be found in the local directory. A good example of the technique, with suggested wording of the message and HELP frames, can be found in the PROSE PLUS application. To see it, rename the EXAMPLECV.GID file that comes with PROSE PLUS to some other name, start up PROSE PLUS and try to edit picture EXAMPLECV.GID.

The file specification in the VDM reference has only the filename and file type.

10.3 THE TABLE FILE

The table file represents an array of string and numeric values, along with some information about the values, and possibly, private information. The array consists of rows and columns; each intersection holds a single value or element. The elements are stored in row order; all the elements of row 1 are stored before the elements of row 2, etc. The elements in a column or

row may be all numeric, all string, or a combination of both.

Tables are stored in an RMS variable-length record file. Each record contains ASCII text -- up to 256 bytes. There need not be a regular mapping from a row of table elements to a single record, or an integral number of records. At least one table element exists in each record, and there may be as many table elements in a record as can fit within the record size of 256 bytes. This limit is set so that applications can allocate a reasonably small record buffer.

10.3.1 Special Record Format

Special records contain information outside of the actual values in the array. There are four kinds of special records -- version, source, size and private.

Version, source, and size records precede the array, while private records follow the array. Each special record starts with an exclamation point character (!) in the first byte, and has a keyword in uppercase letters. The syntax is strictly defined. In the explanations below, the underscore character (_) is used to indicate a required space. No leading spaces or extra embedded spaces are allowed.

All table files have a version number record which is always the first record in the file. This record has the following form:

```
!VERSION_2
```

Table files may have a source record immediately following the version record. The source record identifies the application that created the table file. This record has the following form:

```
!SOURCE_'<name>'
```

where <name> indicates a name string up to 16 nonblank characters in length. The single quotes are required; the angle brackets are part of the notation, and are not entered.

All table files have a table size record immediately preceding the records that contain the elements of the array. The table size record describes the size of the array. The table size record has the following form:

```
!SIZE_<n>,<n>
```

where the <n>s are decimal integers indicating the number of rows and the number of columns in the array. The angle brackets are

part of the notation, and are not entered. A table of 20 rows by 8 columns would have the following table size record:

```
!SIZE_20,8
```

Tables may have private data. If so, the data is stored in separate records after a special record that follows the array and introduces the private data. The private data introducer record has the following form:

```
!PRIVATE
```

10.3.2 Data Record Format

Each data record consists of a list of data values separated by commas.

Spaces and tabs between data items are ignored. The following example shows two typical data records:

```
'Goods', 'Canned', 'January 12th', 11.43
'Shipments (in thousands)', 57.6,58.2,58.2,59
```

Synergy Version 2 table files include four types of data -- unformatted number, formatted number, string, and date. (Version 1 table files included only two types of data -- unformatted number and string. Table files are upward compatible.)

- *Unformatted numeric data items* may be whole numbers, or real numbers in either floating point or exponential (scientific) notation. Dollar signs and commas are not allowed. Numbers may be signed. Numbers must be convertible to 64-bit double-precision quantities. Exponential (E) notation may be used. Examples:

```
1               -2.3
0               0.0
123.456         .123456789123E+23
.0000001        +100.000E-8
```

- *Formatted numeric data items* may contain commas and dollar signs. If commas or a dollar sign are included in a number, the number must be bracketed with the # character. Commas may be used to punctuate the integer portion of the number at thousands intervals. The limitation on the range of the number and the manner in which the number is expressed is the same as unformatted numeric data items. Examples:

```
#$1.00#                  #$2.30#
#-123.456#               #-.123456789123E+23#
#123,456.0000001#        #$123,456,789.88#
```

- *String data items* are bracketed with single-quote characters ('). If the string contains embedded single-quotes, each embedded quote must appear twice (''). The length of string data items must be less than or equal to 132. Examples:

```
'abcdef'
'123'
'''What,'' she asked, ''is that?'''
'Double-quotes (") are not treated specially.'
```

- *Date data items* are represented in two ways:

```
&MM/DD/YY&    or   &MM/DD/YYYY&
```

  MM is the one- or two-digit month, DD is the day, and YY or YYYY is the year. When YY is used for the year, it is interpreted as the 20th century, but only if YY is greater than or equal to 50; and as the 21st century if YY is less then 50. A date is bracketed by the & character, as in this example:

```
&04/09/52&
```

A table entry may be omitted. It's position in the table is said to hold a *null data item*. For example, a pair of unseparated commas, or a comma in the first or last position in a record, represents a null data item. Also, a zero-length record (a blank line) in a table represents a null data item.

Null data items represent missing data, and have no data type. A spreadsheet may have cells that have no value, for example. Applications reading in a table should treat null data items in some reasonable way, perhaps as null strings or zero-valued numbers. However, applications should not output null data items as an abbreviation for zero-length strings or zero-valued numbers.

10.4 TABLE FILE EXAMPLES

The following are examples of table files showing all supported data types:

```
!VERSION 2
```

```
!SOURCE 'YOURPROGRAM'
!SIZE 4,5
,&7/1/84&,&8/1/84&,&9/1/84&,&10/1/84&
'Bill',24.8,24.7,25.2,25.3
'Kate',23.4,22.5,21,22.1
'Totals:',#$48.20#,#$47.20#,#$46.20#,#$47.40#
!PRIVATE
B4=B2+B3
C4=C2+C3
D4=D2+D3
E4=E2+E3
```

The same table in a different format:

```
!VERSION 2
!SOURCE 'YOURPROGRAM'
!SIZE 4,5

&7/1/84&
&8/1/84&
&9/1/84&
&10/1/84&
'Bill'
24.8
24.7
25.2
25.3
'Kate'
23.4
22.5
21
22.1
'Totals:'
#$48.20#
#$47.20#
#$46.20#
#$47.40#
!PRIVATE
B4=B2+B3
C4=C2+C3
D4=D2+D3
E4=E2+E3
```

The text version (.DOC) of these table files is:

	7/1/84	8/1/84	9/1/84	10/1/84
Bill	24.8	24.7	25.2	25.3
Kate	23.4	22.5	21	22.1
Totals:	$48.20	$47.20	$46.20	$47.40

CHAPTER 11

SYNERGY CONVENTIONS

Consistency of the human interface, within an application and from one application to the next, contributes to ease of use. The user learns to recognize familiar words and formats and familiar actions on the screen, and familiar keystrokes on the keyboard. The user develops a consistent conceptual model of the system so that new areas of an application can be explored with a measure of confidence and a minimum of surprise.

Synergy provides such a consistent framework with:

- Application windows that have common features
- A window manager that is independent of all applications
- The clipboard for moving data between applications
- A menu and HELP interface that has a consistent feel from one application to the next

However, there is still a wide margin for variation within the Synergy framework. The wording of menu options, the method of handling files, the movement of the cursor in windows, even the meanings of words, can vary from one application to another.

Sometimes there are valid reasons for this variation. Each application developer has to make the tradeoffs between meeting the needs of the application and being consistent with the rest of the applications.

This chapter presents the Synergy conventions as they were conceived and practiced by Synergy developers. A few of the conventions are arbitrary, but many reflect carefully chosen compromises. Most of the conventions are the result of much trial and error.

Section 11.6 discusses some alternative models that were adopted by the developers of the Synergy applications to handle certain demands of the applications.

11.1 WINDOW CONVENTIONS

These are the conventions that apply to **application** windows, as opposed to the more specialized windows that are used for menus and HELP.

11.1.1 Titles

- Applications should use titles on all windows. A title area is highlighted automatically when the window is the front window, which gives the user a constant cue as to where the center of attention is. Furthermore, the title area can be used for the clock icon and a waiting message when the application starts a time-consuming operation.

- If the application uses only one window, the application name might go in the title. But if the application has two or more windows, the title should probably name the contents of the window. Applications such as Spreadsheet and Calculator accomplish both aims in one word. The Graph application labels one window "Data", and the other "Graph." "Graph Data" and "Graph Picture" are alternate choices.

- Be sure to consider using the clock icon and a message in the title area when your application performs a time-consuming task. You should turn off the blinking cursor in your window when the clock icon is shown. (If the cursor remains on, the user is led to believe that he should be typing. See Cursor Use, below.)

  The waiting message that accompanies the icon should be short and should use a word or phrase from the last menu choice or the last user action. Thus, "Loading spreadsheet," "Writing clipboard," "Searching database."

  For example, the window that Synergy displays when it is loading an application is designed to keep the user aware of what is happening. That window has a blinking cursor as well, even though the user is not being invited to type anything. The idea is to give reassurance that the machine is doing its work.

- If you permit the user to change the size of your application windows, you should gauge the length of the title and the waiting messages so that they make sense in a narrow window, even if they get truncated.

11.1.2 Cursor Use

- The Synergy developers discovered that a blinking cursor is very important on a screen full of windows. The window with the highlighted title is the front window, so the user's attention is naturally attracted to it but even then, a blinking cursor serves to focus the user's attention to a smaller area and to signal that the application is waiting for user input.

  The corollary of this convention is that whenever the application is not prepared to accept input, it should consider turning off the cursor and should indicate why by displaying a wait message in the title line of the window.

  Both of these actions keep the user aware of what is happening. If either action is lacking, the user spends some time wondering if all is well.

- The user also gets information about the state of the application from the shape of the cursor, and whether or not it blinks.

  The blinking cursor should be reserved for the point of greatest interest. On a spreadsheet or data grid the cursor is in the form of a bar while the user is simply moving between cells. As soon as the user begins to type something into a cell, the cursor changes to a single character to emphasize that the next or previous character is the most important point. If the entry of data is echoed on another line (not in the cell), as in Spreadsheet, the cell remains highlighted. In effect, there are then two cursors, a nonblinking cursor showing the cell that is being modified and a blinking cursor showing the focal point of the data being typed by the user.

  Be careful about the size of a blinking cursor. If a large area of the screen begins to blink, it is very hard on the user. When the cursor is large, consider using a blinking "rubber band" around a nonblinking, highlighted area.

11.1.3 Size and Location

- Often it seems that an application should use a window that occupies the full screen. Developers who are familiar with full-screen editors or full-screen spreadsheets think that a smaller window is too restrictive. Indeed, when the user is intent on only one task, a full-screen window may be best since it may have the least distractions. However, when the user wants to do two or more tasks at once (e.g., calculate some results from numbers in a report, and consult a spreadsheet or graph while writing a report summary), it may be advisable to allow the windows to shrink to something smaller than full screen, so that the other application's window can remain in view while action takes place in the front window.

  Of course, there may be practical limitations on the minimum window size.

- Your application should remember the size of the application window from one use of the application to the next. This is easily done by storing the size in the ContextBlock that is saved by the call on the Done service (WIDON) and retrieved for you by the call on the Initialize service (WIINI) at start-up.

  You have the problem of whether the application window is being sized by the user in relation to the data file that he is manipulating or whether the size relates to the application in general. For example, the Graph application records its window sizes and locations in the data file, so that when the user selects that data file, the windows adjust to the location and size that the user considered appropriate for that data. Spreadsheet and PROSE PLUS, however, record the window size and location in the context block since the window size is independent of the spreadsheet or the document.

- You may feel that there is a natural location for an application window. For example, the calculator window belongs on the lower right, near the auxiliary keypad that is used to do the calculations. The user may feel otherwise, however, and you cannot prevent the window from being moved.

  Your application should remember where the user put the window and put it back there on the next start-up of your application.

And, if it becomes necessary to alter the size of the window in the course of the application, and changing the size also necessitates changing the location, the application should restore both the size and the location after the special need is completed. For instance, Calculator expands its window when the user selects the print function, but restores the size and location when printing is ended.

11.2 MENU CONVENTIONS

Many of the menu conventions are built into the high-level menu services. The high-level menu services represent much trial-and-error work on the human interface of menus. Still, there are places in the Synergy applications where a developer resorted to the primitive services in order to achieve a special effect. Some of these are discussed in the following sections.

Two points are worth noting:

- Even when the high-level services are used, there are many conventions to be followed that are not enforced, or even supplied, by the Synergy window server. They require careful attention when the application is implemented.

- If you resort to using the primitive services to achieve a special effect, please consider copying, as much as possible, the actions of a similar high-level service in order to minimize the difference between your menu interface and the standard menu interface.

11.2.1 Placement

- Always place flow control menus at the top left of the screen.

- A pop-up menu (usually a small, single-choice menu) that appears as a result of some action that takes place in the application window, could be displayed close to the point of the window action. A good example of this is the mode selection menu that is displayed by the Calculator when the user presses the MODE key. An alternative approach, when the cursor moves around the application window (PROSE PLUS), is to center the menu in the window. If it is possible to fix the location of the pop-up menu, either within the

application window or on the screen, you may provide more consistency for the user.

- Always center an Old File menu or Any File menu on the screen.

11.2.2 Spelling and Capitalization

Menu options should all observe the same capitalization and punctuation rules.

- Menu options should be short, but not so short as to be cryptic. A verb and an object, or a verb and a phrase are usually the best combinations. The articles "a," "an," and "the" are usually unnecessary.

- All menu options should be left-justified.

- Words are spelled out in full, never abbreviated.

- The first word of a menu option always begins with an uppercase letter, but the remaining words of the option are always lowercase, unless they are proper names or the word HELP.

- Menu options **do not** have terminating punctuation (period or comma).

- Set-up menu options are displayed as two parts, the option text on the left, and the current setting on the right. The option text should end with a colon. The spelling and punctuation of the setting should follow the same rules as a menu option, i.e., leading uppercase only, no terminating punctuation.

- When the settings of a binary option on a set-up menu are similar in spelling, and therefore easy to mistake (such as "Off" and "On"), supply leading spaces on one of the settings, so that when the user switches the setting, the visual change is more dramatic. (See the Cell Formats Menu in Spreadsheet for an example.)

- Try to avoid menu options on the same menu that begin with the same characters, which force the user who likes to type the menu response to resort to the ARROW keys or a long type-in. (See the menu that is used to select the number of decimal places in the Spreadsheet's cell format for a good example.)

Good examples	Poor examples
Load data	Load the data Load Data
Write to clipboard	Write data to the Clipboard.
Decimal places:	Dec Plcs

11.2.3 Structure and Wording

- Menu options that are identical, but occur in separate applications, should have the same wording.

- Flow control menus have considerable visual structure, in that they have a title line and one or more submenus. The Synergy developers discovered that the placement of certain common options into the same submenu, at the same relative position and with the same wording, contributed greatly to the ease with which users could move between applications.

  The titles of the flow control menus for the Synergy applications are shown below:

	F11	F12	F13	ADDTNL OPTIONS
Datamanager	File	Edit	Select	Format
Spreadsheet	File	Edit	Personalize	Format
PROSE PLUS	File/Edit	Attributes	Personalize	Format
Graph	File/Edit	Graph	Text	Format

  Options that pertain to file operations are always grouped into the leftmost submenu under the title, "File." Operations that pertain to generic editing are grouped in the second submenu under the title "Edit," but are included with the file operations if necessary. Operations that seem to be personal preference are grouped in the third submenu under the title "Personalize." Options that pertain to formatting or structure of the window or data are grouped in the rightmost submenu under the title "Format."

  Within the File submenu, the Read/Write clipboard options and the Load/Save options are grouped at the bottom of the submenu, as follows:

Spreadsheet	Graph	PROSE PLUS	Datamanger
.	.	.	.
.	.	.	.
.	.	.	.
Read ...	Read ...		Read ...
	Write ...	Write ...	Write ...
Save data	Save data	Save work area	
	Save graph		
	Load data		Load form

That is, "Read from clipboard" precedes "Write to clipboard" if both are present. The clipboard options precede the Save/Load options. "Save..." options precede "Load..." options.

The standard wording is "Read from clipboard" and "Write to clipboard" and "Save data" and "Load data".

- Set-up menus always have a single column of options. The appearance of the cursor varies on a set-up menu, depending on whether the option under the cursor is calling for a type-in (single character cursor) or a selection with the DO key (cursor bar). It is less distracting for the user if the option types are grouped so that all the type-in options are together, and all the selection options are together. In fact, if there are many options of each kind, you should consider having two set-up menus, one for type-in options and one for selection options.

  You always code the last two option lines of a set-up menu as SKIP options, with standard wording. (Since they are SKIP options, the cursor can not descend to them. The user does not think of them as options, but instead sees them as a reminder of what should be done to exit from the menu. Furthermore, since they are coded as SKIP options, the text is automatically bolded by the window server.) The second-to-last line is blank; the last line is "Press EXIT to accept values." The word EXIT is in a box; this is coded in FCT as,

```
\SKIP
Press $\32+{EXIT}$\32- to accept values.\SKIP
```

  This convention is very important because it reminds the user of an exception to the normal key usage in Synergy. The EXIT key normally means "leave the current activity without taking action," and the DO key is normally used to complete action on a menu, so a user who knows Synergy or P/OS will be more

inclined to attempt to complete the set-up menu by pressing the DO key. However, since the DO key is used to make changes on the set-up menu, the EXIT key is used as the completion key for set-up menus. The user **must have** this bolded reminder at the bottom of the set-up menu to prompt the correct action.

This convention is the result of much experimentation with other keying conventions and menu wordings. Using any other convention only weakens this one and will probably confuse the user.

- Message menus have no options, so the user responds to them with a single keystroke, signifying that the message has been seen. The window server always recognizes the EXIT, MAIN SCREEN, and F5 keys, but you should supply a termination key list so the user can press other keys as well. Your message text should inform the user what key to press. The convention is to let the user press any of RESUME, RETURN, ENTER and DO, even though the message text says only "Press RESUME to continue."

  If the user presses EXIT, you should interpret it as equivalent to RESUME, not as a request to exit from the application.

- HELP menus have structure and wording conventions within the menu and the HELP tree. Both subjects are discussed in Section 11.3.

- Keys on the keyboard are always referred to by using the boxed font. The spelling of the function key names is exactly as it appears on the keyboard keys or label strip, except that the characters are all uppercase. This convention is followed in the printed documentation as well (see Section 11.7). Notice that the boxed font lets you draw the ARROW keys and the delete key as well (see Section 4.3.6).

11.3 HELP CONVENTIONS

11.3.1 Placement

- HELP frames that are context-sensitive are positioned off the window in the hope that they will not obscure the position of the cursor in the window.

- Some HELP frames are not related to the cursor position in the window, and these are best centered on the window. (See the PROSE PLUS overview HELP frame that is provided when HELP is pressed immediately after starting PROSE PLUS.)

11.3.2 Types of HELP Users

The user who presses the HELP key is not necessarily the first-time user. The following questions should be answered by the HELP frame:

- Where am I? The user is confused either because he is a new user, or an infrequent user, or has been distracted and lost his train of thought.

- How do I complete this action? The user sees or knows what is needed at the moment, but needs to be told how to complete this action. This usually means he needs to know what key to press.

- What does this mean? The user wants to read about some feature of the system, perhaps because he tried to use it and failed. He doubts his understanding of what is happening.

For a HELP frame that accompanies a menu option, the primary question is one of meaning. The user usually knows how to complete the action and the menu supplies enough context to answer the "Where am I?" question.

For a HELP frame that accompanies an application window, however, any of the three questions can be paramount in the user's mind.

11.3.3 Structure of HELP

Synergy services provide automatic recognition of the HELP key when a menu or message is being displayed on the screen. The application must recognize the HELP key itself when the application is reading the keyboard.

Once a HELP frame is displayed on the screen, however, Synergy services provide automatic recognition of the NEXT SCREEN and PREV SCREEN keys and the RETURN or DO keys for choosing options on the HELP frames.

- a HELP frame consists of three horizontal bands:

  . The HELP text at the top of the frame. This is the text that explains what the help is about, and supplies the help. You can explain what the help is about by making the first line a centered title; but then you generally have to skip a line and, that uses up two lines. It may be better to work the subject of the help into the opening words of the HELP message and just do without a title. These are all header lines in the frame definition.

  . The "Press RESUME..." line. The last header line in the frame definition must say "Press RESUME to leave HELP." This is the exact wording and capitalization, and RESUME should be boxed. In FCT, the line appears as

    Press \$32+{RESUME}\$32- to leave HELP.

    It is best to provide a visual cue that this line is not part of the HELP text above it. This can be done by centering the text of the line, or preceding it with a blank line or a line of dashes. The extra characters required by a line of dashes or the spaces to center the message (in every HELP frame) will increase the size of the frame file, however.

    The "Press RESUME..." line is a very important part of the HELP frame. Users frequently request HELP, read about how to press other keys to achieve a certain result, but forget that they must press RESUME first, in order to get out of the HELP mode. If this line is omitted, users will press other keys and will become very frustrated at the beeping provided by the Synergy HELP service which is waiting for the RESUME key.

  . The HELP options. HELP frames are like single-choice menus, in that they can have options arranged in a column, row, or matrix. If the HELP text that is provided uses terms or concepts that might be confusing to the user, you may want to provide options that, if chosen, will lead to other HELP frames that explain those terms or concepts.

    If you cannot explain a concept in a single HELP frame, you can provide an option, "More on this topic", that leads to continuation of the HELP message.

If there are many concepts and HELP frames, the user may feel that asking for HELP often leads into a maze, and that he never is sure whether he has read all that he needs to read. You can prevent this feeling of being lost in a maze of HELP frames by providing a HELP frame called the HELP Index, and providing an option on **every** HELP frame that leads to the HELP Index. The HELP Index is explained below. The location of the option that leads to it is normally on the far right of the options (the last option line in the FCT frame definition), and is spelled "HELP index". (Notice capitalization!)

The options on a HELP frame are usually arranged in a single row using the FCT line,

```
.OPTIONS ROWS:1
```

If there is only one option (which leads to the HELP Index), you can move it to the far right by putting one or two SKIP lines ahead of it:

```
.OPTIONS ROWS:1
\SKIP
\SKIP
HELP index\ > HLPIDX
```

- The HELP Index usually has only one header line with a centered title, such as "Calculator HELP Index". (Again, notice capitalization.) It probably should have the "Press RESUME to leave HELP" line, but that can be omitted if the index is large.

  The index is a matrix of options that lead to all the significant HELP frames, the ones that explain concepts. The index can be arranged in any format, with blank (no-choose) options separating the topics into logical groupings. It is probably not useful to arrange the topics alphabetically if there are more natural groupings of topics around certain broad subjects.

  The index should have two important entries that lead to short HELP frames. These are "Suspending ABC" and "Exiting ABC", where ABC is your application name. New or infrequent users may start up your application and then want to know how to get out. They may be too timid to just press EXIT or MAIN SCREEN and want to read about it before doing it.

11.4 KEY USAGE CONVENTIONS

Unless your application uses a mouse or tablet, the user's input device is the keyboard. Synergy observes most of the keyboard conventions that are common to P/OS and adds some more of its own.

11.4.1 The Auxiliary Keypad

Remember that the auxiliary keypad **must** be read in application mode, not in numeric mode. Other applications and the window manager rely on the fact that all bytes in the character-passing buffer have been read in application mode, so that the auxiliary keypad keys are distinguished from their look-alikes on the main array of the keyboard. If your application intends to interpret the auxiliary keypad keys as identical to the look-alike key on the main array, you must read the keypad key in application keypad mode and translate it to its equivalent value.

11.4.2 Individual Keys

- **F5** - The window server does not react to this key, except that it always recognizes it as a terminator of a menu operation and passes it back to the application.

  The application **must** recognize this key, both as a return parameter of any service call and also in its own keyboard input. The application must respond to it by calling the Suspend (WIINT) service to suspend itself.

- **INTERRUPT** - The window server does not react to this key, and the application should ignore it with a beep. P/OS treats INTERRUPT/DO as the equivalent of CTRL/C.

- **RESUME** - On the Synergy Main Menu, the window manager reacts to this key by resuming the application that owns the front window. On a HELP frame, the window server reacts to this key by terminating the HELP frame. Otherwise, the window server does not react to this key.

  The application can react to the key, either by adding it to a termination key list on a service call or by recognizing it in its own keyboard input. In all cases, the meaning of the reaction should be that of resuming the main activity after a digression.

- **CANCEL** - On a menu, the window server reacts to this key by resetting any menu choices or set-up menu changes and by moving the menu cursor to the first option. If the application places CANCEL on the termination key list for a service call, this reaction is bypassed and the key gets returned to the application.

  The application can react to the key, either by adding it to a termination key list on a service call or by recognizing it in its own keyboard input, but the meaning of the reaction should be to reset or undo the most recent action or set of actions.

- **MAIN SCREEN** - The application should react to the key, either as a termination key on a service call or by recognizing it in its own keyboard input. The reaction should be to exit from the application, **with an automatic save of any data files.** The menu that offers a choice of saving or quitting is not displayed when the MAIN SCREEN key is pressed.

- **EXIT** - The application should react to the key, either as a termination key on a service call or by recognizing it in its own keyboard input. The reaction should be to exit from the application, with a normal exit sequence, that displays the Exit Menu, offering a choice of Save or Quit (if there is any data to be saved). See the description of the Exit Menu in Section 11.5.1.

- **F11, F12, F13, ADDTNL OPTIONS** - The application can react to these keys, either by adding them to a termination key list on a service call or by recognizing them in its own keyboard input. The meaning of the reaction should be to display the application's flow control menu, if there is one. Use the individual key to select the proper submenu to display first. (F11 for submenu 1, F12 for submenu 2, etc.) Once the flow control menu is on the screen, the window server reacts automatically to these keys, as well as the left and right ARROW keys.

  Be sure to recognize all four of the keys, even if your flow control menu has fewer than four submenus. Associate all the extra keys on the right with the rightmost submenu. Thus, if you have only two submenus in your flow control menu, the F12, F13 and ADDTNL OPTIONS keys all position the end user on the second submenu.

  If your application does not use a flow control menu, please do not assign other meanings to these keys -- just beep the keyboard.

- **HELP** - The application should react to the key by recognizing it in its own keyboard input. The response should be to display a context-sensitive HELP frame (see Section 11.3).

- **F17, F18, F19, F20** - The application can react to these keys, either by adding them to a termination key list on a service call or by recognizing them in its own keyboard input. The response should be to begin an application-specific action.

  The F17 key is used in Graph and Spreadsheet to mean "enter edit mode," so if your application provides an edit mode (for fields in your application window), consider using F17 to invoke it.

- **FIND** - The application can react to the key, either by adding it to a termination key list on a service call or by recognizing it in its own keyboard input. The response should be to initiate a search or repositioning function.

- **INSERT HERE** - The application can react to the key, either by adding it to a termination key list on a service call or by recognizing it in its own keyboard input. The response should be to initiate a function that adds information (text, records, etc.) to existing information.

- **REMOVE** - The application can react to the key, either by adding it to a termination key list on a service call or by recognizing it in its own keyboard input. The response should be to initiate a function that removes information (text, records, etc.) from existing information.

- **SELECT** - The application can react to the key, either by adding it to a termination key list on a service call or by recognizing it in its own keyboard input. The response should be to choose the item indicated by the cursor position. This key differs from DO in that DO initiates some action as well as making the selection. SELECT implies an intermediate step, whereas DO implies completing the selections and moving on to the action stage.

  It is definitely confusing to give DO and SELECT the same meaning, since that would erode this distinction.

- **PREV SCREEN** - The application can react to the key, either by adding it to a termination key list on a service call or by recognizing it in its own keyboard input. The response should be to move backward in time sequence or backward through ordered information and to repaint the window with other information. Backward means closer to the origin or closer to the top or the left of the data in the window.

- **NEXT SCREEN** - The application can react to the key, either by adding it to a termination key list on a service call or by recognizing it in its own keyboard input. The response should be to move forward in time sequence or forward through ordered information and to repaint the window with other information. Forward means away from the origin or closer to the bottom or the right of the data in the window.

- **ARROW keys** - The application can react to the key by recognizing it in its own keyboard input. The response should be to move the cursor (or the cursor-indicated object) in the indicated direction within the window. When the cursor moves over an expanse of data (text page or a data grid) and hits the window edge, the entire window moves over the data that appears to be behind the window. (Of course, this is done by scrolling the data through the window, not by moving the window on the screen!)

- **DO, RETURN and ENTER** - The application can react to these keys by recognizing them in its own keyboard input. The response should be to begin an action that has been indicated by some previous operation.

- **<X] key** - The application can react to this key by recognizing it in its own keyboard input. The meaning of the response should be to delete the last typed character. The key is reserved for editing typed input, and should not be given other meanings.

11.5 FILE CONVENTIONS

Most applications access one or more files in which the user's data is stored. Synergy provides menus for choosing the name of an existing file and for naming a new file. In addition, the Synergy applications open and close files in standard ways.

11.5.1 File Access

- If your application gives the user the choice of using an existing data file, you should use the Old File service to obtain the name of the file at start-up. If you also want to allow the user to create a new data file at start-up, you can either use the Old File and New File services or use the Any File service.

- The Synergy applications require that the user supply a filename **before** a new file is created. The alternate technique of creating the data file and then asking the user for a filename is not used in Synergy.

- If your application offers the opportunity to use an old file or create a new file, you should either use the Any File service, which provides both opportunities, or use Old File first and include the INSERT HERE key on the termination key list for the Old File menu. If the application gets back the INSERT HERE key from the Old File service, it can then call the New File service to get the name of the file to be created.

  Whether you use Old File or Any File in this sequence, you must include the header line, "Press INSERT HERE to create ..." Look at the Spreadsheet, Graph, or PROSE PLUS menus for examples of the header lines in Any File and Old File menus.

  Be sure to examine the key returned by these calls. If the user has accidentally started an application, he will press EXIT or MAIN SCREEN and will be annoyed if you keep insisting that he choose a filename before you allow early exit from the application.

- Since the file menus use the ADDTNL OPTIONS and FIND keys to permit the user to alter the wildcard file specification, you must be careful not to specify these keys in the termination key list.

- When the user presses the EXIT key to exit from the application and there are open data files, the Synergy convention is to ask the user what to do with the data files. This exiting procedure is highly standardized.

  An Exit Menu consists of a single-choice menu with a menu title and two options arranged vertically. The menu title has the name of the application and the words "Exit Menu", such as "Spreadsheet Exit Menu". The two options consist of the word "Save", and the word "Quit". The "Save" option performs the actions necessary to preserve the user's work on the data file, and the "Quit" option discards the user's work. Again, Graph and Spreadsheet have good examples of this exit procedure.

  Notice that on the PROSE PLUS Exit Menu, there is a third option, "Save without formatting," since PROSE PLUS has to offer two methods of saving the text file.

The Exit Menu is always centered on the application's front window.

You should be careful to examine the key returned by this menu service, since it may be the EXIT key. If the user presses the EXIT key while on the Exit Menu, he is declining the two choices for exiting from the application -- he wants to continue running the application. (The EXIT key is occasionally struck by mistake while running the application. This mistake takes the user into the Exit Menu. You should be sure to give the user a way to recover from this mistake and resume the application. Be sure to explain the use of the EXIT key in the HELP frame for the the Exit Menu. Tell the user something like, "Press EXIT to avoid making either of these choices - the ABC application will continue.")

- Synergy applications that use the Old File/Any File method of start-up and the exit procedure described above also offer "Load data" and "Save data" options on the File submenu of their flow menu. These options give the user explicit control, especially when the user wants to terminate processing with one file and begin processing on another file without leaving the application. Offering "Load" and "Save" options on the File menu does not take the place of the Old File/Any File start-up and the exit procedure, however.

11.5.2 Filenames

- When your application uses an Old File or Any File menu during start-up, you begin by supplying a wildcard file specification.

  The conventional wildcard file specification uses the current user volume (SY:), the current user directory ([]), a wildcard (*) for the filename, and the file type that the application expects. The file service expands the SY:[] portion to show the actual current volume and directory, before displaying it on the screen. By not specifying a version number, you cause the file service to display only the highest-numbered version.

  For example, the PROSE PLUS application might begin by displaying all files that satisfy the wildcard file specification, SY:[]*.DOC. When expanded, this might be displayed as

  ```
  BIGVOLUME:[USERFILES]*.DOC
  ```

The user can edit this file specification, using either the FIND or ADDTNL OPTIONS key (see Section 7.4.1).

- Standard file types are listed in the *Tool Kit User's Guide*. Synergy defines additional file types:

User-Visible	System or Tool Kit
WRK - Spreadsheet data file	SFF - source frame file
GID - file of GIDIS instructions	OFF - object frame file
FRM - Datamanager form file	
DAT - Datamanager data file	
IDX - Datamanager index file	
TBL - table file	

11.6 ALTERNATE CONVENTIONS

Some Synergy applications do not follow every one of these conventions. This section discusses important breaks with convention to help you handle similar situations.

11.6.1 Graph

The set-up requirements on a graph are extensive. The Graph application uses a modified form of flow control menu in order to simplify the process of selecting and modifying these set-up characteristics. Normal flow control menus have hanging submenus that are simple single-choice menus. If this approach were taken in the Graph application, the user would not have been able to see the settings of related characteristics at the flow control menu level. To see or change a setting would have required first making a choice on the flow control menu and having the menu disappear, and then using a set-up menu. To move to another set-up menu would mean exiting the current set-up menu, going back up to the flow control menu, and then down to another set-up menu.

The developers decided to combine the three major set-up menus into the flow control menu. Three of the hanging menus in the Graph flow control menu are actually set-up menus. The entire Graph flow control menu is created by the Graph program without using the flow control menu services. Graph creates a window at the top of the screen that looks like the title part of a flow control menu, then responds to the same keys that menu services uses on flow control menus. It displays the left-most menu by

calling menu services to put up a single-choice menu. Graph positions that menu under the title line, just as it would be positioned by menu services as the submenu of a flow control menu. If the user requests one of the other three menus, Graph uses a menu service to display the appropriate set-up menu, again positioned just as it would be by menu services as the submenu of a flow control menu. Graph alters the rendition of the "titles" in the title line just as they are altered on a flow control menu.

The effect is that the user thinks he is dealing with a flow control menu, and since the three submenus that are actually set-up menus have the bolded message that reminds the user that he is on a set-up menu, the break with convention is minimal.

The Pick Patterns/Colors Menu displays patterns for the graph, and if color is in use, it displays the colors. It is not possible to display patterns and colors in menus using the menu services, hence the developer was forced to create his own menu using another window.

See the example program in Appendix A for a color menu executed in the same way.

11.6.2 Calculator

The developers of the Calculator felt that most users would already have a strong conceptual model of a hand-held calculator. The Professional keyboard has an auxiliary keypad and function keys above it that suggested that these keys should be used exclusively to create an interface to match the hand-held calculator model.

The selection of modes of operation on a hand-held calculator is done entirely through the keys. Hand-held calculators do not present flow control menus. The Calculator application has no flow control menu, but generally uses mode-setting keys on the calculator keyboard. The Calculator application does not recognize keys on the edit keypad or the main array because the developers wanted to avoid altering the model of a hand-held calculator.

The HELP key is used to put the Calculator application in HELP mode, as it is for other applications, but once in HELP mode, the Calculator application allows the user to proceed from the HELP frame that describes one key to the HELP frame that describes another key by a simple press of that key. This is an extension of the normal HELP conventions which do not recognize special keys.

In order to provide this response to the HELP key, the HELP frames were coded in the frame file as message frames with long termination key lists. The calculator code displays them as if they were HELP frames, but calling the EXMESS service. It examines the key that is returned to determine what to do next.

11.7 DOCUMENTATION CONVENTIONS

Consider these guidelines in preparing the user documentation for your application.

11.7.1 Terminology

The *Synergy User's Guide* introduces many terms that are used to describe the Synergy window environment. The use of these same terms in the documentation for your application reinforces their meaning and avoids confusion on the part of the user. The chart below shows the major terms; both spelling and capitalization are important.

Synergy Window Manager	wildcard
Synergy Main Menu	set-up
HELP index	clipboard
HELP tree	

Certain phrases have been deliberately used or deliberately avoided. These are listed below:

- Use "**suspend** the application" rather than "**interrupt** the application" to describe the action in response to the F5 key. Using the word "interrupt" suggests use of the INTERRUPT key.

- Use "**press** DO" rather than "**hit** DO" or "**type** DO." Using "type DO" could confuse the user into entering the letters "D" and "O".

- Use "enter" rather than "type" in giving directions to fill a field or respond to a prompt. Example: "Enter the applicant's social security number."

- Use "screen" rather than "display."

- Use "select" when referring to the action of moving the cursor to a menu option, and "choose" when referring to the complete action of selecting and pressing the DO key. Thus,

Select the "New date" option and press DO to begin the action of ...

or alternatively,

Choose the "New date" option to begin the action of ...

- Use "option" rather than "entry" to describe the text that can be selected on a menu, and put quotes around the option text when referring to it. Thus,

  Choose the "New date" option

  rather than either of these:

  Choose the New date option
  Choose the "New date" entry

- Use "**front** window" rather than "**top** window."

- Use "**hanging** menu" or "**submenu**" to refer to a part of the flow menu, rather than "**pulldown** menu."

- Use "cursor" and "cursor bar" rather than "paddle" to refer to the blinking rectangle on the screen.

- Be sure to capitalize the names of keys. Although the key captions on the keyboard are mixed case, the convention throughout Professional documentation is to print the key names in uppercase, using the same spelling as found on the keyboard caption, e.g., ADDTNL OPTIONS.

11.7.2 Organization

Various users will read your documentation for different reasons. The following organization is designed to meet these diverse needs:

- Start with an "Introduction to ..." section that describes the product's major functionality with appropriate illustrations. Finish this section with a description of the remaining sections of the documentation.

  The Introduction section tells the reader what to expect from the product and from the documentation.

- Provide a "Sample Session" section that walks the user through a session with your application. This must be carefully constructed, so that a user encounters no surprises when using it. It should also be carefully limited to the major features of the application. Avoid the temptation to describe the bells and whistles of the product. This is not a full tutorial. The sample session should be something that a brand new user can get through in a half hour.

  This section satisfies some user's need for assurance during their first exposure to a product. If it is well illustrated with pictures of the screen, it can be read even when away from the computer, so a user can read about the use of the application without the embarassment of appearing unsure at the keyboard. This section can encourage a timid user to approach the application.

  The section is not intended for all users; sophisticated users will skip this section.

- Provide a "Concepts" section that explains what can be accomplished with the application. Define special terms in the Concepts section, and discuss the actions of the application without describing the mechanics of the software, keyboard or screen. Use terminology from the user's experience. If possible, use illustrations that show the screen and its printed output.

  This section is intended to answer the question, "What is happening?" By isolating it from the next section, you can avoid confusion between "what to do" and "how to do it."

  This section may be read only once by a user, and then occasionally consulted as a memory refresher.

  Much of what appears in the Concepts section can be put into the HELP tree that is reached through the on-line HELP Index. There are benefits to writing the HELP frames and the Concepts section at the same time. You should question your own understanding of any concept that you cannot explain in a single HELP frame (two short paragraphs). (See Section 11.3.)

- Provide a "Using the Application" section that gives explicit directions for using each part of the application. Use only the terms that have been described in the Concepts section, and provide cross-references to the Concepts section, so that users can easily refresh their memories on why they need to do a specific action.

The "Using" section answers the user's "how to do it" questions. It should be designed to anticipate the problems that users can encounter.

The "Using" section will be consulted often, and it should be carefully organized and indexed so that it can be used as a reference manual.

Much of what appears in this section is also in the context-sensitive, on-line HELP frames. In fact, both the documentation and the on-line HELP frames can profit from shared design.

- Be sure to carefully index your documentation. A manual without an index is very difficult to use as a reference.

APPENDIX A

APPENDIX A

BATON TWIRLER

A.1 INTRODUCTION TO BATON TWIRLER

The Tool Kit includes a sample Synergy application called Baton Twirler. Baton Twirler provides a simulation of one or more batons twirling in the space enclosed by the window on your screen. You can install this application on your Synergy Main Menu and execute it to see its functionality. The on-line HELP frames provide all the description that is needed for its execution.

All of the files needed to build Baton Twirler are also included on the distribution media. Most of the files are listed in the following sections of this appendix. They provide examples that are referenced from the preceding chapters. The source files contain extensive comments.

Many of the files required to build Baton Twirler are useful in building other applications, if those applications are written in the PRO/PASCAL language. Even if your application is written in another language, you may find the PASCAL examples to be a useful model of a high-level language interface to the Synergy services and to the GIDIS (graphics-mode) part of the terminal subsystem.

The following sections include listings of:

- BATON.PAS - This is the main module containing the logic of the application. It uses the PASCAL %INCLUDE directive to incorporate two other PASCAL files, GIDISOPS.PAS and SYNERGY.PAS.

- GIDISOPS.PAS - This is an include file that supplies the formal definitions of the callable procedures that are used to interface the PASCAL code to the GIDIS part of the terminal subsystem. The procedures thamselves are coded in the GIDIS.PAS file, for separate compilation.

- SYNERGY.PAS - This is an include file that defines a number of constants and structures that are useful in dealing with the Synergy services. It also supplies the formal definitions of the callable Synergy services that are used within BATON.PAS. If you use this file in your application, you will want to add definitions of the other Synergy services that you need.

- GIDIS.PAS - This file contains interfacing routines that take calls from PASCAL code and construct and execute GIDIS instructions.

- BATONFRMS.SFF - This is the source frame file used by Baton Twirler.

- BATON.CMD - This is the command file that is submitted to the PRO/Application Builder (PAB) in order to build BATON.TSK. It references the BATON.ODL file.

- BATON.ODL - This is the Overlay Descriptor File referenced in BATON.CMD.

- BATON.INS - This is the install file that is used to install Baton Twirler as a Synergy application.

- BUILD.CMD - This is an indirect command file that can be used to build the application task image.

A few other files are required to build the entire application. These files are included in the distribution media, but are not listed here, since they do not illustrate any aspects of a Synergy-specific application. You may want to print them, and may find them useful in your work. Their names are: GETAKEY.MAC, KBSERV.MAC, and READMESG.MAC.

(The KBSERV.MAC file contains a subroutine named READKB, which recognizes the call-back code and calls the MGTCB service.)

A.2 THE BATON.PAS FILE

```
[Overlaid] PROGRAM Baton;

{ This PRO/Pascal program is provided with the Synergy Tool Kit to           }
{ demonstrate how a simple application interacts with the Synergy services to }
{ use the windowing facilities.  Also, since this program makes extensive use }
{ of PRO/GIDIS to do its output, this serves as an illustration of to use     }
{ GIDIS effectively in an application.  Lastly, this program shows how to     }
{ issue P/OS Executive Directives directly from a PRO/Pascal program.         }
{                                                                             }
{ The ability to issue directives (EMT 377s) was inadvertantly omitted from   }
{ the PRO/Pascal V1.2 documentation.  There is a new predefined procedure     }
{ named DIR$ which takes a single parameter.  This parameter should be the    }
{ Directive Parameter Block for the directive you are issuing.  The format    }
{ and contents for each directive is described in the P/OS System Reference   }
{ Manual (chapter nine for Tool Kit version V2).  To issue a directive,       }
{ define a RECORD type for the DPB for the directive, and assign the fields   }
{ of the record appropriately (with constants, addresses of buffers, etc.).   }
{ Then simply invoke the DIR$ procedure, as in:                               }
{      DIR$( DPB_Record );                                                    }
{ To check the results of the directive, declare:                             }
{      VAR $DSW: [External] Integer;                                          }
{ and then do:                                                                }
{      IF $DSW = 1 THEN ...                                                   }
{                                                                             }
{ This program takes advantage of color, if the system that it executes on    }
{ has color enabled.  Not only does the program display objects in different  }
{ colors, the program changes the values in the color map.  A Synergy program }
{ has to be somewhat careful when it comes to changing the color map.  In     }
{ most cases, Synergy isolates each window from whatever goes on in every     }
{ other window (that is, application A cannot affect application B's          }
{ windows).  This is made possible through separate software-maintained       }
{ contexts for each window.  For example, each window has its own writing     }
{ mode state, its own cursor position, etc.  However, this is NOT true of the }
{ color map.  On the Professional, the color map applies to ALL windows; thus }
{ if application A changes the color map, application B's windows change too. }
{ So changing the color map has to be done keeping possible "side effects" in }
{ mind.  Most importantly, which of the eight entries in the color map you    }
{ change is crucial.  A program should virtually NEVER change entries 0 or 4. }
{ Entry zero is always BLACK, and is used to display text in windows, for the }
{ windowframe surrounding each window, etc.  Entry four is always WHITE, and  }
{ is used for the white background of all windows.  If a program changes the  }
{ color of those entries, you will get very entertaining results!  This       }
{ sample program takes great care to only change entries 1, 2, 3, 5, 6 and 7. }
```

THE BATON.PAS FILE

```
%Include 'BatonFrms/NoList' { Contains ordinal numbers for Frame IDs          }
%Include 'GIDISOps/NoList'  { Contains declarations needed for GIDIS routines }
%Include 'Synergy/NoList'   { Contains declarations for Synergy services      }


{ Here are the CONSTant definitions specific to this module }

CONST

  GTIM      = 61;   { Directive Identification Code for Get Time directive. }
  GTIM_Len  = 2;    { Length of GTIM Directive Parameter Block. }

  MaxPoints = 200;  { Largest number of segments a single Baton may have. }
  MaxBatons = 10;   { Largets number of simultaneous Batons allowed. }
  MaxColor = 7;     { Number of entries in the Professional-300 color map. }



{ Here are the data structure TYPE definitions specific to this module }

TYPE

  A_Baton           = RECORD  { Data structure describing a Baton }
                        Newest, Hue: Integer;
                        Sticks: ARRAY [ 0..MaxPoints, 0..3 ] OF Integer;
                        Speed:  ARRAY [ 0..3 ] OF Integer;
                      END { A_Baton } ;

  ColorDescriptor   = PACKED RECORD
                        Red:   0..7;  { Contains the RGB intensity }
                        Green: 0..7;  { values for a color.        }
                        Blue:  0..7;
                      END { ColorDescriptor } ;
```

```
PackedColorType  = PACKED RECORD                    { Case variant that maps }
                     CASE Integer OF                { RGB values into a 16-  }
                       1: ( W: Unsigned;         ); { bit binary word for    }
                       2: ( R: ColorDescriptor; ); { compact storage.        }
                   END { PackedColorType } ;

PackedBytes      = PACKED RECORD
                     Low: 0..255;  { Simple structure that allows easy   }
                     Hi:  0..255;  { use of both bytes in a 16-bit word. }
                   END { PackedBytes } ;

ColorMapType     = ARRAY [ 0..MaxColor ] OF PackedColorType;
                   { A table large enough to contain RGB intensity values   }
                   { for all eight entries in the Professional's colot map. }

FunctionKeyNames = ( TextKey,       TabKey,       Return,       FindKey,
                     InsertHere,    RemoveKey,    SelectKey,    PrevScreen,
                     NextScreen,    BreakKey,     SetUpKey,     SuspendKey,
                     InterruptKey,  ResumeKey,    CancelKey,    MainScreenKey,
                     Exit,          F11,          F12,          F13,
                     AddOptnsKey,   HelpKey,      DoKey,        F17,
                     F18,           F19,          F20,          PF1,
                     PF2,           PF3,          PF4,          Up,
                     Down,          Right,        Left,         Delete );
                   { Enumerated type for use with the GetAKey() subroutine. }
                   { Note that the ordering of this list is unrelated to    }
                   { the 16-bit integer Termination Key values that Synergy }
                   { uses.  (This ordering is purely for the convenience of }
                   { the sample program.)                                   }

GTIMdpbType      = PACKED RECORD                 { Directive Parameter Block }
                     D_CODE: [Pos(0,0)] 0..255; { for the Get Time (GTIM)   }
                     D_LGTH: [Pos(1,0)] 0..255; { P/OS Executive Directive. }
                     G_TIBA: [Pos(2,0),Unsafe] Unsigned;
                   END { GTIMdpbType } ;

GTIM$BUF         = PACKED RECORD  { Buffer format used by GTIM directive }
                     G_TIYR: [Pos(0,0)]  Unsigned; { Year   }
                     G_TIMO: [Pos(2,0)]  Unsigned; { Month  }
                     G_TIDA: [Pos(4,0)]  Unsigned; { Day    }
                     G_TIHR: [Pos(6,0)]  Unsigned; { Hour   }
                     G_TIMI: [Pos(8,0)]  Unsigned; { Minute }
                     G_TISC: [Pos(10,0)] Unsigned; { Second }
                     G_TICT: [Pos(12,0)] Unsigned; { Clock ticks of second  }
                     G_TICP: [Pos(14,0)] Unsigned; { Number of ticks/second }
                   END { GTIM$BUF } ;
```

```
{ Here are the data structure VARiable allocations specific to this module }

VAR

  ContextBlock: RECORD
                  CASE Integer OF
                  1: ( C: PACKED ARRAY [ 1..MaxContext ] OF Char; );
                  2: ( R: RECORD
                            WindowX:        Integer;  { Position of the window }
                            WindowY:        Integer;
                            WindowWidth:    Integer;  { Size of the window }
                            WindowHeight:   Integer;
                            CPoints:        Integer;  { How many and what size }
                            CBatons:        Integer;  { Batons the user wants. }
                            SavedColor1:    PackedColorType;  { Values for the }
                            SavedColor2:    PackedColorType;  { entries in the }
                            SavedColor3:    PackedColorType;  { color map that }
                            SavedColor5:    PackedColorType;  { the program    }
                            SavedColor6:    PackedColorType;  { allows the     }
                            SavedColor7:    PackedColorType;  { user to change.}
                            LineThickness: PackedBytes;  { Thickness of lines }
                          END { variant R } );
                END { ContextBlock } ;
    { Allocate the 32-byte Context Block that Synergy keeps for us.  The      }
    { application can store ANYTHING it wants in these 32 bytes.  Almost      }
    { always, you will want to store the X,Y coordinates of your window(s),   }
    { plus the width(s) and height(s).  Beyond that, it's up to you.  This    }
    { sample application uses some of the remaining space to retain program   }
    { variables which the end-user has control of.  This way, the values of   }
    { those variables are preserved one run of the application to the next,   }
    { without the application having to create a file on the disk and store   }
    { the values there, etc.                                                  }

  XStatus: StatusBlock;  { Array used to return success/failure status }
                         { from the various Synergy service calls.     }

  WDesc: WindowDescriptor;  { 32-byte Window Descriptor Block that is   }
                            { used to manage the program's main window. }

  KBBuf: [External,Volatile] Queue;  { Special keyboard buffer GetAKey() uses }

  InputKey:  FunctionKeyNames;  { These two varaibles are used when calling   }
  InputChar: Char;              { the GetAKey() routine to read the keyboard. }

  MainScreenRequested,  { Becomes True if burried MAIN SCREEN is pressed }
  SuspendRequested,     { Becomes True if burried F5 is pressed }
  Done: Boolean;        { Loop control logic variable }

  TitleText: String;  { Buffer, used to read message frames from frame file }
```

```
  ContextLength,  { Used in WIINI/WIDON Synergy calls.                        }
  TitleLength,    { Holds the length of the string TitleText.                 }
  TitleID,        { Holds the frame ID of the frame used for the title string.}
  TitleCounter,   { Counter, keeps track of # of times main program loop runs.}
  i,              { General FOR loop counter for main program body.           }
  Index,          { Another FOR loop counter for main program body.           }
  ColorMapIndex,  { Another FOR loop counter for main program body.           }
  Points,         { How many segments there are per Baton.                    }
  Batons,         { How many Batons there are in the window.                  }
  RndSeed:        { Used as seed for a random number generator.               }
    Integer;

  ColorMap: ColorMapType;  { The program's values for the hardware color map }

  BatonList: ARRAY [ 1..MaxBatons ] OF A_Baton;
    { List of each of the Batons that are moving in the window }



{ Here are the external procedure definitions for this module }

[External(GETMES)]
PROCEDURE GetMessage7( VAR FrameID: [ReadOnly] Integer;
                       VAR Msg1:    [Unsafe]   StringType;
                       VAR Msg2:    [Unsafe]   StringType;
                       VAR Msg3:    [Unsafe]   StringType;
                       VAR Msg4:    [Unsafe]   StringType;
                       VAR Msg5:    [Unsafe]   StringType;
                       VAR Msg6:    [Unsafe]   StringType;
                       VAR Msg7:    [Unsafe]   StringType ); SEQ11;

[External(GETMES)]
PROCEDURE GetMessage9( VAR FrameID: [ReadOnly] Integer;
                       VAR Msg1:    [Unsafe]   StringType;
                       VAR Msg2:    [Unsafe]   StringType;
                       VAR Msg3:    [Unsafe]   StringType;
                       VAR Msg4:    [Unsafe]   StringType;
                       VAR Msg5:    [Unsafe]   StringType;
                       VAR Msg6:    [Unsafe]   StringType;
                       VAR Msg7:    [Unsafe]   StringType;
                       VAR Msg8:    [Unsafe]   StringType;
                       VAR Msg9:    [Unsafe]   StringType ); SEQ11;
  { Both of the above declarations refer to the same procedure.  It is       }
  { written in MACRO to use the PDP-11 R5 calling sequence (SEQ11), and is in }
  { the module READMESG.MAC.                                                 }


FUNCTION GetAKey( VAR CharPressed: Char ) : FunctionKeyNames; EXTERNAL;
  { This routine is written in MACRO to use the PRO/Pascal FUNCTION calling  }
  { sequence, and is in the module GETAKEY.MAC.                              }
```

```
PROCEDURE UnlKeyboard;      EXTERNAL;
PROCEDURE KBASTInitialize; EXTERNAL;
  { These routines are written in MACRO to use the PRO/Pascal PROCEDURE       }
  { calling sequence, and are in the module KBSERV.MAC.                       }


PROCEDURE ExitStatus( Status: Integer ); EXTERNAL;
PROCEDURE Detach;                        EXTERNAL;
  { These routines are contained in the PRO/Pascal Cluster Library, PASRES.   }
  { They may be called from any PRO/Pascal program.                           }




{-----------------------------------------------------------------------------}
{-------------- Now begins the executable code for this module ---------------}


  FUNCTION Rnd : Integer;

  { Simple random number generator.  As used by this program, it returns      }
  { numbers in the range 0..1400.  RndSeed is a global variable.              }

  BEGIN { Rnd }

    RndSeed := ( RndSeed * 13077 + 6925 ) MOD 32768;
    Rnd     := UAND( RndSeed, %0'77777' ) DIV 40

  END { Rnd } ;



  PROCEDURE ReadMessage( FrameID:              Integer;
                         VAR Length:           Integer;
                         VAR Message: [Unsafe] String );

  { ReadMessage reads the specified message frame, returning the first line   }
  { of it in a text string.                                                   }

  VAR
    Offsets: ARRAY [ 1..9 ] OF Integer;
    XStatus: StatusBlock;
    NumLines: Integer;

  BEGIN { ReadMessage }

    WIRMS( XStatus, NumLines, Offsets, Message, FrameID, StringMax );

    Length := Offsets[ 2 ];  { This is the length of the first line }

  END { ReadMessage } ;
```

```
[External(GETMES)]
PROCEDURE GetMessage2( VAR FrameID: [ReadOnly] Integer;
                       VAR Msg1:    [Unsafe] StringType;
                       VAR Msg2:    [Unsafe] StringType ); SEQ11;
{ This definition is needed for the following procedure.                    }


  [Global(FTLERR)]
  PROCEDURE FatalError( FetchHeaders: Boolean;
                        Length: Integer;
                        VAR Message: [ReadOnly,Unsafe] String );

  { FatalError is called to abort the running program whenever it encounters  }
  { any surprise, serious error situation that it cannot cope with/recover    }
  { from.                                                                     }
  { 'FetchHeaders' should be passed as True in the normal case; this means    }
  { that this routine should read some intro header messages from the frame   }
  { file to introduce what has happened to the program.  If 'FetchHeaders' is }
  { passed as false then instead of getting the intro messages from the frame }
  { file, hard-coded strings are used instead.  This is needed in case the    }
  { frame file hasn't been opened at the point the fatal error occurs.        }

  CONST
    JunkString = '_';  { Any random string used as a stub }

  VAR
    Intro1, Intro2: StringType;

  BEGIN { FatalError }

    FlushGIDIS;  { Empty the output buffers }

    IF FetchHeaders
     THEN
       BEGIN  { Get the intro text from the frame file }
         GetMessage2( FERROR, Intro1, Intro2 );
         WIERW( XStatus,                { Use the Synergy Error Window as   }
                Intro1.L, Intro1.S,     { a fail-safe mechanism for getting }
                Intro2.L, Intro2.S,     { some information on the screen.   }
                0, JunkString,
                Length, Message,  { Specific message goes here }
                0, JunkString );
       END { then }

     ELSE  { Need to use the hard-coded strings }
       WIERW( XStatus,
              40, '    This application has encountered    ',
              40, '  the following unexpected problem --   ',
              0, JunkString,
              Length, Message,  { Specific message goes here }
              0, JunkString );

    ExitStatus( 1 );  { Exit the program }

  END { FatalError } ;
```

```
[Global]
PROCEDURE Error( ErrorClass, ErrorNumber, ErrorMsgLength: Integer;
                 VAR ErrorMsg: String;
                 VAR XFile: Text;
                 IOStatus, UserPC: Integer;
                 FileNameLength: Integer;
                 VAR FileName: String );

{ Error is a substitute for the PRO/Pascal run-time system Error handler    }
{ module.  This routine is called whenever some "catastrophic" error        }
{ condition is encountered, for example failed file I/O operations that do  }
{ not specify ERROR=CONTINUE; memory protect traps; odd-address traps; etc. }
{ Most applications should provide their own error handler, since the one   }
{ that PRO/Pascal provides is very large (takes address space), plus the    }
{ information that it displays when an error occurs is generally of little  }
{ help to the user running the application (sort of a register dump).  Your }
{ error handler can display a more friendly message, directing the user to  }
{ some documentation or whatever.                                           }

VAR
  Length: Integer;
  Data: String;

BEGIN { Error }

  ReadMessage( OTSERR, Length, Data );  { Read the error text }
  FatalError( True, Length, Data );  { Display the text & kill the task }

END { Error } ;



[Global] PROCEDURE PVDetach;

{ PVDetach detaches the keyboard, disabling the AST-input mechanism.  Once  }
{ detached, any extra read-ahead input is copied from our private buffer    }
{ (KBBUF) into the Synergy type-ahead buffer (WITBF) so that it is not lost.}

VAR
  i: Integer;   Ch: Char;

BEGIN { PVDetach }

  FlushGIDIS;  { Make sure any buffered output is put out }
  Detach;  { Detach the terminal }
  UnlKeyboard;  { Make sure the keyboard is unlocked }

  WITBF.CurrentLength := KBBUF.Count;  { Copy data from private to public }
  FOR i := 1 TO KBBUF.Count DO
    BEGIN
      RemoveFromQueue( KBBUF, Ch );  { Get next character from KBBUF }
      WITBF.Characters[ i ] := Ch;   { Append it to Synergy WITBF    }
    END { for } ;

END { PVDetach } ;
```

```
PROCEDURE WriteString( VAR Str: [ReadOnly,Unsafe] StringType );

{ WriteString takes a passed string.  This string is then written to the    }
{ screen, taking into account any rendition control sequences embedded in   }
{ the string to change the font in mid-string.                              }
{ NOTE: Normally, the application does NOT have to be concerned with doing  }
{ the grunt-work for rendition control sequences.  That is, whenever you    }
{ call some Synergy service to display some text Synergy will automatically }
{ "do the right thing" to invoke your desired renditions.  You ONLY need to }
{ do it yourself if YOU are going to do the GIDIS output of a string with   }
{ rendition sequences in it to a window yourself                            }

VAR
  Ch: Char;
  Change, Rendition, i: Integer;


  PROCEDURE UseRendition( Rend: Integer );

  { UseRendition is passed a rentition word (16-bits), and does the neces-  }
  { sary things to cause subsequent character output to be in the specified }
  { rendition.                                                              }

  VAR
    WritingMode: WritingModes;
    Intensity, Italics, Alphabet: Integer;

  BEGIN { UseRendition }

    Italics     := 0;           { To start with, assume no italics }
    Alphabet    := WI$Normal;   { Assume normal font               }
    WritingMode := Replace;     { Assume "positive video"          }

    Intensity   := UAND( Rend, 3 );  { Pick out 2 low-order intensity bits }

    IF UAND( Rend, 8 ) = 0
     THEN  { Underline NOT wanted }
       CASE Intensity OF
         0:   Alphabet := WI$Dim;
         1,2: Alphabet := WI$Normal;
         3:   Alphabet := WI$Bold;
       END { case }

     ELSE  { Underline IS wanted }
       CASE Intensity OF
         0,   { Dim-underline is not supported, so use normal }
         1,2: Alphabet := WI$Underline;
         3:   Alphabet := WI$BoldUnderline;
       END { case } ;
```

```
    IF UAND( Rend, 4 ) <> 0
     THEN
       Italics := 2;                    { Use italics   }
    IF UAND( Rend, 16 ) <> 0
     THEN
       WritingMode := ReplaceNegate;    { Inverse Video }
    IF UAND( Rend, 32 ) <> 0
     THEN
       Alphabet := WI$Boxed;            { Boxed font    }

    SetCellRendition( Italics     );  { Now that we know what }
    SetAlphabet(      Alphabet    );  { is wanted, do it.     }
    SetWritingMode(   WritingMode );

  END { UseRendition } ;


BEGIN { WriteString }

  i         := 0;
  Rendition := 2;  { To start with, we want plain, normal intensity text }
  UseRendition( Rendition );  { Select normal rendition }

  WITH Str DO
    WHILE i < L DO
      BEGIN  { Examine (and maybe display) the next character }
        i  := i + 1;
        Ch := S[ i ];  { Get the next character }

        IF Ch <> Chr( 28 )
         THEN
           PutChr( Ch )  { A normal character, print it }
         ELSE
           BEGIN  { The start of a rendition string }
             Change := 0;
             Ch     := '0';  { Prime the pump }

             REPEAT  { Start scanning digit sequence }
               Change := ( Change * 10 ) + Ord( Ch ) - Ord( '0' );
               i      := i + 1;  { Skip to next byte }
               Ch     := S[ i ];  { Get the next character }
             UNTIL ( Ch < '0' ) OR ( Ch > '9' );

             IF Ch = '+'
              THEN Rendition := UOR(  Rendition,       Change   ) {Additive}
              ELSE Rendition := UAND( Rendition, UNOT( Change ) );

             UseRendition( Rendition );  { Invoke new resulting rendition }
           END { else } ;
      END { while } ;

END { WriteString } ;
```

```
FUNCTION CountString( VAR Str: [ReadOnly,Unsafe] StringType ) : Integer;

{ CountString is passed a string record.  It returns the PRINTING length of }
{ the string, once the rendition control sequences to change fonts have      }
{ been accounted for.  (Rendition control sequences start with an ASCII 28,  }
{ then have some digits, then have a "+" or "-".  They are described in the  }
{ Synergy programming manual.)                                               }

VAR
  Ch: Char;       Len, i: Integer;

BEGIN { CountString }

  Len := 0;
  i   := 0;

  WITH Str DO
    WHILE i < L DO
      BEGIN  { Examine the next character }
        i  := i + 1;
        Ch := S[ i ];  { Get the next character }
        IF Ch <> Chr( 28 )
         THEN
           Len := Len + 1  { Normal text character, count it }
         ELSE
           REPEAT  { Start skipping over the rendition sequence }
             i  := i + 1;  { Skip to next character }
             Ch := S[ i ];  { Get the next character }
           UNTIL ( Ch = '+' ) OR ( Ch = '-' );
      END { while } ;

  CountString := Len;  { Pass out the resultant length }

END { CountString } ;


PROCEDURE FillText( NumberOfStrings: Integer;
                    VAR Msg: [ReadOnly,Unsafe] StringArray );

{ FillText simply takes an array of string records, and displays them one    }
{ below the other in the current window.  The strings are scanned to see if  }
{ they contain rendition control sequences and are displayed appropriately.  }

VAR
  i: Integer;

BEGIN { FillText }

  FOR i := 1 TO NumberOfStrings DO
    BEGIN  { Display the next string inside the window }
      SetPosition( 2 * CharacterWidth,
                   (i - 1) * CharacterHeight + ( CharacterHeight DIV 2 ) );
      WriteString( Msg[ i ] );  { Draw the string in its proper position }
    END { for } ;

END { FillText } ;
```

```
PROCEDURE InitializeSyn;

{ This sets up things for the rest of the program.                    }

VAR
  GTIMdpb: GTIMdpbtype;  { Directive Parameter Block for the GTIM directive }
  TimeBuffer: GTIM$BUF;  { Time buffer used by the Get Time P/OS Directive }

BEGIN { InitializeSyn }

  TitleID       := TITLE;  { Assume the normal title string }
  ContextLength := MaxContext;  { Ask for the max }

  WIINI( XStatus, 2, ActualVersion,        { Call Synergy, telling it that  }
         ContextLength, ContextBlock,      { this application has just      }
         ScreenWidth, ScreenHeight,        { started and will be requesting }
         CharacterWidth, CharacterHeight,  { other services.                }
         PixelWidth, PixelHeight,
         ColorWindows );

  WITH ContextBlock.R DO
    BEGIN  { What's in the Context Block from the last time we ran? }

      IF ContextLength = 0
       THEN     { This is the first time this application has run.  In this }
         BEGIN  { case, there is no saved context from before, so we must   }
                { do a one-time initialization of the context data.         }
           TitleID           := TITLE2;  { Special intro title }
           WindowWidth       := ScreenWidth  DIV 2;  { Make beginning size }
           WindowHeight      := ScreenHeight DIV 2;  { of half-screen.     }
           WindowX           := -32763;  { We want the window in     }
           WindowY           := -32763;  { the middle of the screen. }
           CBatons           := 1;    { Start with one Baton,        }
           CPoints           := 17;  { With a tail 17 segments long. }
           SavedColor1.R     := ColorDescriptor( 0, 0, 7 );  { Set up nice }
           SavedColor2.R     := ColorDescriptor( 0, 7, 0 );  { color map   }
           SavedColor3.R     := ColorDescriptor( 7, 7, 0 );  { in case     }
           SavedColor5.R     := ColorDescriptor( 7, 5, 0 );  { COLOR is    }
           SavedColor6.R     := ColorDescriptor( 7, 0, 0 );  { usable.     }
           SavedColor7.R     := ColorDescriptor( 0, 2, 3 );
           LineThickness.Low := PixelWidth;  { Lastly, choose to draw the }
           LineThickness.Hi  := PixelHeight; { segments with thin lines.  }
         END { then } ;

      Batons          := CBatons;  { Now copy permanent values to run-time }
      Points          := CPoints;  { variables for duration of this run.   }
      ColorMap[ 0 ].R := ColorDescriptor( 0, 0, 0 );  { This index is BLACK }
      ColorMap[ 1 ]   := SavedColor1;
      ColorMap[ 2 ]   := SavedColor2;
      ColorMap[ 3 ]   := SavedColor3;
      ColorMap[ 4 ].R := ColorDescriptor( 7, 7, 7 );  { This index is WHITE }
      ColorMap[ 5 ]   := SavedColor5;
      ColorMap[ 6 ]   := SavedColor6;
      ColorMap[ 7 ]   := SavedColor7;
    END { with } ;
```

```
  OPENME( XStatus, $FCTV$, 25, 'LB:[ZZBATON]BATONFRMS.OFF' ); { Open file }
  IF XStatus[ 1 ] <> 1
   THEN
     FatalError( False, 32, '      Can''t open our frame file.' );  { Death }
  { The frame file is stored in LB:[ZZBATON] so that, in P/OS V3 Cluster }
  { configurations, the .OFF file can be shared among multiple users.    }

  WITH GTIMdpb DO
    BEGIN  { Set up the Directive Parameter Block for the GTIM$ directive }
      D_CODE := GTIM;       { Directive Identification Code. }
      D_LGTH := GTIM_Len;   { Parameter Block Length.        }
      G_TIBA := Address( TimeBuffer );  { One Parameter (buffer address) }
    END { with } ;

  DIR$( GTIMdpb );  { Issue the directive, find out the current time }
  WITH TimeBuffer DO  { Use low-order time values to init the RND seed }
    RndSeed := UOR( UOR( G_TIMI * 37, G_TISC * 71 ), G_TICT * 293 );

END { InitializeSyn } ;



PROCEDURE CleanUpSyn;

{ CleanUpSyn does whatever housekeeping is necessary before we leave.       }

BEGIN { CleanUpSyn }

  PVDetach;  { Free up the keyboard }

  WIGEW( XStatus, DescriptorLength, WDesc );  { Get current size & position }

  WITH WDesc, ContextBlock.R DO
    BEGIN  { Load the data that is to be saved in the context block }
      WindowX       := X;               { Remember the window  }
      WindowY       := Y;               { size and position in }
      WindowWidth   := Width;           { the context block.   }
      WindowHeight  := Height;
      CBatons       := Batons;          { Also save the user-      }
      CPoints       := Points;          { settable program values. }
      SavedColor1   := ColorMap[ 1 ];
      SavedColor2   := ColorMap[ 2 ];
      SavedColor3   := ColorMap[ 3 ];
      SavedColor5   := ColorMap[ 5 ];
      SavedColor6   := ColorMap[ 6 ];
      SavedColor7   := ColorMap[ 7 ];
    END { with } ;

  WIDSW( XStatus, WDesc.ID );  { Destroy the main window }
  WIDON( XStatus, MaxContext, ContextBlock );  { Tell Synergy we're through }

END { CleanUpSyn } ;
```

```
PROCEDURE MakeOurWindow;

{ MakeOurWindow simply creates a window for the application, and gives it   }
{ a title.                                                                   }

VAR
  Length: Integer;
  Data: String;  { Used to hold a possible error message string }

BEGIN { MakeOurWindow }

  WITH WDesc, ContextBlock.R DO
    BEGIN  { Fill in the window descriptor block }
      WDesc                := WDescModel;  { Init the bulk of it }
      X                    := WindowX;  { Then fill in the specifics }
      Y                    := WindowY;
      Width                := WindowWidth;
      Height               := WindowHeight;
      Flags.Color          := True;  { We'll use color, if the user lets us }
      Flags.Titled         := True;
      Flags.WhiteBorder    := True;
      Flags.ClearOnChange  := True;
    END { with } ;

  WICRW( XStatus, DescriptorLength, WDesc );  { Create the main window }

  IF XStatus[ 1 ] <> 1
   THEN
     BEGIN  { Yipes!  We got an error creating the window }
       ReadMessage( NOWIND, Length, Data );  { Get error text }
       FatalError( True, Length, Data );  { Display the text & kill task }
     END { then } ;

  WITH WDesc DO
    BEGIN  { Assign the minimum and maximum range for the window size }
      MinWidth  := 100;  { The minimum window   }
      MinHeight := 100;  { size is pretty tiny. }
      MaxWidth  := ScreenWidth;   { The Maximum is   }
      MaxHeight := ScreenHeight;  { the full screen. }
    END { with } ;

  WISWP( XStatus, DescriptorLength, WDesc );  { Tell the values to Synergy }

  ReadMessage( TitleID, TitleLength, TitleText );  { Read a message frame }

  WITTL( XStatus,       TitleLength, TitleText );  { Set the window title }

END { MakeOurWindow } ;
```

```
PROCEDURE InitBaton;

{ This procedure initializes the program-specific data structures.          }

VAR
  Segment, ThisBaton, Inner: Integer;  { Some FOR loop counters and such }

BEGIN { InitBaton }

  IF ColorWindows <> 0
   THEN
     BEGIN  { Color is in use, so initialize the colors }
       ColorMapIndex := 1;

       FOR Inner := 1 TO 7 DO
         IF Inner <> 4  { Leave index 4 (WHITE) alone }
          THEN
            SetColorMap( Inner, ColorMap[ Inner ].R.Red,
                                ColorMap[ Inner ].R.Green,
                                ColorMap[ Inner ].R.Blue );
     END { then } ;

  WITH ContextBlock.R.LineThickness DO
    SetPixelSize( Low, Hi, Low DIV 2, Hi DIV 2 );
  WITH WDesc DO
    SetOutputClippingRegion( 0,0, Width - PixelWidth, Height - PixelHeight );

  SetWritingMode( Replace );
  SetPrimaryColor(   7 );  { Use index 7 for the window background }
  SetSecondaryColor( 7 );

  EraseClippingRegion;  { Clear out window, fill it with background color }

  SetPrimaryColor(   4 );  { Go back to drawing in white }
  SetSecondaryColor( 4 );

  FOR ThisBaton := 1 TO Batons DO  { Initialize each of the Batons }
    WITH BatonList[ ThisBaton ] DO
      BEGIN

        IF ColorWindows = 0
         THEN
           Hue := 0  { Monochrome, draw them in the black colormap entry }
         ELSE
           BEGIN  { Color is present and enabled }
             Hue := UAND( Rnd, 7 );  { Pick a random starting color }

             IF ( Hue >= MaxColor ) OR ( Hue = 0 ) OR ( Hue = 4 )
              THEN
                Hue := 1;  { Make sure that it is a legal color index }
           END { else } ;

        Newest := 0;
```

```
        FOR Inner := 0 TO 3 DO
          BEGIN  { Init the X,Y speeds of both end-points of first segment }
            Speed[ Inner ] := ( Rnd DIV 44 ) + 4;  { Pick starting speed }

            IF Rnd > 450
             THEN
               Speed[ Inner ] := -Speed[ Inner ];  { Randomly flip the sign }

            REPEAT
              Sticks[ 0, Inner ] := Rnd;  { Pick starting X,Y coordinates }
            UNTIL ( Sticks[ 0, Inner ] < ( WDesc.Width  DIV 2 ) )
              AND ( Sticks[ 0, Inner ] < ( WDesc.Height DIV 2 ) )
              AND ( Sticks[ 0, Inner ] > ( 20                 ) );
          END { for } ;

        FOR Segment := 1 TO Points DO  { Propagate base stick into others }
          FOR Inner := 0 TO 3 DO
            Sticks[ Segment, Inner ] := Sticks[ 0, Inner ]

      END { with } ;

END { InitBaton } ;



PROCEDURE Travel( VAR Node, Direction: Integer;
                  WhichBaton, Index, Max: Integer );

{ Travel is called to move one part of a Baton one cycle along its track.   }

BEGIN { Travel }

  WITH BatonList[ WhichBaton ] DO
    BEGIN
      Node := Node + Direction;  { Add position + speed to get new position }

      IF ( Node <= 0 ) OR ( Node >= Max )  { If the Baton hits edge, }
      OR ( Abs( Sticks[ Newest, Index ]    { or it gets too long.    }
             - Sticks[ Newest, UAND( Index + 2, 3 ) ] ) > Max * 3 DIV 5 )
       THEN
         BEGIN  { Reflect this node (bounce it) }
           Direction := -Direction;
           Node      := Node + ( 2 * Direction );
         END { then } ;
    END { with }

END { Travel } ;
```

```
PROCEDURE TwiddleBaton( WhichBaton: Integer );

{ This procedure moves a complete Baton by one step.                    }

VAR
  PriorSegment, Inner: Integer;

BEGIN { TwiddleBaton }

  WITH BatonList[ WhichBaton ] DO
    BEGIN
      IF ColorWindows <> 0  { I.e. on a Color system }
       THEN
         BEGIN  { Roll colors around, so as Baton moves the colors follow }
           Hue := Hue + 1;
           IF Hue = 4
            THEN
              Hue := 5;  { Skip index 4 }
           IF Hue >= MaxColor
            THEN
              Hue := 1;
         END { then } ;

      SetPrimaryColor( 7 );  { Erase oldest segment by redrawing background }
      SetPosition( Sticks[ Newest, 0 ], Sticks[ Newest, 1 ] );
      DrawLine(    Sticks[ Newest, 2 ], Sticks[ Newest, 3 ] );

      PriorSegment := Newest - 1;
      IF PriorSegment < 0
       THEN
         PriorSegment := PriorSegment + Points + 1;

      FOR Inner := 0 TO 3 DO  { Copy 2nd segment into 1st segment }
        Sticks[ Newest, Inner ] := Sticks[ PriorSegment, Inner ];

      WITH WDesc DO
        BEGIN  { Now move X,Y coords of both ends of the first segment }
          Travel( Sticks[ Newest, 0 ], Speed[ 0 ], WhichBaton, 0, Width  );
          Travel( Sticks[ Newest, 1 ], Speed[ 1 ], WhichBaton, 1, Height );
          Travel( Sticks[ Newest, 2 ], Speed[ 2 ], WhichBaton, 2, Width  );
          Travel( Sticks[ Newest, 3 ], Speed[ 3 ], WhichBaton, 3, Height );
        END { with } ;

      SetPrimaryColor( Hue );  { Draw the new (1st) segment in its color }
      SetPosition( Sticks[ Newest, 0 ], Sticks[ Newest, 1 ] );
      DrawLine(    Sticks[ Newest, 2 ], Sticks[ Newest, 3 ] );

      Newest := Newest + 1;  { Remember which segment will be in front }
      IF Newest > Points     { next time, checking for wrap-around.    }
       THEN
         Newest := 0
    END { with }

END { TwiddleBaton } ;
```

```
PROCEDURE EnableColors( VAR NewMap: [ReadOnly] ColorMapType );

{ EnableColors simply refreshes the hardware color map with the current     }
{ programmed color choices.                                                 }

VAR
  i: Integer;  { FOR loop counter }

BEGIN { EnableColors }

  FOR i := 1 TO 7 DO
    IF i <> 4  { Skip index 4, leave it as WHITE }
     THEN
       WITH NewMap[ i ].R DO
         SetColorMap( i, Red, Green, Blue );

END { EnableColors } ;



PROCEDURE Hibernate;

{ Hibernate is called when the user presses F5; the WIINT Synergy service   }
{ is used to suspend the application.  This procedure is responsible for    }
{ knowing what to do if the end-user ever changes the window size.  Usually }
{ this means refreshing the window based on its new width and height.  In   }
{ this program, this is accomplished by merely re-initializing the data     }
{ structures used to contain the X,Y position of the Baton segments.  This  }
{ is done in case the window is reduced in size, such that part or all of a }
{ Baton segment might fall outside the window (there is no reason why this  }
{ cannot be allowed to happen, as GIDIS is fully capable of clipping the    }
{ "overhanging" segments; this program simply chose not to allow it).       }

VAR
  WindowID, NewWidth, NewHeight, WhyChange: Integer;

BEGIN { Hibernate }

  PVDetach;  { Free up the keyboard for Synergy and other applications }

  REPEAT
    FlushGIDIS;  { Force out any buffered GIDIS output }
    WIINT( XStatus, WhyChange, WindowID, NewWidth, NewHeight );  { Suspend }
    IF WhyChange = 1
     THEN
       BEGIN  { Window size was changed, update its contents }
         WIGEW( XStatus, DescriptorLength, WDesc );  { Get size & position }
         InitBaton;  { Re-init the data structures to fit the new size }
       END { then } ;
  UNTIL WhyChange = 0;  { 0 = means the application should resume execution }

  KBASTInitialize;  { Get the keyboard back for AST input }
  EnableColors( ColorMap );  { Refresh the color map to our colors }

END { Hibernate } ;
```

```
PROCEDURE HandleKey;

{ This (large) procedure processes all typed input (function keys) from the }
{ keyboard.                                                                 }


  PROCEDURE HandleFCMKey( InputKey: FunctionkeyNames );

  { HandleFCMKey is called when either F11, F12, F13, or ADDTNL OPTIONS is  }
  { pressed, with the intent being to display the proper leaf in the        }
  { application's main Flow Control Menu.  As it turns out, this            }
  { application only has one leaf, but the code doesn't know that.  It is   }
  { written such that if the frame file were changed to include more leaves }
  { the code would work with them automatically.                            }

  VAR
    MenuID,
    OptionValue,
    KeyPressed: Integer;


    PROCEDURE HandlePersonalizeKey;

    { HandlePersonalizeKey is called when the user chooses the "Change      }
    { Batons" option from the main Flow Control Menu.  What this procedure  }
    { does is to display a setup menu (sometimes called a property sheet)   }
    { to solicit new program values from the user at the keyboard.          }

    VAR
      NumChanged, KeyPressed: Integer;
      ChangedValues: PACKED ARRAY [ 1..2 ] OF 0..255;

    BEGIN { HandlePersonalizeKey }

      WIXPS( XStatus, KeyPressed,    { Display the setup menu }
             NumChanged, ChangedValues,
             DEFQ, 2, 4, Batons, 4, Points );

     IF KeyPressed = PV$F5
      THEN SuspendRequested := True  { Set flag, we'll get it next time }
      ELSE
        IF KeyPressed = PV$MAI
         THEN MainScreenRequested := True
         ELSE
           IF NumChanged > 0
            THEN
              BEGIN  { One or more were changed, so update things }
                IF Batons < 1
                 THEN
                   Batons := 1;  { Do our own bounds checking on the values}
                IF Batons > MaxBatons
                 THEN
                   Batons := MaxBatons;
```

```
            IF Points < 3
             THEN
               Points := 3;
            IF Points > MaxPoints
             THEN
               Points := MaxPoints;

            InitBaton;  { Re-start the new Batons (data structures) }
          END { then } ;

END { HandlePersonalizeKey } ;
```

```
PROCEDURE HandleColorMap;

{ HandleColorMap is called when the user chooses the "Color map" option }
{ from the main Flow Control Menu.  What this procedure does is to      }
{ create and process its own custom, special-purpose interface that     }
{ allws the end user at the keyboard to change the colors in the color  }
{ map.                                                                  }

VAR
  FunctionKey: FunctionKeyNames;
  NewMap: ColorMapType;
  Ch: Char;
  CurSiz, Gap, IndentX, IndentY,
  PriorIndex, ColorIndex: Integer;
  MapWindow: WindowDescriptor;


  PROCEDURE PutCursor;

  { PutCursor positions a semi-graphical cursor on the initial window   }
  { used in changing the color map.  (Initial means the horizontal      }
  { strip of six squares for the six colors that can be changed.)       }
  { The semi-graphical cursor is constructed as follows:  First, a      }
  { heavy black line is drawn around the square that the cursor is      }
  { positioned on.  Then, just inside of that heavy line, the real      }
  { blinking GIDIS Rubber Band cursor is placed.  The net effect is of  }
  { a large hollow square cursor, whose inner edge is blinking.         }

  CONST
    Pixel = 10;  { Thickness of the line for the bulk of the cursor }

  VAR
    NewIndex: Integer;


    PROCEDURE Draw( Index: Integer );

    { This procedure draws the non-blinking first part of the cursor,   }
    { the heavy hollow square.                                          }

    BEGIN  { Draw }

      SetPosition( (Index-1)*(CurSiz+Gap) + IndentX - ( Pixel DIV 2 ),
                   IndentY - ( Pixel DIV 2 ) );
      DrawRelLine(  (Pixel + CurSiz + PixelWidth), 0   );
      DrawRelLine(  0,  (Pixel + CurSiz + PixelHeight) );
      DrawRelLine( -(Pixel + CurSiz + PixelWidth), 0   );
      DrawRelLine(  0, -(Pixel + CurSiz + PixelHeight) );

    END { Draw } ;
```

```
BEGIN { PutCursor }

  SetPixelSize( Pixel, Pixel, Pixel DIV 2, Pixel DIV 2 );  { Fat line }
  SetWritingMode( Erase );  { Prepare to erase any existing cursor }

  IF PriorIndex <> 0
   THEN
     Draw( PriorIndex );  { There was an existing cursor, erase it }

  NewIndex := ColorIndex;
  IF NewIndex > 3
   THEN
     NewIndex := NewIndex - 1;  { Slide upper part over index 4 }

  PriorIndex := NewIndex;  { Remember new cursor position next time }

  SetWritingMode( Replace );  { Prepare to draw the new cursor }
  SetOutputRubberBand( -1, 0, 0 );  { Disable the rubber band }

  IF NewIndex <> 0
   THEN
     BEGIN  { There is a cursor wanted, draw it in its new position }
       Draw( NewIndex );
       SetPosition( ( NewIndex - 1 ) * ( CurSiz + Gap )
                    + IndentX + CurSiz + PixelWidth,
                    IndentY + CurSiz + PixelHeight );
       SetOutputRubberBand( 2, ( NewIndex - 1 ) * ( CurSiz + Gap )
                               + IndentX - PixelWidth,
                            IndentY - PixelHeight );
     END { then } ;

END { PutCursor } ;
```

```
PROCEDURE ModifyEntry( Entry: Integer );

{ ModifyEntry manages the second, inner window portion of changing    }
{ the color map.  This inner window has three black squares, one each }
{ for RED, GREEN, BLUE.  This procedure allows the user to move       }
{ between the three color components, and increase or decrease their  }
{ respective intensities.                                             }

VAR
  FunctionKey: FunctionKeyNames;   Ch: Char;
  EditWindow: WindowDescriptor;    NewColor, OldColor: ColorDescriptor;
  Level, NewRGB, PriorRGB, CurSiz, Gap, IndentX, IndentY: Integer;


  PROCEDURE PutInnerCursor;

  { PutInnerCursor positions a semi-graphical cursor on the second    }
  { window used in changing the color map.  This cursor is produced   }
  { similarly to how the PutCursor procedure above produces the       }
  { cursor for the first, outer window in "Color map."                }

  CONST
    Pixel = 10;  { Thickness of the square line }


    PROCEDURE Draw( Index: Integer );

    BEGIN  { Draw }
      SetPosition( (Index-1)*(CurSiz+Gap) + IndentX - ( Pixel DIV 2 ),
                   IndentY - ( Pixel DIV 2 ) );
      DrawRelLine(  (Pixel + CurSiz + PixelWidth), 0   );
      DrawRelLine(  0,  (Pixel + CurSiz + PixelHeight) );
      DrawRelLine( -(Pixel + CurSiz + PixelWidth), 0   );
      DrawRelLine(  0, -(Pixel + CurSiz + PixelHeight) );
    END { Draw } ;


  BEGIN { PutInnerCursor }
    SetPixelSize( Pixel, Pixel, Pixel DIV 2, Pixel DIV 2 );  { Thick }
    SetWritingMode( Erase );  { Prepare to erase any existing cursor }
    IF PriorRGB <> 0
     THEN
       Draw( PriorRGB );  { There is an existing one, erase it }
    PriorRGB := NewRGB;
    SetWritingMode( Replace );  { Prepare to draw the new cursor }
    SetOutputRubberBand( -1, 0, 0 );  { Disable the rubber band }
    IF NewRGB <> 0
     THEN
       BEGIN  { There is a new one, draw it in its new position }
         Draw( NewRGB );
         SetPosition( (NewRGB-1)*(CurSiz+Gap) + IndentX + CurSiz,
                      IndentY + CurSiz );
         SetOutputRubberBand( 2, (NewRGB-1)*(CurSiz+Gap) + IndentX,
                                 IndentY );
       END { then } ;
  END { PutInnerCursor } ;
```

```
PROCEDURE DisplayEditWindow;

{ DisplayEditWindow creates the second, inner window used in "Color }
{ map."  This window has some instructions, followed by three black }
{ squares representing RED, GREEN, BLUE.                            }

CONST
  EditHeight = 7;  { Height of the inner window in text-lines }

VAR
  Msg: ARRAY [ 1..EditHeight ] OF StringType;
  RGBText: String;
  LongestString, Len, i: Integer;

BEGIN { DisplayEditWindow }

  GetMessage7( EDIMSG,
               Msg[ 1 ], Msg[ 2 ], Msg[ 3 ], Msg[ 4 ],
               Msg[ 5 ], Msg[ 6 ], Msg[ 7 ] );
  { Call GetMessage7 to read the instructional text from a message }
  { frame in the frame file.  GetMessage7 reads the entire frame   }
  { into a buffer, then returns the separate lines of the message. }

  LongestString := 0;
  FOR i := 1 TO EditHeight DO
    BEGIN  { Compute longest string (controls width of the window) }
      Len := CountString( Msg[ i ] );  { Find the displayed length }
      IF Len > LongestString
       THEN
         LongestString := Len;
    END { for } ;

  LongestString := LongestString + 4;  { Add spacers on each side }

  WITH EditWindow DO
    BEGIN  { Fill in the window descriptor block }
      EditWindow         := WDescModel;  { Init the bulk of it }
      Width              := LongestString      * CharacterWidth;
      Height             := ( EditHeight + 1 ) * CharacterHeight;
      X                  := -32760;  { Window centered }
      Y                  := ( 8 * CharacterHeight ) - 2048;
      Flags.Stackable    := True;
      Flags.Color        := True;
      Flags.Titled       := False;
      Flags.WhiteBorder  := True;
    END { with } ;

  FlushGIDIS;  { Flush any pending output to first, outer window }

  WICRW( XStatus, DescriptorLength, EditWindow );  { Create it }

  FillText( EditHeight, Msg );  { Write the text strings to it }
```

```
    ReadMessage( RGBMSG, i, RGBText );  { 3 characters long }
    SetAlphabet( WI$Bold );
    SetCellRendition( 2 );  { Italics }

    FOR i := 1 TO 3 DO
      BEGIN  { Display the 3 squares for R, G, B }
        SetPosition( (i-1)*(CurSiz+Gap) + IndentX, IndentY );
        BeginFill;  { Fill in a square in black }
        DrawRelLine(  (CurSiz),  0         );
        DrawRelLine(  0,         (CurSiz) );
        DrawRelLine( -(CurSiz),  0         );
        DrawRelLine(  0,        -(CurSiz) );
        EndFill;

        SetPosition( (i-1)*(CurSiz+Gap) + IndentX + ( CurSiz DIV 2 ),
                     IndentY + ( CurSiz DIV 2 ) + PixelHeight
                     - ( CharacterHeight DIV 2 ) );
        SetWritingMode( ReplaceNegate );
        PutChr( RGBText[ i ] );  { Label the square "R", "G", or "B" }
        SetWritingMode( Replace );
      END { for } ;

    SetCellRendition( 0 );  { No italics }

  END { DisplayEditWindow } ;


BEGIN { ModifyEntry }

  OldColor := NewMap[ Entry ].R;  { Remember current RGB of color }
  NewColor := OldColor;  { Start the new RGB values with the current }
  NewRGB   := 1;  { The cursor will start on the RED square }
  PriorRGB := 0;  { There is no existing cursor }
  CurSiz   := CharacterWidth  * 4;  { Compute some layout values }
  Gap      := CharacterWidth  * 4;
  IndentX  := CharacterWidth  * 14 - ( PixelWidth * 3 );
  IndentY  := CharacterHeight * 4;

  SetOutputRubberBand( -1, 0, 0 );  { Disable it }
  DisplayEditWindow;  { Display the RGB window }

  REPEAT

    EnableColors( NewMap );  { Refresh color map with current choices }
    PutInnerCursor;  { Draw the cursor wherever it's supposed to be }

    FlushGIDIS;  { Flush pending output prior to any keyboard wait }
    KBASTInitialize;  { Attach the keyboard for AST input }
    FunctionKey := GetAKey( Ch );  { Read a keystroke }
    PVDetach;  { Detach.  This also flushes the GIDIS buffer }

    CASE NewRGB OF
      1:  Level := NewColor.Red;    { Remember the current level of   }
      2:  Level := NewColor.Green;  { whichever component (R,G,B) the }
      3:  Level := NewColor.Blue;   { cursor is on.                     }
    END { case } ;
```

```
    CASE FunctionKey OF  { Process the key that was just pressed }

      SuspendKey,
      MainScreenKey,
      Exit:          ; { Nothing to do right now }

      Right:         IF NewRGB = 3
                      THEN SoundBell  { Can't move right from BLUE }
                      ELSE NewRGB := NewRGB + 1;  { Move right a square }

      Left:          IF NewRGB = 1
                      THEN SoundBell  { Can't move left from RED }
                      ELSE NewRGB := NewRGB - 1;  { Move left a square }

      Up:            IF Level = 7
                      THEN SoundBell  { At maximum component saturation }
                      ELSE Level := Level + 1;  { Increase by a notch }

      Down:          IF Level = 0
                      THEN SoundBell  { At minimum component saturation }
                      ELSE Level := Level - 1;  { decrease by a notch }

      CancelKey:     NewColor := OldColor;  { Reset to original values }

      OTHERWISE      SoundBell;  { Ignore any other keys pressed }
    END { case } ;

    IF ( FunctionKey = Up ) OR ( FunctionKey = Down )
     THEN
       BEGIN  { The component saturation was changed, check it out }
         CASE NewRGB OF
           1:  NewColor.Red   := Level;  { Remember the new level for }
           2:  NewColor.Green := Level;  { whichever component        }
           3:  NewColor.Blue  := Level;  { (R,G,B) the cursor is on.  }
         END { case } ;

         NewMap[ Entry ].R := NewColor;  { Update color map with it }
       END { then } ;

  UNTIL ( FunctionKey = Exit          )  { Signals that the user is }
     OR ( FunctionKey = SuspendKey    )  { done changing this color }
     OR ( FunctionKey = MainScreenKey );{ index now.               }

  IF FunctionKey = SuspendKey
   THEN SuspendRequested := True  { Set flag, we'll get it next time }
   ELSE
     IF FunctionKey = MainScreenKey
      THEN MainScreenRequested := True;

  SetOutputRubberBand( -1, 0, 0 );  { Disable rubber band }
  FlushGIDIS;  { Flush any remaining output to this window }

  WIDSW( XStatus, EditWindow.ID );  { Destroy the window }

END { ModifyEntry } ;
```

```
PROCEDURE DisplayColors;

{ DisplayColors creates the first, outer window used in "Color map."  }
{ This window has some instructions, followed by six colored squares  }
{ representing the six colormap indexes that the user can change.     }

CONST
  WindowHeight = 9;  { Number of lines the window is tall }

VAR
  Msg: ARRAY [ 1..WindowHeight ] OF StringType;
  i, Len,
  LongestString: Integer;

BEGIN { DisplayColors }

  GetMessage9( COLMSG,
               Msg[ 1 ], Msg[ 2 ], Msg[ 3 ],
               Msg[ 4 ], Msg[ 5 ], Msg[ 6 ],
               Msg[ 7 ], Msg[ 8 ], Msg[ 9 ] );
  { Call GetMessage9 to read the instructional text from a message }
  { frame in the frame file.  GetMessage9 reads the entire frame   }
  { into a buffer, then returns the separate lines of the message. }

  LongestString := 0;
  FOR i := 1 TO WindowHeight DO
    BEGIN  { Compute longest string (determines the window width) }
      Len := CountString( Msg[ i ] );
      IF Len > LongestString
       THEN
         LongestString := Len;
    END { for } ;

  LongestString := LongestString + 4;  { Add spacers on each side }

  WITH MapWindow DO
    BEGIN  { Fill in the window descriptor block }
      MapWindow          := WDescModel;  { Init the bulk of it }
      Width              := LongestString        * CharacterWidth;
      Height             := ( WindowHeight + 1 ) * CharacterHeight;
      X                  := -32763;  { Screen centered }
      Y                  := ScreenHeight DIV 8;
      Flags.Stackable    := True;
      Flags.Color        := True;
      Flags.Titled       := False;
      Flags.WhiteBorder  := True;
    END { with } ;

  FlushGIDIS;  { Flush output to current window before creating new }

  WICRW( XStatus, DescriptorLength, MapWindow );  { Create the window }

  FillText( WindowHeight, Msg );  { Write the text strings in window }
```

```
    SetWritingMode( Replace );

    FOR i := 1 TO 6 DO  { Display the 6 color squares }
      BEGIN
        SetPosition( (i-1)*(CurSiz+Gap) + IndentX, IndentY );

        IF i <= 3
         THEN SetPrimaryColor( i     )
         ELSE SetPrimaryColor( i + 1 );  { Skip over color index 4 }

        BeginFill;  { Fill in a square in solid color }
        DrawRelLine(  CurSiz,  0      );
        DrawRelLine(  0,       CurSiz );
        DrawRelLine( -CurSiz,  0      );
        DrawRelLine(  0,      -CurSiz );
        EndFill;
      END { for } ;

    SetPrimaryColor( 0 );

  END { DisplayColors } ;


BEGIN { HandleColorMap }

  NewMap      := ColorMap;  { Init new map values with the current ones }
  ColorIndex  := 1;  { The cursor will start on the first square }
  PriorIndex  := 0;  { There is no existing cursor }
  CurSiz      := CharacterWidth  * 4;
  Gap         := CharacterWidth  * 3;
  IndentX     := CharacterWidth  * 11 DIV 2;
  IndentY     := CharacterHeight * 11 DIV 2;

  DisplayColors;  { Create the outer colormap window with 6 squares }

  REPEAT

    EnableColors( NewMap );  { Make sure the color map is current }
    PutCursor;  { Position the cursor on the chosen square }

    FlushGIDIS;  { Flush pending output prior to any keyboard wait }
    KBASTInitialize;  { Attach the keyboard for AST input }
    FunctionKey := GetAKey( Ch );  { Read a keystroke }
    PVDetach;  { Detach.  This also flushes the GIDIS buffer }
```

```
    CASE FunctionKey OF  { Now process the resulting key }

      SuspendKey,
      MainScreenKey,
      Exit:         ; { Nothing to do right yet }

      Right:        IF ColorIndex = MaxColor
                     THEN SoundBell  { Can't move right from 6th square }
                     ELSE
                       IF ColorIndex = 3
                        THEN ColorIndex := 5  { Skip over index 4 }
                        ELSE ColorIndex := ColorIndex + 1;  { Move right }

      Left:         IF ColorIndex = 1
                     THEN SoundBell  { Can't move left from 1st square }
                     ELSE
                       IF ColorIndex = 5
                        THEN ColorIndex := 3  { Skip back over index 4 }
                        ELSE ColorIndex := ColorIndex - 1;  { Move left }

      DoKey,
      SelectKey,
      Return:       ModifyEntry( ColorIndex );  { Go set RGB components }

      CancelKey:    NewMap := ColorMap;  { Reset to original values }

      OTHERWISE     SoundBell;  { Ignore all other keys }
    END { case } ;

  UNTIL ( FunctionKey = Exit          )   { Signals that the user is }
     OR ( FunctionKey = SuspendKey    )   { done changing all color  }
     OR ( FunctionKey = MainScreenKey );  { indexes for now.        }

  IF FunctionKey = SuspendKey
   THEN SuspendRequested := True  { Set flag, we'll get it next time }
   ELSE
     IF FunctionKey = MainScreenKey
      THEN MainScreenRequested := True;

  ColorIndex := 0;  { Set this up so that...      }
  PutCursor;        { the cursor will be shut off. }
  FlushGIDIS;  { Flush any remaining output to this window }

  WIDSW( XStatus, MapWindow.ID );  { Destroy the window }

  ColorMap := NewMap;  { Remember the new values }
  EnableColors( ColorMap );  { Refresh the hardware colormap }

END { HandleColorMap } ;
```

```
PROCEDURE HandleLineThickness;

{ HandleLineThickness is called when the user chooses the "Line          }
{ thickness" option from the main Flow Control Menu.  What this          }
{ procedure does is to create and process its own custom, special-       }
{ purpose interface that allows the end user at the keyboard to change   }
{ the thickness (both in X and in Y) of the line that the segments of    }
{ the Batons are drawn with.                                             }

VAR
  LineWindow: WindowDescriptor;
  FunctionKey: FunctionKeyNames;
  OldLow, OldHi: Integer;
  Ch: Char;


  PROCEDURE DisplayLineWindow;

  { DisplayLineWindow creates the display used for "Line thickness."     }
  { This display has some instructional text on the left, followed by a  }
  { drawing of a sample Baton on the right.  This routine creates the    }
  { display window and fills in the text on the left.                    }

  CONST
    WindowHeight = 9;  { Number of lines the window is tall }

  VAR
    Msg: ARRAY [ 1..WindowHeight ] OF StringType;
    i, Len, LongestString: Integer;

  BEGIN { DisplayLineWindow }

    GetMessage9( LTHMSG,
                 Msg[ 1 ], Msg[ 2 ], Msg[ 3 ],
                 Msg[ 4 ], Msg[ 5 ], Msg[ 6 ],
                 Msg[ 7 ], Msg[ 8 ], Msg[ 9 ] );
    { Call GetMessage9 to read the instructional text from a message }
    { frame in the frame file.  GetMessage9 reads the entire frame   }
    { into a buffer, then returns the separate lines of the message. }

    LongestString := 0;
    FOR i := 1 TO WindowHeight DO
      BEGIN  { Compute longest string (determines width of window) }
        Len := CountString( Msg[ i ] );
        IF Len > LongestString
         THEN
           LongestString := Len;
      END { for } ;

    LongestString := LongestString + 4;  { Add spacers on each side }
```

```
  WITH LineWindow DO
    BEGIN  { Fill in the window descriptor block }
      LineWindow         := WDescModel;  { Init the bulk of it }
      Width              := LongestString        * CharacterWidth;
      Height             := ( WindowHeight + 1 ) * CharacterHeight;
      X                  := -32760;  { Window centered }
      Y                  := -32760;  { Window centered }
      Flags.Stackable    := True;
      Flags.Color        := True;
      Flags.Titled       := False;
      Flags.WhiteBorder  := True;
    END { with } ;

  FlushGIDIS;  { Flush any remaining output to current window }

  WICRW( XStatus, DescriptorLength, LineWindow );  { Create window }

  FillText( WindowHeight, Msg );  { Write the text strings in window }

END { DisplayLineWindow } ;
```

```
PROCEDURE ShowLines;

{ ShowLines draws the sample Baton in the rightmost portion of the    }
{ "Line thickness" display window, using the current X, Y thickness   }
{ values.                                                             }

VAR
  X1, Y1, X2, Y2, i, Col, BoxSiz, CornerX, CornerY: Integer;

BEGIN { ShowLines }

  BoxSiz  := CharacterWidth  * 13;  { Determine how large it will be }
  CornerX := CharacterWidth  * 31;
  CornerY := CharacterHeight * 2;

  SetOutputClippingRegion( CornerX, CornerY, BoxSiz, BoxSiz );
  EraseClippingRegion;  { Get rid of any old sample Baton }

  WITH ContextBlock.R.LineThickness DO  { Use the new line }
    SetPixelSize( Low, Hi, Low DIV 2, Hi DIV 2 );

  X1  := CornerX + BoxSiz;   { Init the starting }
  Y1  := CornerY;            { position of the   }
  X2  := CornerX;            { sample Baton.     }
  Y2  := CornerY + 80;
  Col := 0;

  FOR i := 1 TO 8 DO  { The sample Baton is 8 segments long }
    BEGIN  { Draw the next segment }
      Col := Col + 1;
      IF Col = 4
       THEN
         Col := 5;  { Skip color index 4 }
      IF Col = MaxColor
       THEN
         Col := 1;  { Cycle back to 1 }

      IF ColorWindows = 0
       THEN SetPrimaryColor( 0   )  { Black if monochrome   }
       ELSE SetPrimaryColor( Col ); { Else use proper color }

      SetPosition( X1, Y1 );
      DrawLine(    X2, Y2 );

      X1 := X1 - 28;  { Move the Baton along its track }
      Y1 := Y1 + 10;
      X2 := X2 + 30;
      Y2 := Y2 + 35;
    END { for } ;

  SetPrimaryColor( 0 );  { Back to drawing in black }

  WITH LineWindow DO
    SetOutputClippingRegion( 0, 0, Width, Height );  { Full window }

END { ShowLines } ;
```

```
BEGIN { HandleLineThickness }

  DisplayLineWindow;  { Create the display and its instructional text }

  EnableColors( ColorMap );  { Refresh the hardware colormap }
  ShowLines;  { Draw the sample Baton using current line thickness }

  WITH ContextBlock.R.LineThickness DO
    BEGIN  { Remember current thickness }
      OldLow := Low;
      OldHi  := Hi;
    END { with } ;

  KBASTInitialize;  { Attach the keyboard for AST input }

  REPEAT

    FlushGIDIS;  { Flush pending output prior to any keyboard wait }
    FunctionKey := GetAKey( Ch );  { Read a keystroke }

    WITH ContextBlock.R.LineThickness DO
      CASE FunctionKey OF  { Now process the keystroke }

        SuspendKey,
        MainScreenKey,
        Exit:          ; { Nothing to do now }

        Left:          IF Low > PixelWidth
                        THEN
                          BEGIN  { Reduce X thickness (Narrower) }
                            Low := Low - PixelWidth;
                            IF ( KBBuf.Count = 0 ) OR ( Low = PixelWidth )
                             THEN
                               ShowLines;  { Update display if needed }
                          END { Left } ;

        Right:         IF Low < 200
                        THEN
                          BEGIN  { Increase X thickness (Wider) }
                            Low := Low + PixelWidth;
                            IF ( KBBuf.Count = 0 ) OR ( Low >= 200 )
                             THEN
                               ShowLines;
                          END { Right } ;

        Up:            IF Hi > PixelHeight
                        THEN
                          BEGIN  { Reduce Y thickness (Shorter) }
                            Hi := Hi - PixelHeight;
                            IF ( KBBuf.Count = 0 ) OR ( Hi = PixelHeight )
                             THEN
                               ShowLines;
                          END { Up } ;
```

```
        Down:          IF Hi < 200
                        THEN
                          BEGIN  { Increase Y thickness (Taller) }
                            Hi := Hi + PixelHeight;
                            IF ( KBBuf.Count = 0 ) OR ( Hi >= 200 )
                             THEN
                               ShowLines;
                          END { Down } ;

        CancelKey:     BEGIN  { Reset to original line thickness }
                         Low := OldLow;
                         Hi  := OldHi;
                         ShowLines;  { And display it that way again }
                       END { CalcelKey } ;

        OTHERWISE      SoundBell;
       END { case } ;

   UNTIL ( FunctionKey = Exit          )  { These keys signal that the }
      OR ( FunctionKey = SuspendKey    )  { user is done changing the  }
      OR ( FunctionKey = MainScreenKey ); { line thickness for now.    }

   IF FunctionKey = SuspendKey
    THEN SuspendRequested := True  { Set flag, so we'll get it next time }
    ELSE
      IF FunctionKey = MainScreenKey
       THEN MainScreenRequested := True;

   PVDetach;  { Detach the keyboard }

   WIDSW( XStatus, LineWindow.ID );  { Destroy "Line thickness" window }

   WITH ContextBlock.R.LineThickness DO
     BEGIN  { Invoke the new line }
       SetPixelSize( Low, Hi, Low DIV 2, Hi DIV 2 );

       IF ( Low < OldLow ) OR ( Hi < OldHi )  { New line is smaller }
        THEN
          InitBaton;  { Start them over to erase old debris }
     END { with } ;

END { HandleLineThickness } ;
```

```
BEGIN { HandleFCMKey }

  PVDetach;  { Detach keyboard to allow Synergy to operate the Flow Menu }

  EXFLOW( XStatus, KeyPressed, MenuID,     { Process the Flow }
          OptionValue, QIXFCM,             { Control Menu.    }
          Ord( InputKey ) - Ord( F11 ) );

  IF ( KeyPressed = PV$DO  )   { A successfull terminator was pressed, }
  OR ( KeyPressed = PV$RET )   { see which option was chosen.          }
   THEN
     CASE OptionValue OF

       1:          IF ColorWindows = 0
                    THEN EXMESS( XStatus, KeyPressed, NOCOLR )
                    ELSE HandleColorMap;  { Go change the color map }

       2:          HandlePersonalizeKey;  { Go change the Batons }

       3:          HandleLineThickness;   { Go change the line thickness }

       OTHERWISE  { Do Nothing } ;
     END { case } ;

  IF KeyPressed = PV$F5
   THEN SuspendRequested := True  { Set flag, we'll get it next time }
   ELSE
     IF KeyPressed = PV$MAI
      THEN MainScreenRequested := True;

  KBASTInitialize;  { Reattach the keyboard for AST input }

END { HandleFCMKey } ;
```

```
BEGIN { HandleKey }

  FlushGIDIS;  { Flush pending output prior to any keyboard input }

  IF SuspendRequested
   THEN
     BEGIN  { F5 was pressed when we were in the middle of something before }
       InputKey         := SuspendKey;  { Fake an F5 }
       SuspendRequested := False;  { Dismiss the request }
     END { then }

   ELSE
     IF MainScreenRequested
      THEN
        BEGIN  { MAIN SCREEN was pressed in the middle of something before }
          InputKey            := MainScreenKey;  { Fake a MAIN SCREEN }
          MainScreenRequested := False;  { Dismiss the request }
        END { then }

      ELSE
        InputKey := GetAKey( InputChar );  { Read the REAL keystroke }


  CASE InputKey OF
    Exit,
    MainScreenKey:  Done := True;  { Set a flag so we will exit }

    SuspendKey:     Hibernate;  { Give control up to the Synergy Main Menu }

    HelpKey:        BEGIN
                      PVDetach;  { Free the keyboard for Synergy's use }
                      EXHELP( XStatus, OVERVW );  { Give HELP }
                      KBASTInitialize;  { Get the keyboard back again }
                    END { help } ;

    F11, F12, F13,
    AddOptnsKey:    HandleFCMKey( InputKey );  { Process Flow Menu request }

    InsertHere:     IF Batons < MaxBatons
                     THEN
                       BEGIN  { Not at max, increase the number of Batons }
                         Batons := Batons + 1;
                         InitBaton;  { Re-start the new Batons }
                       END { then } ;

    RemoveKey:      IF Batons > 1
                     THEN
                       BEGIN  { Not at min, decrease the number of Batons }
                         Batons := Batons - 1;
                         InitBaton;  { Re-start the new Batons }
                       END { then } ;

    OTHERWISE  SoundBell;  { Ignore all other keys }
  END { case } ;

END { HandleKey } ;
```

```
BEGIN { Baton }

  InitializeSyn;     { Set up basic environmental things. }
  MakeOurWindow;     { Create the playing field.          }
  InitBaton;         { Init the Baton data structures.    }
  KBASTInitialize;   { Attach the keyboard for AST input. }

  SuspendRequested    := False;   { Clear a few }
  MainScreenRequested := False;   { variables.  }
  Done                := False;
  TitleCounter        := 0;


  REPEAT  { This is the main program loop }

    WHILE ( KBBUF.Count <> 0 )       { If any keys have been pressed, }
       OR SuspendRequested           { or F5 was pressed a ways back, }
       OR MainScreenRequested DO     { or MAIN SCREEN a ways back,    }
      HandleKey;                     { go process them.               }

    FOR i := 1 TO Batons DO
      TwiddleBaton( i );             { Move each Baton a cycle.       }

    IF ColorWindows <> 0
     THEN
       BEGIN  { On a color system, rotate the colors to follow the Batons }

         FOR i := 1 TO 5 DO
           BEGIN  { Roll the color map }
             Index := i + ColorMapIndex - 1;
             IF Index > 3
              THEN
                Index := Index + 1;  { Skip over 4 }

             IF Index >= MaxColor
              THEN
                BEGIN  { Wrap around }
                  Index := Index - ( MaxColor - 1 );
                  IF Index > 3
                   THEN
                     Index := Index + 1;  { Skip over 4, after wrap }
                END { then } ;

             WITH ColorMap[ Index ].R DO
               IF i <= 3
                THEN SetColorMap( i,     Red, Green, Blue )
                ELSE SetColorMap( i + 1, Red, Green, Blue );  { Skip over 4 }
           END { for } ;

         ColorMapIndex := ColorMapIndex - 1;
         IF ColorMapIndex <= 0
          THEN
            ColorMapIndex := MaxColor - 2;  { Wrap around }

       END { then } ;
```

```
    IF ( TitleID = TITLE2 ) AND ( TitleCounter < 1000 )
     THEN
       BEGIN  { So the Welcome title is up (must be the first time we've run) }

         TitleCounter := TitleCounter + 1;  { Increment a timeout counter }

         IF TitleCounter = 1000
          THEN
            BEGIN  { The Welcome Title's time limit (1000 cycles) has          }
                   { expired, so change the title to the normal title.  (1000 }
                   { loops takes 30-40 seconds.)                              }
              ReadMessage( TITLE, TitleLength, TitleText );
              PVDetach;  { Free the terminal/keyboard for Synergy }

              WITTL( XStatus, TitleLength, TitleText );  { Set the new title }

              KBASTInitialize;  { Get the terminal/keyboard back }
            END { then } ;

       END { then } ;

  UNTIL Done;  { Done is set to True when MAIN SCREEN or EXIT are pressed }

  CleanUpSyn;  { Tidy up after ourselves, and tell Synergy we're done }


  { Once all this is done, when the "END." is reached below, PRO/Pascal will  }
  { terminate the running of this task.  Since this is the first and only     }
  { task in the application, when it terminates, the Synergy Window Manager   }
  { will be notified by P/OS.  Control will thus pass back to the Synergy     }
  { Main Menu (Window Manager), ready for another application to be run.       }

END { Baton } .
```

A.3 THE GIDISOPS.PAS FILE

```
{ GIDISOPS.PAS    Pascal %Include file that defines PRO/GIDIS things.          }

CONST

  { First, define all of the PRO/GIDIS instruction op-codes that are           }
  { implemented in V2.0 of PRO/GIDIS.                                          }

  Begin_Define_Character       = 8452;
  Begin_Filled_Figure          = 7936;
  Create_Alphabet              = 11780;
  Draw_Arcs                    = 5888;   { Plus length }
  Draw_Characters              = 8960;   { Plus length }
  Draw_Lines                   = 6400;   { Plus length }
  Draw_Packed_Characters       = 18944;  { Plus length }
  Draw_Rel_Arcs                = 6912;   { Plus length }
  Draw_Rel_Lines               = 6656;   { Plus length }
  End_Define_Character         = 9216;
  End_Filled_Figure            = 8192;
  End_List                     = -32768;
  End_Picture                  = 6144;
  Erase_Clipping_Region        = 12288;
  Flush_Buffer                 = 7168;
  Initialize                   = 257;
  Load_By_Name                 = 9474;
  Load_Character_Cell          = 8704;   { Plus length }
  New_Picture                  = 1536;
  NOP                          = 0;
  Print_Screen                 = -29434;
  Request_Cell_Standard        = 13824;
  Request_Current_Position     = 14080;
  Request_Output_Size          = 14592;
  Request_Status               = 14848;
  Request_Version_Number       = 18176;
  Scroll_Clipping_Region       = 13314;
  Set_Alphabet                 = 9729;
  Set_Area_Cell_Size           = 17666;
  Set_Area_Texture             = 3586;
  Set_Area_Texture_Size        = 770;
  Set_Cell_Display_Size        = 10242;
  Set_Cell_Explicit_Movement   = 10498;
  Set_Cell_Movement_Mode       = 10753;
```

```
Set_Cell_Oblique              = 16641;
Set_Cell_Rendition            = 11009;
Set_Cell_Rotation             = 11265;
Set_Cell_Unit_Size            = 11522;
Set_Color_Map_Entry           = 4102;
Set_GIDIS_Output_Space        = 2308;
Set_Line_Texture              = 4355;
Set_Output_Clipping_Region    = 1028;
Set_Output_Cursor             = 1286;
Set_Output_Cursor_Rendition = 18433;
Set_Output_IDS                = 3074;
Set_Output_Rubber_Band        = 13571;
Set_Output_Viewport           = 3332;
Set_Pixel_Size                = 4868;
Set_Plane_Mask                = 5121;
Set_Position                  = 7426;
Set_Primary_Color             = 5377;
Set_Rel_Position              = 7682;
Set_Secondary_Color           = 3841;
Set_Writing_Mode              = 5633;


{ Here are a bunch of miscellaneous constant values.                    }

IO_WAL              = %O'410';   { I/O function code, ASCII (text-mode) QIOs }
IO_WSD              = %O'5410';  { I/O function code, GIDIS (graphics) QIOs }
MaxASCII            = 100;  { Size of the ASCII (text-mode) QIO buffer (bytes) }
MaxGIDIS            = 100;  { Size of the GIDIS (graphics) QIO buffer (bytes) }
MaxQueue            = 80;   { Size of the private AST-driven input buffer }
QIOW                = 3;    { Directive Identification Code for QIOW directive }
QIOW_Len            = 12;   { Length of QIO Directive Parameter Block }
SD_GDS              = 1;    { Special data type for IO.WSD signifying GIDIS }
StringMax           = 80;   { Used to allocate character arrays }
```

```
TYPE

  CharCell          = ARRAY [ 1..16 ] OF Integer;  { Holds GIDIS char cells }

  Queue             = RECORD        { Structure of the private, AST-driven }
                        First,      { input buffer that modules KBSERV.MAC }
                        Last,       { and GETAKEY.MAC manage.              }
                        Count: Integer;
                        Data:  PACKED ARRAY [ 1..MaxQueue ] OF Char;
                      END { Queue } ;

  String            = PACKED ARRAY [ 1..StringMax ] OF Char;

  StringType        = RECORD                { Structure that contains }
                        L: Integer;         { a string with a counted }
                        S: String;          { length.                 }
                      END { StringType } ;

  StringArray       = ARRAY [ 1..99 ] OF StringType;
    { Array of arbitrary length used to declare some [Unsafe] parameters.     }

  WritingModes      = ( Transparent, TransparentNegate,  { List of the     }
                        Complement,  ComplementNegate,   { supported GIDIS }
                        Overlay,     OverlayNegate,      { writing modes.  }
                        Replace,     ReplaceNegate,
                        Erase,       EraseNegate,
                        NoMode );

  QIOdpbType        = PACKED RECORD                                { Directive       }
                        D_CODE: [Pos(0,0)]          0..255;      { Parameter block }
                        D_LGTH: [Pos(1,0)]          0..255;      { for the Queue   }
                        Q_IOFN: [Pos(2,0)]          Unsigned;    { I/O Request     }
                        Q_IOLU: [Pos(4,0)]          Unsigned;    { (QIO/QIOW) P/OS }
                        Q_IOEF: [Pos(6,0)]          0..255;      { Executive       }
                        Q_IOPR: [Pos(7,0)]          0..255;      { Directives.     }
                        Q_IOSB: [Pos(8,0),Unsafe]   Unsigned;
                        Q_IOAE: [Pos(10,0),Unsafe]  Unsigned;
                        Q_IOPL: [Pos(12,0),Unsafe]  ARRAY [ 1..6 ] OF Unsigned;
                      END { QIOdpbType } ;
```

THE GIDISOPS.PAS FILE

```
{ Here are definitions for a large number of general-purpose procedures, most }
{ of which are related to doing video output using the PRO/GIDIS interpreter. }


[External] PROCEDURE FlushASCII; EXTERNAL;
[External] PROCEDURE StoreASCII( Count: Integer;
                                 VAR Data: [ReadOnly,Unsafe] String ); EXTERNAL;
[External] PROCEDURE FlushGIDIS; EXTERNAL;
[External] PROCEDURE StoreGIDIS( Count: Integer;
                                 VAR Data: [ReadOnly,Unsafe] String ); EXTERNAL;
[External] PROCEDURE SoundBell; EXTERNAL;
[External] PROCEDURE PutChr( Ch: Char ); EXTERNAL;
[External] PROCEDURE PutString( Length:                          Integer;
                                VAR Chars: [ReadOnly,Unsafe] String ); EXTERNAL;
[External] PROCEDURE SetPosition( X, Y: Integer ); EXTERNAL;
[External] PROCEDURE DrawLine( X, Y: Integer ); EXTERNAL;
[External] PROCEDURE DrawRelLine( X, Y: Integer ); EXTERNAL;
[External(SORB)] PROCEDURE SetOutputRubberBand( BandType,
                                                BaseX, BaseY: Integer ); EXTERNAL;
[External] PROCEDURE SetWritingMode( Mode: WritingModes ); EXTERNAL;
[External(SCME)] PROCEDURE SetColorMapEntry( Map, Index, R, G, B,
                                             Mono: Integer ); EXTERNAL;
[External] PROCEDURE SetAlphabet( WhichAlphabet: Integer ); EXTERNAL;
[External] PROCEDURE SetCellRendition( Rendition: Integer ); EXTERNAL;
[External] PROCEDURE SetPrimaryColor( Color: Integer ); EXTERNAL;
[External] PROCEDURE SetSecondaryColor( Color: Integer ); EXTERNAL;
[External] PROCEDURE SetPixelSize( Width, Height, XOffset,
                                   YOffset: [Unsafe] Integer ); EXTERNAL;
[External] PROCEDURE SetOutputClippingRegion( UpperLeftX, UpperLeftY, Width,
                                              Height: Integer ); EXTERNAL;
[External] PROCEDURE EraseClippingRegion; EXTERNAL;
[External] PROCEDURE BeginFill; EXTERNAL;
[External] PROCEDURE EndFill; EXTERNAL;
[External(CreAlp)] PROCEDURE CreateAlphabet( Width, Height, Extent,
                                             WidthType: Integer ); EXTERNAL;
[External] PROCEDURE LoadCharacterCell( Index, Width, Height: Integer;
                                VAR Raster: [ReadOnly,Unsafe] CharCell ); EXTERNAL;
[External] PROCEDURE SetAreaTexture( Alphabet: Integer;
                                     Index: [Unsafe] Integer ); EXTERNAL;
[External(SCDS)] PROCEDURE SetCellDisplaySize( Width,
                                               Height: Integer ); EXTERNAL;
[External(SetUni)] PROCEDURE SetCellUnitSize( Width,
                                              Height: Integer ); EXTERNAL;
[External] PROCEDURE DrawArc( X, Y, Angle: Integer ); EXTERNAL;
[External(DRARC)] PROCEDURE DrawRelArc( X, Y, Angle: Integer ); EXTERNAL;
[External] PROCEDURE SetColorMap( Index, Red, Green, Blue: Integer ); EXTERNAL;
[External] PROCEDURE EmptyQueue( VAR Que: Queue ); EXTERNAL;
[External] PROCEDURE AddToQueue( VAR Que: Queue; Ch: Char ); EXTERNAL;
[External] PROCEDURE RemoveFromQueue( VAR Que: Queue; VAR Ch: Char ); EXTERNAL;
```

A.4 THE SYNERGY.PAS FILE

```
{ SYNERGY.PAS     Pascal %Include file that defines Synergy things.          }

CONST

  DescriptorLength = 32;  { Length of a window descriptor block (bytes) }
  MaxContext       = 32;  { Size of Synergy Context Block (bytes) }
  MaxTBFSize       = 80;  { Size of the Synergy typeahead buffer WITBF }

  { These are the GIDIS alphabet numbers for the pre-loaded fonts that      }
  { Synergy provides.  Your application can to a GIDIS SET_ALPHABET         }
  { instruction specifying one of these numbers, and then do DRAW_CHARACTERS }
  { and DRAW_PACKED_CHARACTERS instructions to display the normal ASCII,     }
  { DIGITAL Multinational, and VT100 Special Graphics characters on the     }
  { screen (the ASCII, Multinational and Line Drawing characters are all    }
  { folded together into each of the fonts 9 to 14).                        }

  WI$Special       = 7;
  WI$Dim           = 9;
  WI$Normal        = 10;
  WI$Bold          = 11;
  WI$Underline     = 13;
  WI$BoldUnderline = 14;
  WI$Boxed         = 15;

  { These are the values for the Termination Keys that a program passes to  }
  { Synergy in a Termination Key List, and the values that Synergy returns as }
  { KeyPressed values.                                                      }

  PV$RET = 269;
  PV$FND = 513;   PV$INS = 514;   PV$REM = 515;
  PV$SEL = 516;   PV$PRE = 517;   PV$NEX = 518;
  PV$BRK = 525;   PV$SET = 526;   PV$F5  = 527;   PV$INT = 529;
  PV$RES = 530;   PV$CAN = 531;   PV$MAI = 532;   PV$EXI = 533;
  PV$F11 = 535;   PV$F12 = 536;   PV$F13 = 537;   PV$ADD = 538;
  PV$HLP = 540;   PV$DO  = 541;
  PV$F17 = 543;   PV$F18 = 544;   PV$F19 = 545;   PV$F20 = 546;
  PV$PF1 = 547;   PV$PF2 = 548;   PV$PF3 = 549;   PV$PF4 = 550;
  PV$UP  = 551;   PV$DWN = 552;   PV$RIT = 553;   PV$LEF = 554;
  PV$COM = 555;   PV$MIN = 556;   PV$PER = 557;   PV$ENT = 558;
  PV$0   = 559;   PV$1   = 560;   PV$2   = 561;   PV$3   = 562;
  PV$4   = 563;   PV$5   = 564;   PV$6   = 565;   PV$7   = 566;
  PV$8   = 567;   PV$9   = 568;   PV$DEL = 569;
```

```
TYPE

  StatusBlock       = ARRAY [ 1..2 ] OF Integer;
    { Two-word array that success/failure status information is returned in   }
    { from the various Synergy service calls.                                 }

  WindowDescriptor = RECORD  { This is a Synergy Window Descriptor Block }
                       ID:          [Pos(0,0)]  Integer;
                       X:           [Pos(2,0)]  Unsigned;
                       Y:           [Pos(4,0)]  Unsigned;
                       Width:       [Pos(6,0)]  Unsigned;
                       Height:      [Pos(8,0)]  Unsigned;
                       Flags:       [Pos(10,0)] PACKED RECORD
                           Stackable:     [Pos( 0)] Boolean;
                           Titled:        [Pos( 1)] Boolean;
                           Hidden:        [Pos( 2)] Boolean;
                           Color:         [Pos( 3)] Boolean;
                           WhiteBorder:   [Pos( 4)] Boolean;
                           ClearOnChange: [Pos( 5)] Boolean;
                           Unused6:       [Pos( 6)] Boolean;
                           VT100_Style:   [Pos( 7)] Boolean;
                           Invisible:     [Pos( 8)] Boolean;
                           Unused9:       [Pos( 9)] Boolean;
                           ThreePlane:    [Pos(10)] Boolean;
                           Unused11:      [Pos(11)] Boolean;
                           Unused12:      [Pos(12)] Boolean;
                           Unused13:      [Pos(13)] Boolean;
                           Unused14:      [Pos(14)] Boolean;
                           Unused15:      [Pos(15)] Boolean;
                         END { Flags } ;
                       MinWidth:    [Pos(12,0)] Unsigned;
                       MinHeight:   [Pos(14,0)] Unsigned;
                       MaxWidth:    [Pos(16,0)] Unsigned;
                       MaxHeight:   [Pos(18,0)] Unsigned;
                       XOffset:     [Pos(20,0)] Unsigned;
                       YOffset:     [Pos(22,0)] Unsigned;
                       FrameWidth:  [Pos(24,0)] Unsigned;
                       FrameHeight: [Pos(26,0)] Unsigned;
                       OwnerTaskID: [Pos(28,0)] ARRAY [ 1..2 ] OF Unsigned;
                     END { WindowDescriptor } ;

  WordArray         = ARRAY [ 1..99 ] OF Integer;
    { Used in miscellaneous places where arrays of indeterminant size are     }
    { used.  This is made possible through the PRO/Pascal [Unsafe] attribute. }
```

```
CONST  { Structured constants }

  WDescModel = WindowDescriptor( 0, 0, 0, 0, 0, ( 16 OF False ),
                                 0, 0, 0, 0, 0, 0, 0, 0, ( 0, 0 ) );
    { This may be used in an assignment statement to fill in all the fields }
    { of a window descriptor block in one fell swoop.                        }


VAR  { Some variables that virtually all Synergy applications will use }

  WITBF: [External] RECORD
                      MaxSize:       Integer;
                      CurrentLength: Integer;
                      Characters:    PACKED ARRAY [ 1..MaxTBFSize ] OF Char;
                    END { WITBF } ;
    { Definition of the Synergy-provided type-ahead buffer that is used to    }
    { pass read-ahead characters from the application to Synergy and back.    }

  ActualVersion,  { Returned from WIINI }
  ColorWindows:   { Returned from WIINI }
    integer;

  CharacterHeight: [External(CHARHI)] Unsigned;  { Returned from WIINI }
  CharacterWidth:  [External(CHARWI)] Unsigned;  { Returned from WIINI }

  PixelHeight,    { Returned from WIINI }
  PixelWidth,     { Returned from WIINI }
  ScreenWidth,    { Returned from WIINI }
  ScreenHeight:   { Returned from WIINI }
    Unsigned;
```

```
{ These are some procedure definitions for a number of the Synergy services.  }

PROCEDURE WIINI( VAR Status:                        StatusBlock;
                 VAR ExpectedVersion: [ReadOnly] Integer;
                 VAR ActualVersion:             Integer;
                 VAR ContextLength:             Integer;
                 VAR ContextBlock:    [Unsafe]  String;
                 VAR ScreenWidth:               Unsigned;
                 VAR ScreenHeight:              Unsigned;
                 VAR CharacterWidth:            Unsigned;
                 VAR CharacterHeight:           Unsigned;
                 VAR PixelWidth:                Unsigned;
                 VAR PixelHeight:               Unsigned;
                 VAR ColorWindows:              Integer ); SEQ11;

PROCEDURE WIDON( VAR Status:                              StatusBlock;
                 VAR ContextLength: [ReadOnly]            Integer;
                 VAR ContextBlock:  [ReadOnly,Unsafe] String ); SEQ11;

PROCEDURE WIINT( VAR Status:    StatusBlock;
                 VAR WhyChange: Integer;
                 VAR WindowID:  Integer;
                 VAR NewWidth:  Integer;
                 VAR NewHeight: Integer ); SEQ11;

PROCEDURE WICRW( VAR Status:                       StatusBlock;
                 VAR DescriptorLength: [ReadOnly] Integer;
                 VAR Descriptor:                  WindowDescriptor ); SEQ11;

PROCEDURE WIDSW( VAR Status:              StatusBlock;
                 VAR WindowID: [ReadOnly] Integer ); SEQ11;

PROCEDURE WIGEW( VAR Status:                       StatusBlock;
                 VAR DescriptorLength: [ReadOnly] Integer;
                 VAR Descriptor:                  WindowDescriptor ); SEQ11;

PROCEDURE WISWP( VAR Status:                       StatusBlock;
                 VAR DescriptorLength: [ReadOnly] Integer;
                 VAR Descriptor:                  WindowDescriptor ); SEQ11;

PROCEDURE WITTL( VAR Status:                         StatusBlock;
                 VAR TitleLength: [ReadOnly]         Integer;
                 VAR TitleText:   [ReadOnly,Unsafe] String ); SEQ11;
```

```
PROCEDURE WIERW( VAR Status:                    StatusBlock;
                 VAR Len1: [ReadOnly]           Integer;
                 VAR Msg1: [ReadOnly,Unsafe]    String;
                 VAR Len2: [ReadOnly]           Integer;
                 VAR Msg2: [ReadOnly,Unsafe]    String;
                 VAR Len3: [ReadOnly]           Integer;
                 VAR Msg3: [ReadOnly,Unsafe]    String;
                 VAR Len4: [ReadOnly]           Integer;
                 VAR Msg4: [ReadOnly,Unsafe]    String;
                 VAR Len5: [ReadOnly]           Integer;
                 VAR Msg5: [ReadOnly,Unsafe]    String ); SEQ11;

PROCEDURE OPENME( VAR Status:                                   StatusBlock;
                  VAR FileVersionNumber: [ReadOnly]             Integer;
                  VAR FilenameLength:    [ReadOnly]             Integer;
                  VAR Filename:          [ReadOnly,Unsafe]      String ); SEQ11;

PROCEDURE WIRMS( VAR Status:                    StatusBlock;
                 VAR LineCount:                 Integer;
                 VAR LineOffsets:   [Unsafe]    WordArray;
                 VAR MessageBuffer: [Unsafe]    String;
                 VAR FrameID:       [ReadOnly]  Integer;
                 VAR BufferSize:    [ReadOnly]  Integer ); SEQ11;

PROCEDURE WIXPS( VAR Status:                    StatusBlock;
                 VAR KeyPressed:                Integer;
                 VAR NumChanged:                Integer;
                 VAR ChangedValues: [Unsafe]    WordArray;
                 VAR FrameID:       [ReadOnly]  Integer;
                 VAR NumEntries:    [ReadOnly]  Integer;
                 VAR Entry1Class:   [ReadOnly]  Integer;
                 VAR Entry1Value:               Integer;
                 VAR Entry2Class:   [ReadOnly]  Integer;
                 VAR Entry2Value:               Integer ); SEQ11;

PROCEDURE EXMESS( VAR Status:                   StatusBlock;
                  VAR KeyPressed:               Integer;
                  VAR MenuID:       [ReadOnly]  Integer ); SEQ11;

PROCEDURE EXFLOW( VAR Status:                    StatusBlock;
                  VAR KeyPressed:                Integer;
                  VAR MenuID:                    Integer;
                  VAR OptionValue:               Integer;
                  VAR FrameID:      [ReadOnly]   Integer;
                  VAR InitialLeaf:  [ReadOnly]   Integer ); SEQ11;

PROCEDURE EXHELP( VAR Status:                StatusBlock;
                  VAR FrameID: [ReadOnly]    Integer ); SEQ11;
```

A.5 THE GIDIS.PAS FILE

```
MODULE GIDIS;

%Include 'GIDISOps/NoList'  { Contains declarations needed for GIDIS routines }
%Include 'Synergy/NoList'   { Contains declarations for Synergy services      }


CONST  { Structured constants }
  QIOdpbModel = QIOdpbType( QIOW, QIOW_Len, IO_WSD, 5, 1, 0, 0, 0,
                            ( 0, 0, 0, SD_GDS, 0, 0 ) );


VAR

  XCharacterHeight: [Global(CHARHI)] Unsigned;  { Returned by WIINI }
  XCharacterWidth:  [Global(CHARWI)] Unsigned;  { Returned by WIINI }

  ASCIIBuffer: [Global] PACKED ARRAY [ 1..MaxASCII ] OF Char;
  GIDISBuffer: [Global] PACKED ARRAY [ 1..MaxGIDIS ] OF Char;
    { The two output buffers used by the routines in this module to do their  }
    { stuff.  The output is buffered (QIOs done when buffer fills, buffer     }
    { then emptied) in order to get increased video performance.              }

  ASCIIQIOdpb: [Global,Volatile] QIOdpbType;
  QIOdpb:      [Global,Volatile] QIOdpbType;
    { Directive Parameter Blocks used by these routines to issue QIOW         }
    { Executive Directives, to actually cause output data to be sent from the }
    { application task, to the P/OS Terminal Subsystem for subsequent display.}

  State: [Global] PACKED RECORD         { Some miscellaneous flags }
                    CursorOn: Boolean;
                  END { State } ;

  KBBuf: [External,Volatile] Queue;
    { The private AST-driven type-ahead buffer used solely by this            }
    { application.  Note that it is DIFFERENT than the WITBF buffer defined   }
    { above.  Our private buffer (KBBUF) is circular, whereas the Synergy     }
    { buffer (WITBF) is linear.  Due to their different structure, we cannot  }
    { make double-duty of the Synergy buffer; thus we create our own.         }
```

```
[Initialize] PROCEDURE InitializeIO;

{ This procedure is automatically run by the PRO/Pascal run-time system     }
{ when the program starts up.  This routine initializes some data           }
{ structures that the rest of this module depends on.                       }

BEGIN { InitializeSyn }

  State.CursorOn := False;  { The graphics cursor is initially off }
  ASCIIQIOdpb    := QIOdpbModel;  { Initialize the Directive Parameter }
  QIOdpb         := QIOdpbModel;  { Blocks for the output QIOWs.       }

  WITH QIOdpb DO
    BEGIN
      Q_IOPL[ 1 ] := ( Address( GIDISBuffer ) )::Unsigned;
      Q_IOPL[ 2 ] := 0;  { Set the length of the data to zero }
    END { with } ;

  WITH ASCIIQIOdpb DO
    BEGIN
      Q_IOFN      := IO_WAL;
      Q_IOPL[ 1 ] := ( Address( ASCIIBuffer ) )::Unsigned;
      Q_IOPL[ 2 ] := 0;  { Set the length of the data to zero }
      Q_IOPL[ 4 ] := 0;
    END { with } ;

END { InitializeIO } ;
```

```
PROCEDURE FlushG; FORWARD;


PROCEDURE FlushA;

{ FlushA forces out any buffered ASCII (text-mode) data.                    }

BEGIN { FlushA }

  IF ASCIIQIOdpb.Q_IOPL[ 2 ] > 0
   THEN
     BEGIN  { There is data in the buffer }
       DIR$( ASCIIQIOdpb );  { Issue the directive to write text-mode }
       ASCIIQIOdpb.Q_IOPL[ 2 ] := 0;  { Reset the buffer length to zero }
       State.CursorOn := False;  { This has disables the cursor }
     END { then } ;

END { FlushA } ;


[Global] PROCEDURE FlushASCII;

BEGIN { FlushASCII }
  FlushG;  { Make sure any buffered GIDIS is put out first }
  FlushA;  { Then flush any text-mode data }
END { FlushASCII } ;



[Global] PROCEDURE StoreASCII( Count: Integer;
                               VAR Data: [ReadOnly,Unsafe] String );

{ StoreASCII takes the passed byte data, and appends it to the current      }
{ contents of the text-mode QIOW output buffer.  Whenever the buffer fills, }
{ the data pending in the buffer is written out, and the buffer is then     }
{ reset.                                                                    }

VAR
  i: Integer;

BEGIN { StoreASCII }

  i := 0;

  WITH ASCIIQIOdpb DO
    WHILE i < Count DO  { For each byte being output... }
      BEGIN
        IF Q_IOPL[ 2 ] = MaxASCII
         THEN
           FlushASCII;  { Buffer is full, dump it }
        i := i + 1;  { Increment to next byte to store }
        Q_IOPL[ 2 ] := Q_IOPL[ 2 ] + 1;  { Increment count of stored bytes }
        ASCIIBuffer[ Q_IOPL[ 2 ] ] := Data[ i ];  { Store the byte in list }
      END { while } ;

END { StoreASCII } ;
```

```
PROCEDURE FlushG;

{ FlushG forces out any buffered GIDIS (graphics mode) data.                    }

BEGIN { FlushG }

  WITH QIOdpb DO
    IF Q_IOPL[ 2 ] > 0
     THEN
       BEGIN  { There is data in the buffer }
         DIR$( QIOdpb );  { Issue the write QIOW to GIDIS }
         Q_IOPL[ 2 ] := 0;  { Reset the buffer length to zero }
       END { then } ;

END { FlushG } ;


[Global] PROCEDURE FlushGIDIS;

BEGIN { FlushGIDIS }
  FlushA;  { Make sure any buffered ASCII (text-mode) data is put out }
  FlushG;  { Then output any GIDIS data }
END { FlushGIDIS } ;


[Global] PROCEDURE StoreGIDIS( Count: Integer;
                               VAR Data: [ReadOnly,Unsafe] String );

{ StoreGIDIS takes the passed byte data, and appends it to the current         }
{ contents of the graphics-mode GIDIS QIOQ output buffer.  Whenever the        }
{ buffer fills, the data pending in the buffer is written out, and the         }
{ buffer is then reset.                                                        }

VAR
  i: Integer;

BEGIN { StoreGIDIS }

  i := 0;

  WITH QIOdpb DO
    WHILE i < Count DO  { For each data byte to output... }
      BEGIN
        IF Q_IOPL[ 2 ] = MaxGIDIS
         THEN
           FlushGIDIS;  { Buffer is full, dump it }
        i := i + 1;  { Increment to next byte to store }
        Q_IOPL[ 2 ] := Q_IOPL[ 2 ] + 1;  { Increment count of stored bytes }
        GIDISBuffer[ Q_IOPL[ 2 ] ] := Data[ i ];  { Store the byte in list }
      END { while } ;

END { StoreGIDIS } ;
```

```
[Global] PROCEDURE EmptyQueue( VAR Que: Queue );

{ EmptyQueue zeroes the passed queue.                                    }

BEGIN { EmptyQueue }

  WITH Que DO
    BEGIN
      First := 1;
      Last  := 0;
      Count := 0;
    END { with } ;

END { EmptyQueue } ;


[Global] PROCEDURE AddToQueue( VAR Que: Queue; Ch: Char );

{ AddToQueue adds the passed character to the passed queue.              }

BEGIN { AddToQueue }

  WITH Que DO
    IF Count < MaxQueue
     THEN
       BEGIN  { The queue is not full, proceed }
         IF Last >= MaxQueue
          THEN
            Last := 0;  { Circular list; wrap around }
         Last          := Last + 1;  { Advance pointer }
         Count         := Count + 1;  { Count this new byte }
         Data[ Last ] := Ch;  { Add the byte to the buffer }
       END { then } ;

END { AddToQueue } ;


[Global] PROCEDURE RemoveFromQueue( VAR Que: Queue; VAR Ch: Char );

{ RemoveFromQueue pops the oldest byte from the passed queue.            }

BEGIN { RemoveFromQueue }

  WITH Que DO
    IF Count > 0
     THEN
       BEGIN  { There is data in the queue, get the next byte }
         Ch := Data[ First ];  { Pass out the oldest byte }
         IF First >= MaxQueue
          THEN
            First := 0;  { Circular pointer; wrap around }
         First := First + 1;  { Advance pointer to next byte }
         Count := Count - 1;  { Decrease count by one }
       END { then } ;

END { RemoveFromQueue } ;
```

```
[Global] PROCEDURE SoundBell;

BEGIN { SoundBell }
  StoreASCII( 1, 7 );  { Store a single text-mode ASCII character, BELL }
  FlushASCII;  { And force it out right now }
END { SoundBell } ;


[Global] PROCEDURE PutChr( Ch: Char );

{ PutChr takes a character, and adds it to the GIDIS output buffer.          }

VAR
  Data: ARRAY [ 1..2 ] OF Integer;

BEGIN { PutChr }
  Data[ 1 ] := Draw_Characters + 1;  { Op-code, plus param count of 1 }
  Data[ 2 ] := Ord( Ch );
  StoreGIDIS( 4, Data );  { Send the GIDIS sequence to the output buffer }
END { PutChr } ;


[Global] PROCEDURE PutString( Length:                        Integer;
                              VAR Chars: [ReadOnly,Unsafe] String );

{ PutString takes a sequence of characters, and adds them to the GIDIS       }
{ output buffer (this is done with a DRAW_PACKED_CHARACTERS instruction for  }
{ performance sake).                                                         }

VAR
  i: Integer;
  Data: PACKED RECORD
          ParamCount: [Pos(0,0)] 0..255;
          OpCode:     [Pos(1,0)] 0..255;
          Text:       [Pos(2,0)] PACKED ARRAY [ 1..82 ] OF Char;
        END { Data } ;

BEGIN { PutString }

  WITH Data DO
    BEGIN
      FOR i := 1 TO Length DO  { Copy in the characters to draw }
        Text[ i ] := Chars[ i ];
      OpCode      := Draw_Packed_Characters DIV 256;  { Compute opcode byte }
      ParamCount  := ( Length + 1 ) DIV 2;  { Make a word count, rounded up }
      i := Length + 2;  { Count of text, plus Op-code and Length bytes }
      IF Odd( Length )
       THEN
         BEGIN  { String being written is odd length, must make it even }
           Text[ Length + 1 ] := Chr( 255 );  { Packed padding byte }
           i := i + 1;  { Count the extra byte }
         END { then } ;
      StoreGIDIS( i, Data );  { Add it to the GIDIS output buffer }
    END { with } ;

END { PutString } ;
```

```
{ The following procedures do a wide variety of individual low-level GIDIS    }
{ operations.                                                                  }


  [Global] PROCEDURE SetPosition( X, Y: Integer );

  VAR
    Data: ARRAY [ 1..3 ] OF Integer;

  BEGIN { SetPosition }
    Data[ 1 ] := Set_Position;
    Data[ 2 ] := X;
    Data[ 3 ] := Y;
    StoreGIDIS( 6, Data );
  END { SetPosition } ;


  [Global] PROCEDURE DrawLine( X, Y: Integer );

  VAR
    Data: ARRAY [ 1..3 ] OF Integer;

  BEGIN { DrawLine }
    Data[ 1 ] := Draw_Lines + 2;  { Op-code word plus 2 params (1 coord pair) }
    Data[ 2 ] := X;
    Data[ 3 ] := Y;
    StoreGIDIS( 6, Data );
  END { DrawLine } ;


  [Global] PROCEDURE DrawRelLine( X, Y: Integer );

  VAR
    Data: ARRAY [ 1..3 ] OF Integer;

  BEGIN { DrawRelLine }
    Data[ 1 ] := Draw_Rel_Lines + 2;  { Op-code word plus 2 params (1 pair) }
    Data[ 2 ] := X;
    Data[ 3 ] := Y;
    StoreGIDIS( 6, Data );
  END { DrawRelLine } ;


  [Global] PROCEDURE DrawArc( X, Y, Angle: Integer );

  VAR
    Data: ARRAY [ 1..4 ] OF Integer;

  BEGIN { DrawArc }
    Data[ 1 ] := Draw_Arcs + 3;
    Data[ 2 ] := X;
    Data[ 3 ] := Y;
    Data[ 4 ] := Angle;
    StoreGIDIS( 8, Data );
  END { DrawArc } ;
```

```
[Global(DRARC)] PROCEDURE DrawRelArc( X, Y, Angle: Integer );

VAR
  Data: ARRAY [ 1..4 ] OF Integer;

BEGIN { DrawRelArc }
  Data[ 1 ] := Draw_Rel_Arcs + 3;
  Data[ 2 ] := X;
  Data[ 3 ] := Y;
  Data[ 4 ] := Angle;
  StoreGIDIS( 8, Data );
END { DrawRelArc } ;


[Global(SORB)] PROCEDURE SetOutputRubberBand( BandType,
                                              BaseX, BaseY: Integer );

VAR
  Data: ARRAY [ 1..4 ] OF Integer;

BEGIN { SetOutputRubberBand }
  Data[ 1 ] := Set_Output_Rubber_Band;
  Data[ 2 ] := BandType;
  Data[ 3 ] := BaseX;
  Data[ 4 ] := BaseY;
  StoreGIDIS( 8, Data );
END { SetOutputRubberBand } ;


[Global] PROCEDURE SetWritingMode( Mode: WritingModes );

VAR
  Data: ARRAY [ 1..2 ] OF Integer;

BEGIN { SetWritingMode }
  Data[ 1 ] := Set_Writing_Mode;
  Data[ 2 ] := Ord( Mode );
  StoreGIDIS( 4, Data );
END { SetWritingMode } ;


[Global] PROCEDURE SetAlphabet( WhichAlphabet: Integer );

VAR
  Data: ARRAY [ 1..2 ] OF Integer;

BEGIN { SetAlphabet }
  Data[ 1 ] := Set_Alphabet;
  Data[ 2 ] := WhichAlphabet;
  StoreGIDIS( 4, Data );
END { SetAlphabet } ;
```

```
[Global] PROCEDURE SetCellRendition( Rendition: Integer );

VAR
  Data: ARRAY [ 1..2 ] OF Integer;

BEGIN { SetCellRendition }
  Data[ 1 ] := Set_Cell_Rendition;
  Data[ 2 ] := Rendition;
  StoreGIDIS( 4, Data );
END { SetCellRendition } ;


[Global(SCME)] PROCEDURE SetColorMapEntry( Map, Index, R, G, B,
                                           Mono: Integer );

VAR
  Data: ARRAY [ 1..7 ] OF Integer;

BEGIN { SetColorMapEntry }
  Data[ 1 ] := Set_Color_Map_Entry;
  Data[ 2 ] := Ord( Map   );
  Data[ 3 ] := Ord( Index );
  Data[ 4 ] := Ord( R     );
  Data[ 5 ] := Ord( G     );
  Data[ 6 ] := Ord( B     );
  Data[ 7 ] := Ord( Mono  );
  StoreGIDIS( 14, Data );
END { SetColorMapEntry } ;


[Global] PROCEDURE SetPrimaryColor( Color: Integer );

VAR
  Data: ARRAY [ 1..2 ] OF Integer;

BEGIN { SetPrimaryColor }
  Data[ 1 ] := Set_Primary_Color;
  Data[ 2 ] := Color;
  StoreGIDIS( 4, Data );
END { SetPrimaryColor } ;


[Global] PROCEDURE SetSecondaryColor( Color: Integer );

VAR
  Data: ARRAY [ 1..2 ] OF Integer;

BEGIN { SetSecondaryColor }
  Data[ 1 ] := Set_Secondary_Color;
  Data[ 2 ] := Color;
  StoreGIDIS( 4, Data );
END { SetSecondaryColor } ;
```

```
[Global] PROCEDURE SetPixelSize( Width, Height,
                                 XOffset, YOffset: [Unsafe] Integer );

VAR
  Data: ARRAY [ 1..5 ] OF Integer;

BEGIN { SetPixelSize }
  Data[ 1 ] := Set_Pixel_Size;
  Data[ 2 ] := Width;
  Data[ 3 ] := Height;
  Data[ 4 ] := XOffset;
  Data[ 5 ] := YOffset;
  StoreGIDIS( 10, Data );
END { SetPixelSize } ;


[Global] PROCEDURE SetOutputClippingRegion( UpperLeftX, UpperLeftY,
                                            Width, Height: Integer );
VAR
  Data: ARRAY [ 1..5 ] OF Integer;

BEGIN { SetOutputClippingRegion }
  Data[ 1 ] := Set_Output_Clipping_Region;
  Data[ 2 ] := UpperLeftX;
  Data[ 3 ] := UpperLeftY;
  Data[ 4 ] := Width;
  Data[ 5 ] := Height;
  StoreGIDIS( 10, Data );
END { SetOutputClippingRegion } ;


[Global] PROCEDURE EraseClippingRegion;

BEGIN { EraseClippingRegion }
  StoreGIDIS( 2, Erase_Clipping_Region );
END { EraseClippingRegion } ;


[Global] PROCEDURE BeginFill;

BEGIN { BeginFill }
  StoreGIDIS( 2, Begin_Filled_Figure );
END { BeginFill } ;


[Global] PROCEDURE EndFill;

BEGIN { EndFill }
  StoreGIDIS( 2, End_Filled_Figure );
END { Set_Current_Position } ;
```

```
[Global(CREALP)] PROCEDURE CreateAlphabet( Width, Height, Extent,
                                           WidthType: Integer );

VAR
  Data: ARRAY [ 1..5 ] OF Integer;

BEGIN { CreateAlphabet }
  Data[ 1 ] := Create_Alphabet;
  Data[ 2 ] := Width;
  Data[ 3 ] := Height;
  Data[ 4 ] := Extent;
  Data[ 5 ] := WidthType;
  StoreGIDIS( 10, Data );
END { CreateAlphabet } ;


[Global]
PROCEDURE LoadCharacterCell( Index, Width, Height: Integer;
                             VAR Raster: [ReadOnly,Unsafe] CharCell );

VAR
  i: Integer;
  Data: ARRAY [ 1..19 ] OF Integer;

BEGIN { LoadCharacterCell }
  Data[ 1 ] := Load_Character_Cell + Height + 2;
  Data[ 2 ] := Index;
  Data[ 3 ] := Width;
  FOR i := 1 TO Height DO  { Copy in the cell pattern data }
    Data[ 3 + i ] := Raster[ i ];
  StoreGIDIS( ( Height * 2 ) + 6, Data );
END { LoadCharacterCell } ;


[Global] PROCEDURE SetAreaTexture( Alphabet: Integer;
                                   Index: [Unsafe] Integer );

VAR
  Data: ARRAY [ 1..3 ] OF Integer;

BEGIN { SetAreaTexture }
  Data[ 1 ] := Set_Area_Texture;
  Data[ 2 ] := Alphabet;
  Data[ 3 ] := UAND( Index, %0'377' );  { Only use low-order byte }
  StoreGIDIS( 6, Data );
END { SetAreaTexture } ;
```

```
[Global(SCDS)] PROCEDURE SetCellDisplaySize( Width, Height: Integer );

VAR
  Data: ARRAY [ 1..3 ] OF Integer;

BEGIN { SetCellDisplaySize }
   Data[ 1 ] := Set_Cell_Display_Size;
   Data[ 2 ] := Width;
   Data[ 3 ] := Height;
   StoreGIDIS( 6, Data );
END { SetCellDisplaySize } ;


[Global(SETUNI)] PROCEDURE SetCellUnitSize( Width, Height: Integer );

VAR
  Data: ARRAY [ 1..3 ] OF Integer;

BEGIN { SetCellUnitSize }
   Data[ 1 ] := Set_Cell_Unit_Size;
   Data[ 2 ] := Width;
   Data[ 3 ] := Height;
   StoreGIDIS( 6, Data );
END { SetCellUnitSize } ;


PROCEDURE SetColorMap( Index, Red, Green, Blue: Integer );

{ SetColorMap provides a easy interface to setting the Professional's color }
{ map.  The RED, GREEN, BLUE intensity values range from 0 (nothing) to 7   }
{ (full on).                                                                }

VAR
  Mono: Unsigned;

BEGIN { SetColorMap }

  Mono := ( Red*2 + Green*4 + Blue ) DIV 7;  { NTSC conversion approximation}

  SetColorMapEntry( 0, Index, Red * 8192, Green * 8192, Blue * 8192, Mono );

END { SetColorMap } ;

{ End of module GIDIS.PAS } ;
```

A.6 THE BATONFRMS.SFF FILE

This is actually a listing of the BATONFRMS.LST file that is produced by the Frame Compiler Tool. The line numbers and cross-reference listing are produced by FCT.

```
File: BATONFRMS   FCT version 2.1  on 31-Oct-85 at 11:23 PM

  1     .!      Frame file for the Synergy demonstration program "Baton
  2     .!      Twirler".
  3
  4     .!-------------------------------------------------------------------
  5     .!      First is the vector table containing the frames that the source
  6     .!      code refers to directly.  Add new frames to the end.
  7
  8     .TABLE
  9     COLMSG, DEFQ,   EDIMSG, FERROR, LTHMSG, NOCOLR
 10     NOWIND, OTSERR, OVERVW, QIXFCM, RGBMSG, TITLE, TITLE2
 11
 12
 13     .!-------------------------------------------------------------------
 14     .!      This frame contains text strings used on the Color Map window.
 15     .!      This frame must have EXACTLY nine (9) lines of text.
 16
 17     .Frame COLMSG Message
 18
 19                     \$3+Edit Colors
 20     \
 21     Choose the color to edit with the \$32+<l>\$32- and \$32+<r>\$32-
 22     keys, and press \$32+<DO>\$32- to edit the chosen color.
 23     \       ! (must be blank)
 24     \       ! (must be blank)
 25     \       ! (must be blank)
 26     \       ! (must be blank)
 27             \$3+Press \$32+<EXIT>\$32- to accept values.
 28
 29
 30     .!-------------------------------------------------------------------
 31     .!      This frame describes the "Color map" option on the Flow Control
 32     .!      Menu (which is itself stored in the frame QIXFCM).
 33
 34     .Frame CMHELP Help
 35     .Home Column:Window:Off Row:Window:Off
 36
 37                         \$3+Color Map\ < INDEX
 38     \
 39     When run on a color system, the Batons are drawn
 40     in color.  Six separate colors are used, producing
 41     a travelling ribbon with the moving Batons.  You
 42     can control the individual 6 colors that are used.
 43     \
 44     Press \$32+<RESUME>\$32- to leave HELP.
 45     .Options Rows:1
 46     \             Skip
 47     \             Skip
 48     HELP index\  > INDEX
```

```
49
50
51    .!-----------------------------------------------------------------------
52    .!      This frame describes "Change Batons" on the Flow Menu.
53
54    .Frame CQHELP Help
55    .Home Column:Window:Off Row:Window:Off
56
57                        \$3+Change Batons\ < INDEX
58    \
59    This option displays a set-up menu that enables
60    you to change how many individual Batons there
61    are, and to change how long their tails are.
62    \
63    You can also press \$32+{INSERT HERE}\$32- and \$32+{REMOVE}\$32-
64    to add or subtract Batons.
65    \
66    Press \$32+{RESUME}\$32- to leave HELP.
67    .Options Rows:1
68    \             Skip
69    \             Skip
70    HELP index\   > INDEX
71
72
73    .!-----------------------------------------------------------------------
74    .!      This is the setup menu that controls how many Batons there are.
75
76    .Frame DEFQ Setup
77    .Home Column:Window:Center Row:Window:Center
78
79    \$3+Choose how many and of what
80    \$3+size Batons you want.
81
82    .Options Columns:1
83    Number of Batons (1-10):\                          1 NumericString
84    Length of tails (3-200):\                          2 NumericString
85    \                                                  Skip
86    Press \$32+{EXIT}\$32- to accept values.\          Skip
87
88
89    .!-----------------------------------------------------------------------
90    .!      This frame contains text strings used on the RGB color editing
91    .!      window.  This frame must have EXACTLY seven (7) lines of text.
92
93    .Frame EDIMSG Message
94
95    Choose red, green, or blue with the \$32+{l}\$32- and
96    \$32+{r}\$32- keys.  Use \$32+{u}\$32- and \$32+{d}\$32- to increase and
97    decrease the RGB components.
98    \       ! (must be blank)
99    \       ! (must be blank)
100   \       ! (must be blank)
101          \$3+Press \$32+{EXIT}\$32- to accept values.
```

```
102
103
104     .!-------------------------------------------------------------------
105     .!      This HELP frame is the main HELP Index for the application.
106
107     .Frame INDEX Help
108     .Home Column:Window:Off Row:Window:Off
109
110     \$3+Baton Twirler HELP Index
111
112     .Options Columns:2 Rows: 6
113
114     .!      First column of entries:
115
116     The Flow Menu\           > SAMPLE
117       Change Batons\         > CQHELP
118       Line thickness\        > LTHELP
119       Color map\             > CMHELP
120     \                        ! Empty spacer.
121     Another Topic\           > SAMPLE
122
123     .!      Second column of entries:
124
125     More Topics\             > SAMPLE
126       Suspending\            > SAMPLE
127       Exiting\               > SAMPLE
128       Window size\           > SAMPLE
129     \                        ! Empty spacer.
130     \                        ! Empty spacer.
131
132
133     .!-------------------------------------------------------------------
134     .!      This frame describes "Line thickness" on the Flow Menu.
135
136     .Frame LTHELP Help
137     .Home Column:Window:Off Row:Window:Off
138
139                     \$3+Line Thickness\ < INDEX
140     \
141     The Batons are initially drawn with thin lines.
142     This option lets you change the pen or brush that
143     the lines are drawn with.  The size of the brush
144     in X and Y can be controlled independently.
145     \
146     Press \$32+{RESUME}\$32- to leave HELP.
147     .Options Rows:1
148     \            Skip
149     \            Skip
150     HELP index\  > INDEX
```

```
151
152
153     .!-------------------------------------------------------------------
154     .!      This frame contains text strings used on the Line Thickness
155     .!      window.  This frame must have EXACTLY nine (9) lines of text.
156
157     .Frame LTHMSG Message
158
159     Use the arrow keys to change
160     the line thickness:
161                                                 \    ! Space for picture.
162                 \$4+Short
163                 \$32+{u}\$32-
164       \$4+Narrow\$4- \$32+{l}\$32- \$32+{d}\$32- \$32+{r}\$32-  \$4+Wide
165                 \$4+Tall
166     \
167     \$3+Press \$32+{EXIT}\$32- to accept values.
168
169
170     .!-------------------------------------------------------------------
171     .!      This message frame must contain two lines of text; they can be
172     .!      at most 40 printing characters long (80 total).  The lines are
173     .!      displayed in an error window by the FatalError() procedure.
174
175     .Frame FERROR Message
176
177     .!3456789012345678901234567890123456789O     Ruler for 40 columns.
178        This application has encountered
179       the following unexpected problem --
180
181
182     .!-------------------------------------------------------------------
183     .!      This message is displayed if the user chooses the "Color map"
184     .!      option on the Flow Control Menu, when color is not available.
185
186     .Frame NOCOLR Message
187     .Home Column:Window:Center Row:Window:Center
188     .Keys RESUME, RETURN, DO, ENTER
189
190     Because your system is not running in color
191     right now, you cannot manipulate the color
192     map.  If your Professional does have an EBO
193     option and a color monitor, and you want to
194     use color, tell Synergy by pressing \$32+{SETUP}\$32-
195     from the Synergy Main Menu.
196     \
197     Press \$32+{RESUME}\$32- to continue.
```

```
198
199
200     .!-------------------------------------------------------------------
201     .!      This message frame must contain one line of text; this line can
202     .!      be at most 40 printing characters long (80 total).  The line is
203     .!      displayed in an error window if the program cannot create its
204     .!      main window (for any reason).
205
206     .Frame NOWIND Message
207
208     .!3456789012345678901234567890123456789O     Ruler for 40 columns.
209       Cannot create the application window.
210
211
212     .!-------------------------------------------------------------------
213     .!      This message frame must contain one line of text; this line can
214     .!      be at most 40 printing characters long (80 total).  The line is
215     .!      displayed in an error window if the program ever blows up with
216     .!      some unexpected error fault.
217
218     .Frame OTSERR Message
219
220     .!3456789012345678901234567890123456789O     Ruler for 40 columns.
221        Something has really gone wrong!
222
223
224     .!-------------------------------------------------------------------
225     .!      This HELP frame is displayed if the user presses HELP while the
226     .!      program is in "free-running" mode.  In this case, there is no
227     .!      specific state or context that the program might give a
228     .!      context-sensitive HELP frame about; so general introductory-
229     .!      type HELP is given.
230
231     .Frame OVERVW Help
232     .Home Column:Window:Off Row:Window:Off
233
234                     \$3+Overview of Baton Twirler
235     \
236     Welcome to the Synergy demonstration application, Baton
237     Twirler.  At any time while the Batons are twirling, you
238     can press \$32+{F5}\$32- to temporarily pause Baton Twirler and
239     return to the Synergy Main Menu (and then change the
240     window size or move it around).  Pressing \$32+{EXIT}\$32- will
241     stop Baton Twirler and return to the Synergy Main Menu.
242     \
243     You can also press \$32+{F11}\$32-, which will display a menu of
244     options that enable you to modify the Batons in a variety
245     of ways.
246     \
247     Press \$32+{RESUME}\$32- to leave HELP.
248     .Options Rows:1
249     \               Skip
250     \               Skip
251     \               Skip
252     HELP index\   > INDEX
253
```

```
254
255     .!-----------------------------------------------------------------------
256     .!      This is the main Flow Control Menu for the program.
257
258     .Frame QIXFCM Flow
259     .Home Column:Screen:Left Row:Screen:Top
260
261     .Options Rows:1
262     Personalize
263
264     .Options Columns:1      ! Personalize Menu
265     Change Batons\          2 ?CQHELP
266     Line thickness\         3 ?LTHELP
267     Color map\              1 ?CMHELP
268
269
270     .!-----------------------------------------------------------------------
271     .!      This message contains exactly one line, of exactly three
272     .!      characters.  The characters are the labels to apply to the RED,
273     .!      GREEN, BLUE color component squares that the editing window of
274     .!      the "Color map" option displays.
275
276     .Frame RGBMSG Message
277     RGB\
278
279
280     .!-----------------------------------------------------------------------
281     .!      This HELP frame is just a sample.
282
283     .Frame SAMPLE Help
284     .Home Column:Window:Off Row:Window:Off
285
286                     \$3+Sample HELP Frame\ < INDEX
287     \
288     This is a sample Synergy HELP frame that shows the
289     sorts of things that can be done in any Synergy
290     frame (HELP, menu, setup, etc.).  First, here's a
291     sample use of the DEC Multinational (@¼½àéîõü) and
292     Special Graphics characters <X80><PU1><VTS><PLU><PU1><X80> that are
293     available in the Synergy fonts.
294     \
295     Here is a sample use of renditions:  You can use
296     \$3-Dim\$1+, Normal and \$3+bold\$2- intensities plus \$8+underline\$8-,
297      \$4+Italic\$4- and \$16+ Hilight \$16- for emphasis, and for
298     keyboard function key notation the \$32+<BOXED>\$32- font.
299     \
300     Press \$32+<RESUME>\$32- to leave HELP.
301     .Options Rows:1
302     \               Skip
303     \               Skip
304     HELP index\   > INDEX
```

```
305
306
307     .!---------------------------------------------------------------------
308     .!      This message line is the title for the main program window.
309
310     .Frame TITLE Message
311     Baton Twirler\
312
313
314     .!---------------------------------------------------------------------
315     .!      This message line is the secondary title for the main program
316     .!      window.  This title is only used the VERY first time the
317     .!      application is run, and then ONLY for the first thirty seconds
318     .!      or so that the program runs; after that time is up, the program
319     .!      changes the title to use the text in frame TITLE (above).
320
321     .Frame TITLE2 Message
322     Baton Twirler (press HELP for details)\

**** No errors.
**** 8 blocks in BATONFRMS.OFF.
**** 18 frames.

Cross Reference listing.

LINE  FRAME    Referenced from line/frame.

  34  CMHELP    119/INDEX     267/QIXFCM
  17  COLMSG      9/.TABLE
  54  CQHELP    117/INDEX     265/QIXFCM
  76  DEFQ        9/.TABLE
  93  EDIMSG      9/.TABLE
 175  FERROR      9/.TABLE
 107  INDEX      37/CMHELP     48/CMHELP     57/CQHELP     70/CQHELP    139/LTHELP
                 150/LTHELP    252/OVERVW    286/SAMPLE    304/SAMPLE
 136  LTHELP    118/INDEX     266/QIXFCM
 157  LTHMSG      9/.TABLE
 186  NOCOLR      9/.TABLE
 206  NOWIND     10/.TABLE
 218  OTSERR     10/.TABLE
 231  OVERVW     10/.TABLE
 258  QIXFCM     10/.TABLE
 276  RGBMSG     10/.TABLE
 283  SAMPLE    116/INDEX     121/INDEX     125/INDEX     126/INDEX     127/INDEX
                 128/INDEX
 310  TITLE      10/.TABLE
 321  TITLE2     10/.TABLE
```

A.7 THE BATON.CMD FILE

```
; Task builder Command (.CMD) File for the sample Synergy program.

; The first line tells PAB to create the task image file with checkpointing
; enabled and floating point context space allocated (both of which you almost
; ALWAYS want).  The details of the task structure are contained in the .ODL
; file, referenced by the /MP switch.

Baton/CP/FP = Baton/MP

; We can give the eventual running task any name we want.  This name can be
; used by other tasks running under P/OS to communicate with our task.

Task    = BATON

; PRO/Pascal does not use the conventional RSX-11M-PLUS (and thus P/OS) area of
; the task reserved for stack space usage.  Rather, it allocates the stack
; itself in the PSECT $SDAT$.  Thus it would be wasting address space for us to
; allocate room to the P/OS default stack (PAB defaults to 256 words).
; Allocating zero words would be risky though, since Pascal DOES use the
; default stack momentarily until its run-time system is initialized.

Stack   = 30

; This task needs the PRO/Pascal run-time system (PASRES), which in turn needs
; the Record Management Services library (RMSRES).  At our option we have
; included the DECnet extensions to RMS (DAPRES) to support remote file access.

CLSTR   = PASRES,RMSRES,DAPRES:RO

; The following line modifies a part of the PRO/Pascal run-time system.  As
; described above, the Pascal Stack (containing subroutine linkage and local
; variable storage) is allocated in the $SDAT$ PSECT.  Normally, the Pascal
; Heap (containing dynamic memory from NEW/DISPOSE, file I/O buffers, etc.) is
; automatically allocated at run-time by extending the task address region
; (with EXTK$ directives).  But this requires a checkpoint operation whenever
; the task is extended, and checkpointing is slow.  So we can ask PRO/Pascal
; not to ever extend the task, and instead use the bottom portion of the stack
; space (the $SDAT$ PSECT) to double as the heap.  CAUTION: If you ever do
; this, MAKE SURE you have allocated a large enough $SDAT$ PSECT!  This usually
; takes some trial and error to allocate enough space (but if you allocate lots
; of extra room to be on the safe side, you use up extra address space).

GBLPAT  = Baton:$NOEXT:240

; Lastly, define values for some global variables used by the program.

GBLDEF  = SCRLUN:5      ; Logical unit number for video output
GBLDEF  = SCREFN:1      ; Event Flag Number for video output writes.
GBLDEF  = IDLE$:6       ; Event Flag Number for keyboard input idles.
//
```

A.8 THE BATON.ODL FILE

```
; Overlay Description (.ODL) File for the sample Synergy program.

; The Pascal $ODAT$ psect (where all the outer level variables live in a
; program with the [Overlaid] attribute) and the Heap (where all dynamic
; memory comes from) are made non-disk resident to reduce the size of the task
; image file.
; This is accomplished by grouping those modules of the program into one
; segment (called OFLINE here), and including in that segment a dummy module
; created with the .NAME directive (called SAVER here) that is given the NODSK
; attribute.  Any segment which has the NODSK attribute will not be allocated
; any space in the on-disk .TSK task image file by the Task Builder.  For
; simplicity, this whole segment (OFLINE) is placed as a co-tree off the end of
; the main .ROOT structure.

        .PSECT  $ODAT$, RW, D, GBL, REL, OVR    ; Pascal Overlaid data.
        .PSECT  $SDAT$, RW, D, GBL, REL, OVR    ; Pascal stack/heap.
        .PSECT  $SDAT0, RW, D, GBL, REL, OVR    ; Top of stack/heap.

        .NAME   SAVER, NODSK                    ; Dummy module for NODSK.

; Here is the entire segment put together:

OFLINE: .FCTR   $ODAT$ - $SDAT$ - $SDAT0 - SAVER


; Here are the actual code modules for this task:

LIBS:   .FCTR   LB:[1,5]PasLib/LB - LB:[1,5]WinLib/LB
MODULS: .FCTR   Baton - GIDIS - GetAKey - KBServ - ReadMesg


; We need RMS in this task.  For P/OS V2.0 and greater, we might as well
; include PRO/DECnet facilities in the task so that we could do transparent
; remote file accesses if we ever worked with files.  To include DECnet,
; simply reference DAPRLX instead of RMSRLX, and add DAPRES to the CLSTR line
; in the task builder command file.

@LB:[1,5]DAPRLX


        .ROOT   MODULS - LIBS - RMSROT, OFLINE
        .END
```

A.9 THE BATON.INS FILE

```
!Synergy/I2
Name "Baton Twirler"
!
File [ZZPROVUE]SYNCHK2.TSK/Delete
File [ZZPROVUE]SYNERR.HLP/Keep
Execute [ZZPROVUE]SYNCHK2.TSK/Ins
Execute [ZZPROVUE]REMEXE.TSK/Rem
!
File BATON.TSK/Delete
File [ZZBATON]BATONFRMS.OFF/Delete
Execute [ZZPROVUE]INSAPP.TSK/Ins
!Install [ZZSYS]PASRES.TSK/Library
!Install BATON.TSK/Task
!Run BATON
!
Install [ZZPROVUE]SYNRUN.TSK/Task
Run WI$MGR
```

A.9.1 The BATON.INB File

```
!Synergy/I2
Name "Baton Twirler"
File [ZZPROVUE]SYNCHK2.TSK/Delete
File [ZZPROVUE]SYNERR.HLP/Keep
EXECUTE [ZZPROVUE]SYNCHK2.TSK/Ins/USR
EXECUTE [ZZPROVUE]SYNCHK2.TSK/Ins
!
Execute [ZZPROVUE]REMEXE.TSK/Rem/USR
!
File BATON.TSK/Delete/Network
File [ZZBATON]BATONFRMS.OFF/Delete/Cluster
!
Execute [ZZPROVUE]INSAPP.TSK/Ins/USR
Execute [ZZPROVUE]INSAPP.TSK/Ins
!
!Install [ZZSYS]PASRES.TSK/Library
!Install BATON.TSK/Task/Network
!Run BATON
!
Install [ZZPROVUE]SYNRUN.TSK/Task/Cluster
Run WI$MGR
```

A.10 THE BUILD.CMD FILE

The Baton Twirler source files can be recompiled if you have Version 1.2 (or later) of PRO/PASCAL installed on your Tool Kit system. This indirect command file can be used to compile the sources and link the result into a task image.

```
! Indirect Command File to build the sample Synergy program.
! This command file assumes that the frame file BATONFRMS.SFF has been run
! through the Frame Compiler Tool to produce BATONFRMS.PAS and BATONFRMS.OFF.
! (To do this, invoke FCT and specify "BATONFRMS" as the file to compile.)
!
! You invoke this command file by typing @BUILD from PRO/Tool Kit DCL.
!
PASCAL Baton/NoDebug/NoCheck
PASCAL GIDIS/NoDebug/NoCheck
MACRO GetAKey
MACRO KBServ
MACRO ReadMesg
!
LINK @Baton
```

APPENDIX B

APPENDIX B

TABLE OF SYNERGY SERVICES

Table B-1: Table of Synergy Services

SYMBOL	NAME	PAGE
CLOSEM	Close Frame File	7-3
DFLOW	Dynamic Flow Control Menu	7-12
DMESSA	Dynamic Message Frame	7-17
DMULTI	Dynamic Multiple-Choice Menu	7-10
DSINGL	Dynamic Single-Choice Menu	7-9
EXFLOW	Static Flow Control Menu	7-12
EXHELP	Static Help Menu	7-10
EXMESS	Static Message Frame	7-16
EXMULT	Static Multiple-Choice Menu	7-10
EXSING	Static Single-Choice Menu	7-9
MGTCB	Expand Call-Back Code	4-22
NEWFLE	Dynamic New File Name	7-23
OLDFLE	Dynamic Old File Name	7-21
OPENME	Open Frame File	7-2
WICHW	Change Size and Position of Window	6-5
WICOLD	Get Selected File Name	7-21
WICRM	Create Menu Window	7-30
WICRS	Create String Editing Window	7-26
WICRW	Create a Window	6-5
WIDEM	Destroy Menu Window	7-33
WIDES	Destroy String Editing Window	7-28
WIDON	Application Done	5-1
WIDSW	Destroy a Window	6-6
WIEF	Edit String Field	7-28
WIENM	Change Option in a Menu	7-33
WIERW	Display Error Window	6-7
WIEWT	End Wait Message	6-7
WIGEW	Get Window Parameters	6-8
WIGKM	Get Key from a Menu	7-33
WIGKS	Get Key from String Editing Window	7-29
WIHDR	Change Header Line	7-34
WIHDW	Hide a Window	6-8
WIIDA	ID of a Window at a Point	6-9

Glossary

active application
: The application using the front window; the current application.

application
: A task, or a group of closely interacting tasks.

character-passing buffer
: An 80-character buffer that is passed between the window server and all Synergy applications, used to hold keyboard characters that have been read but not yet processed.

dynamic call
: A call on a Synergy service in which all information is passed by parameters.

EBO
: Extended Bitmap Option.

front window
: The window that is addressable through the GIDIS interpreter.

GIDIS
: General Image Display Instruction Set; a Digital proprietary graphics protocol supported on the Professional 300 Series.

GOS
: GIDIS Output Space - an isotropic mapping of screen coordinates used in all GIDIS instructions.

hardware pixel
The picture element defined by the video display hardware.

header lines
Lines of text at the top of a window.

isotropic
Measured in equal units along the vertical and horizontal axes.

logical pixel
A software defined picture element that is isotropic and is mapped to hardware pixels.

options
The items that can be selected on a menu.

owner task
The task in an application whose task name is used to represent the owner of the windows created by the application.

P/OS
The Professional Operating System.

positioning unit
A specified number of GOS units in both the horizontal and vertical directions, on which a window may be positioned.

raster
A rectangular section of a bitmap display.

stackable windows
A sequence of windows defined and manipulated as a stack, so that only the window on the front of the stack may be manipulated.

static call
A call on a Synergy service in which a frame ID is a parameter. The frame ID locates a frame in the frame file that supplies most of the information needed by the service.

stopped
A state in P/OS in which a task does not execute and does not compete for memory.

suspend
An action in which an application relinquishes control to Synergy and does no terminal output or keyboard input until Synergy permits it to resume.

task
The basic unit of execution in P/OS.

title line
A line of text that is incorporated into the top edge of the windowframe.

white border
An optional, narrow white area that separates the windowframe from the writable area of a window.

wildcard
An asterisk used in a portion of a file specification in order that matching will be bypassed on that portion.

window
A rectangular section of the screen that an application can use to display information.

windowframe
A thin, dark line that surrounds a window on the Synergy screen.

window manager
A Synergy task that displays the Synergy Main Menu and interacts with the user to control window size and position and to start and stop applications.

window server
A Synergy task that does the actual work involved in manipulating windows and in providing menu and HELP services.

writable area
The area of a window that can be addressed in GIDIS output instructions by the application that owns the window.

Synergy
Programmer's Manual
Order No. AA-EU61A-TH

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized?
Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or
Country

Please cut along this line.

--- Do Not Tear - Fold Here and Tape ---

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

Professional 300 Series Publications
DIGITAL EQUIPMENT CORPORATION
146 MAIN STREET MLO5-2/T77
MAYNARD, MA 01754-2571

---- Do Not Tear - Fold Here ----

Cut Along Dotted Line