Spooler Programmer’s Guide

Abstract

This manual describes the Spooler subsystem, its uses, and its applications. This guide is intended for application programmers who want to use the Spooler interface procedures to spool data programmatically.

Product Version

Spooler D48, H01, and H02

Supported Releases

This manual supports D48.03, G06.15, and H06.03 and all subsequent release version updates until otherwise indicated in a new edition.

Part Number	Published
522287-002	August 2012

Document History

Part Number	Product Version	Published
106813	Spooler D41	January 1995
103190	Spooler D41	July 1997
135720	Spooler D41	August 1997
522287-001	Spooler D48	August 2002
522287-002	Spooler D48, H01, and H02	August 2012

Legal Notices

Legal Notice

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial

Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor’s standard commercial license.

The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

Export of the information contained in this publication may require authorization from the U.S. Department of Commerce.

Microsoft, Windows, and Windows NT are U.S. registered trademarks of Microsoft Corporation.

Intel, Pentium, and Celeron are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java is a registered trademark of Oracle and/or its affiliates.

Motif, OSF/1, UNIX, X/Open, and the "X" device are registered trademarks, and IT DialTone and The Open Group are trademarks of The Open Group in the U.S. and other countries.

Open Software Foundation, OSF, the OSF logo, OSF/1, OSF/Motif, and Motif are trademarks of the Open Software Foundation, Inc. OSF MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THE OSF MATERIAL PROVIDED HEREIN, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. OSF shall not be liable for errors contained herein or for incidental consequential damages in connection with the furnishing, performance, or use of this material.

© 1990, 1991, 1992, 1993 Open Software Foundation, Inc. The OSF documentation and the OSF software to which it relates are derived in part from materials supplied by the following:© 1987, 1988, 1989 Carnegie-Mellon University. © 1989, 1990, 1991 Digital Equipment Corporation.

© 1985, 1988, 1989, 1990 Encore Computer Corporation. © 1988 Free Software Foundation, Inc. © 1987, 1988, 1989, 1990, 1991

Hewlett-Packard Company. © 1985, 1987, 1988, 1989, 1990, 1991, 1992 International Business Machines Corporation. © 1988, 1989 Massachusetts Institute of Technology. © 1988, 1989, 1990 Mentat Inc. © 1988 Microsoft Corporation. © 1987, 1988, 1989, 1990, 1991, 1992 SecureWare, Inc. © 1990, 1991 Siemens Nixdorf Informationssysteme AG. © 1986, 1989, 1996, 1997 Sun Microsystems, Inc. © 1989,1990, 1991 Transarc Corporation.OSF software and documentation are based in part on the Fourth Berkeley Software Distribution under license from The Regents of the University of California. OSF acknowledges the following individuals and institutions for their role in its development: Kenneth C.R.C. Arnold, Gregory S. Couch, Conrad C. Huang, Ed James, Symmetric Computer Systems, Robert Elz. © 1980, 1981, 1982, 1983, 1985, 1986, 1987, 1988, 1989 Regents of the University of California.

Spooler Programmer's Guide

Index | Examples | Figures | Tables

1. Introduction to the Spooler Subsystem (continued)

2. Using the Spooler Interface Procedures

3. Using the Spooler Print Procedures, Print Processes, and Perusal Processes

4. Spooler Procedure Calls

4. Spooler Procedure Calls (continued)

Index

Examples

Figures

Tables

Tables (continued)

What's New in This Manual

Manual Information

Abstract

This manual describes the Spooler subsystem, its uses, and its applications. This guide is intended for application programmers who want to use the Spooler interface procedures to spool data programmatically.

Product Version

Spooler D48, H01, and H02

Supported Releases

This manual supports D48.03, G06.15, and H06.03 and all subsequent release version updates until otherwise indicated in a new edition.

Part Number	Published
522287-002	August 2012

Document History

Part Number	Product Version	Published
106813	Spooler D41	January 1995
103190	Spooler D41	July 1997
135720	Spooler D41	August 1997
522287-001	Spooler D48	August 2002
522287-002	Spooler D48, H01, and H02	August 2012

Changes in the August 2012 manual:

- Updated information about SPOOLEREQUEST error code under SPOOLEREQUEST[2] Procedure on page 4-42.
- Updated the SPOOLEREQUEST error code details on C-8.
- Added an additional criteria for SPOOLERSTATUS2 on 4-51.
- Added a note about the status of a collector for large values on 4-53.

Changes in the August 2002 manual:

The following additions and changes have been added to the guide:

- In Section 1, Introduction to the Spooler Subsystem new information about the specification of the unit size and buffer size for SPOOLCOM has been added to the Unit Size subsection.
- In Section 4, Spooler Procedure Calls:
  - Material has been added to the description of the PRINTSTART[2] Procedure call.
  - A complete list for flag and devflagx bit values defined in the SPOOLERSTATUS2 Procedure call has been added.
  - The extended-level-3-buff definition has been corrected in the SPOOLSTART Procedure.

This publication has been updated to reflect new product names:

- Since product names are changing over time, this publication might contain both HP and Compaq product names.
- Product names in graphic representations are consistent with the current product interface.

About This Manual

The *Spooler Programmer's Guide* describes the Spooler subsystem, its uses, and its applications for experienced programmers.

This guide includes

- Detailed information about the Spooler subsystem and its components
- An explanation of how to use the Spooler interface and print procedures
- Complete syntax and considerations for all Spooler-related procedures
- Sample print and perusal processes
- Descriptions of Spooler-related errors

Who Should Use This Manual

This guide is intended for application programmers who want to use the Spooler interface procedures to spool data programmatically. Users who want to use the spooler interactively should refer to the *Spooler Utilities Reference Manual*.

Note. Some of the tasks described in this guide are normally performed by a system operator (user ID 255, *n*).

How This Manual Is Organized

Table i summarizes the contents of this manual.

Table i. Contents

Section or Appendix	Title	Contents
1	Introduction to the Spooler Subsystem	Provides detailed information on the Spooler subsystem and its components that is of special interest to programmers. All programmers should read this section before reading any of the other parts of this guide.
2	Using the Spooler Interface Procedures	Describes the use of the Spooler interface procedures. This section supplies examples of spooling on three levels, along with COBOL spooling and spooling from a NonStop process pair for most of these levels. The latter examples include discussion of zero and nonzero sync depth.
3	Using the Spooler Print Procedures, Print Processes, and Perusal Processes	Describes the use of the Spooler print procedures. Information on print processes includes reading the startup message, retrieving and printing the spooled data, and communicating with the spooler supervisor. This section also discusses how to handle device errors and PRINTREAD errors, gives pointers on debugging print processes, and covers combining data retrieval with spooler communication. This section also describes a user-written perusal process which can access spooled data without communicating with the spooler supervisor.
4	Spooler Procedure Calls	Presents the complete syntax and considerations for all Spooler-related procedures. The SPOOLERSTATUS Struts are of particular interest to application programmers for use in finding the status of Spooler components.
A	Sample Print Process	Presents a sample print process.
B	Sample Perusal Process	Presents a sample perusal process.
C	Spooler-Related Errors	Describes errors codes returned by the Spooler interface and by the print and utility procedures.

Related Manuals

Before reading this guide, you should be familiar with the following manuals:

- *Spooler Utilities Reference Manual*

  This manual contains the complete syntax, considerations, and examples for the utilities Peruse, Spoolcom, Font, and RPSetup.

- *Guardian Procedure Errors and Messages Manual*

  This manual describes in detail file-system and other types of errors that are referred to by number in some Peruse and Spoolcom messages

Also recommended, although not necessary, are the following manuals:

- *Guardian Programmer's Guide*,

  This manual describes the system procedures that programmers can call from within their programs.

- *Guardian Procedure Calls Reference Manual*

  This manual contains the procedure-call syntax for procedures supported by HP that can be called from the Transaction Application Language (TAL).

- *TAL Reference Manual*

  This manual provides syntax descriptions and error messages for TAL (Transaction Application Language) for system and application programmers.

- *COBOL85 Manual*

  This manual describes the HP implementation of the 1985 version of the COBOL language.

Notation Conventions

General Syntax Notation

The following list summarizes the notation conventions for syntax presentation in this manual.

UPPERCASE LETTERS. Uppercase letters indicate keywords and reserved words; enter these items exactly as shown. Items not enclosed in brackets are required. For example:

```
MAXATTACH
```

lowercase italic letters. Lowercase italic letters indicate variable items that you supply. Items not enclosed in brackets are required. For example:

```
file-name
```

computer type. `Computer type` letters within text indicate C and Open System Services (OSS) keywords and reserved words; enter these items exactly as shown. Items not enclosed in brackets are required. For example:

```
myfile.c
```

italic computer type. *`Italic computer type`* letters within text indicate C and Open System Services (OSS) variable items that you supply. Items not enclosed in brackets are required. For example:

```
pathname
```

[ ] Brackets. Brackets enclose optional syntax items. For example:

```
TERM [\system-name.]$terminal-name

INT[ERRUPTS]
```

A group of items enclosed in brackets is a list from which you can choose one item or none. The items in the list may be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
LIGHTS [ ON              ]
       [ OFF             ]
       [ SMOOTH [ num ] ]

K [ X | D ] address-1
```

{ } Braces. A group of items enclosed in braces is a list from which you are required to choose one item. The items in the list may be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
LISTOPENS PROCESS { $appl-mgr-name }
                  { $process-name  }

ALLOWSU { ON | OFF }
```

| Vertical Line. A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
INSPECT { OFF | ON | SAVEABEND }
```

… Ellipsis. An ellipsis immediately following a pair of brackets or braces indicates that you can repeat the enclosed sequence of syntax items any number of times. For example:

```
M address-1 [ , new-value ]...

[ - ] {0|1|2|3|4|5|6|7|8|9}...
```

An ellipsis immediately following a single syntax item indicates that you can repeat that syntax item any number of times. For example:

```
"s-char..."
```

Punctuation. Parentheses, commas, semicolons, and other symbols not previously described must be entered as shown. For example:

```
error := NEXTFILENAME ( file-name ) ;

LISTOPENS SU $process-name.#su-name
```

Quotation marks around a symbol such as a bracket or brace indicate the symbol is a required character that you must enter as shown. For example:

```
"[" repetition-constant-list "]"
```

Item Spacing. Spaces shown between items are required unless one of the items is a punctuation symbol such as a parenthesis or a comma. For example:

```
CALL STEPMOM ( process-id ) ;
```

If there is no space between two items, spaces are not permitted. In the following example, there are no spaces permitted between the period and any other items:

```
$process-name.#su-name
```

Line Spacing. If the syntax of a command is too long to fit on a single line, each continuation line is indented three spaces and is separated from the preceding line by a blank line. This spacing distinguishes items in a continuation line from items in a vertical list of selections. For example:

```
ALTER [ / OUT file-spec / ] CONTROLLER

   [ , attribute-spec ]...
```

!i and !o. In procedure calls, the !i notation follows an input parameter (one that passes data to the called procedure); the !o notation follows an output parameter (one that returns data to the calling program). For example:

```
CALL CHECKRESIZESEGMENT (  segment-id                    !i
                         , error        ) ;              !o
```

!i,o. In procedure calls, the !i,o notation follows an input/output parameter (one that both passes data to the called procedure and returns data to the calling program). For example:

```
error := COMPRESSEDIT ( filenum ) ;                      !i,o
```

!i:i. In procedure calls, the !i:i notation follows an input string parameter that has a corresponding parameter specifying the length of the string in bytes. For example:

```
error := FILENAME_COMPARE_ (  filename1:length           !i:i
                            , filename2:length ) ;       !i:i
```

!o:i. In procedure calls, the !o:i notation follows an output buffer parameter that has a corresponding input parameter specifying the maximum length of the output buffer in bytes. For example:

```
error := FILE_GETINFO_ (  filenum                          !i
                        , [ filename:maxlen ] ) ;          !o:i
```

Notation for Messages

The following list summarizes the notation conventions for the presentation of displayed messages in this manual.

Nonitalic text. Nonitalic letters, numbers, and punctuation indicate text that is displayed or returned exactly as shown. For example:

```
Backup Up.
```

lowercase italic letters. Lowercase italic letters indicate variable items whose values are displayed or returned. For example:

```
p-register
```

```
process-name
```

[ ] Brackets. Brackets enclose items that are sometimes, but not always, displayed. For example:

```
Event number = number [ Subject = first-subject-value ]
```

A group of items enclosed in brackets is a list of all possible items that can be displayed, of which one or none might actually be displayed. The items in the list might be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
LDEV ldev [ CU %ccu | CU %... ] UP [ (cpu,chan,%ctlr,%unit) ]
```

{ } Braces. A group of items enclosed in braces is a list of all possible items that can be displayed, of which one is actually displayed. The items in the list might be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
LBU { X | Y } POWER FAIL
```

```
process-name State changed from old-objstate to objstate
{ Operator Request. }
{ Unknown.          }
```

| Vertical Line. A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
Transfer status: { OK | Failed }
```

% Percent Sign. A percent sign precedes a number that is not in decimal notation. The %þnotation precedes an octal number. The %Bþnotation precedes a binary number. The %Hþnotation precedes a hexadecimal number. For example:

```
%005400
```

```
P=%p-register E=%e-register
```

Notation for Management Programming Interfaces

UPPERCASE LETTERS. Uppercase letters indicate names from definition files; enter these names exactly as shown. For example:

```
ZCOM-TKN-SUBJ-SERV
```

lowercase letters. Words in lowercase letters are words that are part of the notation, including Data Definition Language (DDL) keywords. For example:

```
token-type
```

!r. The !r notation following a token or field name indicates that the token or field is required. For example:

```
ZCOM-TKN-OBJNAME      token-type ZSPI-TYP-STRING.          !r
```

!o. The !o notation following a token or field name indicates that the token or field is optional. For example:

```
ZSPI-TKN-MANAGER      token-type ZSPI-TYP-FNAME32.         !o
```

Change Bar Notation

Change bars are used to indicate substantive differences between this edition of the manual and the preceding edition. Change bars are vertical rules placed in the right margin of changed portions of text, figures, tables, examples, and so on. Change bars highlight new or revised information. For example:

> The message types specified in the REPORT clause are different in the COBOL85 environment and the Common Run-Time Environment (CRE).
>
> The CRE has many new message types and some new message type codes for old message types. In the CRE, the message type SYSTEM includes all messages except LOGICAL-CLOSE and LOGICAL-OPEN.

HP Encourages Your Comments

HP encourages your comments concerning this document. We are committed to providing documentation that meets your needs. Send any errors found, suggestions for improvement, or compliments to **docsfeedback@hp.com**. Include the document title, part number, and any comment, error found, or suggestion for improvement concerning this document.

1

Introduction to the Spooler Subsystem

The Spooler subsystem serves as a buffer between an application writing to a print device and the device itself, allowing applications to create hard-copy output without affecting the status of print devices. The Spooler subsystem performs its work by storing the output from each application on disk and monitoring the status of print devices. When the appropriate device becomes available, the Spooler subsystem prints the output.

Topics described in this section include

Spooler and Spooler Plus Comparison

Spooler Plus is an optional product containing Spoolcom and Peruse modules that can be used to replace the Spoolcom and Peruse modules provided by the D41 or later product versions of the Spooler subsystem. You can use the Spooler Plus Spoolcom and Peruse utilities to configure and manage expanded configurations of the Spooler subsystem. You *must* use these utilities if the maximum jobs has been configured above 8191.

The Spooler Plus subsystem is described in the *Spooler Plus Utilities Reference Manual*.

Spooler Features

Features of the Spooler subsystem include:

- Continuous operation. The Spooler subsystem keeps working even if a processor fails.
- Fault-tolerant applications. You can run application processes on the HP NonStop system to avoid loss or duplication of data in the event of a failure.
- Custom print processes. You can write your own print processes or use the print processes supplied by HP.
- Flexible routing structure. The Spooler subsystem allows the destination of program output to be changed after the program has been run.
- Interactive user control. The Spoolcom utility permits you to inspect or alter the status of jobs and devices, specify the routing structure, and initialize the Spooler subsystem.
- Support for batch processing. You can specify attributes for jobs before they are created using the CLASS SPOOL DEFINE procedure call.

Note. Because **spooler** is an industry-standard term used to describe a printer spooling system, it used throughout the remainder of this guide to refer to the Spooler subsystem.

Spooler Components

The spooler consists of a set of related processes and procedures, as shown in Table 1-1.

Table 1-1. Spooler Processes and Procedures

Process	Function	Related Procedures
Spooler supervisor	Monitors and controls operation of the spooler	
Collector	Transfers data from applications to disk	Spooler interface procedures
Print process	Transfers data from disk to print devices	Spooler print procedures allow users to write their own print processes
Perusal processes	Reads spooled data from jobs and modifies attributes of jobs	Spooler utility procedures and print procedures work together
Spoolcom	Provides system operator's interface to the spooler	Spooler utility procedures issue commands to the spooler

Spooler Supervisor

The spooler supervisor process monitors and communicates with the other spooler processes and decides when and where to print jobs.

Collector

All output that is to be printed is sent to the collector. Spooler collection processes (collectors) accept output from applications and store the output on disk.

Print Processes

A set of print processes performs the task of retrieving and printing spooled data. The FASTP print process is supplied with the spooler; to write your own print processes, you use the print procedures described in Section 3, Using the Spooler Print Procedures, Print Processes, and Perusal Processes.

Perusal Processes

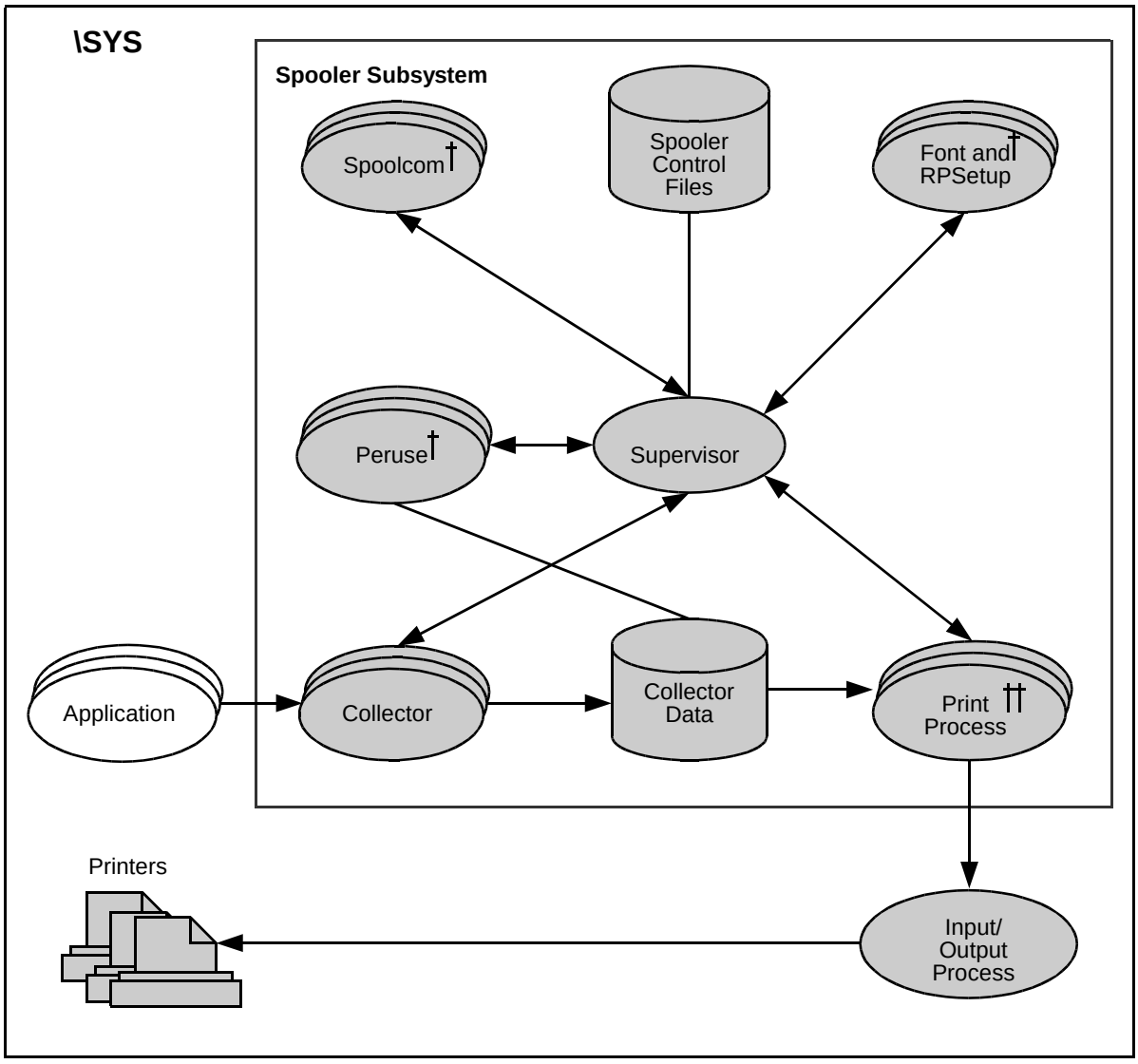
Perusal processes examine a job's data file without intervention by the spooler supervisor. They use the print procedures to read data from a collector's data file. Perusal processes, unlike print processes, function without interactive communication with the supervisor. They are useful for studying the spooled data from a job before it is printed. The Peruse utility is an example of a perusal process. The complete syntax and considerations of Peruse appear in the *Spooler Utilities Reference Manual*.

Spoolcom

The Spoolcom utility allows you to declare and initialize collectors and print processes, define and modify the routing structure, control the printing of jobs, and obtain the status of any component of the spooler. You can run Spoolcom from a terminal to translate your commands into messages to the spooler supervisor, which carries out your instructions. You can enter Spoolcom commands either interactively from a terminal or from a command file, which you specify by using the IN option of the Spoolcom command. Spoolcom performs its work by calling the spooler utility procedures, which can also be used by programmers who want to control the spooler from their programs. The complete syntax and considerations of Spoolcom appear in the *Spooler Utilities Reference Manual*.

An overview of a spooler subsystem is shown in Figure 1-1, which includes the spooler utilities: Peruse, Spoolcom, Font, and RPSetup. By using print and spooler procedures described in this manual, you can write an application that performs some or all the functions of these utilities.

Figure 1-1. Components of the Spooler



Legend

† Applications can perform all the functions of Peruse, Spoolcom, Font, and RPSetup; therefore, these programs can be considered applications.

†† A print process can be FASTP, PSPOOL, PSPOOLB, or user written

VST011.vsd

Disk Files Maintained by the Spooler

The spooler maintains two sets of disk files: data files and control files.

Data files are unstructured files containing the spooled data from all the jobs in the spooler subsystem. Each collector has its own data file to hold jobs written to it. The collector maintains an internal structure in the data file to ensure the integrity of jobs and most efficiently use the storage space. The spooler library print routines are used to fetch jobs from the collector data file. You can create data files at any time prior to starting the collector.

Control files contain information regarding the attributes of all components of the spooler: devices, processes, jobs, and so on.

The control file name has the same form as a disk file name. The control file name you supply when you start the spooler is used to create these control files. You can specify only seven letters or digits in the control file name; the spooler appends a integer (0 through 9) to the control file name you specified. All 10 control file names should be reserved for use by the spooler.

A particular control file defines a particular spooler subsystem. When a spooler has been brought to an orderly halt (by means of the Spoolcom command SPOOLER DRAIN), you can always restart it without any initialization by passing the control file name to the supervisor (using the SPOOL command).

See the *Guardian System Operations Guide* for more information on how data file names and control file names are selected and assigned.

Multiple Spoolers

The spooler can handle up to 8191 jobs and 511 output devices at one time, which should suffice for most applications. However, if you need more spooling capability, you can run any number of spooler subsystems at the same time.

To create additional spooler subsystems, simply follow the starting procedure as many times as needed, each time using a different control file name and process name. When running multiple spoolers, you should consider the following:

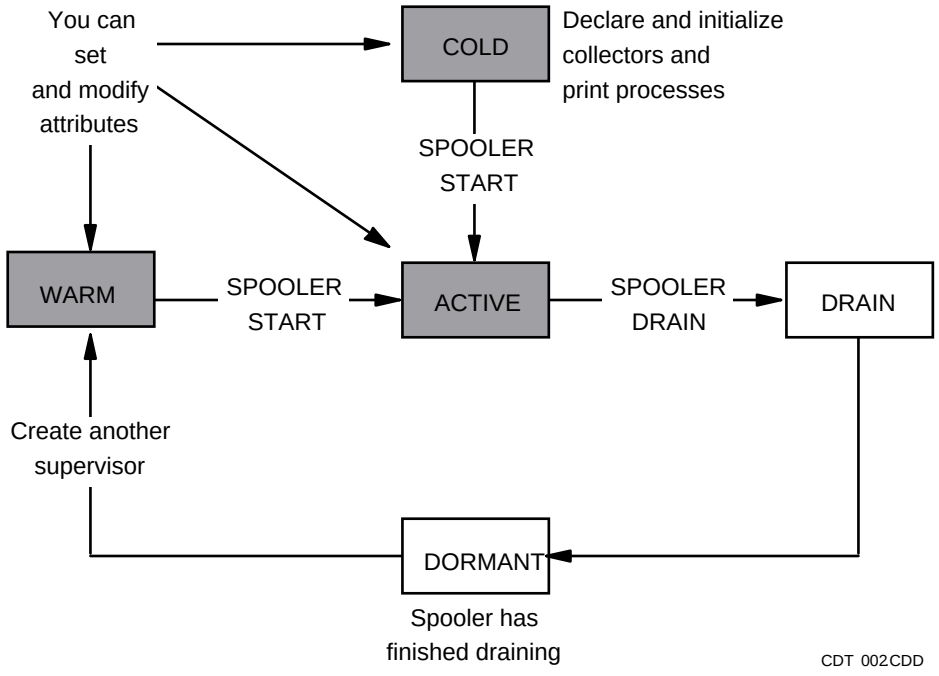
- Spoolcom communicates with the spooler supervisor designated by the Spoolcom OPEN command. The default supervisor name is `$SPLS`.
- A spooler is always characterized by its control file. When a spooler is stopped and then restarted, the control file name (not the process name of the spooler supervisor) determines which spooler gets started.
- Ensure that all collector and print process names are unique across all spooler subsystems created on a given system. The presence of duplicate collector or print process names is not immediately apparent when multiple spoolers are starting. As long as the supervisor names are unique, all of the spooler subsystems will start without error. However, nonunique collector or print process

names will cause the affected components to fail. When starting multiple spooler subsystems, monitor the spooler error log file (usually $0, the event log file for operator messages) for reports of process creation errors that indicate that the specified collector or print process could not be started because the name given was already in use.

Spooler States

During its life, the spooler cycles through the five states shown in Figure 1-2.

Figure 1-2. Spooler Life Cycle



The meanings of the spooler states are as follows:

COLD — The first step in creating the spooler is to run the spooler supervisor. The spooler is in the COLD state as soon as you start the supervisor. At this time, use Spoolcom to declare and initialize the collectors and print processes.

ACTIVE — After declaring and initializing the collectors and print processes, issue the Spoolcom command SPOOLER START, which puts the spooler into the ACTIVE state. In this state, the spooler is fully operational and ready to accept output from application processes. You can then use Spoolcom to add, delete, or modify collectors and print processes.

DRAIN — It is sometimes necessary to halt the spooler. You should never issue a TACL STOP command for any spooler process, because the spooler recovery from STOP can be time-consuming. Instead, bring the spooler to an orderly halt by issuing the Spoolcom command SPOOLER DRAIN. This command puts the spooler into the DRAIN state. When the spooler is in this state, the following events take place:

- Each collector stops accepting new jobs, rejects new opens with file-system error 66 (device downed by operator), finishes accepting and storing any jobs that are currently open, and stops.
- Each print process finishes the jobs currently printing and stops.
- The supervisor updates its control files and stops.
- Any attempt to print to a stopped spooler is rejected with file-system error 14 (device does not exist).

DORMANT — Once drained, a spooler is in the DORMANT state. In this state, it consists solely of a set of disk files, including: program files containing object code, data files containing spooled jobs, and control files containing the names and attributes of the components and jobs known to the spooler.

You cannot use Spoolcom to obtain information regarding a DORMANT spooler, because in the DORMANT state there is no supervisor for Spoolcom to communicate with.

WARM — When you start another supervisor (from TACL using the RUN command), the spooler enters the WARM state. This state is the same as the COLD state in that the supervisor is the only process running and that collectors and print processes can be added, deleted, or modified. The only difference is that the WARM state indicates the restarting of a spooler that was formerly ACTIVE, while the COLD state indicates the starting of a new spooler.

To bring the spooler from the WARM state to the ACTIVE state, enter the command SPOOLER START.

Spooling From an Application Program

Spooling from an application program can be performed with the Guardian file-system procedures or with the spooler interface procedures described in Section 2, Using the Spooler Interface Procedures. These procedures give you complete control of the contents and attributes of your job. You can find the procedure syntax for the spooler and print procedures in Section 4, Spooler Procedure Calls. See the *Guardian Procedure Calls Reference Manual* for the operating system procedure calls.

The spooler includes one or more collectors, described in Collectors on page 1-10. Applications can direct their output to a particular collector by treating the collector as a file (that is, an application can open a file to any collector and begin writing its output by using the Guardian file-system WRITE[X] procedure). In this case, the collector assigns default attributes to the job as shown in Table 1-2.

Table 1-2. Default Attributes for Jobs

Attribute	Default Value
Priority	4
Form name	blanks
Copies	1
Hold flag	Off
Hold after flag	Off
Location	#DEFAULT
Report name	Owner's group and username

As an alternative method of spooling a job from a program, use the spooler interface procedures, described in Section 2, Using the Spooler Interface Procedures. These procedures provide an application process with the following abilities:

- To specify job attributes (see SPOOLSTART Procedure on page 4-64)
- To compress and block data and write it to the collection process (see SPOOLWRITE Procedure on page 4-69)
- To send CONTROL, CONTROLBUF, and SETMODE instructions to the collector (see SPOOLCONTROL Procedure on page 4-24, SPOOLCONTROLBUF Procedure on page 4-26, and SPOOLSETMODE Procedure on page 4-62)

Data Compression

All spooled data is compressed before being stored on disk. Data spooled with the Guardian file-system procedures is compressed by the collector, while data spooled with SPOOLWRITE is compressed before being written to the collector.

Nulls, zeros, and spaces are compressed on word by word basis. Each sequence of consecutive words containing %0 (two nulls), %020040 (two spaces), or %030060 (two

zeros) is replaced by one word describing the character and the number of words being compressed. For example, an 80-character line containing all blanks would be compressed into a single word.

Job States While Spooling From a Program

The term **job** refers to the data written by an application process to a collector. The collector creates a job when an application opens a file to it and either issues a Guardian file-system WRITE[X], CONTROL, CONTROLBUF, or SETMODE procedure call to the collector, or calls SPOOLSTART.

At this point, the job is in the open state, meaning that the application is sending data to the spooler.

Spooling is completed either when the application closes the file to the collector or when an application using the interface procedures calls SPOOLEND.

By using the CONTROL, CONTROLBUF, and SETMODE procedures or the SPOOLCONTROL, SPOOLCONTROLBUF, and SPOOLSETMODE procedures, the spooler gives applications complete control of the devices on which their jobs are printing. For this reason, applications should issue a top-of-form control, using CONTROL or SPOOLCONTROL, to guarantee the state of the print device before and after the job has been printed. Otherwise, data from two different jobs could appear on the same printed page. All software supported by HP performs this step when spooling a job.

If an application using the interface procedures closes the file to the collector without first calling SPOOLEND, the job is completed but the collector assumes that an abnormal termination has occurred and places the job in the hold state.

Collectors

The spooler includes one or more collectors, each of which is a continuously running copy of the program in `$SYSTEM.SYSTEM.CSPOOL`. Applications can direct output from an application program to a collector by treating the collector as their OUT file; that is, an application can open a file to any collector and begin writing its output using the Guardian file-system WRITE[X] procedure. Applications can also use the spooler interface procedures to spool their output (described in Section 2, Using the Spooler Interface Procedures).

While the spooler is in the cold or warm state, you can declare and initialize the spooler collectors by using the Spoolcom COLLECT command, specifying attributes such as execution priority and program file name.

Collector Attributes

Table 1-3 lists the default attributes of collectors and the Spoolcom COLLECT subcommands used to specify them.

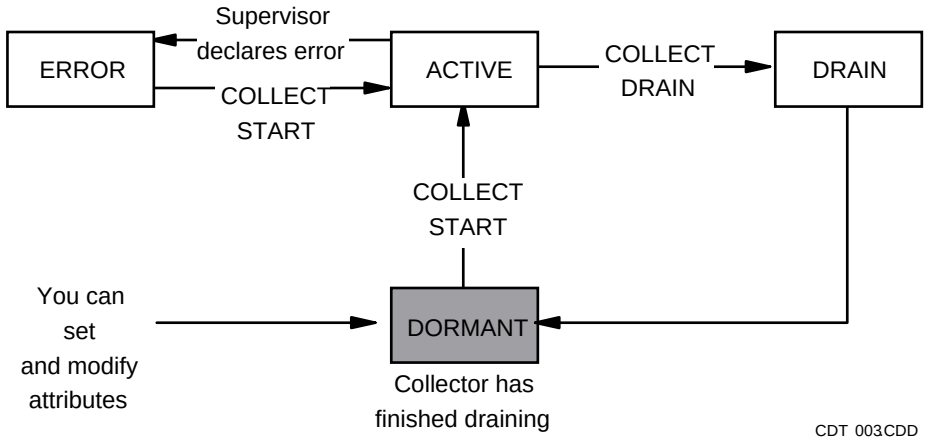
Table 1-3. Collector Attributes

Collector Attribute	Spoolcom COLLECT Subcommand	Default Value
Program file	FILE	$SYSTEM.SYSTEM.CSPOOL
Primary CPU	CPU	Processor of supervisor
Backup CPU	BACKUP	No backup processor
Execution priority	PRI	145
Data file	DATA	None
Unit size	UNIT	4
Page size	PAGESIZE	60

Collector States

The collector is always in one of four states, as shown in Figure 1-3.

Figure 1-3. Collector States



The meanings of the collector states are as follows:

DORMANT — A collector in the dormant state cannot accept new jobs for spooling, and no jobs are currently being spooled. You can set and modify collector attributes while it is in the dormant state. The Spoolcom command COLLECT START puts a dormant collector in the active state.

ACTIVE — A collector in the active state can accept new jobs for spooling. You cannot change a collector's attributes while it is in the active state. The Spoolcom command COLLECT DRAIN puts an active collector in the drain state.

DRAIN — A collector in the drain state will not accept new jobs for spooling, but jobs currently being spooled will continue until completion. When all open jobs have been completed, the collector enters the dormant state. You cannot change a collector's attributes while it is in the drain state.

ERROR — A collector in the error state cannot function. The Spoolcom COLLECT STATUS command tells you whether the collector is in an error state. The octal error number is either %1000 plus a Guardian file-system error number or %100000 plus a NEWPROCESS error number.

You can declare, initialize, and delete collectors whenever the spooler is in either the cold or the warm state or whenever the collector is in the dormant state. At other times, these operations are rejected.

When you issue a SPOOLER START or COLLECT START command, any collector attributes that have not been specified take their default values. You must specify the data file, however, because it has no default attributes. If you issue the SPOOLER START command and the collector has no data file specified, the collector will abnormally terminate.

Unit Size

The Spoolcom COLLECT UNIT command is used to specify a unit size for the collector. The unit size specifies the number of 512-word blocks the collector allocates from its data file each time it needs more space for a job. The maximum number of units is limited only by the size of the file.

The larger the unit size, the less often the collector must allocate a new unit. For this reason, you should specify a relatively large unit size if you expect that most spooled jobs will be large. You should also use a large extent size if you are using large block sizes.

A smaller unit size provides more efficient use of disk space, because once the controller reserves space for a job, that space cannot be used by any other job. If the unit size is 10 and a spooled job requires only 1 block, the other 9 blocks are wasted. It is best to use a unit size that is a whole multiple of the buffer size.

The collector file size can be any size allowed by the Guardian file system. The file's buffer size attribute is used to set the collector's internal buffer size. The Collector unit size attribute must be a whole multiple of the buffer size attribute, or the buffer size must be a whole multiple of the unit size. You must set the buffer size to at least 2K bytes. You can also set the buffer size to a multiple of 2K bytes. For best performance, HP recommends that you set the buffer size to 4K bytes and the unit size to 4K bytes.

You should set the unit size of a collector once and not change it. If a different unit size is required, delete the old collector and start a new one.

It can be useful to have two collectors, one with a large unit size and the other with a small unit size, to be used for large and small jobs. For example, compiler listings are spooled to the collector with a large unit size, while short jobs (one-page or two-page memos, for example) are spooled to the collector with the small unit size.

Print Processes

Each output device known to the spooler is assigned a print process, which has the task of getting spooled data from disk and writing it to the device. You declare print processes when initializing the spooler; they are run by the supervisor as needed.

Some print processes are copies of the `$SYSTEM.SYSTEM.FASTP` program supplied by HP. However, you can write others. See Section 4, Spooler Procedure Calls, for full syntax and considerations for the print procedures. See Section 3, Using the Spooler Print Procedures, Print Processes, and Perusal Processes, for descriptions of how you can declare and initialize print processes.

Print Process Attributes

Table 1-4 lists the default attributes of a print process and the Spoolcom PRINT subcommands used to specify them.

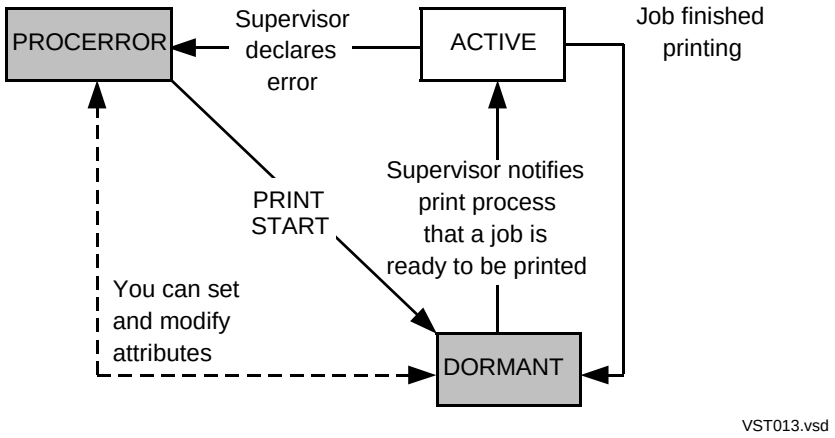
Table 1-4. Spoolcom PRINT Subcommands and Print Process Default Values

Print Process Attribute	Spoolcom PRINT Subcommand	Default Value
Program file	FILE	Independent process
Primary processor	CPU	Processor of supervisor
Backup processor	BACKUP	No backup processor
Execution priority	PRI	145
Print process parameter	PARM	0
Debug mode	DEBUG	OFF

Print Process States

A print process, once declared, is always in one of three states: active, dormant, or procerror, as shown in Figure 1-4.

Figure 1-4. Print Process States



The print process states are as follows:

ACTIVE — The print process is running. A print process enters the ACTIVE state for one of three reasons:

- The print process is printing a job.
- A device controlled by the print process has been declared exclusive and must therefore be kept open even when a job is not printing.
- The print process is independent and is always running, as described in Independent Print Processes on page 1-15.

DORMANT — The print process is not running. The print process enters the DORMANT state whenever it has no job to print, controls only shared devices, and is not an independent print process.

You can modify print process attributes when the print process is in the DORMANT state.

PROCERROR The supervisor has determined that the print process is not responding correctly. When the supervisor places a print process into this state, it writes a message to the error log file. (These messages are typically sent to the console, although their destination is set at system startup time and can be changed with the Spoolcom SPOOLER ERRLOG command. The error log messages are described in the *Operator Messages Manual*.) You should debug the print process to determine the cause of failure.

You can modify print process attributes when the process is in the PROCERROR state. To remove a print process from the PROCERROR state, use the PRINT START command.

Independent Print Processes

A print process can be started independently of the supervisor. You do not need to start the print process before starting the supervisor.

To define an independent print process, do not specify the FILE subcommand in the Spoolcom PRINT command. The spooler supervisor flags a null print process file name as an independent print process.

When started, the independent print process receives two startup messages. The first startup message is the normal message sent by TACL. The second startup message is sent by the supervisor and contains the process name of the spooler supervisor in the OUTFILE field. This supervisor message is in the same format as the TACL message. See the *Guardian Procedure Errors and Messages Manual* for an explanation of this format. An independent print process can be started, stopped, and restarted without the knowledge of the supervisor, as long as no jobs are active for that print process. If no jobs are active, the second startup message is not sent.

If a job targeted for an independent print process arrives before the print process is started, the process is placed in the error state with error %5016 (device does not exist). The device is placed in the PROCERROR state, and an error message is written to the spooler log file. To clear the error state and begin the print job, issue this Spoolcom command:

```
PRINT print-process-name, START
```

Once an independent print process is started and the startup messages are read, the independent print process runs in the same manner as a standard print process. The spooler supervisor does not stop an independent print process. Independent print processes remain ACTIVE until you remove them from the ACTIVE state.

Stopping an independent print process is a two-step procedure. First, issue this Spoolcom command:

```
PRINT print-process-name, DELETE
```

The Spoolcom PRINT DELETE command removes the independent print process from the spooler subsystem but does not stop the physical process. Second, stop the physical process by issuing the TACL STOP command. When you use both of these commands to stop an independent print process, no error message is written to the log file.

Stopping an independent print process without using the Spoolcom PRINT DELETE command or abending an independent print process results in error %5311 (current path to device is down). This error occurs regardless of the state of the independent print process prior to the stop or abend. An error message is written to the spooler log file.

To restart an abended print process or one stopped without the Spoolcom PRINT DELETE command, start the physical print process first. Then restart the independent print process using the following Spoolcom command:

```
PRINT print-process-name, START
```

The Spoolcom PRINT START command clears the PRINT and DEV error states and begins the print job.

When restarting an independent print process that was stopped with both the Spoolcom PRINT DELETE and TACL STOP commands, the order of the procedures does not matter. You can run the physical process before or after reconfiguring and starting the print process using Spoolcom.

Devices

Most jobs eventually print on a spooler device, which can be a physical device (such as a line printer, terminal, or tape drive), a process, or a virtual device (explained in Declaring and Initializing Devices on page 1-20).

Device Attributes

Table 1-5 lists the default attributes of spooler devices and the Spoolcom DEV subcommands used to specify them. Refer to the *Spooler Utilities Reference Manual* for a complete description of the Spoolcom DEV subcommands.

Table 1-5. Device Attributes (page 1 of 2)

Device Attribute	Spoolcom DEV Subcommand	Default Value	Comment
Form name	FORM	All blanks	Guarantees that only certain types of jobs print on the device. Most commonly used when device is loaded with special paper or ribbon.
Speed	SPEED	100 (lines per minute)	Value used by supervisor in calculating how long jobs will take to print on the device. Used only for job selection; has no effect on device printing speed.
Print process name	PROCESS	(None)	Each device must have a print process associated with it if it is going to print jobs. The print process retrieves jobs from disk and writes the spooled data to the device.
Ownership mode	EXCLUSIVE	EXCLUSIVE OFF	
Truncation mode	TRUNC	TRUNC OFF	Tells a standard print process whether to truncate or to wrap around long lines.
Device width	WIDTH	Value from DEVICEINFO	If user does not supply this, standard print process calls Guardian file-system DEVICEINFO procedure to obtain record size of the device.
Device parameter	PARM	0	Affects the FASTP print process as described in the Spoolcom DEV command.
Selection algorithm	FIFO	FIFO OFF	FIFO orders the jobs on a first-in, first-out basis. FIFO OFF (the default) orders the printing queue based on the length of the incoming job and how long the other jobs have been in the queue.
Retry interval	RETRY	5 (seconds)	
Number of retries	TIMEOUT	360 (seconds)	

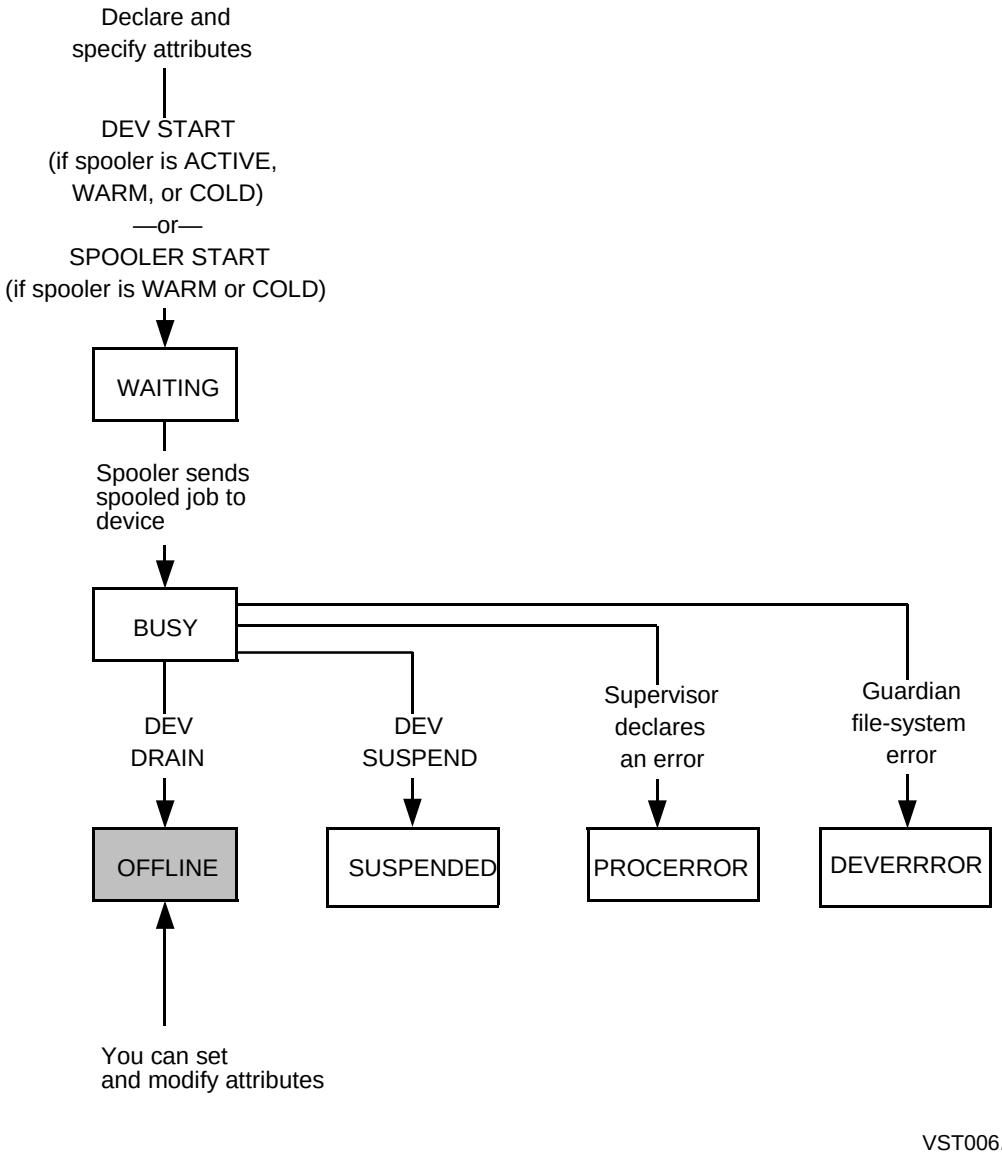
Table 1-5. Device Attributes (page 2 of 2)

Device Attribute	Spoolcom DEV Subcommand	Default Value	Comment
Header message	HEADER	HEADER	Printed at beginning (and optionally at end) of every job. Its type is determined by the print process. The standard print process header contains the job report name, location, and job number.
Restart	RESTART	OFF	Controls automatic restart of devices that have encountered non-retryable errors or have timed out on retryable errors.
Multibyte character set	CHARMAP	NONE	Specifies whether or not character set translation is required.

Device States

At any particular time, each device in the spooler subsystem is in one of several states, as shown in Figure 1-5.

Figure 1-5. Device States



The state of a device describes what the device is doing and determines which Spoolcom commands are valid. The meanings of the device states are as follows:

BUSY	The device is currently printing a job.
WAITING	The device is ready to print a job, but no job is available to print on the device, because either there are no jobs in the device queue or none of the jobs in the queue have a form name that matches the form name of the device.
OFFLINE	The device is not available for printing jobs. Device attributes can be changed only when the device is in the offline state.
SUSPENDED	A job was printing on the device, but printing has stopped as a result of a Spoolcom DEV SUSPEND command.
DEVERROR	A Guardian file-system error occurred on the device while a job was printing.
PROCERROR	The supervisor has determined that the device print process is not working correctly and writes a message to the error log file. (These messages are typically sent to the console, although their destination is set at system startup time and can be changed with the Spoolcom SPOOLER ERRLOG command. The error log messages are described in the *Operator Messages Manual*.) The device is therefore unusable until the print process is restarted.

Declaring and Initializing Devices

Devices are declared and initialized using Spoolcom commands, and the spooler can be active, warm, or cold.

A device is declared with the DEV command. As soon as you issue the DEV command, the device is considered to be part of the spooler subsystem, with all default attributes, but it is in the offline state. You can then change the default attributes or leave them as they are. After specifying all attributes, bring the device into the waiting state (in which it is ready to print jobs) with the DEV START command.

If you initialize the device with the spooler in a warm or cold state, you can bring the device into the waiting state with the SPOOLER START command.

You can change the attributes of a device only after putting it in the offline state with the DEV DRAIN command. The DRAIN subcommand allows the job currently printing (if any) to finish, then puts the device offline.

The usual way to cause a job to leave the spooler subsystem is to send it to an output device (the only other way is to delete the job from the system). In most cases, the output device is a printer; however, it can also be a terminal, tape drive, disk, or virtual device, as explained in Declaring and Initializing Devices on page 1-20.

Although the supervisor makes the decision to print a job on a particular device, the supervisor does not directly interface to a device. Instead, each device has an

associated print process that performs the task of retrieving and printing jobs on that device. Refer to Section 3, Using the Spooler Print Procedures, Print Processes, and Perusal Processes, for more information on using print processes.

Virtual Devices

When a device becomes available, the supervisor tells the print process associated with the device to retrieve and print the next job. It then becomes the responsibility of the print process to access the spooled data and issue a succession of Guardian file-system WRITE[X] procedure calls to the correct device.

A print process can perform a function with the spooled data other than writing it to a device. If, for example, you want to perform a statistical analysis of numerical data that has been spooled from an application, you can write a print process that retrieves the spooled data and performs the desired analysis. To do this you must declare a fictitious (virtual) device by using the Spoolcom DEV command. Then assign the print process to control that device.

Device Ownership

Physical devices that are declared as part of the spooler subsystem can be accessed by other processes when they are not being used by the spooler. A device that can be accessed by other processes is called a shared device.

The print process opens a shared device when the spooler wants to print a job on the device. When the job is finished printing, the print process closes the device to allow access to the device by other processes. The print process competes with all other processes for access to a shared device.

An exclusive device, on the other hand, is kept open all the time, preventing any other process from gaining access.

If a print process controls only shared devices, it runs only when one of the devices is actually being used by the spooler. If a print process controls an exclusive device, it must run all the time to keep the device open.

Device Queues

Associated with each device is a device queue, which is simply a list of jobs waiting for that device in the order that they are scheduled to print. The device queue for a particular device contains jobs routed to all locations connected to the device. A job is added to a device queue whenever the job becomes ready to print.

Device queues are not queues in the usual sense of the term, because jobs are not always added to device queues on a first-in, first-out (FIFO) basis. The default queueing method orders the print jobs in the device queue based on the length of the incoming job and how long the other jobs have been in the queue.

Routing Structure

The spooler includes a flexible routing structure whose function is to direct jobs to print devices. The routing structure consists of a set of locations, logically organized into groups, and connections between locations and print devices.

Locations

A location is the logical destination of a job (as opposed to the physical destination, which is a print device). At the time each job enters the spooler subsystem, it is associated with a location. If a device is associated with that location, then the job prints on that device.

Location names have two parts: a group name and a destination name. The group name is always preceded by the # symbol. For example, `#RED.LP` is a valid location name.

Table 1-6 lists the default attributes of locations and the Spoolcom LOC subcommands used to change them.

Table 1-6. Location Attributes

Location Attribute	Spoolcom LOC Subcommand	Default Value
Device	DEV	None
Location	FONT	None
Broadcast mode	BROADCAST	BROADCAST OFF

For a job to print on a device, the job location must be connected to that device.

Connecting Devices and Locations

The Spoolcom LOC DEV command establishes connections between devices and locations. You can connect a specific location to a device or all locations with a specific destination name to a device.

To connect a specific location to a device, you make a single connection. You make additional connections to that device with subsequent LOC DEV commands. To connect all locations with a specific destination name to a device, you make several connections at the same time. However, the LOC DEV command applies only to existing locations. For example, if the existing routing structure includes two locations, `#RED.LP` and `#BLUE.LP`, the command LOC LP, DEV $LP would associate the device $LP to both locations. If you create a third location, `#GREEN.LP`, use another LOC DEV command to connect $LP.

It is not possible to connect more than one device to the same location. However, it is possible to connect multiple devices to a single group. In this case, the group will contain at least one destination name for each device connected to it. Using Spoolcom

LOC, you can set the broadcast mode attribute for the group to print a job at every device associated with the group (BROADCAST ON) or only at the device that would finish the job first (BROADCAST OFF).

Devices can have any number of locations connected to them, but each location can have only one device connected to it. A group can have several devices associated with it. Each device must be connected to a different destination contained in the group. Make these connections in any order that best suits the particular needs of your system.

Jobs

The term **job** refers to the data written by an application process to a collector.

Job Attributes

A set of attributes is specified for each job when the job is created (that is, when it enters the open state). These attributes can be changed with the Spoolcom JOB command. Table 1-7 lists the default attributes of jobs and the Spoolcom JOB subcommands used to specify them.

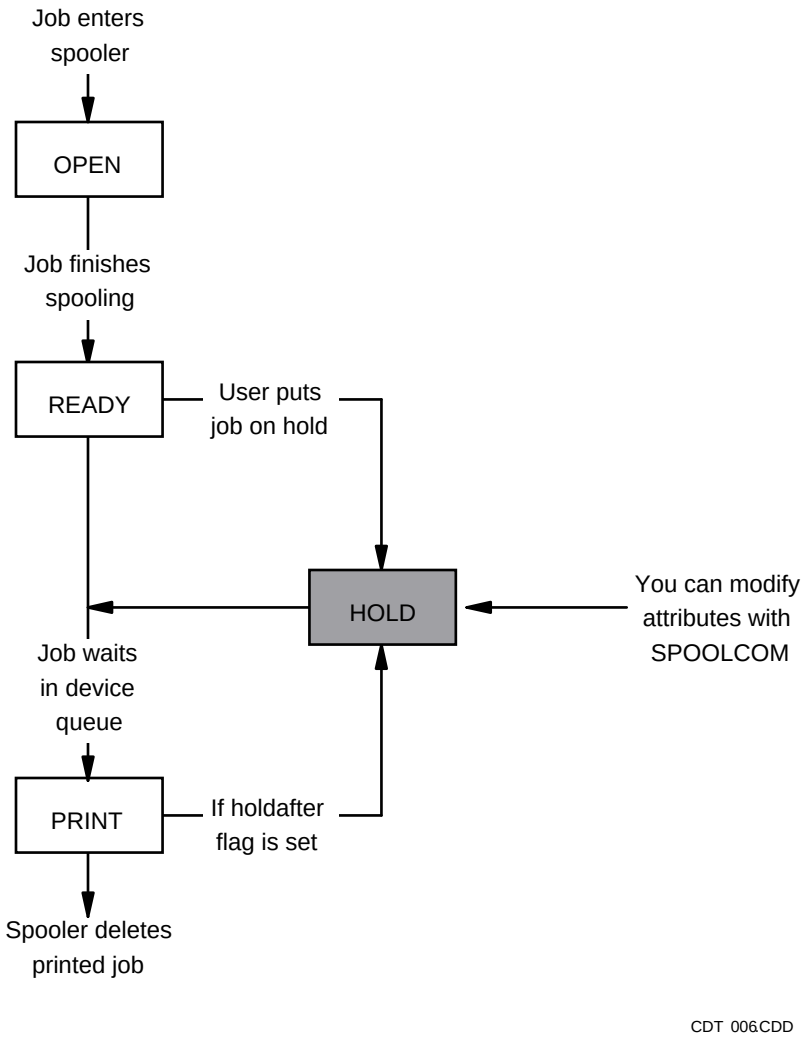
Table 1-7. Job Attributes

Job Attribute	Spoolcom JOB Subcommand	Default Value
Form name	FORM	Blanks
Report name	REPORT	User ID
Number of copies	COPIES	1
Selection priority	SELPRI	4
Location	LOC	#DEFAULT
Holdafter flag	HOLDAFTER	OFF
Hold flag	HOLD	OFF
Owner name	OWNER	User ID
Batch Name	BATCHNAME	Blanks
Maximum # of lines	MAXPRINTLINES	No limit
Maximum # of pages	MAXPRINTPAGES	No limit
Page size	PAGESIZE	60

Job States

The status of a job is described by its state. At any time, each job in the spooler subsystem is in one of several states, shown in Figure 1-6.

Figure 1-6. Job States



Different Spoolcom JOB subcommands are valid in different job states. The state of a job determines which Spoolcom commands are valid. Refer to the Spoolcom JOB command in the *Spooler Utilities Reference Manual* for a description of the JOB subcommands to use in which job states.

The meanings of the job states are as follows:

OPEN The job has been added to the spooler. It remains in this state until it has finished spooling.

READY The job is ready to print, but it has not yet begun to print because another job is ahead of it in the device queue or because its location is not connected to a device.

HOLD You can place a job on HOLD in order to prevent it from printing or to change its attributes. You can put a job on HOLD at any time except when it is in the OPEN state.

If you put on HOLD a job that has multiple occurrences, then all occurrences of the job lose their place in their respective device queues. A currently printing job is also placed on HOLD.

If you put a job on HOLD and then immediately take it off HOLD, you remove the job from the device queue and then add it back to the queue. This causes the job to lose its place in the device queue.

PRINT The job is being printed.

If you set the HOLDAFTER flag on a job, the spooler places the job on HOLD after printing is complete, rather than deleting it. When you later remove the HOLD, the job will print another time and again enter the HOLD state until you either delete it or remove the HOLD.

The description of the Spoolcom JOB command in the *Spooler Utilities Reference Manual* lists all the ways a user can alter a job. To change job attributes, the job must be in the hold state. The JOB subcommands HOLD, START, and DELETE allow a job to be put on hold, taken off hold, or deleted from the spooler subsystem.

Job Numbers

At the time it is created, each job is assigned a number in the range 1 through the maximum number of jobs allowed in the spooler subsystem. That maximum is specified when the spooler is first initialized and cannot exceed 65535. Job numbers are assigned consecutively. If the last number assigned is the maximum, the next job number assigned will be 1. If the number that would be assigned to a job is already in use, the next available number is assigned.

A job numbered 0 indicates a corrupted control file. Recovery requires at least a warmstart and rebuild of the spooler; if this is not successful, a coldstart is required.

Occurrences of Jobs

A job routed to a group enters the device queue of each device connected to the locations in that group. Each entry in a device queue is a separate occurrence of the job, indicating that the job has been routed to multiple locations.

Controlling Jobs

The Spoolcom JOB subcommands HOLD, START, and DELETE allow a job to be put on hold, taken off hold, or deleted from the spooler subsystem. To change job attributes, the job must be in the hold state. You can place a job on hold at any time except when it is open.

Placing a job on hold takes the job off any device queue that it is on. If a job has multiple occurrences, then all occurrences of the job lose their place in their respective device queues as a result of a HOLD command. Any occurrence currently printing is also placed on hold.

When a job enters the ready state and a device is connected to the job's location, it is added to a device queue. Putting a job on hold and then immediately taking it off hold causes the job to be removed from the device queue and then added to the queue. This causes the job to lose its place in the device queue.

The Spooler and Batch Jobs

In the spooler, a **batch job** is a group of associated spooler jobs. Typically, these are jobs from a NetBatch application. Each spooler job has certain attributes, called key attributes, which allow jobs to be linked together in a continuous listing when printed. The values of these key attributes when the spooler collector is opened dictate whether or not a job enters the queue as part of a batch job. Any job that does not meet the key attribute criteria remains a normal job.

The first job that has all the necessary key attributes for a batch job is assigned a batch number by the spooler. Any subsequent job whose attributes match those of the first job in the batch job is linked to that same batch job and assigned the same batch number.

The meanings of the key attributes that determine whether a spooler job is part of a batch job are as follows:

JOBID — This identifier is the same for all jobs in a batch job. A job that is not part of a batch job does not have this identifier. (JOBID can be set through a TACL RUN command or a DEFINE for the application generating the batch job.)

FORM — Each job in a batch job must use the same print device and paper.

OWNER — Each job in a batch job must be owned by the same user.

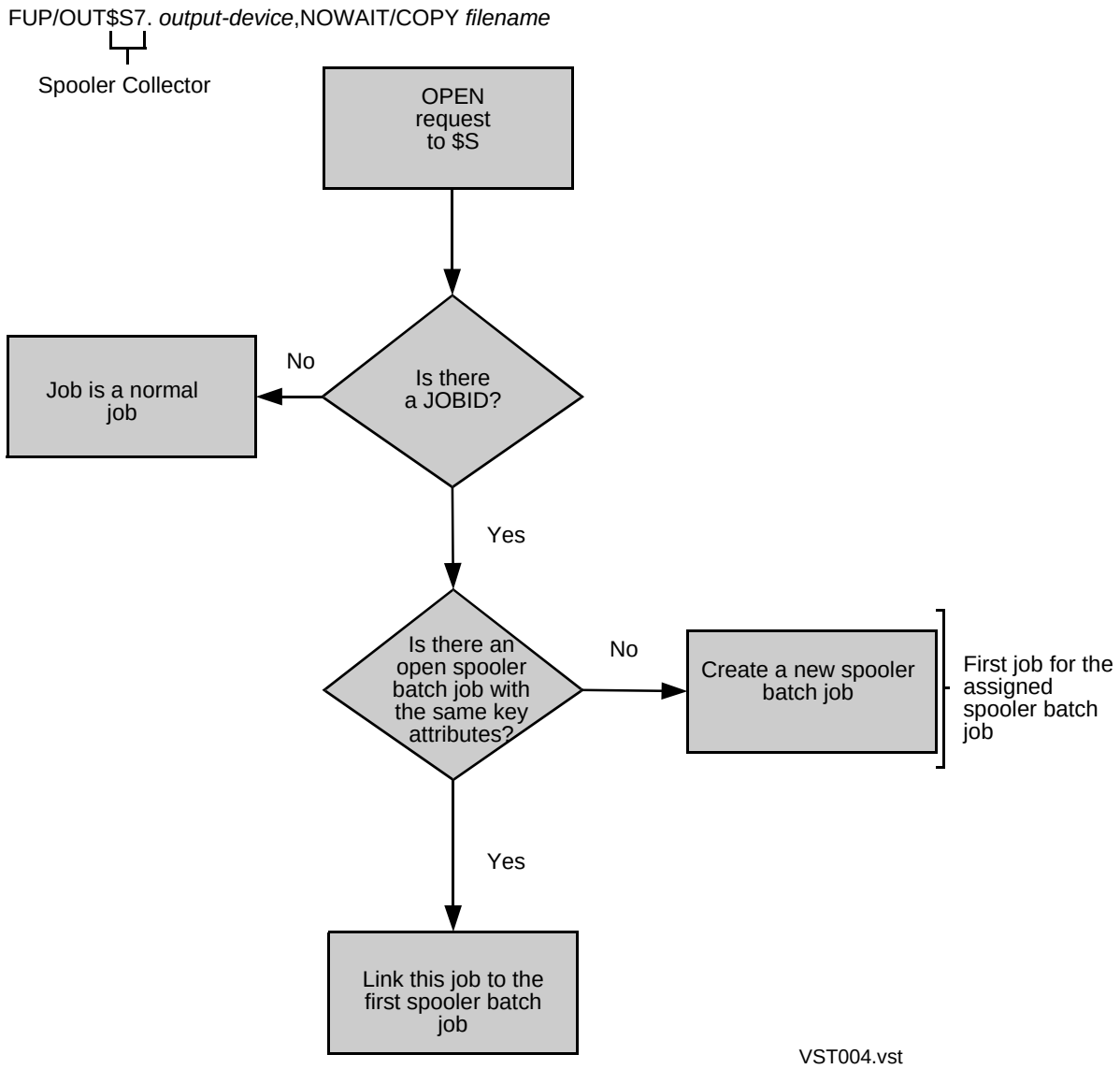
LOC — The device associated with each logical location within the spooler must be the same for all members of a batch job. A location name has two parts: #*group.destination*, where *destination* represents the device. You can specify the destination in the location name used by the application or configured for the location through a Spoolcom DEV command. For more information see the *Spooler Utilities Reference Manual*.

The method that the spooler uses to determine if a job becomes a normal job, the first job in a batch job, or a job to be linked to an existing batch job is shown in Figure 1-7.

If a job is part of a batch job, you cannot alter the key attributes unless you use the Peruse UNLINK command or the UNLINK option of the Spoolcom BATCH command to unlink the job from the batch job.

Other job attributes have no effect on whether the job is linked to a batch job. Some attributes might have no meaning for a job that is part of a batch job. For example, if every job within a batch job contains a different report name, only the report name associated with the first job in the batch job is printed on the header pages for that batch job. Attributes that have meaning only to the first job in a batch job are the report name, batch name, and selection priority.

Figure 1-7. How the Spooler Determines Which Jobs Are to Be Batched



2

Using the Spooler Interface Procedures

Application programs can spool jobs (that is, write the data for a spooler job) using a group of procedures that act as an interface between an application program and a collector process in a spooler subsystem. These procedures are usually referred to as spooler interface procedures. This section describes how to use these procedures in an application program.

Table 2-1 contains a summary of the spooler interface procedures. Section 4, Spooler Procedure Calls, contains complete descriptions of the procedures and their parameters. Refer to Appendix C, Spooler-Related Errors, for those error codes that are relevant to spooler interface procedures, along with file-system errors that have special significance for the spooler.

Table 2-1. Summary of Spooler Interface Procedures

Procedure	Function
SPOOLBATCHNAME	Returns the name of the spooler batch job currently being spooled to the collector.
SPOOLCONTROL	Replaces the Guardian file-system CONTROL procedure when spooling at level 3.
SPOOLCONTROLBUF	Replaces the Guardian file-system CONTROLBUF procedure when spooling at level 3.
SPOOLEND	Writes any remaining blocked data to the spooler and signals end of job; can be used to modify the job attributes.
SPOOLERCOMMAND	Issues a SPOOLCOM command to the supervisor.
SPOOLEREQUEST	Obtains a Startup message from the supervisor suitable for reading a job.
SPOOLEREQUEST2	Obtains a startup message from the supervisor suitable for reading a job. Includes batch enhancements to SPOOLEREQUEST.
SPOOLERSTATUS	Obtains status of spooler components.
SPOOLERSTATUS2	Obtains status of spooler components. Includes batch enhancements to SPOOLERSTATUS.
SPOOLJOBNUM	Returns the job number of the job currently being spooled to the collector.
SPOOLSETMODE	Replaces the Guardian file-system SETMODE procedure when spooling at level 3.
SPOOLSTART	Specifies job attributes and optionally initializes a level-3 buffer.
SPOOLWRITE	Compresses, blocks, and sends data to the spooler.

External Declarations for Spooler Interface Procedures

To use spooler interface procedures in a TAL program, you must declare them to be external to your program. The external declarations for the interface procedures are located in the file `$SYSTEM.SYSTEM.EXTDECS0`. They can be sourced into your program with the following compiler command:

```
?SOURCE $SYSTEM.SYSTEM.EXTDECS0 (SPOOLSTART, SPOOLWRITE, ... )
```

See the *TAL Reference Manual* for a full explanation of the EXTERNAL procedure declaration and the ?SOURCE compiler directive.

Levels of Spooling From an Application Program

The three ways for an application program to spool a job to a collector are as follows:

- Level-1 spooling
  - Job attributes default to the default attribute values.
  - The application sends data for a job to the collector by using the WRITE[X], CONTROL, and SETMODE Guardian file-system procedures.
- Level-2 spooling
  - Job attributes can be specified using the SPOOLSTART procedure.
  - The application sends data for a job to the collector by using the WRITE[X], CONTROL, and SETMODE Guardian file-system procedures.
- Level-3 spooling
  - Job attributes can be specified using the SPOOLSTART procedure.
  - The application sends data for a job to the collector by using the SPOOLWRITE, SPOOLCONTROL, SPOOLCONTROLBUF, and SPOOLSETMODE procedures.

    Instead of requiring the transfer of data from the procedure buffer to the collector each time a procedure is called, the spooler interface procedures allow you to collect (but not transfer) the data in a special buffer. Each time the SPOOLWRITE procedure is called, it checks to see whether the write operation can cause this buffer to overflow. If it can, the procedure initiates the transfer of the buffer contents and then begins filling the buffer again.

    Two types of buffer areas for the data can be used in the spooler interface procedures:

- A buffer allocated above the data stack that is limited to 500 bytes. This buffer is indicated by the *level-3-buff* parameter.
- A buffer allocated in an extended data segment. Extended data segments can be much larger than 512 bytes This buffer is indicated by the *extended-level-3-buff* parameter. Extended data segments are discussed in the *Guardian Programmer's Guide*.

  Use either the *level-3-buff* or *extended-level-3-buff,* but not both. The buffer is specified in the SPOOLSTART procedure call and must be used in all subsequent spooler interface procedure calls.

A program can spool data for several jobs concurrently, but a separate file must be open to a collector for each job. A single application can spool up to 256 jobs to each collector.

Note. The collector cannot accept more than 1024 jobs simultaneously. Thus, if there is more than one process spooling to the same collector at the same time, the actual limit for one particular process might be lower than 1024.

Jobs spooled concurrently can be spooled at different levels. For example, an application might open three files to the collector and send it data for three separate jobs, spooling one job at each level. However, one job cannot be spooled at two different levels at the same time (in other words, you cannot use both the Guardian file-system WRITE[X] procedure and the SPOOLWRITE procedure on the same job).

Opening a File to a Collector

Before a program can send data to the spooler, it must have a file open to a collector. This is accomplished by using the Guardian file-system OPEN procedure or the FILE_OPEN_ procedure. The program must open the collector with shared access.

If an application is spooling at level 1, the file can be opened with either waited or nowait input/output (I/O). However, if level-2 or level-3 spooling will be performed, you must open the file to the collector with waited I/O.

To open a collector, the program must specify the collector and location for the job as follows:

- A call to the OPEN procedure must provide the collector name and location for the job in 12-word internal format. The format is
  - Words[0:3] contain the $*collector-name*, blank filled.
  - Words[4:7] contain the *#group-name,* blank filled.
  - Words[8:11] contain the *destination-name*, blank filled.
- The FILE_OPEN_ procedure provides extended features not available from the OPEN procedure. On a call to FILE_OPEN_ , you must provide a variable-length

string for the collector and location (as the *filename* input parameter) and the length of the string.

Note. If the filename input parameter is not fully qualified, FILE_OPEN_ uses the current settings, including the system name, in the =_DEFAULTS DEFINE, for unspecified parts including the system.

The parameters required in the FILE_OPEN_ procedure call are

```
CALL FILE_OPEN_ (filename:length,filenum) ;
```

For example, if the *filename* parameter specifies the collector name and location $S.#LP2, the *length* parameter must be 7. The system returns a value in *filenum*.

All considerations of job routing described under Routing Structure on page 1-22 apply to routes specified implicitly when using OPEN or FILE_OPEN_ to establish communication with a collector.

$collector-name is required in the call to OPEN and *$collector-name* and *length* are required in the call to FILE_OPEN_. If *#group-name* or *destination-name* is filled with blanks in a call to OPEN or is not present in a call to FILE_OPEN, then the job will be assigned a location according to the rules for default routing.

In level-2 and level-3 spooling, you can change the job location in the call to SPOOLSTART.

Summary of Spooling From an Application Program

The following points summarize spooling from an application program:

- All three spooling levels have in common the requirement that the application must have a file open to a collector before spooling can begin.
- An application can spool data from several jobs concurrently. You must open a separate file to a collector for each job.
- The application process can use different levels of spooling to the same collector at the same time. However, the levels cannot be combined with one another to spool data for the same job.
- Level-1 spooling does not involve any of the interface procedures. Jobs spooled in this manner are assigned the default job attributes.
- Level-2 spooling is the same as level 1, except that you can specify job attributes using SPOOLSTART.
- Level-3 spooling requires the use of the spooler interface procedures: SPOOLSTART, SPOOLJOBNUM, SPOOLWRITE, SPOOLEND, SPOOLCONTROL, SPOOLCONTROLBUF, and SPOOLSETMODE. These allow a more efficient data transfer than is possible at levels 1 or 2.

- SPOOLCONTROL, SPOOLCONTROLBUF, and SPOOLSETMODE take the place of the Guardian file-system procedures CONTROL, CONTROLBUF, and SETMODE, for level-3 spooling.
- SPOOLWRITE compresses and stores the data from several successive calls. Only when its buffer is full does it write to the spooler.
- Applications should issue a top-of-form control at the beginning and end of all jobs. This ensures that data from two different jobs will not be printed on the same page.

Example of a Level-1 Application Program

You send data to the spooler at level 1 in 3 steps:

1. Open a file to the collector using the Guardian file-system OPEN procedure.
2. Write data to the collector with calls to the Guardian file-system procedures WRITE[X], CONTROL, CONTROLBUF, and SETMODE. The amount of data written by a given call might vary.
3. Signal the end of the job by using the Guardian file-system CLOSE procedure to close the file to the collector. When the collector receives a system message telling it that the file has been closed, it removes the job from the open state and places it in the ready state at the appropriate destination.

Example 2-1 is an example of level-1 spooling.

Example 2-1. Annotated Example of Level-1 Spooling (page 1 of 2)

```
! This program is an example of level-1 spooling. It consists of 3
! procedures: error, getline, and root, and it calls the
! Guardian procedures OPEN, CLOSE, WRITE, and STOP.


! error --this procedure handles I/O errors.  It performs the
!   necessary steps for recovery or it aborts the program.


! sperror --this procedure handles Spooler errors.  It performs the
!   necessary steps for recovery or it aborts the program.  It has
!   a single INT parameter that is the error code returned from a
!   Spooler Interface Procedure.


! getline --this procedure returns a line of data for spooling.
!   It is an INT procedure that returns a zero (FALSE) value when
!   it has no data to spool.  It has two parameters: line and
!   length.  line is a reference to a 40-word (80-byte) array.
!   The array is filled with the line of data to be spooled.
!   length is a reference to an INT that is set to the number of
!   bytes to be written from line.


! root --this is the main procedure.  It performs all the file
!   management to the collector and calls the other procedures in
!   the program as needed.

?nolist
INT counter := 0;
?SOURCE $SYSTEM.SYSTEM.EXTDECS(OPEN, CLOSE, WRITE, STOP)

PROC error;
   BEGIN
   CALL STOP;
   END;
INT PROC getline( line, length);
   INT .line,
       .length;
```

Example 2-1. Annotated Example of Level-1 Spooling (page 2 of 2)

```
   BEGIN
   int temp, done;
   temp :=0;
   temp := counter.<13:15>     ;
   temp := temp * 5;
   line [0] ':=' "  ";
   line [1] ':=' line[0] for 39;
   line [temp] ':=' "0123456789";
   length := 80;
   IF counter >  120 THEN done := 0 ELSE done := 1;
   counter := counter + 1;
   return done;
   END;
?list
PROC root MAIN;
BEGIN
   ! Declarations
   INT collector [0:11] := "$S      #LLP    LLP     ",
       ! contains the collector and location name in
       ! internal format
       line [0:39],
       ! contains the line of data to spool
       length,
       ! contains the number of bytes to write from line
       collectnum;
       ! contains the collector's file number returned from open

   ! Open file to collector, and check for errors.
   CALL OPEN( collector, collectnum);
   IF <> THEN CALL error;
   ! Get a line of data and test for done.
   ! If done, fall through.
   WHILE getline(line, length) DO
      BEGIN
      ! Write the line to the collector and check for errors
      CALL WRITE ( collectnum, line, length);
      IF <> THEN CALL error;
      END;
   ! Close the file to the collector and stop the program
   CALL CLOSE(collectnum);
   IF <> THEN CALL error;
   CALL STOP;

   END;
```

Example of a Level-2 Application Program

You send data to the spooler at level 2 in 4 steps:

1. Open a file to a collector with the Guardian file-system OPEN procedure. This file must be opened with waited I/O.
2. Call SPOOLSTART, passing the file number. You can specify the location, form name, report name, number of copies, page size, and instructions on whether or not to place the job in the HOLD state.
3. Send data to the spooler with calls to file-system procedures WRITE[X], CONTROL, CONTROLBUF, and SETMODE. Each call to WRITE[X] is limited to 900 bytes or fewer.

4. The job leaves the open state when you close the file to the collector with the file-system CLOSE procedure. Depending on the hold option specified in the call to SPOOLSTART, the job will go to the READY state or the HOLD state at the location specified.

Example 2-2 is an example of level-2 spooling.

Example 2-2. Annotated Example of Level-2 Spooling (page 1 of 3)

```
! This program is an example of level-2 spooling. It consists of 4
! procedures:  error, sperror, getline, and root, and it calls the
! Guardian procedures OPEN, CLOSE, WRITE, and STOP.  It also
! calls SPOOLSTART to specify the attributes of the job.

! error --this procedure handles I/O errors.  It performs
!   the necessary steps for recovery or it aborts the program.

! sperror --this procedure handles spooler errors.  It
!   performs the necessary steps for recovery or it aborts the
!   program.  It has a single INT parameter that is the error
!   code returned from the spooler interface procedure.

! getline --this procedure returns a line of data for
!   spooling. It is an INT procedure that returns a zero
!   (FALSE) value when it has no data to spool.  It has two
!   parameters:  line and length.  line is a reference to a
!   40-word (80-byte) array.  The array is filled with the line
!   of data to be spooled.  length is a reference to an INT that
!   is set to the number of bytes to be written from line.

! root --this is the main procedure.  It performs all
!   the file management to the collector and calls the other
!   procedures in the program as needed.
```

Example 2-2. Annotated Example of Level-2 Spooling (page 2 of 3)

```
?nolist
INT counter := 0;
?SOURCE $SYSTEM.SYSTEM.EXTDECS(OPEN, CLOSE, WRITE, STOP,
?SPOOLSTART)

PROC error;
   BEGIN
   CALL STOP;
   END;
PROC sperror (errnum);
   INT errnum;
   BEGIN
   CALL STOP;
   END;
INT PROC getline( line, length);
   INT .line,
       .length;
   BEGIN
   int temp, done;
   temp :=0;
   temp := counter.<13:15>     ;
   temp := temp * 5;
   line [0] ':=' "  ";
   line [1] ':=' line[0] for 39;
   line [temp] ':=' "0123456789";
   length := 80;
   IF counter >  120 THEN done := 0 ELSE done := 1;
   counter := counter + 1;
   RETURN done;
   END;
?list
PROC root MAIN;
BEGIN
   ! Declarations
   INT collector [0:11] := "$S      #LP3    LP3     ",
       ! contains the collector and location name in internal
       ! format
       line [0:39],
       ! contains the line of data to spool
       length,
       ! contains the number of bytes to write from line
       collectnum,
       ! contains the collector's file number returned from OPEN
       location [0:7] := "#LPRMT3         ",
       ! contains the job's new location
       sperrnum;
       ! receives SPOOLSTART error code
```

Example 2-2. Annotated Example of Level-2 Spooling (page 3 of 3)

```
! Open file to collector, and check for errors.
CALL OPEN( collector, collectnum);
IF <> THEN CALL error;
! Call SPOOLSTART to specify job's attributes
!     location              #LPRMT3
!     form name             blanks  (default)
!     report name           user's name and group name (default)
!     number of copies      1 (default)
!     page size             40
!     flags
!       hold                off
!       holdafter           on
!       NonStop bit         off
!       priority            7
sperrnum := SPOOLSTART(collectnum,,location,,,,
  40,%B0000000000100111);
! Test for an error from SPOOLSTART
IF sperrnum THEN CALL sperror(sperrnum);
! Get a line of data and test for done.
! If done, fall through.
WHILE getline(line, length) DO
   BEGIN
   ! Write the line to the collector and test for errors
   CALL WRITE ( collectnum, line, length);
   IF <> THEN CALL error;
   END;

! Close the file to the collector and stop the program
CALL CLOSE(collectnum);
IF <> THEN CALL error;
CALL STOP;
END;
```

Example of a Level-3 Application Program

You send data to the spooler at level 3 in 5 steps:

1. Open a file to a collector with the file-system OPEN procedure. This file must be opened with waited I/O.
2. Call SPOOLSTART, including the *level-3-buffer* parameter.
3. Send data to the collector with calls to the SPOOLWRITE, SPOOLCONTROL, SPOOLCONTROLBUF, and SPOOLSETMODE procedures. As noted earlier, not all calls to these procedures actually write data to the spooler. However, the blocking of data is transparent to the user.
4. The application signals the end of the job by calling the SPOOLEND procedure. Because you call SPOOLEND instead of the file-system CLOSE procedure to close the file, you can begin spooling another job without reopening a file to the collector.
5. At the end of the program run, close the collector file.

Example 2-3 is an example of level-3 spooling.

Example 2-3. Annotated Example of Level-3 Spooling (page 1 of 3)

```
! This program is an example of level-3 spooling. It consists of 4
! procedures:  error, sperror, getline, and root, and it calls the
! Guardian procedures OPEN, CLOSE, and STOP.  It uses the
! spooler interface procedures SPOOLSTART, SPOOLWRITE, and SPOOLEND
! to spool the job.

! error --this procedure handles I/O errors.  It performs the
!   necessary steps for recovery or it aborts the program.

! sperror --this procedure handles spooler errors.  It performs the
!   necessary steps for recovery or it aborts the program.  It has
!   a single INT value parameter that is the error code returned
!   from the spooler interface procedures.

! getline --this procedure returns a line of data for spooling.
!   It is an INT procedure that returns a zero (FALSE) value when
!   it has no data to spool.  It has two parameters:  line and
!   length.  line is a  reference to a 40-word (80-byte) array.
!   The array is filled with the line of data to be spooled.
!   length is a reference to an INT that is set to the number of
!   bytes to be written from line.

! root --this is the main procedure.  It performs all the file
!    management to the collector and calls the other procedures in
!    the program as needed.

?nolist
INT counter := 0;
```

Example 2-3. Annotated Example of Level-3 Spooling (page 2 of 3)

```
?SOURCE $SYSTEM.SYSTEM.EXTDECS (OPEN, CLOSE, WRITE, STOP, ?SPOOLSTART,
SPOOLWRITE, SPOOLEND)
PROC error;
   BEGIN
   CALL STOP;
   END;
PROC sperror (errnum);
INT errnum;
   BEGIN
   CALL STOP;
   END;
INT PROC getline( line, length);
   INT .line,
       .length;
   BEGIN
   int temp, done;
   temp :=0;
   temp := counter.<13:15>     ;
   temp := temp * 5;
   line [0] ':=' "  ";
   line [1] ':=' line[0] for 39;
   line [temp] ':=' "0123456789";
   length := 80;
   IF counter >  120 THEN done := 0 ELSE done := 1;
   counter := counter + 1;
   RETURN done;
   END;
?list

PROC root MAIN;
BEGIN
   ! Declarations
   INT collector [0:11] := "$S      #LP3    LP3     ",
       ! contains the file name of the collector and
       ! location in internal format
       line [0:39],
       ! contains the line of data to spool
       length,
       ! contains the number of bytes to write from line
       collectnum,
       ! contains the file number returned from OPEN
       location [0:7] := "#LPRMT3         ",
       ! contains the new location for the job
       sperrnum,
       ! receives SPOOLSTART error code
       .buffer[0:511];
       ! this is the level-3 buffer
   ! Open file to collector, and check for errors.
```

Example 2-3. Annotated Example of Level-3 Spooling (page 3 of 3)

```
CALL OPEN( collector, collectnum);
IF <> THEN CALL error;
! Call SPOOLSTART to specify new job attributes
!     location              #LPRMT3
!     form name             blanks  (default)
!     report name           user's name and group name (default)
!     number of copies      1 (default)
!     page size             40
!     flags
!       hold                off
!       holdafter           on
!       NonStop bit         off
!       priority            7

sperrnum :=
   SPOOLSTART(collectnum, buffer, location,,,, 40,
     %B0000000000100111);

! Test for an error from SPOOLSTART
If sperrnum THEN CALL sperror(sperrnum);

! Get a line of data and test for done.
! If done, fall through.

WHILE getline(line, length) DO
   BEGIN
   ! Write line to collector and check for errors
   sperrnum := SPOOLWRITE (buffer, line, length);
   IF sperrnum THEN CALL sperror(sperrnum);
   END;

! End this job and change the flag settings to
!     flags
!       cancel               off
!       hold                 on
!       holdafter            off
!       NonStop bit          off
!       priority             4
sperrnum := SPOOLEND(buffer, %B0000000001000100);
IF sperrnum THEN CALL sperror(sperrnum);

! Close the file to the collector and stop the program

CALL CLOSE(collectnum);
IF <> THEN CALL error;
CALL STOP;
END;
```

COBOL Spooling

Programs written in COBOL can spool their data at level 2 or 3 in the same manner as a TAL program. COBOL users typically use the spooling facilities provided by COBOL. See the *COBOL85 Manual* for more information.

COBOL Spooling—Level 1

Level-1 spooling from COBOL is not supported.

COBOL Spooling—Levels 2 and 3

To spool from COBOL, use one of the COBOL utility routines:

- COBOLSPOOLOPEN permits level-2 spooling for COBOL users.
- COBOL85^SPECIAL^OPEN permits level-2 or level-3 spooling for COBOL85 users.
- COBOL_SPECIAL_OPEN permits level-2 or level-3 spooling.

Use any of the above procedures to perform both the OPEN and SPOOLSTART operations. Following the OPEN, write the data to the collector as follows:

1. Using a SELECT statement, assign a COBOL file descriptor name to the process name of the collector. For example, if $SPL is a collector, the Input-Output Section of the Environment Division would contain the statement:

   ```
   SELECT SPOOLER ASSIGN "$SPL".
   ```

2. In the File Section of the Data Division, use a file definition (FD) to specify the record to be associated with writes to the spooler:

   ```
   FD SPOOLER RECORD CONTAINS 80 CHARACTERS LABEL RECORDS ARE
   OMITTED.  01 SPOOL-LINE    PIC X(80).
   ```

3. In the Procedure Division, open the spooler as the output file:

   ```
   OPEN OUTPUT SPOOLER.
   ```

4. Write each line of data to the spooler. For example, if the data to be spooled is in DATA-LINE, the WRITE statement for the above FD would look like the following:

   ```
   WRITE SPOOL-LINE FROM DATA-LINE.
   ```

Example 2-4 is an example of a program that performs COBOL spooling. A specification in the Working Storage Section determines whether level-2 or level-3 spooling is used.

Example 2-4. Example of Spooling From COBOL (page 1 of 2)

```
 IDENTIFICATION DIVISION.
   PROGRAM-ID.  SPOOLER.

 ENVIRONMENT DIVISION.

   CONFIGURATION SECTION.
     SOURCE-COMPUTER.  HP.
     OBJECT-COMPUTER.  HP.
     SPECIAL-NAMES.  FILE "$SYSTEM.SYSTEM.CBL85UTL" IS CBL85UTL.
   INPUT-OUTPUT SECTION.
     FILE-CONTROL.
       SELECT IN-FILE
              ASSIGN TO "SPOOLEE"
              FILE STATUS IS FILE-STAT.
       SELECT OUT-FILE
              ASSIGN TO "$S"
              FILE STATUS IS FILE-STAT.

 DATA DIVISION.
 FILE SECTION.

 FD  IN-FILE
     RECORD CONTAINS 1 TO 80 CHARACTERS,
     LABEL RECORDS ARE OMITTED.
 01  IN-REC            PIC X(80).

 FD  OUT-FILE
     LABEL RECORDS ARE OMITTED.
 01  OUT-REC           PIC X(132).

 WORKING-STORAGE SECTION.

 01  FILE-STAT.
     05  FILE-STAT-1   PIC X.
         88 IN-FILE-EOF VALUE "1".
     05  FILE-STAT-2   PIC X.

 01  FLAGS             PIC 9(4)  COMP  VALUE 99.
*                                        99 = 64  (HOLD)
*                                            +32  (HOLDAFTER)
*                                            + 3  (PRIORITY 3)
 01  ERROR-CODE        PIC 9(4).
 01  LOCATION.
     03 GROUP-NAME     PIC X(8) VALUE "#LP     ".
     03 LOCATION-NAME  PIC X(8) VALUE SPACES.
 01  FORM-NAME         PIC X(16)       VALUE "PREPRINTED".
 01  REPORT-NAME       PIC X(16)       VALUE "NAME OF REPORT".
 01  SPOOLER-OPEN      PIC S9(4) COMP  VALUE 1.
 01  LEVEL-2           PIC S9(4) COMP  VALUE 0.
 01  LEVEL-3           PIC S9(4) COMP  VALUE 1.
```

Example 2-4. Example of Spooling From COBOL (page 2 of 2)

```
PROCEDURE DIVISION.
DECLARATIVES.
    UA-IN-FILE SECTION.
        USE AFTER ERROR PROCEDURE ON IN-FILE.
    UA-IN-FILE-PROC.
        IF NOT IN-FILE-EOF DISPLAY "IN-FILE ERROR=" FILE-STAT.
    UA-OUT-FILE SECTION.
        USE AFTER ERROR PROCEDURE ON OUT-FILE.
    UA-OUT-FILE-PROC.
        IF NOT IN-FILE-EOF DISPLAY "OUT-FILE ERROR=" FILE-STAT.
END DECLARATIVES.

MAIN SECTION.
BEGIN-PROGRAM.
    PERFORM A-INIT
    PERFORM B-DO-IT UNTIL IN-FILE-EOF
    PERFORM C-EOJ
    STOP RUN.
A-INIT.
    OPEN INPUT IN-FILE
    ENTER "COBOL85^SPECIAL^OPEN" OF CBL85UTL
        USING OUT-FILE
              SPOOLER-OPEN
              OMITTED
              OMITTED
              OMITTED
              LEVEL-2
              LOCATION
              FORM-NAME
              REPORT-NAME
              OMITTED
              OMITTED
              FLAGS
        GIVING ERROR-CODE
    IF ERROR-CODE NOT = 0
       DISPLAY "COBOL85^SPECIAL^OPEN ERROR=" ERROR-CODE
       STOP RUN
    END-IF
    .
B-DO-IT.
    READ IN-FILE
        NOT AT END WRITE OUT-REC FROM IN-REC
    END-READ
    .
C-EOJ.
    CLOSE IN-FILE
          OUT-FILE
    .
```

Spooling From a NonStop Process Pair

An application process sending data to a collector can run as a NonStop process pair. Programmers writing such applications should be aware of the checkpointing considerations described in this subsection.

This discussion assumes that you already have knowledge of fault-tolerant programming (coding NonStop process pairs).

For an application program to run as a NonStop process pair, the system spooler should also be running as a NonStop process pair. Otherwise, an attempt at NonStop spooling from the application will not be fully effective.

If the collector is running as a NonStop process pair, you can spool with or without the spooler interface procedures. However, the considerations are different in each case.

Note. The examples in this subsection deal with checkpointing only to ensure the integrity of the application-collector interface. Checkpointing the remainder of the program is left to the programmer.

You must decide whether or not the application can tolerate duplication of a line of data in the event of a failure of the application's primary process. If the application cannot tolerate duplication of any lines, then sync depth must be specified when the collector is opened.

Use of Sync Depth

When a NonStop process pair opens the collector, it can set the sync-depth parameter of the file-system OPEN or FILE_OPEN_ procedure to a value of 1 or more, up to 15. Each write to the collector is then tagged by the file system with a sync ID. This ensures that, in the event of a failure of the primary processor, the collector will recognize any line rewritten by the backup.

If you have opened the collector with a nonzero sync depth, the information checkpointed should include the synchronization block of the file to the collector. A sync depth greater than 1 allows the application to perform less-frequent checkpoints. If you open the file to the collector with a sync depth of *n*, the application need only checkpoint before every *n*th write operation.

Spooling—Levels 1 and 2

The considerations for spooling on levels 1 and 2 from a NonStop process pair are very similar. Your main concern while performing such spooling from an application program is whether you can afford duplication in your spooled job.

Spooling With a Zero Sync Depth

If duplication of data lines can be tolerated, then a reasonable checkpointing strategy is to checkpoint the primary processor stack, the line of data to be spooled, and the

synchronization block for the file to the collector immediately before every write of a line of data to the collector.

In the event of a failure after the checkpoint, the backup process will execute the write with the correct line.

If the failure occurs following the write but before the next checkpoint, the backup will re-execute the write of the last line sent. When you print this job, the listing will contain two copies of the line.

Note. When you open a job with a sync depth of 0, the collector does not checkpoint as often for level-1 jobs as it does for jobs with a higher sync depth. Level-1 jobs opened with a sync depth of 0 could lose lines of data when a collector takeover occurs.

Example 2-5 illustrates this checkpointing strategy in level-1 spooling.

Example 2-5. Annotated Example of Level-1 Spooling From a NonStop Process Pair With a Zero Sync Depth (page 1 of 3)

```
! This program is an example of level-1 NonStop spooling with a
! zero sync depth.  It consists of 5 procedures:  error, cherror,
! stbackup, getline, and root, and it calls the Guardian
! procedures OPEN, CLOSE, WRITE, STOP, CHECKOPEN, and CHECKPOINT.

! error --this procedure handles I/O errors.  It performs the
!   necessary steps for recovery or it aborts the program.

! cherror --this procedure handles checkpointing errors.
!   It performs the necessary steps for recovery or it aborts the
!   program.  It has a single INT parameter that is the back error
!   returned from CHECKOPEN or the status word returned from
!   CHECKPOINT.  If it is called with a 0 value, it will stop the
!   backup process.

! stbackup --this procedure opens $RECEIVE, checks the Startup
!   message, and decides whether this program is the primary or
!   backup procedure.  If it is the primary, it starts a backup.
!   Otherwise, it waits for checkpointing information from the
!   primary.

! getline --this procedure returns a line of data for spooling.
!   It is an INT procedure that returns a zero (FALSE) value when
!   it has no data to spool.  It has two parameters:  line and
!   length. line is a reference to a 40-word (80-byte) array.
!   length is a reference to an INT that is set to the number of
!   bytes to be written from line.

! root --this is the main procedure.  It performs all the file
!   management to the collector and calls the other procedures in
!   the program as needed.

?nolist
INT counter := 0;
```

Example 2-5. Annotated Example of Level-1 Spooling From a NonStop Process Pair With a Zero Sync Depth (page 2 of 3)

```
?SOURCE $SYSTEM.SYSTEM.EXTDECS(OPEN, CLOSE, WRITE, STOP,
?CHECKOPEN,CHECKPOINT)

PROC error;
   BEGIN
   CALL STOP;
   END;
PROC cherror (george);
   INT george;
   BEGIN
   CALL STOP;
   END;
PROC stbackup;
   BEGIN
   counter := counter;
   END;
INT PROC getline( line, length);
   INT .line,
       .length;
   BEGIN
   int temp, done;
   temp :=0;
   temp := counter.<13:15>     ;
   temp := temp * 5;
   line [0] ':=' "  ";
   line [1] ':=' line[0] for 39;
   line [temp] ':=' "0123456789";
   length := 80;
   IF counter >  120 THEN done := 0 ELSE done := 1;
   counter := counter + 1;
   RETURN done;
   END;
?list
PROC root MAIN;
BEGIN
   ! Declarations
   INT collector [0:11] := "$S      #LP3    LP3     ",
       ! contains the collector and location name in
       ! internal format
       collectnum,
       ! contains the collector's file number returned from open
       err,
       ! contains the CHECKOPEN back error or the CHECKPOINT status
       ! word
       flags := %B0100100000000000,
       ! contains the bit pattern for the flags parameter to OPEN
       ! and CHECKOPEN
```

Example 2-5. Annotated Example of Level-1 Spooling From a NonStop Process Pair With a Zero Sync Depth (page 3 of 3)

```
      line [0:39],
      ! contains the line of data to spool
      length;
      ! contains the number of bytes to write from line

! Start the backup and check if this is the backup

CALL stbackup;

! Open file to collector, and check for errors.
CALL OPEN( collector, collectnum, flags, 0);
IF <> THEN CALL error;

! CHECKOPEN the successful open to the collector
! and test for error.
CALL CHECKOPEN( collector, collectnum, flags,0,,, err);
IF <> THEN CALL cherror(err);

! Get a line of data and test for done.
! If done, fall through.

WHILE getline(line, length) DO
   BEGIN

   ! CHECKPOINT stack (including line of data) and
   ! synchronization block

   err := CHECKPOINT(line, , collectnum);

   ! Write the line to the collector and check for errors
   CALL WRITE ( collectnum, line, length);
   IF <> THEN CALL error;
   END;

! Close the file to the collector

CALL CLOSE(collectnum);
IF <> THEN CALL error;

! Stop backup and stop primary

CALL cherror(0);
CALL STOP;
END;
```

Spooling With a Nonzero Sync Depth

To prevent a line from being duplicated, you can take advantage of the sync depth feature of the file system.

To use this feature, you should open the collector with a nonzero sync depth. The program should perform a checkpoint before every *n*th write to the collector, where *n* is the sync depth. At that time, the following should be checkpointed:

- The data stack
- The data line about to be written to the collector
- The synchronization block of the file to the collector, including the current sync ID and the access control block

The collector will keep track of up to 15 sync IDs; therefore, an application program can open the collector with any sync depth up to and including 15.

Example 2-6 is an example of level-2 spooling with a nonzero sync depth.

Example 2-6. Annotated Example of Level-2 Spooling From a NonStop Process Pair With a Nonzero Sync Depth (page 1 of 4)

```
! This program is an example of level-2 NonStop spooling with a
! sync depth of 5.  It consists of 6 procedures:  cherror,
! stbackup, error, sperror, getline, and root, and it calls the
! Guardian procedures OPEN, CLOSE, WRITE, STOP, CHECKOPEN, and
! CHECKPOINT.  It also calls SPOOLSTART to specify the attributes
! of the job.

! cherror --this procedure handles checkpointing errors.
!   It performs the necessary steps for recovery or it aborts the
!   program.  It has a single INT parameter that is the back error
!   returned from CHECKOPEN or the status word returned from
!   CHECKPOINT.  If it is called with a 0 value, it will stop the
!   backup process.

! stbackup --this procedure decides whether this program is the
!   primary or backup procedure.  If it is the primary, it starts a
!   backup.  Otherwise, it waits for checkpointing information from
!   the primary.  Opens $RECEIVE and reads the Startup message.

! error --this procedure handles I/O errors.  It performs the
!   necessary steps for recovery or it aborts the program.

! sperror --this procedure handles spooler errors.  It performs the
!   necessary steps for recovery or it aborts the program.  It has
!   a single INT parameter that is the error code returned from the
!   spooler interface procedure.
```

Example 2-6. Annotated Example of Level-2 Spooling From a NonStop Process Pair With a Nonzero Sync Depth (page 2 of 4)

```
! getline --this procedure returns a line of data for spooling.
!   It is an INT procedure that returns a zero (FALSE) value when
!   it has no data to spool.  It has two parameters:  line and
!   length.  line is a reference to a 40-word (80-byte) array.
!   The array is filled with the line of data to be spooled.
!   length is a reference to an INT that is set to the number of
!   bytes to be written from line.

! root --this is the main procedure.  It performs all the file
!   management to the collector and calls the other procedures in
!   the program as needed.

?nolist
INT counter := 0;

?SOURCE $SYSTEM.SYSTEM.EXTDECS(OPEN, CLOSE, WRITE, STOP,
? SPOOLSTART, CHECKOPEN, CHECKPOINT)
PROC cherror (george);
   INT george;
   BEGIN
   CALL STOP;
   END;
PROC stbackup;
   BEGIN
   counter := counter;
   END;
PROC error;
   BEGIN
   CALL STOP;
   END;
PROC sperror (errnum);
   INT errnum;
   BEGIN
   CALL STOP;
   END;
INT PROC getline( line, length);
   INT .line,
       .length;
   BEGIN
   int temp, done;
   temp :=0;
   temp := counter.<13:15>     ;
   temp := temp * 5;
   line [0] ':=' "  ";
   line [1] ':=' line[0] for 39;
   line [temp] ':=' "0123456789";
   length := 80;
```

Example 2-6. Annotated Example of Level-2 Spooling From a NonStop Process Pair With a Nonzero Sync Depth (page 3 of 4)

```
   IF counter >  120 THEN done := 0 ELSE done := 1;
   counter := counter + 1;
   RETURN done;
   END;
?list
PROC root MAIN;
BEGIN
   ! Declarations
   INT collector [0:11] := "$S      #LP3    LP3     ",
       ! contains the collector and location name in
       ! internal format
       collectnum,
       ! contains the collector's file number returned from OPEN
       flags := %B0100100000000000,
       ! contains the bit pattern for the flags parameter to OPEN
       ! and CHECKOPEN
       location [0:7] := "#LPRMT3         ",
       ! contains the job's new location
       line [0:39],
       ! contains the line of data to spool
       length,
       ! contains the number of bytes to write from line
       chcount := 0,
       ! contains the number of uncheckpointed writes pending
       err,
       ! contains the CHECKOPEN back error or the CHECKPOINT
       ! status word
       sperrnum;
       ! receives SPOOLSTART error code
   ! Open file to collector
   !  flags
   !     receive messages     on
   !     access mode          write only
   !     open process nowait  off
   !     exclusion mode       shared
   !     wait/nowait          wait
   !  sync depth              5
   CALL OPEN( collector, collectnum, flags, 5);
   ! Check for errors
   IF <> THEN CALL error;
   ! CHECKOPEN the successful open to the collector
   CALL CHECKOPEN(collector,collectnum, flags, 5,,, err);
   ! Check for a CHECKOPEN error
   IF <> THEN CALL cherror(err);
```

Example 2-6. Annotated Example of Level-2 Spooling From a NonStop Process Pair With a Nonzero Sync Depth (page 4 of 4)

```
! Checkpoint call to SPOOLSTART (in this example none of the
! parameters to SPOOLSTART have been computed so this checkpoint
! is not necessary, but most practical programs would need this
! checkpoint).
CALL CHECKPOINT( location);
! Call SPOOLSTART to specify job's attributes
!     location              #LPRMT3
!     form name             blanks  (default)
!     report name           user's name and group name (default)
!     number of copies      1 (default)
!     page size             40
!     flags
!       hold                off
!       holdafter           on
!       NonStop bit         off
!       priority            7

sperrnum := SPOOLSTART(collectnum,,location,,,,
  40,%B0000000000100111);

! Test for an error from SPOOLSTART
If sperrnum THEN CALL sperror(sperrnum);
! Get a line of data and test for done.
! If done, fall through.
WHILE getline(line, length) DO
   BEGIN
   ! Increment sync depth counter and test for
   ! sync depth overflow
   chcount := chcount + 1;
   IF chcount = 5 THEN
      BEGIN
      ! Checkpoint the stack (including line of data)
      ! and synchronization block. Reset sync count.
      CALL CHECKPOINT( line, , collectnum);
      chcount := 1;
      END;
   ! Write the line to the collector and test for errors
   CALL WRITE ( collectnum, line, length);
   IF <> THEN CALL error;
   END;
! Close the file to the collector
CALL CLOSE(collectnum);
IF <> THEN CALL error;
! Stop backup and stop primary
CALL cherror (0);
CALL STOP;
END;
```

Spooling—Level 3

An application process spooling at level 3 writes data to the collector using the spooler interface procedures. SPOOLWRITE, SPOOLCONTROL, SPOOLCONTROLBUF, and SPOOLSETMODE do not actually write a line of data to the collector each time one of them is called. Instead, they put the data in the *level-3-buffer* you specified in an earlier call to SPOOLSTART.

At the point that data specified in a call to a SPOOLWRITE, SPOOLCONTROL, SPOOLCONTROLBUF, or SPOOLSETMODE procedure would cause the *level-3-buffer* to overflow, the procedure checks to see whether bit 11 of the SPOOLSTART *flags* parameter has been set to 1. If that bit is 0, the procedure writes the buffer to the collector and begins refilling the buffer with the data line that would have overflowed the buffer.

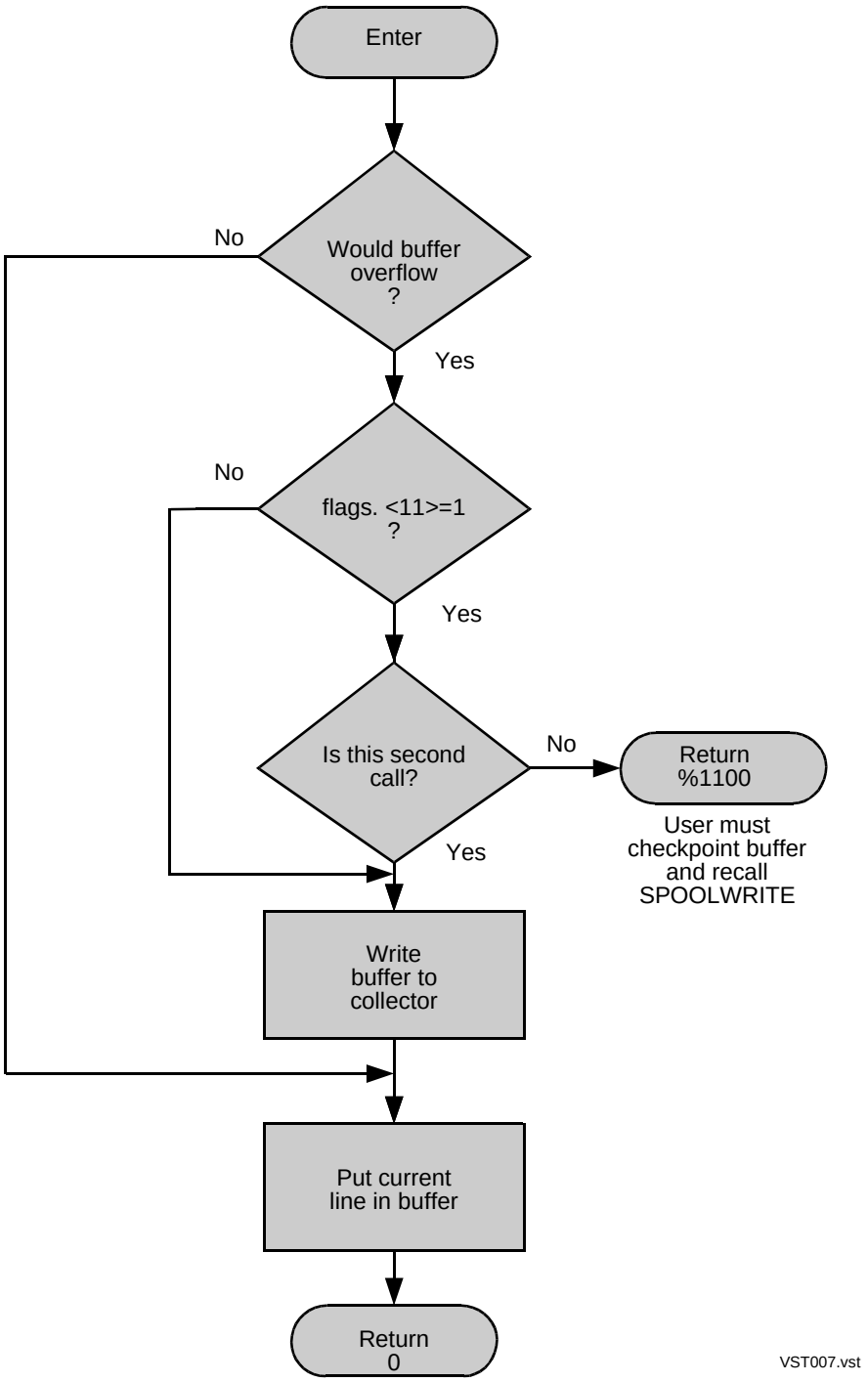
If bit 11 of *flags* is 1, however, the procedure exits before writing the *level-3-buffer* to the collector and returns a spooler error code value of %11000. This indicates to the application process that the *level-3-buffer* is about to be written to the collector and that a checkpoint should be performed.

After checkpointing, you must use the same data again to call the procedure that returned the %11000. This time, that procedure writes the buffer to the collector and begins refilling the buffer with the data line.

When using the spooler interface procedures, you can perform spooling from a NonStop process pair with or without checkpointing. In fact, you do not even have to use the fault-tolerant bit in SPOOLSTART (bit 11 of the *flags* parameter), but this would lead to inefficient checkpointing. Only fault-tolerant programs that use bit 11 of *flags* are considered. The application must checkpoint the data stack, the lbuffer, the latest line of data to be written, and the synchronization block of the file to the collector.

Figure 2-1 shows a flow chart of SPOOLWRITE, SPOOLSETMODE, SPOOLCONTROL, and SPOOLCONTROLBUF procedures for handling a call that would overflow the *level-3-buffer*.

Figure 2-1. Buffer Overflow Logic



Spooling With a Zero Sync Depth

After opening and initializing the *level-3-buffer*, you can begin spooling with the interface procedures.

If SPOOLCONTROL, SPOOLSETMODE, SPOOLCONTROLBUF, or SPOOLWRITE returns an error code of %11000, then the data given in the last call to that procedure would cause the *level-3-buffer* to be written to the collector. The following information should be checkpointed:

- The data stack
- The *level-3-buffer* (the number of bytes of *level-3-buffer* containing valid data is returned in the *bytes-written-to-buffer* parameter of the SPOOLWRITE, SPOOLSETMODE, SPOOLCONTROL, and SPOOLCONTROLBUF procedures)
- The data line that caused the %11000 return

After this information has been checkpointed, again call the procedure that returned the %11000 as an error code, using the same data. It writes the *level-3-buffer* to the collector on the second call and then begins refilling the buffer with the last line of data that was resubmitted.

If the primary fails after the *level-3-buffer* has been written to the collector but before the next checkpoint, it is possible that the entire *level-3-buffer* will be written twice. To prevent this duplication, a nonzero sync depth is required.

Example 2-7 is an example of level-3 spooling with a zero sync depth.

Example 2-7. Annotated Example of Level-3 Spooling From a NonStop Process Pair With a Zero Sync Depth (page 1 of 4)

```
! This program is an example of level-3 NonStop spooling with a
! zero sync depth.  It consists of 6 procedures:  cherror,
! stbackup, error, sperror, getline, and root, and it calls the
! Guardian procedures OPEN, CLOSE, STOP, CHECKOPEN, and
! CHECKPOINT.  It uses the spooler interface procedures SPOOLSTART,
! SPOOLWRITE, and SPOOLEND to spool the job.

! cherror --this procedure handles checkpointing errors.
!   It performs the necessary steps for recovery or it aborts the
!   program.  It has a single INT parameter that is the back error
!   returned from CHECKOPEN or the status word returned from
!   CHECKPOINT.  If it is called with a 0 value, it will stop the
!   backup process.

! stbackup --this procedure decides whether this program is the
!   primary or backup procedure.  If it is the primary, it starts a
!   backup.  Otherwise, it waits for checkpointing information from
!   the primary.
```

Example 2-7. Annotated Example of Level-3 Spooling From a NonStop Process Pair With a Zero Sync Depth (page 2 of 4)

```
! error --this procedure handles I/O errors. It performs the
!   necessary steps for recovery or it aborts the program.

! sperror --this procedure handles spooler errors. It performs the
!   necessary steps for recovery or it aborts the program.  It has
!   a single INT value parameter that is the error code returned
!   from the spooler interface procedures.

! getline --this procedure returns a line of data for spooling.
!   It is an INT procedure that returns a zero (FALSE) value when
!   it has no data to spool.  It has two parameters: line and
!   length.  line is a  reference to a 40-word (80-byte) array.
!   The array is filled with the line of data to be spooled.
!   length is a reference to an INT that is set to the number of
!   bytes to be written from line.

! root --this is the main procedure.  It performs all the file
!   management to the collector and calls the other procedures in
!   the program as needed.

?nolist
INT counter := 0;
?SOURCE $SYSTEM.SYSTEM.EXTDECS(OPEN, CLOSE, WRITE, STOP, ?SPOOLSTART,
SPOOLWRITE, SPOOLEND,CHECKOPEN, CHECKPOINT)
PROC cherror (george);
   INT george;
   BEGIN
   CALL STOP;
   END;
PROC stbackup;
   BEGIN
   counter := counter;
   END;
PROC error;
   BEGIN
   CALL STOP;
   END;
PROC sperror (errnum);
INT errnum;
   BEGIN
   CALL STOP;
   END;
INT PROC getline( line, length);
   INT .line,
       .length;
   BEGIN
   INT temp, done;
   temp :=0;
   temp := counter.<13:15>     ;
   temp := temp * 5;
   line [0] ':=' "  ";
   line [1] ':=' line[0] for 39;
   line [temp] ':=' "0123456789";
   length := 80;
   IF counter >  120 THEN done := 0 ELSE done := 1;
   counter := counter + 1;
   RETURN done;
   END;
```

Example 2-7. Annotated Example of Level-3 Spooling From a NonStop Process Pair With a Zero Sync Depth (page 3 of 4)

```
?list
PROC root MAIN;
BEGIN
   ! Declarations
   INT collector [0:11] := "$S      #LP3    LP3     ",
       ! contains the file name of the collector and
       ! location in internal format
       collectnum,
       ! contains the file number returned from OPEN
       location [0:7] := "#LPRMT3         ",
       ! contains the new location for the job
       flags := %B0100100000000000,
       ! contains the bit pattern for the flags parameter to OPEN
       ! and CHECKOPEN
       line [0:39],
       ! contains the line of data to spool
       length,
       ! contains the number of bytes to write from line
       err,
       ! contains the CHECKOPEN back error or the CHECKPOINT
       ! status word
       sperrnum,
       ! receives spooler error code
       .buffer[0:511],
       ! this is the level 3 buffer
       bytecount;
       ! contains the number of bytes written to the buffer

   ! Open file to collector, and check for errors.
   CALL OPEN( collector, collectnum, flags);
   IF <> THEN CALL error;

  ! CHECKOPEN successful open of the collector
   CALL CHECKOPEN(collector,collectnum, flags,,,, err);
   ! Check for a CHECKOPEN error
   IF <> THEN CALL cherror(err);
   ! Checkpoint call to SPOOLSTART (in this example none of the
   ! parameters to SPOOLSTART have been computed so this checkpoint
   ! is not necessary, but most practical programs would need this
   ! checkpoint)
   CALL CHECKPOINT( location);
   ! Call SPOOLSTART to specify new job attributes and level-3
   ! buffer
   !     location              #LPRMT3
   !     form name             blanks  (default)
   !     report name           user's name and group name (default)
   !     number of copies      1 (default)
   !     page size             40
   !     flags
   !       hold                off
   !       holdafter           on
   !       NonStop bit         on
   !       priority            7
```

Example 2-7. Annotated Example of Level-3 Spooling From a NonStop Process Pair With a Zero Sync Depth (page 4 of 4)

```
 sperrnum :=
   SPOOLSTART(collectnum, buffer, location,,,, 40,
   %B0000000000110111);

! Test for an error from SPOOLSTART
If sperrnum THEN CALL sperror(sperrnum);

! Get a line of data and test for done.
! If done, fall through.

WHILE getline(line, length) DO
   BEGIN
   ! Write line to collector and check for errors
   sperrnum := SPOOLWRITE (buffer, line, length);
   ! If buffer is ready to be written to collector
   ! call CHECKPOINT, then write buffer
   IF sperrnum = %11000 THEN
      BEGIN
      err := CHECKPOINT(line, buffer, bytecount,, collectnum);
      IF err THEN CALL cherror(err);
      sperrnum := SPOOLWRITE(buffer, line, length, bytecount);
      END;

   ! If there was an error from the first or second call to
   ! SPOOLWRITE, call sperror with the error code.
   IF sperrnum THEN CALL sperror(sperrnum);
   END;
! Call checkpoint before final write to collector by
! SPOOLEND
err := CHECKPOINT(line, buffer, bytecount,,collectnum);
IF err THEN CALL cherror(err);
! End this job and change the flag setting to
!     flags
!       cancel              off
!       hold                on
!       holdafter           off
!       NonStop bit         off
!       priority            4
sperrnum := SPOOLEND(buffer, %B0000000001000100);
IF sperrnum THEN CALL sperror(sperrnum);
! Close the file to the collector
CALL CLOSE(collectnum);
IF <> THEN CALL error;
! Stop backup and stop primary
CALL cherror (0);
CALL STOP;
END;
```

Spooling With a Nonzero Sync Depth

In this type of spooling, you must specify the sync depth when you open the collector and set bit 11 of *flags* in SPOOLSTART. Your program need not perform a checkpoint each time SPOOLWRITE, SPOOLSETMODE, SPOOLCONTROL, or SPOOLCONTROLBUF returns an error code of %11000. You should perform a checkpoint each time the number of writes performed since the last checkpoint equals the sync depth. When you do checkpoint data, the following information should be checkpointed:

- The data stack
- The *level-3-buffer* (the number of bytes of *level-3-buffer* containing valid data is returned in the *bytes-written-to-buffer* parameter of the SPOOLWRITE, SPOOLSETMODE, and SPOOLCONTROL procedures)
- The data line that caused the %11000 return

Example 2-8 is an example of level-3 spooling from a NonStop process pair with a sync depth greater than zero.

Example 2-8. Annotated Example of Level-3 Spooling From a NonStop Process Pair With a Nonzero Sync Depth (page 1 of 5)

```
! This program is an example of level-3 NonStop spooling with a
! sync depth of 3.  It consists of 6 procedures:  cherror,
! stbackup, error, sperror, getline, and root, and it calls the
! Guardian procedures OPEN, CLOSE, STOP, CHECKOPEN, and
! CHECKPOINT.  It uses the spooler interface procedures SPOOLSTART,
! SPOOLWRITE, and SPOOLEND to spool the job.

! cherror --this procedure handles checkpointing errors.
!   It performs the necessary steps for recovery or it aborts the
!   program.  It has a single INT parameter that is the back error
!   returned from CHECKOPEN or the status word returned from
!   CHECKPOINT.  If it is called with a 0 value, it will stop the
!   backup process.

! stbackup --this procedure decides whether this program is the
!   primary or backup procedure.  If it is the primary, it starts a
!   backup.  Otherwise, it waits for checkpointing information from
!   the primary.

! error --this procedure handles I/O errors.  It performs the
!   necessary steps for recovery or it aborts the program.

! sperror --this procedure handles spooler errors.  It performs
!   the necessary steps for recovery or it aborts the program.
!   It has a single INT value parameter that is the error code
!   returned from the spooler interface procedures.
```

Example 2-8. Annotated Example of Level-3 Spooling From a NonStop Process Pair With a Nonzero Sync Depth (page 2 of 5)

```
! getline --this procedure returns a line of data for spooling.
!   It is an INT procedure that returns a zero (FALSE) value when
!   it has no data to spool.  It has two parameters: line and
!   length.  line is a  reference to a 40-word (80-byte) array.
!   The array is filled with the line of data to be spooled.
!   length is a reference to an INT that is set to the number of
!   bytes to be written from line.

! root --this is the main procedure.  It performs all the file
!   management to the collector and calls the other procedures in
!   the program as needed.

?nolist
INT counter := 0;
?SOURCE $SYSTEM.SYSTEM.EXTDECS(OPEN, CLOSE, WRITE, STOP, ?SPOOLSTART,
SPOOLWRITE, SPOOLEND,CHECKOPEN, CHECKPOINT)
PROC cherror (george);
   INT george;
   BEGIN
   CALL STOP;
   END;

PROC stbackup;
   BEGIN
   counter := counter;
   END;
PROC error;
   BEGIN
   CALL STOP;
   END;
PROC sperror (errnum);
INT errnum;
   BEGIN
   CALL STOP;
   END;
INT PROC getline( line, length);
   INT .line,
       .length;
   BEGIN
   INT temp, done;
   temp :=0;
   temp := counter.<13:15>     ;
   temp := temp * 5;
   line [0] ':=' "  ";
   line [1] ':=' line[0] for 39;
   line [temp] ':=' "0123456789";
   length := 80;
```

Example 2-8. Annotated Example of Level-3 Spooling From a NonStop Process Pair With a Nonzero Sync Depth (page 3 of 5)

```
   IF counter >  120 THEN done := 0 ELSE done := 1;
   counter := counter + 1;
   RETURN done;
   END;
?list
PROC root MAIN;
BEGIN
   ! Declarations
   INT collector [0:11] := "$S      #LLP    LLP     ",
       ! contains the file name of the collector and
       ! location in internal format
       collectnum,
       ! contains the file number returned from OPEN
       location [0:7] := "#LPRMT3         ",
       ! contains the new location for the job
       flags := %B0100100000000000,
       ! contains the bit pattern for the flags parameter to OPEN
       ! and CHECKOPEN
       line [0:39],
       ! contains the line of data to spool
       length,
       ! contains the number of bytes to write from line
       syncount := 0,
       ! contains the count of synchronized writes to the collector
       err,
       ! contains the CHECKOPEN back error or the CHECKPOINT
       ! status word
       sperrnum,
       ! receives spooler error code
       .buffer[0:511],
       ! this is the level-3 buffer
       bytecount;
       !contains the number of bytes already written to the buffer
   ! Open file to collector, and check for errors.
   CALL OPEN( collector, collectnum, flags, 5);
   IF <> THEN CALL error;
   ! CHECKOPEN successful open of the collector
   CALL CHECKOPEN(collector,collectnum, flags, 5,,, err);
   ! Check for a CHECKOPEN error
   IF <> THEN CALL cherror(err);
```

Example 2-8. Annotated Example of Level-3 Spooling From a NonStop Process Pair With a Nonzero Sync Depth (page 4 of 5)

```
! Checkpoint call to SPOOLSTART (in this example none of the
! parameters to SPOOLSTART have been computed so this checkpoint
! is not necessary, but most practical programs would need this
! checkpoint)
CALL CHECKPOINT( location);
! Call SPOOLSTART to specify new job attributes and level-3
! buffer
!     location              #LPRMT3
!     form name             blanks  (default)
!     report name           user's name and group name (default)
!     number of copies      1 (default)
!     page size             40
!     flags
!       hold                off
!       holdafter           on
!       NonStop bit         on
!       priority            7

sperrnum :=
   SPOOLSTART(collectnum, buffer, location,,,, 40,
   %B0000000000110111);

! Test for an error from SPOOLSTART
IF sperrnum THEN CALL sperror(sperrnum);
! Get a line of data and test for done.
! If done fall through.
WHILE getline(line, length) DO
   BEGIN
   ! Write line to collector and check for errors
   sperrnum := SPOOLWRITE (buffer, line, length);
   ! If buffer is ready to be written to collector
   ! call CHECKPOINT then write buffer
   IF sperrnum = %11000 THEN
      BEGIN
      !Increment sync depth counter and check for overflow
      syncount := syncount + 1 ;
      IF syncount = 3 THEN
         BEGIN
         err := CHECKPOINT(line, buffer, bytecount,,collectnum);
         IF err THEN CALL cherror(err);
         syncount := 1
         END;
```

Example 2-8. Annotated Example of Level-3 Spooling From a NonStop Process Pair With a Nonzero Sync Depth (page 5 of 5)

```
        ! Write buffer to collector
        sperrnum := SPOOLWRITE(buffer, line, length, bytecount);
        IF sperrnum THEN CALL sperror(sperrnum);
        END
     ELSE CALL sperror(sperrnum);
     END;
! Call checkpoint before final write to collector by
! SPOOLEND
err := CHECKPOINT(line, buffer, bytecount,,collectnum);
IF err THEN CALL cherror(err);
! End this job and change the flag setting to
!     flags
!       cancel              off
!       hold                on
!       holdafter           off
!       NonStop bit         off
!       priority            4

sperrnum := SPOOLEND(buffer, %B0000000001000100);
IF sperrnum THEN CALL sperror(sperrnum);
! Close the file to the collector
CALL CLOSE(collectnum);
IF <> THEN CALL error;
! stop backup and stop primary
CALL cherror (0);
CALL STOP;
END;
```

3

Using the Spooler Print Procedures, Print Processes, and Perusal Processes

The spooler print procedures enable print and perusal processes to access spooled data and enable all print processes to communicate with the spooler supervisor. This section describes how to use the spooler print procedures and write print and perusal processes.

Print and Perusal Processes

A **print process** retrieves spooled job data from disk and sends it to the appropriate output device. The spooler can include several print processes, each controlling a number of different output devices.

Every device known to the supervisor has a print process associated with it. When a device becomes available and a job is waiting to print on that device, the supervisor instructs the print process associated with the device to begin reading and printing the job.

The standard print processes communicate with the supervisor and keep track of devices and jobs in a manner that is entirely transparent to the users of the spooler. However, if you choose to write your own print process, be aware of the interaction between print processes and the supervisor. The print procedures make this task easier. An example of a user-written print process is shown in Appendix A, Sample Print Process.

A process that can access spooled data without communicating with the supervisor is called a **perusal process**. The Peruse utility is the perusal process supported by HP. An example of a perusal process is shown in Appendix B, Sample Perusal Process.

Summary of Print Procedures

The procedures involved in accessing spooled data are PRINTSTART, PRINTSTART2, PRINTINFO, and PRINTREAD.

The procedures involved in communicating with the spooler control process are PRINTINIT, PRINTINIT2, PRINTSTATUS, PRINTSTATUS2, PRINTREADCOMMAND, PRINTCOMPLETE, and PRINTCOMPLETE2.

The program filename of the standard print process is `$SYSTEM.SYSTEM.FASTP`.

Table 3-1 contains a summary of the print procedures. You can find the complete syntax and considerations of the spooler print procedures in Section 4, Spooler

Procedure Calls. Refer to Print Procedure Errors on page C-10 for a list of print procedure error codes.

Table 3-1. Summary of Print Procedures

Procedure	Function
PRINTCOMPLETE	Obtains a message from the spooler.
PRINTCOMPLETE2	Obtains a message from the spooler. This procedure includes batch enhancements to PRINTCOMPLETE.
PRINTINFO	Obtains information regarding a job being printed by the print process.
PRINTINIT	Initializes the print control buffer.
PRINTINIT2	Initializes the print control buffer. This procedure includes batch enhancements to PRINTINIT.
PRINTREAD	Obtains a line of spooled data.
PRINTREADCOMMAND	Interprets a message from the spooler supervisor.
PRINTSTART	Initializes a job buffer for a new job.
PRINTSTART2	Initializes a job buffer for a new job. This procedure includes batch enhancements to PRINTSTART.
PRINTSTATUS	Sends a message to the supervisor.
PRINTSTATUS2	Sends a message to the supervisor. This procedure includes batch enhancements to PRINTSTATUS.

How the Print Process Handles a Job

The following sequence of events occurs when the print process prints a job:

1. The spooler supervisor starts the print process associated with the device (unless it is already running) and sends it a startup message.
2. The print process opens a file to the supervisor and calls PRINTINIT.
3. When the supervisor tries to send a message to the print process, the print process calls PRINTCOMPLETE to obtain the message.
4. The message from the supervisor is interpreted by PRINTREADCOMMAND. If it is a start job message, the print process opens the data file and the device specified in the message.
5. The print process passes the file number of the data file to PRINTSTART.

6. The print process now reads the job, one line at a time, by a series of calls to PRINTREAD, and writes each line to the device until an end of file is returned.

   Note. Be aware that when some devices (notably drum printers) encounter a non-ASCII character, the printer can lock up unless you specified CTRLTOSPACE in the system generation configuration for the printer. Refer to the appropriate system generation manual for more information.

7. The print process informs the supervisor that the job is complete by calling PRINTSTATUS.
8. If the device is shared, the supervisor sends a “close device” indication to the print process. The print process closes the device and calls STOP.

In addition to the above action, the print process periodically calls AWAITIO[X] to check for an incoming message from the spooler supervisor. Such a message might indicate the start of another job (if the print process is capable of handling multiple concurrent jobs) or an instruction requiring specific action on the part of the print process, such as “skip to page 17” or “stop job.”

Errors occurring on the device are sent by the print process to the supervisor by using the PRINTSTATUS command.

The details of the interaction between the spooler supervisor and a print process are described later in this section.

External Declarations for Print Procedures

To use the print procedures in a TAL program, you must declare them to be external to your program. The external declarations for all procedures related to the spooler, including the print procedures, are located in the file `$SYSTEM.SYSTEM.EXTDECS0`. They can be sourced into your program with the following compiler command:

```
?SOURCE $SYSTEM.SYSTEM.EXTDECS0 ( PRINTSTART , PRINTINIT ,
?          PRINTINFO ,  PRINTCOMPLETE , PRINTREADCOMMAND ,
?          PRINTSTATUS , PRINTREAD )
```

See the *TAL Reference Manual* for a full explanation of the EXTERNAL procedure declaration and the ?SOURCE compiler directive.

Writing a Print Process

The following pages discuss the considerations associated with writing your own print process. Appendix A, Sample Print Process, contains an example of a user-written print process.

A print process must perform three tasks:

1. Read the Startup message.
2. Retrieve and print spooled data (using PRINTSTART and PRINTREAD).

3. Communicate with the spooler supervisor (using PRINTCOMPLETE, PRINTSTATUS, and PRINTREADCOMMAND).

Print Process Startup Message

A print process not currently running is started by the spooler supervisor for one of two reasons:

- A job is ready to be printed on a device controlled by the print process.
- A device controlled by the print process has been declared exclusive.

Immediately after starting a print process, the supervisor sends the print process a Startup message, passing it the process name of the supervisor.

The format of the Startup message from the supervisor is shown in Table 3-2.

The first action taken by a print process should be to read its Startup message to obtain the name of the supervisor, its backup processor, and its print process device parameter.

Table 3-2. Startup Message From the Spooler Supervisor

Word	Byte	Contents
[0]	[0:1]	-1 (all bits on).
[1:20]	[2:41]	ASCII blanks.
[21:32]	[42:65]	Process name of supervisor, blank-filled. This can be passed by the print process directly to the OPEN procedure.
[33]	[66:67]	Backup processor number, in ASCII, specified in the Spoolcom command PRINT BACKUP. If no backup was specified, this word contains -1 in ASCII (that is, %026461).
[34 left byte]	[68]	ASCII comma (,).
[34 right byte: 37 left byte]	[69:74]	Print process parameter; field contains six ASCII characters from the Spoolcom command PRINT PARM.
[37 right byte]	[75]	ASCII null (0).

Retrieving and Printing Spooled Data

The sequence of events involved in retrieving and printing a job is as follows:

1. Open the spooler supervisor with nowait I/O, call PRINTINIT to format a print control buffer, and call AWAITIO[X] to wait for a message from the supervisor. Call PRINTCOMPLETE to get the *print-control-buffer* from the supervisor.
2. Call PRINTREADCOMMAND to interpret the information contained in the supervisor *print-control-buffer*. PRINTREADCOMMAND returns the name of the data file, the job route and attributes, and the name of the device on which the job is to be printed.

3. Open the data file and the device, then call PRINTSTART. This places control information for the job into a *job-control-buffer*. The control information in the *job-control-buffer* consists of pointers to the current page and line number, the file numbers of the supervisor and data file, and so on.
4. Call PRINTREAD to get one line of spooled data. PRINTREAD also updates the control information in the print process buffer to point to the next line.
5. You can now access the job with a succession of calls to PRINTREAD.

   In theory, the print process can take any action whatsoever with the data; the usual action, though, is to write the data to the device specified by PRINTREADCOMMAND.

Note. You can write a print process that handles as many concurrent jobs as you like, subject to time and memory limitations. A separate *job-control-buffer* must be maintained for each job, and there must be some way of keeping track of which job control buffers go with which devices. The supervisor might attempt to start a job on each device that has been associated with a print process (using Spoolcom). Therefore, if a print process is associated with devices X, Y, and Z, it must be capable of handling three jobs concurrently.

Communicating With the Spooler Supervisor

Communication with the supervisor falls into two categories: responding to messages and sending errors.

Responding to Spooler Supervisor Messages

The print process must periodically check for a completion on the file to the supervisor. When a completion is detected, PRINTCOMPLETE is called to obtain the message, which is then passed to PRINTREADCOMMAND for interpretation. PRINTREADCOMMAND returns a control number to indicate the nature of the message.

It is up to the programmer to decide how often to check for a supervisor message. However, the more often you check for a message, the more responsive your print process will be to Spoolcom commands. For example, a print process that checks for completion after every write to the device can respond immediately to a message such as “skip to page 3,” “send job status,” or “suspend job.”

The supervisor times out any print process that waits more than 10 minutes to respond to a message. A print process that times out is put into the procerror state by the supervisor, and any devices controlled by that print process are considered unusable.

The *controlnum* values returned by PRINTREADCOMMAND, and the action that the print process should take in response, are

0 Open the device specified in the *device* parameter of PRINTREADCOMMAND.

1 Close the device specified in the *device* parameter of PRINTREADCOMMAND.

2 Start the job with attributes as specified in the PRINTREADCOMMAND parameters.

3 Cancel any incomplete I/O associated with the job on the *device* specified in the device parameter of PRINTREADCOMMAND. Then stop the job. After successfully stopping the job, send an “end of job” status to the supervisor by calling PRINTSTATUS with a *msg-type* of 2.

4 Resume the job on the device specified in the *device* parameter of PRINTREADCOMMAND.

5 Suspend the job on the device specified in the *device* parameter of PRINTREADCOMMAND.

Suspend reading and printing activity on the job pending another instruction; no new job can be started on the device.

6 Print a form-alignment template on the device specified in the *device* parameter of PRINTREADCOMMAND.

7 Start printing on the specified page on the device specified in the *device* parameter of PRINTREADCOMMAND.

8 Start printing on the page that is offset from the current page by the specified number of pages on the device specified in the *device* parameter of PRINTREADCOMMAND.

9 Send the status of the job printing on the device specified in the *device* parameter of PRINTREADCOMMAND.

Sending Error Messages to the Spooler Supervisor

The only messages that a print process sends to the supervisor without a supervisor request are error messages. The print process informs the supervisor of errors by sending the error to the supervisor using the PRINTSTATUS procedure. A print process can encounter errors from a print device during a call to WRITE[X] or from a job during a call to PRINTREAD.

Device Errors

When errors are encountered on a print device during the printing of a job, the print process should call PRINTSTATUS with the following messages:

- For the first occurrence of a device error following any number of successful operations on the device, the print process should specify a *msg-type* of 5 and pass the error number in the *error* parameter.

- If the print process encounters subsequent errors on the same device without an intervening successful operation, the print process should specify a *msg-type* of 1 and pass the error number in the *error* parameter.

The print process, however, continues printing other jobs on other devices.

For example, following a series of successful I/O operations, a print process receives an error 102 (paper out) from one of the devices it controls. The print process calls PRINTSTATUS with *msg-type* equals 5 and *error* equals 102. The print process suspends the job that was printing on that device. Later, the print process receives a "resume job" message from the supervisor. The print process attempts to resume the job on the specified device, but receives an error 100 (device not ready) on the first I/O operation. This time, the print process calls PRINTSTATUS with a *msg-type* of 1 and an *error* of 100.

The print process suspends the job on which the error occurred. A suspended job is resumed when the print process receives a "resume job" message from the supervisor. If the error was caused by the job data (Guardian file-system CONTROL and SETMODE operations), the error could be repeated several times.

Note. A print process can elect to perform its own error checking; for example, it could retry an operation that returned "device not ready" several times on its own before informing the supervisor.

PRINTREAD Errors

Most errors returned by PRINTREAD cause an abnormal job termination when passed to PRINTSTATUS. Error %12001, however, simply indicates the end of a copy.

Errors %12000 and %12002 cause the spooler supervisor to delete the job. For example, a print process calls PRINTREAD to get the next line of data for a job, and PRINTREAD returns error %12002, invalid data file. The print process terminates the job by closing the data file and permanently suspending activity on the job. In a call to PRINTSTATUS, the print process passes *msg-type* 2 (end of job) to the supervisor, along with the error (%12002) and the name of the device on which the job was printing. The supervisor logs the abnormal termination and the error number and purges the job. Refer to Appendix C, Spooler-Related Errors, for a list of print procedure error codes.

Errors %12003, %12004, and %12005 signify that the caller should call the appropriate file-system operation (CONTROL, SETMODE, or CONTROLBUF).

All other PRINTREAD errors put the job on hold.

Combining Data Retrieval With Spooler Communication

A print process must perform the tasks of retrieving and printing jobs and responding to supervisor messages concurrently. This can be accomplished in any manner that you find convenient. The only limitation is that a print process must respond to a supervisor message within 10 minutes. However, it is important that the print process respond as quickly as possible because the supervisor cannot do any useful work until it gets a response.

To maximize print process responsiveness to Spoolcom commands, however, the file to the supervisor should be checked for completion following the writing of every line of data. HP recommends that you use the following procedure:

1. Open each device with nowait I/O. After each write to a device, monitor completion on any file with AWAITIO[X].
2. If a device has finished, the print process calls PRINTREAD to get the next line of data for the job printing on that device and starts an I/O operation on the device.
3. If the file to the supervisor finishes, the print process calls PRINTCOMPLETE and PRINTREADCOMMAND to obtain the message and executes the supervisor instruction.

After executing the supervisor instruction, the print process loops back to AWAITIO[X].

Note. As long as a job is being printed, there is always an operation on the device outstanding after a supervisor instruction has been executed.

Debugging Print Processes

To debug a print process using the interactive debugging facility Debug (described in the *Debug Manual)*, the subcommand DEBUG is specified at the time the print process is initialized with the Spoolcom PRINT command:

```
)PRINT $trial, FILE $yrvol.yrsub.pproc, PRI 145, DEBUG
```

The DEBUG subcommand causes the print process to run in Debug mode, which has two effects:

- The print process immediately enters the Debug state upon being started by the supervisor. The Debug facility prompts the terminal where the supervisor was run.
- The supervisor does not time out the print process if the print process fails to respond to a message within 10 minutes.

To interactively debug a print process, follow this procedure:

1. Coldstart or warmstart a spooler, initializing the print process with the DEBUG subcommand.

   Note. Start a separate spooler dedicated solely to debugging your print process. Because the supervisor waits indefinitely for message responses from a print process in Debug mode, the spooler halts when a print process is being debugged.

   ---

2. Declare a print device, specify its print process to be the one you are debugging, and connect the device to a location.
3. Run a simple job to your debugging spooler—for example, try a TACL FILES or STATUS command, specifying your collector as the OUT file.
4. When the command interpreter prompt (for example, `TACL 17>`) comes back, indicating that the spooler has accepted your job, enter PAUSE.

The Debug prompt should appear at your terminal, and you can now debug your print process.

To see how to debug spooler with another debugging program such as Garth or a similar HP product, see the appropriate documentation.

Writing a Perusal Process

The preceding subsection explains how a print process communicates with the spooler supervisor to access spooled data. This subsection describes how a process can access spooled data without communicating with the supervisor. This kind of process is called a perusal process, an example of which appears in Appendix 3, Using the Spooler Print Procedures, Print Processes, and Perusal Processes. Peruse is the perusal process supported by HP.

For a perusal process to get the necessary control information that a print process obtains from a spooler job-start message, it calls the spooler utility procedure SPOOLEREQUEST, which provides it with a message identical to the message that the supervisor sends when starting a job. This message allows a perusal process to use the PRINTREADCOMMAND, PRINTSTART, PRINTSTART2, PRINTREAD, and PRINTINFO procedures as if it were a print process. However, a perusal process cannot use the PRINTCOMPLETE, PRINTCOMPLETE2, PRINTINFO, PRINTINIT, PRINTINIT2, PRINTSTATUS, and PRINTSTATUS2 procedures.

The key difference between a print process and a perusal process is that the print process operates in conjunction with and under the control of the supervisor, while a perusal process operates on its own. This independence of a perusal process from the supervisor makes perusal programs easy to code, because the perusal process need not monitor and respond to the supervisor.

Nevertheless, a perusal process should monitor its own messages to the supervisor in case the supervisor goes down (resulting in an error message in response to a message to it) and prevents the perusal process from running successfully.

A perusal program has no way of preventing a job from being deleted from it. If this should happen, PRINTREAD returns %12002, invalid data file. When a perusal process encounters this condition, it can no longer read data from the job.

You can use a perusal process for such things as scanning a job (such as a compiler listing) in order to decide whether the job should be printed. You can also display data on a special terminal in a customized way. You can write an interactive job scanner that allows a user to select a job for scanning and displays pages of the job on the

home terminal. Commands would allow you to display a specific page, skip ahead or back pages, and so forth. You can implement the “display page” command with the *pagenum* parameter of PRINTREAD; you can implement skipping relative to the present page by calling PRINTINFO to determine the present page number and computing the desired new page.

You could also use such a job scanner in conjunction with the SPOOLERCOMMAND utility procedure, giving the job scanner the ability to delete a job (for example, if the compilation contained too many errors) or to change the location of a job.

Outline of the Basic Perusal Process

The following steps summarize the perusal process:

1. Open a file to the spooler supervisor:

```
CALL OPEN( cntrlr^name, cntrlr^fnum );   ! must be waited
```

2. Pass to SPOOLEREQUEST the file number of the supervisor (obtained in Step 1) and the job number of the job to be accessed. SPOOLEREQUEST returns a message whose format is identical to the format of the message that the supervisor sends to a print process to start a new job:

```
err := SPOOLEREQUEST( cntrlr^fnum, job^num, msg, );
IF err THEN ...                          ! error occurred
ELSE ...                                 ! successful
```

3. Call PRINTREADCOMMAND to obtain the name of the data file in which the job is located. If desired, you can also obtain job attributes. Open the data file:

```
CALL PRINTREADCOMMAND( msg,,,,,,, data^file,, location );
CALL OPEN( data^file, data^file^fnum );
```

4. Call PRINTSTART, passing a 560-word *job-buffer*:

```
err := PRINTSTART( job^buf, msg, data^filenum );
IF err THEN ...              ! PRINTSTART error
ELSE ...                     ! OK to begin reading the job
```

5. The job can now be accessed with a succession of calls to PRINTREAD:

```
err := PRINTREAD( job^buf, data^line, read^count );
IF err THEN ...              ! PRINTREAD error
ELSE ...                     ! data^line contains the
                               ! next line of spooled data
```

Note. Some PRINTREAD messages occur normally, such as end of file, end of copy, CONTROL, and SETMODE.

At this point, the perusal process performs an operation with data^line, such as writing it to a terminal. The next call to PRINTREAD returns the next line of the job, and so on, until end of file.

4 Spooler Procedure Calls

There are two types of spooler procedures: print procedures, whose names start with PRINT, and spooler procedures, whose names start with SPOOL. These procedures are summarized in Table 4-1 and their use is described in the remainder of this section.

Table 4-1. Summary of Spooler and Print Procedures (page 1 of 2)

Procedure	Description
PRINTCOMPLETE	Obtains a message from the spooler.
PRINTCOMPLETE[2]	Obtains a message from the spooler. This procedure includes batch enhancements to PRINTCOMPLETE.
PRINTINFO	Obtains information regarding a job being printed by the print process.
PRINTINIT	Initializes the print control buffer.
PRINTINIT2	Initializes the print control buffer. This procedure includes batch enhancements to PRINTINIT.
PRINTREAD	Obtains a line of spooled data.
PRINTREADCOMMAND	Interprets a message from the spooler supervisor.
PRINTSTART	Initializes a job buffer for a new job.
PRINTSTART2	Initializes a job buffer for a new job. This procedure includes batch enhancements to PRINTSTART.
PRINTSTATUS	Sends a message to the supervisor.
PRINTSTATUS2	Sends a message to the supervisor. This procedure includes batch enhancements to PRINTSTATUS.
SPOOLBATCHNAME	Returns the name of the spooler batch job currently being spooled to the collector.
SPOOLCONTROL	Replaces the Guardian file-system CONTROL procedure when spooling at level 3.
SPOOLCONTROLBUF	Replaces the Guardian file-system CONTROLBUF procedure when spooling at level 3.
SPOOLEND	Writes any remaining blocked data to the spooler and signals end of job; can be used to modify the job attributes.
SPOOLERCOMMAND	Issues a Spoolcom command to the supervisor.
SPOOLEREQUEST	Obtains a Startup message from the supervisor suitable for reading a job.
SPOOLEREQUEST2	Obtains a Startup message from the supervisor suitable for reading a job. Includes batch enhancements to SPOOLEREQUEST.
SPOOLERSTATUS	Obtains status of spooler components.
SPOOLERSTATUS2	Obtains status of spooler components. Includes batch enhancements to SPOOLERSTATUS.

Table 4-1. Summary of Spooler and Print Procedures (page 2 of 2)

Procedure	Description
SPOOLJOBNUM	Returns the job number of the job currently being spooled to the collector.
SPOOLSETMODE	Replaces the Guardian file-system SETMODE procedure when spooling at level 3.
SPOOLSTART	Specifies job attributes and optionally initializes a level-3 buffer.
SPOOLWRITE	Compresses, blocks, and sends data to the spooler.

PRINTCOMPLETE[2] Procedure

The PRINTCOMPLETE[2] procedure is used by a print process to communicate with the spooler supervisor.

The PRINTCOMPLETE procedure obtains a message from the supervisor.

PRINTCOMPLETE[2] is the same procedure with a larger buffer.

```
error-code := PRINTCOMPLETE[2] ( filenum-of-supervisor    ! i
                                ,print-control-buffer );  ! o
```

error-code returned value

INT

returns the following spooler error codes:

%3000-%3377	Supervisor file error (<8:15> contains a file error). This error indicates a communication problem with the supervisor. A print process receiving this error can call ABEND, retry the operation a number of times, or continue reading and printing jobs without any further communication with the supervisor.
%14015	The process is not a spooler supervisor.

filenum-of-supervisor input

INT:value

is the file number of an open supervisor file. The file number is returned when the supervisor is opened.

print-control-buffer output

INT:ref:64 (Use with PRINTCOMPLETE)

INT:ref:128 (Use with PRINTCOMPLETE[2])

on return, contains a message from the supervisor.

Considerations

The following considerations apply to the use of the PRINTCOMPLETE[2] procedure:

- PRINTCOMPLETE[2] should not be used by a perusal process. The print process operates with, and under the control of, the supervisor, while a perusal process operates on its own.
- The message returned by PRINTCOMPLETE[2] is interpreted through a call to PRINTREADCOMMAND.

- PRINTCOMPLETE[2] must be called immediately following the completion of a call on the file to the supervisor.
- In addition to obtaining the supervisor message, PRINTCOMPLETE[2] also initiates a nowait operation to the supervisor file. Thus, a call to AWAITIO[X] must be issued to the supervisor file at some time after a call to PRINTCOMPLETE[2].
- You can use PRINTCOMPLETE to access jobs that reside in the D41 and later releases of the spooler only if they are in the form of file code 129 job files; otherwise you must use PRINTCOMPLETE2.

Example

```
PRINT^ERROR := PRINTCOMPLETE ( FILENUM^SUP , PRINT^BUFF );
```

PRINTINFO Procedure

The PRINTINFO procedure is used in print processes to communicate with the spooler supervisor.

The PRINTINFO procedure returns information regarding a job to the supervisor in response to a “send status” request. This includes spooler job files.

```
error-code := PRINTINFO ( job-buffer                  ! i
                        ,[ copies-remaining ]         ! o
                        ,[ current-page ]             ! o
                        ,[ current-line ]             ! o
                          ,[ lines-printed ] );         ! o
```

error-code returned value

INT

returns the following spooler error code:

%1000 1	Parameter present, but its content is wrong.

job-buffer input

INT:ref:560

contains control information for the job started. PRINTINFO interprets the contents of *job-buffer*.

copies-remaining output

INT:ref:1

returns the number of copies of the job that are left to print, including the current copy.

current-page output

INT:ref:1

returns the page number of the current page.

current-line output

INT:ref:1

returns the current line of the current page being printed.

lines-printed output

INT:ref:1

returns the total number of lines printed for this copy of the job.

Considerations

The following considerations apply to the use of the PRINTINFO procedure:

- PRINTINFO should not be used by a perusal process. A print process operates with, and under the control of, the supervisor, while a perusal process operates on its own.
- PRINTINFO is used by a print process (the PRINTSTART or PRINTREAD procedure) to respond to a status request from the supervisor.
- The *lines-printed* parameter is not always an indication of how many lines remain to be printed on a job, because it includes lines that are printed more than once as a result of a page-skip action.

PRINTINIT[2] Procedure

The PRINTINIT[2] procedure is used in print processes to initialize communication with the spooler supervisor.

The PRINTINIT procedure initializes the print process's print control buffer, which is used in calls to other print procedures.

PRINTINIT[2] is the same procedure with a larger buffer.

```
error-code := PRINTINIT[2] ( filenum-of-supervisor     ! i
                           ,print-control-buffer ); ! i,o
```

error-code returned value

INT

returns one of the following spooler error codes:

%2000-%2377	File error on data file (bits <8:15> contain a Guardian file-system error number).
%3000-%3377	Supervisor file error (<8:15> contains a file error). This error indicates a communication problem with the supervisor. A print process receiving this error can call ABEND, retry the operation a number of times, or continue reading and printing jobs without any further communication with the supervisor.
%4000-%4377	Device error sent to the supervisor by the print process (bits <8:15> contain a file-system error number).
%10000	Missing parameter.
%10001	Parameter present, but its content is wrong.

filenum-of-supervisor input

INT:value

is the file number of an open supervisor file. The file number is returned when the supervisor is opened. This file must be opened nowait.

print-control-buffer input, output

INT:ref:64 (Use with PRINTINIT)

INT:ref:128 (Use with PRINTINIT2)

is formatted by PRINTINIT[2] and should be passed unaltered to other print procedures.

Considerations

The following considerations apply to the use of the PRINTINIT[2] procedure:

- PRINTINIT[2] should not be used by a perusal process. A print process operates with, and under the control of, the supervisor, while a perusal process operates on its own.
- Before calling PRINTINIT[2], a print process must have a file open to the supervisor with nowait I/O and a sync depth of, at most, 1.
- PRINTINIT[2] must be followed at some point by a call to AWAITIO[X].
- The *print-control-buffer* returned by PRINTINIT[2] is used by the supervisor to send messages to the print process. The print process should never alter this buffer except with calls to the print procedures.
- Usually, PRINTINIT[2] is called only once by a print process.

PRINTREAD Procedure

The PRINTREAD procedure can be used in print and perusal processes to access spooled data and to allow print processes to communicate with the spooler supervisor. This includes spooler data stored in a spooler job file.

The PRINTREAD procedure returns one line of spooled data.

```
error-code := PRINTREAD ( job-buffer              ! i,o
                        ,data-line                ! o
                        ,read-count               ! i
                        ,[ count-read ]           ! o
                        ,[ pagenum ] );           ! i
```

error-code returned value

INT

returns a spooler error code. Certain nonzero *error-code*s from PRINTREAD have special significance:

%12000	End of file. All lines in the job have been transferred (send an “end job” message to the supervisor by PRINTSTATUS; this error is returned only for print processes—not for perusal processes).
%12001	End of copy.
%12002	Invalid data file.
%12003	CONTROL found.
%12004	SETMODE found.
%12005	CONTROLBUF found.

See Appendix C, Spooler-Related Errors, for a list of spooler errors and their meanings.

job-buffer input, output

INT:ref:560

is the job buffer for the job being read.

data-line output

INT:ref:450 (or less)

returns a line of spooled data.

read-count input

INT:value

specifies the maximum number of bytes to be read.

count-read output

INT:ref:1

is the number of bytes actually read.

pagenum input

INT:value

returns one of the following:

> 0	PRINTREAD returns the first line on the page specified by this parameter.
< 0	PRINTREAD repeats the last line returned.
= 0 or absent	PRINTREAD returns the next sequential line.

Considerations

The following considerations apply to the use of the PRINTREAD procedure:

- The size of *data-line* never exceeds 450 words; however, in most cases, it is smaller.
- Errors returned from PRINTREAD other than %12000-%12001 and %12003-%12005 are critical. In the case of a print process (not for perusal processes), the error should be sent to the supervisor using PRINTSTATUS.
- When the *error-code* returns %12003 = CONTROL found, the data line contains a file-system CONTROL message for the print device. The format of the CONTROL message is

  data-line [0] = operation
  data-line [1] = parameter

  See the *Guardian Procedure Calls Reference Manual* for a description of the CONTROL operations.

- When the *error-code* returns %12004 = SETMODE found, the data line contains a file-system SETMODE instruction. The SETMODE instruction format is

  data-line [0] = SETMODE function
  data-line [1] = *param-1*
  data-line [2] = *param-2*

See the *Guardian Procedure Calls Reference Manual* for a description of the SETMODE functions.

Example

```
READ^ERROR := PRINTREAD ( JOB^BUFF , LINE , COUNT ,
  COUNT^READ );
```

PRINTREADCOMMAND Procedure

The PRINTREADCOMMAND procedure can be used in print and perusal processes to access spooled data and to allow print processes to communicate with the spooler supervisor.

The PRINTREADCOMMAND procedure interprets the information contained in the print control buffer returned from a call to PRINTCOMPLETE (in a print process) or SPOOLEREQUEST[2] (in a perusal process).

```
error-code := PRINTREADCOMMAND  ( print-control-buffer   ! i
                                 ,[ controlnum ]         ! o
                                 ,[ device ]             ! o
                                 ,[ devflags ]           ! o
                                 ,[ devparam ]           ! o
                                 ,[ devwidth ]           ! o
                                 ,[ skipnum ]            ! o
                                 ,[ data-file ]          ! o
                                 ,[ jobnum ]             ! o
                                 ,[ location ]           ! o
                                 ,[ form-name ]          ! o
                                 ,[ report-name ]        ! o
                                 ,[ pagesize ]           ! o
                                 ,[ batchname ]          ! o
                                 ,[ batchid ]            ! o
                                 ,[ owner ]              ! o
                                 ,[ charmap ]            ! o
                                 ,[ devflagx ] );        ! o
```

error-code returned value

INT

returns one of the following spooler error codes:

%1000 0 Missing parameter.

%1000 1 Parameter is present, but its content is wrong.

print-control-buffer input

INT:ref:64

is passed exactly as it is returned from PRINTCOMPLETE[2] or SPOOLEREQUEST[2].

controlnum output

INT:value

specifies the action requested by the supervisor:

```
0 = Open device
1 = Close device
2 = Start job on device
3 = Stop job on device
4 = Resume job on device
5 = Suspend job on device
6 = Print form-alignment template on device
7 = Skip to page
8 = Skip over pages
9 = Send status of job printing on device
```

device output

INT:ref:12

specifies the particular device referred to by the control number.

devflags output

INT:ref:1

indicates the state of the device's truncation and header flags.
Where applicable: 1 = on, 0 = off

<0:3> Device type

<4:5> Startff on/off

<6> Specifies job is a font job

<7> NetBatch job initial form feed (TOF)

<8> Job is dependent on downloaded font

<9> Batch header

<10> Truncation flag

<11> Device reset on/off

<12> Reserved (set to 0)

<13> Header flag

<14> Exclusive on/off

<15> Endff on/off

devparam output

INT:ref:1

is the parameter specified in the Spoolcom DEV PARM command.

devwidth output

INT:ref:1

is the device width specified in the Spoolcom DEV WIDTH command.

skipnum output

INT:ref:1

returns the number of pages to skip. The meaning of this number depends on the control number:

controlnum = 7	(skip to page). *skipnum* specifies the page that should be skipped to.
controlnum = 8	(skip over pages). *skipnum* specifies the number of pages relative to the current page that should be skipped.

controlnum is neither 7 nor 8. The *skipnum* parameter has no meaning.

data-file output

INT:ref:12

is the data file in which the job is stored if the control number is 2 (start job); otherwise, this parameter has no meaning.

jobnum output

INT:ref:1

is the number of the job referred to if the control number is 2 (start job); otherwise, this parameter has no meaning.

location output

INT:ref:8

is the location of the job started if the control number is 2 (start job); otherwise, this parameter has no meaning.

form-name output

INT:ref:8

is the form name of the job being referred to if the control number is 2 (start job); otherwise, this parameter has no meaning.

report-name output

INT:ref:8

is the report name of the job being referred to if the control number is 2 (start job); otherwise, this parameter has no meaning.

pagesize output

INT:ref:1

is the page size of the job being referred to if the control number is 2 (start job); otherwise, this parameter has no meaning.

batchname output

INT:ref:16

is the name of the batch job.

batchid output

INT:ref:1

is the batch number.

owner output

INT:ref:1

is the owner of the job.

charmap output

INT:ref:1

returns one of the following codes indicating which character sets are supported:

-1 Device does not support MBCS characters.

2 Device supports IBMKANJIKANA characters.

5 Device supports JEFKANJIKANA characters.

8 Device supports JISKANJIKANA characters.

devflagx output

INT:ref:1

When *devflagx*.<1> = 0, the device is not in pretranslate mode.

When *devflagx*.<1> = 1, the device is in pretranslate mode.

All other bits in *devflagx* are reserved for use by the spooler.

Considerations

The following considerations apply to the use of the PRINTREADCOMMAND procedure:

- If desired, PRINTREADCOMMAND can be called once to get the *controlnum* and then a second time to get whatever particular information is needed.
- The print process can use the *location*, *form-name*, *report-name*, and *devflags* values to print out a header message.

- If the control number is 7, the *skipnum* parameter can pass directly to PRINTREAD.
- If the control number is 8, PRINTINFO must be called to get the *current-page*. *skipnum* must then be added to the *current-page* to find the page number to pass to PRINTREAD.
- A print process can ignore the header and truncation flags and the *devwidth* parameter.

Example

```
READ^ERROR:= PRINTREADCOMMAND ( PRINT^BUFFER  ! print buffer
                              , CNTRL^NUM    ! control number
                              ,              ! device
                              ,              ! device flags
                              ,              ! device parameter
                              ,              ! device width
                              ,              ! pages to skip
                                , DATA^FILE );  ! data file
```

PRINTSTART[2] Procedure

The PRINTSTART procedure formats the job buffer for a spooler job being started. The buffer is used in subsequent calls to PRINTREAD.

PRINTSTART[2] is the same procedure with a larger buffer.

```
error-code := PRINTSTART[2] ( job-buffer                ! o
                            ,print-control-buffer       ! i
                            ,data-filenum );            ! i
```

error-code returned value

INT

returns one of the following spooler error codes:

0	Successful operation.
%2000-%2377	File error on data file (bits <8:15> contain a file-system error number).
%3000-%3377	Supervisor file error (<8:15> contains a file error). This error indicates a communication problem with the supervisor. A print process receiving this error can call ABEND, retry the operation a number of times, or continue reading and printing jobs without any further communication with the supervisor.
%4000-%4377	Device error sent to the supervisor by the print process (bits <8:15> contain a file-system error number).
%10001	Parameter is present, but its content is wrong.

job-buffer output

INT:ref:560

contains control information for the job being started in a form suitable for passing to other print procedures.

print-control-buffer input

INT:ref:64 (Use with PRINTSTART)

INT:ref:128 (Use with PRINTSTART2)

is the buffer obtained from PRINTCOMPLETE[2] or SPOOLEREQUEST[2] for spooler jobs. For spooler job files, *print-control-buffer* must be passed but previous buffer contents are ignored.

data-filenum input

INT:value

is the file number of the data file containing the started job.

Considerations

The following considerations apply to the use of the PRINTSTART[2] procedure:

- In addition to containing control information for the job, the PRINTREAD procedure uses *job-buffer* to store a block of spooled data.
- PRINTSTART[2] is called once for each job started on a device.
- The job buffer should not be altered by the print or perusal process.
- A spooler job file can be initialized for reading by passing its file number as *data-filenum*. In this case, neither PRINTCOMPLETE[2] nor SPOOLEREQUEST[2] needs to be called prior to calling PRINTSTART[2] to initialize *print-control-buffer*.
- You can use PRINTSTART to access jobs that reside in the D41 and later releases of the spooler only if they are in the form of file code 129 job files; otherwise you must use PRINTSTART2. If the files do not have a file code of 129, you will receive file system error number %14015.

PRINTSTATUS[2] Procedure

The PRINTSTATUS procedure can be used in print processes to communicate with the supervisor and to send an unsolicited status message to the spooler supervisor.

PRINTSTATUS[2] is the same procedure with a larger buffer.

```
error-code := PRINTSTATUS[2] ( filenum-of-supervisor     ! i
                              ,print-control-buffer      ! i
                              ,msg-type                  ! i
                              ,device                    ! i
                              ,[ error ]                 ! i
                              ,[ num-copies ]            ! i
                              ,[ page ]                  ! i
                              ,[ line ]                  ! i
                              ,[ lines-printed ] );      ! i
```

error-code returned value

INT

returns one of the following spooler error codes:

%2000-%2377	File error on data file (bits <8:15> contain a file-system error number).
%3000-%3377	Supervisor file error (bits <8:15> contain a file-system error number). This error indicates a communication problem with the supervisor. A print process receiving this error can call ABEND, retry the operation a number of times, or continue reading and printing jobs without any further communication with the supervisor.
%4000-%4377	Device error sent to the supervisor by the print process (bits <8:15> contain a file-system error number).

filenum-of-supervisor input

INT:value

is the file number of an open supervisor file. The file number is returned when the supervisor is opened.

print-control-buffer input

INT:ref:64 (Use with PRINTSTATUS)

INT:ref:128 (Use with PRINTSTATUS2)

is the buffer obtained from the PRINTCOMPLETE[2] procedure.

msg-type input

INT:value

specifies the type of message being sent, as follows:

```
0 = Sending status of job
1 = Error occurred on print device; previous operation was
unsuccessful
2 = End of job
3 = Unable to open device
4 = Invalid operation in this state
5 = Error occurred on print device; previous operation was
successful
```

device input

INT:ref:12

is the name of the device on which an error occurred.

error input

INT:value

is the error that caused this call to PRINTSTATUS[2]. It is sent to the supervisor. The list of errors follows.

%4000-%4377	Device error sent to the supervisor by the print process (bits <8:15> contain a Guardian file-system error number).
%13000	No such device.
%13001	Device already open.
%13002	No job on device.
%13003	Job is running.
%13004	TABLE IS FULL is sent by a print process to the supervisor when the print process is already handling as many jobs as it can, and the supervisor instructs it to start another job.

Refer to Appendix C, Spooler-Related Errors, for a complete list and description of spooler errors.

num-copies input

INT:value

is the number of copies of the job remaining to be printed.

page input

INT:value

is the current page number.

line input

INT:value

if present, is the current line number (from PRINTINFO).

lines-printed input

INT:value

is the number of lines printed.

Considerations

The following considerations apply to the use of the PRINTSTATUS[2] procedure:

- PRINTSTATUS[2] should not be used by a perusal process. A print process operates with, and under the control of, the supervisor, while a perusal process operates on its own.
- The file number to supervisor, print control buffer, message type, and device parameters are required parameters and must always be present in a call to PRINTSTATUS[2]. The remaining parameters are optional; PRINTSTATUS[2] might need these parameters, however, depending on the message type.

Table 4-2 shows which parameters are needed for each message type.

Table 4-2. PRINTSTATUS[2] Message Type and Parameters

Message Type	Error	Number of Copies	Page	Lines Printed
0		X	X	X
1	X			
2				
3				
4	X			
5	X			

- PRINTSTATUS[2] is a nowait operation and must be completed with a call to AWAITIO[X].
- Message types 1 and 5 inform the supervisor of an error occurring on a print device.
- Message type 5 is sent if the previous operation on the device was successful. When it receives the message, the supervisor can instruct the print process to retry the operation. If the operation fails again, the print process sends message type 1, which indicates to the supervisor that a retry of an operation failed.

  Message type 5 causes the supervisor to reset its retry count for that device.

Example

```
STATUS^ERROR := PRINTSTATUS ( FILENUM^SUP
                            , PRINT^BUFF
                            , MSG
                            , DEVICE
                            ,                ! error
                            ,                ! num copies
                            , PAGE
                            ,                ! line
                            , NUM^LINES );
```

SPOOLBATCHNAME Procedure

The SPOOLBATCHNAME procedure returns the name of the batch job currently being spooled to the collector. This procedure can be used when spooling at levels 1, 2, or 3.

```
error-code := SPOOLBATCHNAME ( filenum-of-collector       ! i
                              ,batchname );               ! o
```

error-code returned value

INT

returns one of the following spooler error codes:

0	Successful operation
%1000-%1377	Error on file to collector (bits <8:15> contain a file-system error number
%10000	Missing parameter
%11001	Attempted to write to collector without opening the file first

filenum-of-collector input

INT:value

is the file number of an open supervisor file. The file number is returned when the supervisor is opened.

batchname output

INT:ref:16

is the name of the batch job currently being spooled to the collector through the specified file number. If the spooler job does not belong to a batch job, blanks are returned.

Considerations

The following considerations apply to the use of the SPOOLBATCHNAME procedure:

- A call to SPOOLBATCHNAME can be issued by an application spooling at any level.
- When spooling at level 1, a job is not created until after the WRITE[X], SETMODE, or CONTROL procedure is called once.
- When spooling at level 2 or 3, a job is not created until after the SPOOLSTART procedure is called.

SPOOLCONTROL Procedure

The SPOOLCONTROL procedure is used to perform device-dependent I/O operations when the application process is spooling at level 3.

If a level-3 buffer is specified in a call to SPOOLSTART, the SPOOLCONTROL procedure must be used in place of the CONTROL procedure.

```
error-code := SPOOLCONTROL ( level-3-buff                  ! i,o
                           ,operation                      ! i
                           ,param                          ! i
                           ,[ bytes-written-to-buff ] )    ! o
                           ,[ extended-level-3-buff ] ); ! i,o
```

error-code returned value

INT

returns one of the following spooler error codes:

0	Successful operation
%1000-%1377	Error on file to collector (bits <8:15> contain a file-system error number; see Considerations on page 4-25)
%10000	Missing parameter
%10001	Parameter is present, but its content is wrong
%11000	Checkpoint exit
%11001	Attempted to write to the collector without first opening the file

level-3-buff input, output

INT:ref:512

is the *level-3-buff* specified in the SPOOLSTART procedure.

operation input

INT:value

is a CONTROL operation value (see the CONTROL procedure in the *Guardian Procedure Calls Reference Manual* for information about CONTROL operations).

param input

INT:value

is a *parameter* for the specified CONTROL operation (see the *Guardian Procedure Calls Reference Manual* for more information on CONTROL operations).

bytes-written-to-buff output

INT:ref:1

returns the number of bytes to be checkpointed from the *level-3-buff*. This parameter is used by fault-tolerant applications.

extended-level-3-buff input,output

INT:.EXT.ref.*

is the *extended-level-3-buff* specified in the SPOOLSTART procedure.

Considerations

The following considerations apply to the use of the SPOOLCONTROL procedure:

- If *flags*.<11> of SPOOLSTART is set to 1, a return of %11000 from SPOOLCONTROL indicates that the *level-3-buff* is about to be written to the collector. The buffer should be checkpointed, and SPOOLCONTROL should be called again.
- Some file-system errors have special significance to a process sending data to a collector; these errors are described in the *Guardian Procedure Errors and Messages Manual*.

  A program using level-1 or level-2 spooling gets these errors from the WRITE[X], OPEN, or FILE_OPEN_ procedure while a program spooling at level 3 obtains these errors in bits <8:15> of a spooler error code in the %1000 range.

SPOOLCONTROLBUF Procedure

The SPOOLCONTROLBUF procedure is used to perform device-dependent I/O operations requiring a data buffer when the application process is spooling at level 3.

This procedure must be used in place of CONTROLBUF if a level-3 buffer is specified in a call to SPOOLSTART.

```
error-code := SPOOLCONTROLBUF ( level-3-buff                ! i,o
                               ,operation                    ! i
                               ,buffer                       ! i
                               ,count                        ! i
                               ,[bytes-written-to-buff]      ! o
                               ,[extended-level-3-buff] );   ! i,o
```

error-code returned value

INT

returns one of the following spooler error codes:

0	Successful operation
%1000-%1377	Error on file to collector (bits <8:15> contain a file-system error number; see Considerations on page 4-27)
%10000	Missing parameter
%10001	Parameter is present, but its content is wrong
%11000	Checkpoint exit
%11001	Attempted to write to the collector without first opening the file

level-3-buff input, output

INT:ref:512

is the *level-3-buff* specified in the SPOOLSTART procedure.

operation input

INT:value

is a SPOOLCONTROLBUF operation as follows:

operation	**Definition**
1	Load DAVFU (printer subtype 4) *buffer* = VFU buffer to be loaded *count* = number of bytes contained in *buffer*

buffer input

INT:ref:*

is an array containing the control information to be sent to the print device.

count input

INT:value

is the number of bytes of information contained in the buffer.

bytes-written-to-buff output

INT:ref:1

returns the number of bytes to be checkpointed from the *level-3-buff*. This parameter is used by fault-tolerant applications.

extended-level-3-buff input,output

INT:.EXT.ref.*

is the *extended-level-3-buff* specified in the SPOOLSTART procedure.

Considerations

The following considerations apply to the use of the SPOOLCONTROLBUF procedure:

- If *flags*.<11> of SPOOLSTART is set to 1, a return of %11000 from SPOOLCONTROLBUF indicates that the *level-3-buff* is about to be written to the collector. The buffer should be checkpointed, and SPOOLCONTROLBUF should be called again.
- Some file-system errors have special significance to a process sending data to a collector; these errors are described in the *Guardian Procedure Errors and Messages Manual*.

  A program using level-1 or level-2 spooling gets these errors from the WRITE[X] or OPEN procedure, while a program spooling at level 3 obtains these errors in bits <8:15> of a spooler error code in the %1000 range.

Example

```
ERROR := SPOOLCONTROLBUF (COLL^BUFF , 1 , CON^BUFF , COUNT );
```

SPOOLEND Procedure

The SPOOLEND procedure can be used to complete a job being spooled at level 3.

The SPOOLEND procedure writes any remaining data in the collection process buffer to the collector, sends the collection process a termination message, and optionally modifies the job's attributes.

```
error-code := SPOOLEND ( level-3-buff                    ! i,o
                       ,[ flags ] );                     ! i
                       ,[ extended-level-3-buff ] );     ! i,o
```

error-code returned value

INT

returns one of the following spooler error codes:

0	Successful operation
%1000-%1377	Error on file to collector (bits <8:15> contain a file-system error number; see Considerations on page 4-29)
%10000	Missing parameter
%10001	Parameter is present, but its content is wrong
%11000	Checkpoint exit
%11001	Attempted to write to the collector without first opening the file

level-3-buff input,output

INT:ref:512

is the *level-3 buff* specified in the call to SPOOLSTART.

flags input

INT:value

overrides the flags specified in SPOOLSTART. If bit <8> is set to 1, the job is canceled rather than printed.

<0:7>	Reserved for use by the collector	
<8>	Cancel job flag:	0 = off
		1 = on
<9>	HOLD flag:	0 = off
		1 = on

<10> HOLDAFTER flag: 0 = off

1 = on

<11> Reserved; must be 0

<12> Close the file

<13:15> Job priority

If this parameter is omitted, the attributes established by the call to SPOOLSTART are not changed.

extended-level-3-buff input,output

INT:.EXT.ref.*

is the *extended-level-3-buff* specified in the SPOOLSTART procedure.

Considerations

The following considerations apply to the use of the SPOOLEND procedure:

- A call to SPOOLEND causes any existing data in the collection process buffer to be written to the collector.
- Following a call to SPOOLEND, a new job can be started without reopening a file to the collector.
- SPOOLEND must be called to terminate the spooling of any job being spooled at level 3 (using the spooler interface procedures).
- Calling SPOOLEND with the *flags* cancel job flag bit (bit 8) set to 1 has the same effect as if you never started the job.
- Including *flags* causes all bit fields to override the values specified in the *flags* parameter of SPOOLSTART.
- Some file-system errors have special significance to a process sending data to a collector; these errors are described in the *Guardian Procedure Errors and Messages Manual*.

  A program using level-1 or level-2 spooling gets these errors from the WRITE[X] or OPEN procedure, while a program spooling at level 3 obtains these errors in bits <8:15> of a spooler error code in the %1000 range.

Example

```
SPERRNUM := SPOOLEND ( BUFFER , %B0000000001000100 );
   ! put job on HOLD and set priority 4.
```

SPOOLERCOMMAND Procedure

The SPOOLERCOMMAND procedure is used to perform Spoolcom and Peruse operations from within applications.

The SPOOLERCOMMAND procedure allows a process to send a Spoolcom command to the spooler supervisor.

```
error-code := SPOOLERCOMMAND ( filenum-of-supervisor       ! i
                             ,command-code                 ! i
                             ,[ command-parameter ]        ! i
                             ,subcommand-code              ! i
                             ,[ subcommand-parameter ] );  ! i
```

error-code returned value

INT

returns one of the following spooler error codes:

0	Successful operation
%3000-%3377	Error on file to supervisor (file-system error in bits <8:15>); refer to the *Guardian Procedure Errors and Messages Manual*
%10000	Parameter missing
%10001	Parameter in error
%14000	Invalid command
%14001	Command parameter missing
%14002	Command parameter in error
%14003	Invalid subcommand
%14004	Subcommand missing
%14005	Subcommand parameter in error
%14010	Cannot add entry to tables
%14011	Cannot find entry requested
%14012	Entry not in proper state for requested operation
%14013	Entry in use; cannot be deleted
%14014	Security violation
%14015	Process not a spooler supervisor

%14017	A job associated with a font cannot be deleted
%14020	Command not valid on spooler batch job
%14021-%14023	Spooler batch job error code print process (NEWPROCESS error in bits <8:15>, refer to the *Guardian Procedure Errors and Messages Manual*)

filenum-of-supervisor input

INT:value

is the file number of an open supervisor file. The file number is returned when the supervisor is opened.

command-code input

INT:value

is the number of the command to be executed. *command-code* values are described in SPOOLERCOMMAND Procedure and Subcommand Parameters on page 4-32.

command-parameter input

INT:ref:*

is a buffer containing the parameter to be used with the *command-code* being executed. The size and content of this parameter for each command are shown in SPOOLERCOMMAND Procedure and Subcommand Parameters on page 4-32.

subcommand-code input

INT:value

is the number of the subcommand to be executed. *subcommand-code* values are described in SPOOLERCOMMAND Procedure and Subcommand Parameters on page 4-32.

subcommand-parameter input

INT:ref:*

is a buffer containing the parameter to the subcommand being executed. See the following explanation for the size and content of this parameter for each subcommand.

SPOOLERCOMMAND Procedure and Subcommand Parameters

The SPOOLERCOMMAND procedure is useful to programmers who want to perform Spoolcom operations from within application programs. The Spoolcom commands that can be sent are COLLECT, DEV, JOB, LOC, PRINT, and SPOOLER.

The remaining commands—EXIT, FC, HELP, and OPEN—are executed by Spoolcom, and therefore they are not related to the supervisor.

Note. Before calling the SPOOLERCOMMAND procedure, you must open a file to the spooler supervisor. You must specify waited I/O.

To send a Spoolcom command to the spooler supervisor from within an application, you specify its equivalent using SPOOLERCOMMAND procedure commands. The commands are described here in command-code order, starting with 1.

Spoolcom Command	SPOOLERCOMMAND Command Code
DEV	1
JOB	2
LOC	3
COLLECT	4
PRINT	5
SPOOLER	6
FONT	7
BATCH	8

Spoolcom DEV Command Parameters

To send the equivalent of a Spoolcom DEV command, specify a file number for the *filenum-of-supervisor* parameter, a *command-code* of 1, and the *command-parameter*, which in this case is a print device name of up to 32 characters in internal filename format. Find the *subcommand-code* and the *subcommand-parameter* for the Spoolcom DEV parameters you want from Table 4-3.

filenum-of-supervisor: a file number
command-code: 1
command-parameter: a printer name

For example, to specify the speed of printer $LP3, you could enter the following:

```
INT .DEV [0:15] := ["         $LP3                          "];
COM^ERROR := SPOOLERCOMMAND (4, 1, DEV, 101, 300);
```

Refer to the *Spooler Utilities Reference Manual* for a description of the Spoolcom DEV parameters.

Table 4-3. SPOOLERCOMMAND Parameters for Spoolcom DEV (page 1 of 2)

Spoolcom DEV Subcommand	*subcommand-code* (Fourth Parameter)	*subcommand-parameter* (Fifth Parameter)
Blank	100	None
ALIGN	113	None
CHARMAP	133	–1 = does not support MBCS 2 = supports IBMKANJIKANA 5 = supports JEFKANJIKANA 8 = supports JISKANJIKANA
CLEAR	112	1 = DEL (INT:1) 0 = no DEL
DELETE	116	None
DEVRESET	136	0 = off, 1 = on (INT:1)
DEVTYPE	140	Blank (INT:2) LU1 LU3 7 8 9 10
DRAIN	114	None
ENDFF	138	0 = off, 1 = on (INT:1)
EXCLUSIVE	103	0 = off (INT:1) 1 = on 2 = off !
FIFO	104	0 = off, 1 = on (INT:1)
FORM	107	[*form-name*] (INT:8)
HEADER	110	0 = off (INT:1) 1 = on 2 = batch
JOB	117	*job-code* (INT:1)
LUEOLVALUE	156	0 = CRLF (INT:1) 1 = NL
LUEOLWHEN	155	0 = LT132 (INT:1) 1 = ALWAYS 2 = LTWIDTH 3 = NEVER
LUTOFVALUE	154	0 = CRFFCR (INT:1) 1 = FFCR 2 = FF 3 = NEVER
PARM	148	(INT:1)

Table 4-3. SPOOLERCOMMAND Parameters for Spoolcom DEV (page 2 of 2)

Spoolcom DEV Subcommand	*subcommand-code* (Fourth Parameter)	*subcommand-parameter* (Fifth Parameter)	
PREXLATE	134	0 = off, 1 = on	(INT:1)
PROCESS	102	*process-name*	(INT:3)
RESTART	120	*interval*	(INT:1)
RETRY	105	*interval*	(INT:1)
SKIP	108	+ *num-pages* – *num-pages*	(INT:1)
SKIPTO	109	*page-num*	(INT:1)
SPEED	101	*lpm*	(INT:1)
START	115	none	
STARTFF	137	0 = off 1 = on 2 = off !	(INT:1)
SUSPEND	111	none	
TIMEOUT	106	*num-retries*	(INT:1)
TRUNC	118	0 = off, 1 = on	(INT:1)
WIDTH	119	*device-width*	(INT:1)

Spoolcom JOB Command Parameters

To send the equivalent of a Spoolcom JOB command, specify a file number for the *filenum-of-supervisor* parameter, a *command-code* of 2, and the *command-parameter*, which in this case is a job number. Find the *subcommand-code* and the *subcommand-parameter* for the Spoolcom JOB parameters you want from Table 4-4.

filenum-of-supervisor: a file number
command-code: 2
command-parameter: a job number

For example, to specify the priority of job number 412 as 6, enter the following:

```
COM^ERROR := SPOOLERCOMMAND (4, 2, 412, 126, 6);
```

Refer to the *Spooler Utilities Reference Manual* for a description of the Spoolcom JOB parameters.

Table 4-4. SPOOLERCOMMAND Parameters for Spoolcom JOB

Spoolcom JOB Subcommand	*subcommand-code* (Fourth Parameter)	*subcommand-parameter* (Fifth Parameter)	
COPIES	123	*num-copies*	(INT:1)
DELETE	116	None	
FORM	107	[*form-name*]	(STRING:8)
HOLD	122	None	
HOLDAFTER	121	0 = off 1 = on	(INT:1)
LOC	125	[*location-name*]	(INT:8)
OWNER	127	*group* = <0:7> *user* = <8:15>	(INT:1)
REPORT	124	[*report-name*]	(STRING:8)
SELPRI	126	*selection priority*	(INT:1)
START	115	None	
SUMMARY	167	None	

Spoolcom LOC Command Parameters

To send the equivalent of a Spoolcom LOC command, specify a file number for the *filenum-of-supervisor* parameter and a *command-code* of 3. Find the *command-parameter*, *subcommand-code*, and *subcommand-parameter* for the Spoolcom LOC parameters you want from Table 4-5.

filenum-of-supervisor: a file number
command-code: 3

For example, to specify $LP as a default print device, you could enter the following:

```
INT .LOC [0:15] := ["#PRIN   DEFAULT                  "];
INT .DEV [0:15] := ["        $LP                      "];
COM^ERROR := SPOOLERCOMMAND (4, 3, LOC, 131, DEV);
```

This is equivalent to the interactive command:

```
SPOOLCOM LOC #PRIN.DEFAULT, DEV $LP
```

Refer to the *Spooler Utilities Reference Manual* for a description of the Spoolcom LOC parameters.

Table 4-5. SPOOLERCOMMAND Parameters for Spoolcom LOC

Spoolcom LOC Subcommand	*command-parameter* (Third Parameter)		*subcommand-code* (Fourth Parameter)	*subcommand-parameter* (Fifth Parameter)	
Blank	*group dest*	(INT:16)	100	none	
BROADCAST	*group*	(INT:16)	132	0 = off 1 = on	(INT:1)
DELETE	*group* [*dest*]	(INT:16)	116	None	
DEV	[*group*] *dest*	(INT:16)	131	[*device-name*]	(INT:16)
FONT	*group dest*	(INT:16)	149	[*font-name*]	(INT:8)

The *command-parameter* is an INT:16; the first 8 characters contain the group name, the second 8 characters contain the destination name, and the remaining characters are not used. The device name is in internal file-name format. The font name is left-justified and blank-filled.

Spoolcom COLLECT Command Parameters

To send the equivalent of a Spoolcom COLLECT command, specify a file number for the *filenum-of-supervisor* parameter, a *command-code* of 4, and then the *command-parameter*, which in this case is the name of the collection process, not exceeding three words in length. Find the *subcommand-code* and the *subcommand-parameter* for the Spoolcom COLLECT parameters you want from Table 4-6.

filenum-of-supervisor: a file number
command-code: 4
command-parameter: a collection process

For example, to specify the processor that the collector is to run in, you could enter the following:

```
INT .COL [0:15] := ["$S                                    "];
COM^ERROR := SPOOLERCOMMAND (4, 4, COL, 142, 6);
```

Refer to the *Spooler Utilities Reference Manual* for a description of the Spoolcom COLLECT parameters.

Table 4-6. SPOOLERCOMMAND Parameters for Spoolcom COLLECT

Spoolcom COLLECT Subcommand	*subcommand-code* (Fourth Parameter)	*subcommand-parameter* (Fifth Parameter)	
blank	100	None	
BACKUP	143	*backup-cpu*	(INT:1)
CPU	142	*cpu*	(INT:1)
DATA	145	*data-filename*	(STRING:12)
DELETE	116	None	
DRAIN	114	None	
FILE	141	*program-filename*	(STRING:12)
PRI	144	*process-priority*	(INT:1)
START	115	None	
SUMMARY	167	None	
UNIT	146	*unit-size*	(INT:1)

The *data-filename* and the *program-filename* parameters can be specified as either an INT:12 or a STRING:24. The format in either case is

$Volume name in bytes 0-7
Subvolume name in bytes 8-15
File name in bytes 16-23

Spoolcom PRINT Command Parameters

To send the equivalent of a Spoolcom PRINT command, specify a file number for the *filenum-of-supervisor* parameter, a *command-code* of 5, and the *command-parameter*, which in this case is the name of the print process, not exceeding three words in length. Find the *subcommand-code* and the *subcommand-parameter* for the Spoolcom PRINT parameters you want from Table 4-7.

filenum-of-supervisor: a file number
command-code: 5
command-parameter: a print process

For example, to specify the backup processor that the print process is to run in (CPU2), you could enter the following:

```
INT .PP [0:2] := ["$SPLP   "];
COM^ERROR := SPOOLERCOMMAND (4, 5, PP, 143, 2);
```

Refer to the *Spooler Utilities Reference Manual* for a description of the Spoolcom PRINT parameters.

Table 4-7. SPOOLERCOMMAND Parameters for Spoolcom PRINT

Spoolcom PRINT Subcommand	*subcommand-code* (Fourth Parameter)	*subcommand-parameter* (Fifth Parameter)	
blank	100	None	
BACKUP	143	*backup-cpu*	(INT:1)
CPU	142	*cpu*	(INT:1)
DEBUG	147	0 = off, 1 = on	(INT:1)
DELETE	116	None	
FILE	141	*program-filename*	(STRING:12)
PARM	148	*print-process-param*	(INT:1)
PRI	144	*process-priority*	(INT:1)
START	115	None	
SUMMARY	167	None	

The *program-filename* parameter can be specified as either an INT:12 or a STRING:24. The format in either case is

$Volume name in bytes 0-7
Subvolume name in bytes 8-15
File name in bytes 16-23

Spoolcom SPOOLER Command Parameters

To send the equivalent of a Spoolcom SPOOLER command, specify a file number for the *filenum-of-supervisor* parameter and a *command-code* of 6. The *command-parameter* does not exist, but a place-holder comma for it must be supplied. Find the *subcommand-code* and the *subcommand-parameter* for the Spoolcom SPOOLER parameters you want from Table 4-8.

filenum-of-supervisor: a file number
command-code: 6

For example, to issue the DRAIN subcommand to stop the spooler in an orderly manner, you could enter the following:

```
COM^ERROR := SPOOLERCOMMAND (4, 6, , 114, );
```

Refer to the *Spooler Utilities Reference Manual* for a description of the Spoolcom SPOOLER parameters.

Table 4-8. SPOOLERCOMMAND Parameters for Spoolcom SPOOLER

Spoolcom SPOOLER Subcommand	*subcommand-code* (Fourth Parameter)	*subcommand-parameter* (Fifth Parameter)	
DRAIN	114	None	
DUMP	157	*filename*	(INT:12)
ERRLOG	151	*filename*	(INT:12)
MGRACCESS	158	0 = off, 1 = on	(INT:1)
START	115	None	

The *filename* parameter can be specified as either an INT:12 or a STRING:24. The format in either case is

$Volume name in bytes 0-7
Subvolume name in bytes 8-15
File name in bytes 16-23

Spoolcom FONT Command Parameters

To send the equivalent of a Spoolcom FONT command, specify a file number for the *filenum-of-supervisor* parameter and a *command-code* of 7. The *command-parameter* in this case is the font name, not to exceed eight words in length. Find the *subcommand-code* and the *subcommand-parameter* for the Spoolcom FONT parameters you want from Table 4-9.

filenum-of-supervisor: a file number
command-code: 7
command-parameter: a font name

For example, to create a font called PRTPAYCHK and associate font job 1843 with this font, you could enter the following:

```
INT .FONTNAME [0:15] := ["PRTPAYCHK                          "];
COM^ERROR := SPOOLERCOMMAND (4, 7, FONTNAME, 117, 1843);
```

Refer to the *Spooler Utilities Reference Manual* for a description of the Spoolcom FONT parameters.

Table 4-9. SPOOLERCOMMAND Parameters for Spoolcom FONT

Spoolcom FONT Subcommand	*subcommand-code* (Fourth Parameter)	*subcommand-parameter* (Fifth Parameter)	
DELETE	116	None	
JOB	117	*job number*	(INT:1)

Spoolcom BATCH Command Parameters

To send the equivalent of a Spoolcom BATCH command, specify a file number for the *filenum-of-supervisor* parameter and a *command-code* of 8. The *command-parameter* in this case is the batch number, not to exceed one word in length. Find the *subcommand-code* and the *subcommand-parameter* for the Spoolcom BATCH parameters you want from Table 4-10.

filenum-of-supervisor: a file number
command-code: 8
command-parameter: a batch number

For example, to specify that batch 78 requires a special form, you could enter the following:

```
STRING .PAYROLL [0:7] := ["PAYROLL "];
COM^ERROR := SPOOLERCOMMAND (4, 8, 78, 107, PAYROLL);
```

Refer to the *Spooler Utilities Reference Manual* for a description of the Spoolcom BATCH parameters.

Table 4-10. SPOOLERCOMMAND Parameters for Spoolcom BATCH

Spoolcom BATCH Subcommand	*subcommand-code* (Fourth Parameter)	*subcommand-parameter* (Fifth Parameter)	
COPIES	123	*num-copies*	(INT:1)
DELETE	116	None	
FORM	107	[*form-name*]	(STRING:8)
HOLD	122	None	
HOLDAFTER	121	0 = off, 1 = on	(INT:1)
LINK	129	*job number*	(INT:1)
LOC	125	[*loc-name*]	(INT:8)
OWNER	127	*group* = <0:7> *user* = <8:15>	(INT:1)
REPORT	124	[*report-name*]	(STRING:8)
SELPRI	126	*sel-pri*	(INT:1)
START	115	None	
UNLINK	130	*job number*	(INT:1)

Considerations

The following considerations apply to the use of the SPOOLERCOMMAND procedure:

- Note that *subcommand-code* is a required parameter on every call to SPOOLERCOMMAND.

- Commands not accompanied by subcommands should have code 100 as their *subcommand-code* (for example, creating a component with all default parameters).
- You must put a batch on hold by using the Spoolcom JOB HOLD command before attempting to use the Spoolcom BATCH LINK and UNLINK subcommands.
- Error %14017 is returned if an attempt is made to delete a job associated with a font.
- Error %14020 is returned from a Spoolcom JOB command when the command cannot be done on a portion of a spooler batch job.
- When a Spoolcom BATCH command returns an error code in the range %14021 through %14023, it indicates a problem with the LINK or UNLINK operation. For more information on spooler-related errors, see Appendix C, Spooler-Related Errors.

Example

```
COM^ERROR := SPOOLERCOMMAND ( FILENUM
                            , COM^CODE
                            ,            ! command parameter
                              , SUB^CODE );
```

SPOOLEREQUEST[2] Procedure

The SPOOLEREQUEST procedure allows a perusal process to access a spooled job outside the control of the spooler supervisor.

SPOOLEREQUEST2 is the same procedure with a larger buffer.

```
error-code := SPOOLEREQUEST[2] ( supervisor-filenum       ! i
                                ,job-num                  ! i
                                ,print-control-buffer );  ! o
```

error-code returned value

INT

returns one of the following spooler error codes:

0	Successful operation
%3000- %3377	Error on file to supervisor (file-system error in bits <8:15>); refer to the *Guardian Procedure Errors and Messages Manual*
%10000	Parameter missing
%10001	Parameter in error
%14001	Command parameter missing
%14002	Command parameter in error
%14007	Cannot find entry requested
%14014	Security violation
%14015	Process not a spooler supervisor

supervisor-filenum input

INT:value

is the file number of an open supervisor file. The file number is returned when the supervisor is opened.

job-num input

INT:value

is the number of the job to be accessed.

print-control-buffer output

INT:ref:64 (Use with SPOOLEREQUEST)

INT:ref:128 (Use with SPOOLEREQUEST2)

returns a “start job” message suitable for passing to the PRINTREADCOMMAND procedure.

Considerations

The following considerations apply to the use of the SPOOLEREQUEST[2] procedure:

- Before calling the SPOOLEREQUEST[2] procedure, you must open a file to the spooler supervisor. You must specify waited I/O.
- The PRINTCOMPLETE[2], PRINTINFO, PRINTINIT[2], and PRINTSTATUS[2] spooler print procedures cannot be used by a perusal process.
- SPOOLEREQUEST[2] must be used with the spooler print procedures (PRINTREADCOMMAND Procedure on page 4-12, PRINTREAD Procedure on page 4-9, and PRINTSTART[2] Procedure on page 4-17).
- Because the supervisor does not know that the data file is being accessed, it allows the job to be deleted. If this occurs, PRINTREAD returns an “invalid data file” error (%12002) when attempting to read a line of data that is no longer there.
- SPOOLEREQUEST[2] returns job information only if the process access ID (PAID) of the process calling SPOOLEREQUEST[2] (for example, the user executing SPOOLEREQUEST[2]) matches the job’s owner.
- You can use SPOOLEREQUEST to access jobs that reside in the D41 and later releases of the spooler only if they are in the form of file code 129 job files; otherwise you must use SPOOLEREQUEST2.

SPOOLERSTATUS2 Procedure

The SPOOLERSTATUS2 procedure performs Spoolcom and Peruse operations from within applications. SPOOLERSTATUS allows a process to obtain the status of spooler components.

SPOOLERSTATUS2 differs from SPOOLERSTATUS in that *command-code* 13 has been added to support font information, *command-code* 14 has been added to support batch processing, and the size of the *status-buffer* has been doubled.

```
error-code := SPOOLERSTATUS2 ( supervisor-filenum       ! i
                             ,command-code              ! i
                             ,scan-type                 ! i
                             ,status-buffer );          ! i,o
```

error-code returned value

INT

returns one of the following spooler error codes:

0	Successful operation
%3000-%3377	Error on file to supervisor (file-system error in bits <8:15>; refer to Appendix C, Spooler-Related Errors, or to the *Guardian Procedure Errors and Messages Manual*).
%10000	Parameter missing
%10001	Parameter in error
%14001	Command parameter missing
%14002	Command parameter in error
%14006	End of SPOOLERSTATUS2 entries
%14007	Entry not found by SPOOLERSTATUS2
%14015	Process not a spooler supervisor
%14016	SPOOLERSTATUS2 request in progress

supervisor-filenum input

INT:value

is the file number of an open supervisor file. The file number is returned when the supervisor is opened.

command-code input

INT:value

specifies the spooler component whose status is being sought. The range of values and their meanings are listed in Table 4-11.

Table 4-11. SPOOLERSTATUS2 Command Codes

Command Code	Component
1	Device
2	Job
3	Location
4	Collector
5	Print process
6	Spooler
7	Jobs on a particular device queue
8	Occurrences of a particular job
9	Jobs with a particular location
10	Cross-reference by location
11	Cross-reference by device
12	Cross-reference by print process
13	Font information (SPOOLERSTATUS2 only)
14	Batch information (SPOOLERSTATUS2 only)
25	Collector LISTOPENS data

scan-type input

INT:value

specifies the type of scan desired as follows:

0 = Status of the item specified in the *status-buffer*
1 = Status of the item that follows the item specified in the *status-buffer*

status-buffer input, output

INT:ref:64 (Use with SPOOLERSTATUS)

INT:ref:128 (Use with SPOOLERSTATUS2)

is a 64-word or 128-word buffer where the status is returned. The format of the status buffer depends on the particular command code.

Considerations

The following considerations apply to the use of the SPOOLERSTATUS2 procedure:

- Before calling the SPOOLERSTATUS2 procedure, you must open a file to the spooler supervisor. You must specify waited I/O.

Obtaining the Spooler Statistics and Status

The spooler supervisor maintains a separate list for each type of spooler component. The lists of print processes, devices, collectors, and locations are in alphabetical order. The list of jobs is in ascending numerical order by job number. The SPOOLERSTATUS2 procedure allows you to access the elements of these lists sequentially or individually.

Listed below are the spooler components in *command-code* order, where *command-code* is a parameter of SPOOLERSTATUS2 as described in Table 4-11. For each type of component, you will find the STRUCTs that determine the format of the *status-buffer* where the status is returned.

Obtaining the Status of a Device (Command Code 1)

To obtain the status of a device, set the *command-code* parameter of SPOOLERSTATUS2 to 1 and pass either a 64-word status buffer to SPOOLERSTATUS or a 128-word status buffer to SPOOLERSTATUS2. The following STRUCT shows the fields of the buffer:

```
STRUCT device;

BEGIN
   INT name [0:15],                    ! $device name
                                       ! name[0:3]   = \system name
                                       ! name[4:7]   = $device name
                                       ! name[8:11]  = #subdevice
                                       ! name[12:15] = (blank-filled)

       state,                          ! 1 = device waiting
                                       ! 2 = device busy
                                       ! 3 = device suspended
                                       ! 4 = device error (deverror)
                                       ! 5 = device offline
                                       ! 6 = print process error
                                       !     (procerror)

       last^error,                     ! the last spooler error code
                                       ! recorded on the device

       flags;                          ! device flags (0=off, 1=on)
                                       ! flags.<0> = Devreset
                                       ! flags.<1:2>=Startff
                                          00 = OFF
                                          01 = ON
                                          10 =OFF!
                                       ! flags.<3>   = Endff
                                       ! flags.<4:7> = DEVTYPE
                                       0001 = SNAX LU1
                                       0010 = SNAX LU3
                                       0111 = 7  SUB-TYPE FOR 5515,
```

```
                                                  5516, 5518
                                          1000 = 8  SUB-TYPE FOR 5573,5574
                                          1001 = 9  SUB-TYPE FOR 5512
                                          1010 = 10 SUB-TYPE FOR 5577
                                          ! flags. <8> = Batch header
                                          ! flags. <9> = Exclusive OFF!
                                          ! flags.<10> = Truncation
                                          ! flags.<11> = Job printing
                                          ! flags.<12> = Device draining
                                          ! flags.<13> = Header
                                          ! flags.<14> = Exclusive
                                          ! flags.<15> = FIFO

STRING form^name [0:15];                  ! form name of device,
                                          !   blank-filled

INT retry^interval,                       ! minimum period to wait between
                                          !   retries

     time^out,                            ! number of retries to attempt
                                          !   before timing out device

     speed;                               ! device speed specified in the
                                          !   Spoolcom command (not
                                          !   actual speed of the device)


STRING print^process [0:5];               ! print process name,
                                          !   blank-filled

INT job^number,                           ! job number of job currently
                                          !   being printed

     parameter,                           ! parameter specified in
                                          !   Spoolcom command

     width,                               ! device width specified in the
                                          !   Spoolcom command (not
                                          !   necessarily actual width
                                          !   of device)

     retries,                             ! number of retries attempted on
                                          !   device.  Valid only when in
                                          !   deverror state.

     busy^time[0:1],                      ! amount of time (in seconds)
                                          !   required for current job to
                                          !   complete printing. Valid
                                          !   only when device is in
                                          !   print state.

     restart^interval,                    ! automatic device restart
                                          !   specification

     charmap,                             ! multibyte character set
                                          !   translation flag, where
                                          !    -1 = no translation
                                          !     2 = IBMKANJIKANA
                                          !     5 = JEFKANJIKANA
                                          !     8 = JISKANJIKANA


devflagx;                                 ! Optional Device Flag
                                          !  (0 = OFF, 1 = ON)
                                          ! devflagx.<1> = Prexlate
                                          ! DEVFLAGX.<10:11>= Lutofvalue
                                             00 = CRFFCR
                                             01 = FFCR
                                             10 = FF
```

```
   11 = NEVER
(for FASTP substitute for
 DEV PARM bits 10:11)
! DEVFLAGX.<12:13>= Lueolwhen
   00 = LT132
   11 = ALWAYS
   10 = LTWIDTH
   11 = NEVER
(for FASTP substitute for
 DEV PARM bits 12:13)
! DEVFLAGX.<15>= Lueolvalue
   0 = CRLF
   1 = NL
(for FASTP substitute for
 DEV PARM bit 15)
END;
```

If you want the status of all devices in the spooler subsystem, fill the `device.name` field with blanks and enter a 1 as the *`scan-type`*. The first call returns the status of the device whose name comes first alphabetically. Continue calling SPOOLERSTATUS2 until it returns error %14006 (end of entries).

If you want the status of a particular device, fill the `device.name` field with the name of the device whose status you want (remember to fill the right side of the field with blanks). Then call SPOOLERSTATUS2 with *`scan-type`* set to 0.

The values that can be returned for the field `device.charmap` are as follows:

-1 specifies that the device does not support multibyte character set translation.

2 specifies that the device supports IBMKANJIKANA characters.

5 specifies that the device supports JEFKANJIKANA characters.

8 specifies that the device supports JISKANJIKANA characters.

Because `device.charmap` is at the end of the structure, you do not have to include it in your program. Programs created before the addition of this device attribute will continue to execute properly.

The values that can be returned for the field `device.devflagx.<1>` are as follows:

0 specifies that the device is not in pretranslate mode.

1 specifies that the device is in pretranslate mode.

All other bits in `devflagx` are reserved for use by the spooler.

Obtaining the Status of a Job (Command Code 2)

To obtain the status of a job, set the *`command-code`* parameter of SPOOLERSTATUS2 to 2 and pass either a 64-word status buffer to SPOOLERSTATUS or a 128-word status buffer to SPOOLERSTATUS2. The last five

fields of the following STRUCT support spooler batch jobs and require the larger status buffer with SPOOLERSTATUS2. The following STRUCT shows the fields of the buffer:

```
STRUCT job^buffer;

BEGIN

   INT number,                          ! job number
                                        ! if number.<0> = 0, then
                                        !   SPOOLERSTATUS2 returns
                                        !   the status of all jobs
                                        !   in the spooler subsystem.


        state;                          ! 1 = Open
                                        ! 2 = Ready
                                        ! 3 = Hold
                                        ! 4 = Printing

STRUCT location;                        ! location of job in
BEGIN                                   !   internal form
   STRING group [0:7],                  ! #group name, blank-filled
          destination [0:7];            ! destination, blank-filled
END;

STRING form^name [0:15],                ! job form name, blank-filled
       report^name [0:15];              ! job report name, blank-filled

INT flags,                              ! device flags (0=off, 1=on)
                                        ! flags.<6> = Abnormal close
                                        !   (print process failed to
                                        !   call SPOOLEND prior to
                                        !   closing the file to the
                                        !   collector)
                                        ! flags.<9> = Hold
                                        ! flags.<10> = Holdafter
                                        ! flags.<13:15> = Selection
                                        !                  priority
   page^size,                           ! number of lines per page
   owner^ID,                            ! owner^ID.<0:7> = owner's group ID
                                        ! owner^ID.<8:15> = owner's user ID
   copies,                              ! number of copies to be printed
   pages,                               ! total number of pages in job
   lines,                               ! total number of lines in job
   time^opened [0:2],                   ! time opened (48-bit timestamp)
   time^closed [0:2];                   ! time closed (48-bit timestamp)

STRUCT data^file;                       ! name of file containing the job
BEGIN
   INT volume [0:3],                    ! $volume, blank-filled
       subvolume [0:3],                 ! subvolume, blank-filled
       filename [0:3];                  ! file name, blank-filled
END;

STRING collector^process^name [0:5];
                                        ! process name of collector
                                        !   used to create this job

INT units^allocated;                    ! number of units allocated
                                        !   by collector to the job
INT jobid;                              ! job id for this job
INT (32) max^lines;                     ! maximum number of lines
                                        !   allocated to the job
INT (32) max^pages;                     ! maximum number of pages
                                        !   allocated to the job
INT batch^name [0:15],                  ! batch name that includes
                                        !   this job
```

```
    batch^id;                          ! batch number
INT gmom^crtpid [0:3]                  ! crtpid of Netbatch monitor
END;
```

If you want the status of all jobs in the spooler subsystem, set the `job^buffer.number` field to 0 and pass a 1 as the *`scan-type`* parameter. The first call will return the status of the job with the lowest job number. Continue calling SPOOLERSTATUS2 until it returns error %14006 (end of entries).

If you want the status of jobs belonging to a specific owner ID, set bit 0 of `job^buffer.number` to 1, set `job^buffer.owner^ID` to the access ID of the person whose jobs you want returned, and pass 1 as the SPOOLERSTATUS2 *`scan-type`*. Note that SPOOLERSTATUS2 can return (without any error code) jobs that belong to other users (because the supervisor limits its search to 32 jobs and can return a job number even if the owner ID does not match). Your program should examine the returned buffer to verify that the owner is correct. If it is not, reissue the request and continue.

If you want the status of a particular job, set `job^buffer.number` to the job number of the job whose status you want. Then call SPOOLERSTATUS2 with *`scan-type`* set to 0.

If you want the status of only certain jobs in the spooler subsystem, set bit 1 of `job^buffer.number` to 1 and pass a 1 as the SPOOLERSTATUS2 *`scan-type`*. SPOOLERSTATUS2 returns error %14016 (request in progress) if the supervisor is unable to find a job that meets the qualifications after searching 32 jobs. Do not use the contents of the returned buffer. Instead, reassign the buffer and call SPOOLERSTATUS2 until %14006 (end of entries) is returned. SPOOLERSTATUS2 returns only the jobs that meet the following criteria:

- If state is not equal to 0, only jobs in the specified state are returned.
- If `location.group` is nonblank, only jobs with the specified group and location are returned.
- If `location.destination` is nonblank, only jobs with the specified destination are returned.
- Only jobs with the same `form^name` are returned.
- If `report^name` is nonblank, only jobs with the specified report name are returned.
- If `pages` is greater than 0, then all jobs with more than the number of pages specified are returned. If `pages` is less than 0, then all jobs with less than the absolute value of the number of pages specified are returned. Setting `pages` to 0 allows all jobs to be returned.
- All jobs that have closing timestamps less than `time^closed` and greater than `time^opened` are returned. Open jobs have infinite closing timestamps.
- If `collector^process^name` is nonblank, only jobs collected by the specified collector are returned.

- If `data^file` is nonblank, only jobs stored in the specified data file are returned.

When using a SPOOLERSTATUS2 *scan-type* of 1, if either bit 0 or bit 1 of `number` is set to 1 by the caller, these bits must be restored between calls. The low-order bits of `number` must not be allowed to change between calls. The easiest method to ensure this does not occur is to logically OR these bits with `number`. When using a *scan-type* of 1 with bit 1 of `number`, all fields except `number` must be reassigned.

SPOOLERSTATUS2 returns %14006 (end of entries) when all jobs have been returned.

Obtaining the Status of a Location (Command Code 3)

To obtain the status of a location, set the *command-code* parameter of SPOOLERSTATUS2 to 3 and pass either a 64-word status buffer to SPOOLERSTATUS or a 128-word status buffer to SPOOLERSTATUS2. The following STRUCT shows the fields of the buffer:

```
STRUCT location;
BEGIN
   STRUCT name;                          ! name of location
   BEGIN
      STRING group [0:7],                ! #group name, blank-filled
           destination [0:7];            ! destination, blank-filled
   END;
   INT flags,                            ! location flags
                                         !    (1=on, 0=off)
                                         !    flags.<8> =  broadcast
       device^name [0:15],               ! device name
                                         ! name[0:3]   = \system name*
                                         ! name[4:7]   = $device name*
                                         ! name[8:11]  = #subdev name*
! name[12:15] = (blank-filled)
       fontname [0:7];                   ! font associated with location
END;                                     !         * field is blank-filled
```

If you want the status of all locations in the spooler subsystem, fill `location.name` (both group and destination) with blanks and enter a 1 as the SPOOLERSTATUS2 *scan-type* parameter. The first call will return the status of the location whose name comes first alphabetically. Continue calling SPOOLERSTATUS2 until it returns error %14006 (end of entries).

When the group changes, you must make two calls to SPOOLERSTATUS2 in order to get complete information. The first call returns the group number, with the destination as blank; the next call returns all destinations within this group. In other words, the first return you receive is not the first location for this group; it is the group description itself.

If you want the status of a particular location, fill `location.name` with the group name and, as the destination, the name of the location whose status you want (both group and destination must be blank-filled). Then call SPOOLERSTATUS2 with the *scan-type* parameter set to 0.

Obtaining the Status of a Collector (Command Code 4)

To obtain the status of a collector, set the *command-code* parameter of SPOOLERSTATUS2 to 4 and pass either a 64-word status buffer to SPOOLERSTATUS or a 128-word status buffer to SPOOLERSTATUS2. The following STRUCT shows the fields of the buffer:

```
STRUCT collector;
BEGIN
   STRING name [0:5];                ! collector process name, in
                                     !   local form, blank-filled
   INT state,                        ! collector state:
                                     !   1 = Active
                                     !   2 = Dormant
                                     !   3 = Procerror
                                     !   4 = Drain
      reserved,                      ! reserved for use by HP
      last^error;                    ! last error recorded on collector

STRUCT program^file;                 ! file name of collector
BEGIN                                !    program
   INT volume [0:3],                 ! $volume, blank-filled
       subvolume [0:3],              ! subvolume, blank-filled
       filename [0:3];               ! file name, blank-filled
END;

INT cpus,                            ! CPUs executing the
                                     !   collector program
                                     ! cpus.<0:7> = primary
                                     ! cpus.<8:15> = backup
    priority;                        ! execution priority of
                                     !   the collector process

STRUCT data^file;                    ! file name of collector
BEGIN                                !   output file
   INT volume [0:3],                 ! $volume, blank-filled
       subvolume [0:3],              ! subvolume, blank-filled
       filename [0:3];               ! file name, blank-filled
END;

   INT unit^size,                    ! number of blocks reserved
                                     !   for when it needs more
                                     !   disk space
       units^allocated,              ! number of blocks already
                                     !   allocated
       total^units;                  ! total number of units
                                     !   available in data file
   INT pagesize;                     ! default page size

END;
```

If you want the status of all collectors in the spooler subsystem, fill the `collector.name` field with blanks and enter a 1 as the SPOOLERSTATUS2 *scan-type* parameter. The first call will return the status of the device whose name comes first alphabetically. Continue calling SPOOLERSTATUS2 until it returns error %14006 (end of entries).

If you want the status of a particular collector, fill the `collector.name` field with the name of the collector whose status you want (blank-fill the right side of the field). Then call SPOOLERSTATUS2 with the *scan-type* parameter set to 0.

Note. To obtain the status of a collector for large values (the values exceed the range of INT) of `units^allocated` and `total^units`, set the command-code parameter of SPOOLERSTATUS2 to 18. For more information on command-code 18, see the *Spooler Plus Programmer's Guide.*

Obtaining the Status of a Print Process (Command Code 5)

To obtain the status of a print process, set the *command-code* parameter of SPOOLERSTATUS2 to 5 and pass either a 64-word status buffer to SPOOLERSTATUS or a 128-word status buffer to SPOOLERSTATUS2. The following STRUCT shows the fields of the buffer:

```
STRUCT print;
BEGIN
   STRING process^name [0:5];          ! print process name,
                                       !   blank-filled
   INT state,                          ! print process state:
                                       !   1 = Active
                                       !   2 = Dormant
                                       !   3 = Procerror
                                       !   4 = Drain
      flags,                           ! print process flags
                                       !   (1 = yes, 0 = no)
                                       ! flags.<14> = Associate
                                       !              print process
                                       ! flags.<15> = Debug mode
      last^error;                      ! last error logged on
                                       !    print process

STRUCT program^file;                   ! program file of print process
BEGIN
  INT volume [0:3],                    ! $volume, blank-filled
      subvolume [0:3],                 ! subvolume, blank-filled
      filename [0:3];                  ! file name, blank-filled
END;

INT cpus,                              ! CPUs executing print
                                       !    process program
                                       ! cpus.<0:7> = primary
                                       ! cpus.<8:15> = backup
    priority,                          ! execution priority of
                                       !    the print process
    parameter;                         ! parameter from Spoolcom
                                       !    command
END;
```

If you want the status of all print processes in the spooler subsystem, fill the `print.process^name` field with blanks and enter a 1 as the *scan-type*. The first call will return the status of the print process whose name comes first alphabetically. Continue calling SPOOLERSTATUS2 until it returns error %14006 (end of entries).

If you want the status of a particular print process, fill the `print.process^name` field with the name of the print process whose status you want (fill the right side of the field with blanks). Then call SPOOLERSTATUS2 with the *scan-type* parameter set to 0.

Obtaining the Status of the Spooler (Command Code 6)

To obtain the status of the spooler, set the *command-code* parameter of SPOOLERSTATUS2 to 6, pass either a 64-word status buffer to SPOOLERSTATUS or a 128-word status buffer to SPOOLERSTATUS2, and set the *scan-type* parameter to 0. Because the file number specifies the spooler with which SPOOLERSTATUS2 is communicating, none of the status buffer fields needs to be filled. The following STRUCT shows the fields of the buffer:

```
STRUCT spooler^buffer;
BEGIN
   INT state,                          ! spooler state:
                                       !   1 = Active
                                       !   2 = Warm
                                       !   3 = Cold
                                       !   4 = Drain
      reserved;                        ! reserved for use by HP

STRUCT logfile;                        ! supervisor error log file
BEGIN
   INT volume [0:3],                   ! $volume, blank-filled
       subvolume [0:3],                ! subvolume, blank-filled
       filename [0:3];                 ! file name, blank-filled
END;

INT last^error;                        ! last error recorded in log
                                       ! file
END;
```

Obtaining the Status of Jobs Waiting to Print (Command Codes 7, 8, and 9)

To obtain the status of a job on the ready list, set the *command-code* parameter of SPOOLERSTATUS2 for the type of scan you want to perform (7, 8, or 9) and pass either a 64-word status buffer to SPOOLERSTATUS or a 128-word status buffer to SPOOLERSTATUS2. The following STRUCT shows the fields of the buffer:

```
STRUCT job^queue^status;
BEGIN
   INT state;                          ! job state:
                                       !   1 = Open
                                       !   2 = Ready
                                       !   3 = Hold
                                       !   4 = Printing
   INT device^name [0:15];             ! name of device job is or
                                       !   will be printing on
                                       ! [0:3]   = \system name
                                       ! [4:7]   = $device name
                                       ! [8:11]  = #subdevice
                                       ! [12:15] = (blank-filled)
   INT sequence^number;                ! device queue sequence
                                       !   number of job
   INT number;                         ! job number

STRUCT location;                       ! location of job in
BEGIN                                  !   internal form
   STRING group [0:7],                 ! #group name, blank-filled
         destination [0:7];            ! destination, blank-filled
END;
```

```
INT copies,                          ! number of copies to be printed
    pages,                           ! number of pages in job
    current^line,                    ! reserved for use by HP
    total^lines;                     ! total number of lines in
                                     !   the job

STRING form^name [0:15];             ! job form name, blank-filled
INT owner^ID;                        ! owner^ID.<0:7> = owner's group ID
                                     ! owner^ID.<8:15> = owner's user ID
END;
```

Obtaining the Status of Jobs in a Device Queue (Command Code 7)

To obtain the status of a job in a particular device queue, set the *command-code* parameter of SPOOLERSTATUS2 to 7 and the *scan-type* parameter to 1.

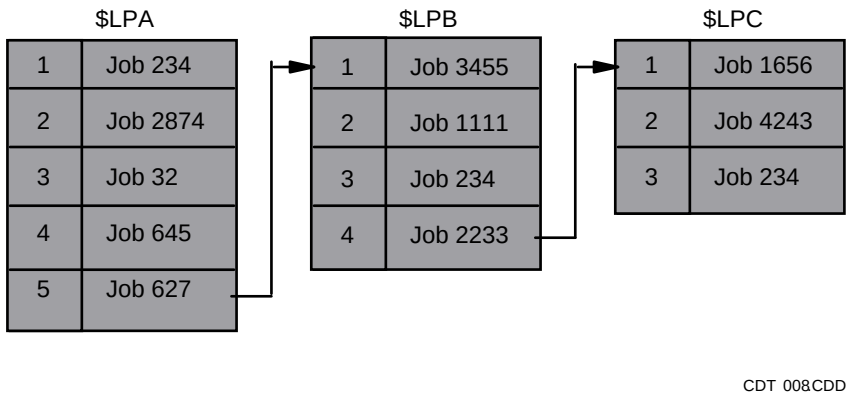
Put the name of the device whose queue you want to examine in the `job^queue^status.device^name` field (blank-filled), and set the `sequence^number` field to 0. The first call returns the job at the head of the device queue. Continue calling SPOOLERSTATUS2, but do not change the `device^name` or `sequence^number` fields. Each subsequent call returns the next job in the queue. When SPOOLERSTATUS2 returns %14006, the status of all jobs in the queue has been returned.

In addition to the lists of spooler components, the supervisor also keeps a ready list of all jobs in a device queue waiting to print. The ready list is composed of all device queues in the system. The end of one device queue links to the beginning of the next. The device queues are linked in alphabetical order according to the name of the device.

The device queue sequentially lists each job in the queue. The job at the front of the queue is assigned sequence number 1, the next job is sequence number 2, and so on. Sometimes the same job can appear in more than one queue. Each entry for the job in a device queue is called a job occurrence.

Figure 4-1 illustrates this data structure. The sequence number of each job is shown to the left of each job number. Note that Job 234 has a separate occurrence queued for each of the three devices.

Figure 4-1. Spooler Ready List



Obtaining the Status of Occurrences of a Job (Command Code 8)

To obtain the status of an occurrence of a particular job, set the *command-code* parameter of SPOOLERSTATUS2 to 8 and the *scan-type* parameter to 1.

Put the job number of the job whose occurrence you want to examine in the number field of the STRUCT. Fill the `device^name`, `location.group`, and `location.destination` fields with blanks and set `sequence^number` to zero.

Each call to SPOOLERSTATUS2 returns a different occurrence of the job. Do not change the `number` and `device^name` fields between calls. When SPOOLERSTATUS2 returns %14006 as the error code, the status of all occurrences of the job has been returned.

Obtaining the Status of Jobs at a Location (Command Code 9)

To obtain the status of a job at a specified location, set the *command-code* parameter of SPOOLERSTATUS2 to 9 and the *scan-type* parameter to 1.

Set the `location.group` and `location.destination` fields of the STRUCT to the location that you want to examine (blank-fill both fields on the right), and set the `sequence^number` field to 0. Each call to SPOOLERSTATUS2 returns the status of a different job at the specified location. Do not change the `sequence^number`, `device^name`, `location.group`, or `location.destination` fields between calls. When SPOOLERSTATUS2 returns %14006 as the error code, the status of all jobs at the specified location has been returned.

Obtaining a Cross-Reference (Command Codes 10, 11, and 12)

To obtain a cross-reference of locations, devices, or print processes, set the *command-code* parameter of SPOOLERSTATUS2 for the type of cross-reference you want and pass either a 64-word cross-reference buffer to SPOOLERSTATUS or a 128-word cross-reference buffer to SPOOLERSTATUS2. The command codes are as follows:

10 = by location

11 = by device

12 = by print process

The following STRUCT shows the fields of the buffer:

```
STRUCT xref^buffer;
BEGIN
   STRUCT location;
   BEGIN
      STRING group[0:7],              ! group name, blank-filled
             destination[0:7];        ! destination, blank-filled
   END;

   INT device^name[0:15];             ! device name (internal
                                      !    file-name format)
                                      !
   STRING process^name[0:5];          ! print process name,
                                      !   blank-filled
   INT marker;                        ! must be initialized to 0
END;
```

If you want a complete cross-reference, initialize the field to zero, set all other fields to blanks, and use a *scan-type* of 1.

If you want a cross-reference for a particular item, assign that field, initialize marker to zero and the other fields to blanks, and use a *scan-type* of 0.

In both cases, keep calling SPOOLERSTATUS2 until it returns error %14006 (end of entries).

Because device and print process cross-references can tie up the spooler supervisor for extended periods, SPOOLERSTATUS2 can return error %14016 (request in progress). When this error code is returned, the information in the buffer is invalid. Call SPOOLERSTATUS2 again until %14006 is returned (end of entries).

Note. Do not change any of the fields in the buffer between calls. Unexpected results can occur.

Obtaining the Status of a Font (Command Code 13)

To obtain the status of a font, set the *command-code* parameter of SPOOLERSTATUS2 to 13 and pass either a 64-word status buffer to

SPOOLERSTATUS or a 128-word status buffer to SPOOLERSTATUS2. The following STRUCT shows the fields of the buffer:

```
STRUCT font;
BEGIN
   INT fontname [0:7];      ! font name
   INT job;                 ! job associated with the font
END;
```

If you want the status of all fonts in the spooler subsystem, fill the `font.fontname` field with blanks and pass a 1 as the SPOOLERSTATUS *`scan-type`* parameter. The first call will return the status of the font whose name comes first alphabetically. Continue calling SPOOLERSTATUS until it returns error %14006 (end of entries).

If you want the status of a particular font, fill the `font.fontname` field with the name of the font and then call SPOOLERSTATUS2 with *`scan-type`* set to 0.

Obtaining the Status of a Batch (Command Code 14)

To obtain the status of a batch, set the *`command-code`* parameter of SPOOLERSTATUS2 to 14 and pass a 128-word status buffer to SPOOLERSTATUS2. The following STRUCT shows the fields of the buffer:

```
STRUCT batch;
BEGIN
   INT batch^id,                    ! batch number
       job,                         ! job number
       batch^name [0:15],           ! name of the batch
       jobs^in^batch;               ! number of jobs in batch
END;
```

If you want the status of a particular spooler batch job, set `batch.batch^id` to the batch number of the spooler batch job whose status you want. Then call SPOOLERSTATUS2 with *`scan-type`* set to 0. Note that `batch.job` returns with the number of the first spooler job in the spooler batch job. For batch jobs created by NetBatch, the first job is the user log file.

If you want the status of all spooler batch jobs in the spooler subsystem, set `batch.batch^id` to 0 and pass 1 as the *`scan-type`* parameter. The first call returns the status of the spooler batch job with the lowest batch number. Continue calling SPOOLERSTATUS2 until it returns error %14006 (end of entries). Note that `batch.job` returns with the number of the first spooler job in each batch job returned in this manner.

If you want the status of all spooler jobs in a spooler batch job, set `batch.batch^id` to the batch number of the spooler batch job whose spooler jobs you want. Set bit <0> of `batch.job` to 1, and set bits <1:15> of `batch.job` to 0. Then call SPOOLERSTATUS2 with *`scan-type`* set to 1. The first call will return the job number of the first spooler job in the spooler batch job. Continue calling SPOOLERSTATUS2 to get each job number in the spooler batch job. These job numbers are not in numerical sequence. They are in the order in which they were linked to the spooler batch job.

Example

```
STATUS^ERROR := SPOOLERSTATUS ( FILENUM , COM^CODE , TYPE ,
BUFF );
```

Obtaining Collector LISTOPENS (Command Code 25)

To obtain a list of the jobs that a specified collector currently has open, along with the processor and process identification number (PIN) of the processes that are spooling those jobs, set the *command-code* parameter of SPOOLERSTATUS2 to 25 and pass a 128-word status buffer to SPOOLERSTATUS2. The following STRUCT shows the fields in the buffer:

```
STRUCT sstatus^listopens;              !
BEGIN                                  !
 INT collector[0:2];                   !input: collector process name
 INT entry^offset;                     !input/output: entry number requested
 INT numitems;                         !number of OPENDATA items returned
 INT status;                           !status returned from collector
   STRUCT opendata [0:18];             !table of returned LISTOPENS data
   BEGIN
      INT jobid;                       !job number
      INT procname[0:2];               !process with job open
      INT cpu;                         !processor
      INT pin;                         !process identification number
   END;
END;
```

Set the `collector` field to the name of the collector for the LISTOPENS information needed and set the `entry^offset` field to 0. The call SPOOLERSTATUS2 with bit 15 of the *scan-type* parameter set to 1. The `numitems` field returned to the status buffer indicates how many valid occurrences of items exist in the `opendata` structure.

Each call to SPOOLERSTATUS2 will return data for up to 19 jobs. More than one call may be required to obtain all the LISTOPENS information. The `status` word returned will indicate whether all the data has been obtained. If the `status` word returned is 0, then there may be more data to be retrieved; if the `status` word is 1, then all the LISTOPENS data has been retrieved.

If you make subsequent calls to SPOOLERSTATUS2, make the additional calls without changing the input parameters or buffer contents, including `entry^offset`. The return from the first call will provide the internal `entry^offset` into the list of openers for the collector needed for the additional calls.

SPOOLJOBNUM Procedure

The SPOOLJOBNUM procedure returns the job number of the job currently being spooled to the collector. This procedure can be used when spooling at level 1, 2, or 3.

```
error-code := SPOOLJOBNUM ( filenum-of-collector     ! i
                          ,job-num );                ! o
```

error-code returned value

INT

returns one of the following spooler error codes:

0	Successful operation
%1000-%1377	Error on file to collector (bits <8:15> contain a file-system error number; see Considerations on page 4-60)
%10000	Missing parameter
%11001	Attempted to write to the collector without first opening the file

filenum-of-collector input

INT:value

is the file number of the collector. The file number is returned when the collector is opened.

job-num output

INT:ref:1

is the job number of the job currently being spooled to the collector through the specified file number. The value -1 is returned to *job-num* when *filenum-of-collector* is a spooler job file.

Considerations

The following considerations apply to the use of this procedure:

- A call to SPOOLJOBNUM can be issued by an application spooling at any level.
- SPOOLJOBNUM cannot be called with nowait I/O.
- When spooling at level 1, a job is not created until after the WRITE[X], SETMODE, or CONTROL procedure is called once.
- When spooling at level 2 or 3, a job is not created until after the SPOOLSTART procedure is called.

- Some file-system errors have special significance to a process sending data to a collector; many of these errors are described in Appendix C, Spooler-Related Errors. All of the file-system errors are listed in the *Guardian Procedure Errors and Messages Manual*.

  A program using level-1 or level-2 spooling gets these errors from the WRITE[X] or OPEN procedures, while a program spooling at level 3 obtains these errors in bits <8:15> of a spooler error code in the %1000 range.

Example

```
ERROR := SPOOLJOBNUM ( FILENUM^COLL , JOB^NUM );
```

SPOOLSETMODE Procedure

The SPOOLSETMODE procedure is used to set device-dependent functions when an application process is using the spooler interface procedures.

This procedure must be used in place of the SETMODE procedure if a level-3 buffer is specified in a call to SPOOLSTART.

```
error-code := SPOOLSETMODE ( level-3-buff                  ! i,o
                           ,function                       ! i
                           ,[ param1 ]                     ! i
                           ,[ param2 ]                     ! i
                           ,[ bytes-written-to-buff ] )    ! o
                           ,[ extended-level-3-buff ] ); ! i,o
```

error-code returned value

INT

returns one of the following spooler error codes:

0	Successful operation
%1000-%1377	Error on file to collector (bits <8:15> contain a file-system error number; see Considerations on page 4-63)
%11000	Checkpoint exit
%11001	Attempted to write to the collector without first opening the file

level-3-buff input, output

INT:ref:512

is the *level-3-buff* specified in the call to SPOOLSTART.

function input

INT:value

is a SETMODE function (see the *Guardian Procedure Calls Reference Manual)*.

param1 input

INT:value

is a parameter for the specified SETMODE function (see the *Guardian Procedure Calls Reference Manual)*.

param2 input

INT:value

is a parameter for the specified SETMODE function (see the *Guardian Procedure Calls Reference Manual)*.

bytes-written-to-buff output

INT:ref:1

returns the number of bytes to be checkpointed from the *level-3-buff*. This parameter is used by fault-tolerant applications.

extended-level-3-buff input,output

INT:.EXT.ref.*

is the *extended-level-3-buff* specified in the SPOOLSTART procedure.

Considerations

The following considerations apply to the use of the SPOOLSETMODE procedure:

- If *flags*.<11> of SPOOLSTART is set to 1, a return of %11000 from SPOOLSETMODE indicates that the *level-3-buff* is about to be written to the collector. The buffer should be checkpointed, and SPOOLSETMODE should be called again.
- Some file-system errors have special significance to a process sending data to a collector; these errors are listed in the *Guardian Procedure Errors and Messages Manual*.

  A program using level-1 or level-2 spooling gets these errors from the WRITE[X], OPEN, or FILE_OPEN_ procedure, while a program spooling at level 3 obtains these errors in bits <8:15> of a spooler error code in the %1000 range.

Example

```
ERROR := SPOOLSETMODE ( COLL^BUFF , 68 , 2 ); ! select expanded
                                                ! print.
```

SPOOLSTART Procedure

The SPOOLSTART procedure formats a spooler buffer suitable for passing to other spooler interface procedures. It is used to specify job attributes, establish a level-2 or level-3 spooling session with a spooler collector, or establish a level-3 spooling session to a spooler job file.

```
error-code := SPOOLSTART ( [filenum-of-collector ]       ! i
                         ,[ level-3-buff ]               ! o
                         ,[ location ]                   ! i
                         ,[ form-name ]                  ! i
                         ,[ report-name ]                ! i
                         ,[ num-of-copies ]              ! i
                         ,[ page-size ]                  ! i
                         ,[ flags ]                      ! i
                         ,[ owner ]                      ! i
                         ,[ max-lines ]                  ! i
                         ,[ max-pages ]                  ! i
                         ,[ file-name ]                  ! i,o
                         ,[ filenum ] )                  ! i,o
                         ,[ extended-level-3-buff ] ); ! i
```

error-code returned value

INT

returns one of the following spooler error codes:

0	Successful operation
%1000-%1377	Error on file to collector (bits <8:15> contain a file-system error number; see Considerations on page 4-67)
%10000	Missing parameter
%10001	Parameter is present, but its content is wrong
%11000	Checkpoint exit
%11001	Attempted to write to the collector without first opening the file

filenum-of-collector input

INT:value

is the file number of the collector or spooler job file obtained through a call to the system OPEN or FILE_OPEN_ procedure. The collector must be opened for waited I/O.

level-3-buff output

INT:ref:512

indicates that the spooler interface procedures are used to send data to the collector. The data is put into a buffer. The address of this buffer is returned by the SPOOLSTART procedure and must be passed to other interface procedures. The buffer is initialized and its address returned as a result of this call. The buffer is located in the data stack and is limited to 512 bytes. If a buffer area in extended memory is needed, use the *extended-level-3-buff* parameter instead of the *level-3-buff* parameter. Use either *level-3-buff* or *extended-level-3-buff*, but not both.

location input

INT:ref:8

specifies a location for this job and overrides the location specified in the call to OPEN or FILE_OPEN_. The default location for the job is the location specified when the collector was opened.

form-name input

INT:ref:8

specifies a form name for the job. The form name can contain letters, digits, and blanks. The default *form-name* is all blank-filled.

report-name input

INT:ref:8

specifies a report name for the job. The report name can contain letters, digits, and blanks.

The default *report-name* is the user name of the person executing the application program.

num-of-copies input

INT:value

specifies the number of copies to print. The default *num-of-copies* is 1.

page-size input

INT:value

specifies the page size used by Peruse when a PAGE or LIST command is given. The default *page-size* is 60.

flags input

INT:value

specifies certain attributes of the job.

The bit fields are as follows:

<0>	Reserved for use by the collector	
<1>	ASCII compression:	0 = off 1 = on
<2:8>	Reserved for use by the collector	
<9>	HOLD flag:	0 = off 1 = on
<10>	HOLDAFTER flag:	0 = off 1 = on
<11>	SPOOLWRITE, SPOOLCONTROL, SPOOLCONTROLBUF, and SPOOLSETMODE exit before writing the level-3 buffer to the collector process, so that user can checkpoint:	0 = no 1 = yes
<12>	Delete existing data in spooler job file:	0 = no 1 = yes
<13:15>	Job priority	

The default job priority is 4; all other bits are set to 0.

owner input

INT:value

allows the caller to assign job ownership. The owner name is group number:user number.

The default *owner* of a job is the user who opened the file to the collector.

max-lines input

INT(32):value

is the maximum number of lines to allow for the job. The current range is 0 through 65534. This parameter is a 32-bit integer to allow for future expansion of the upper limit. If *max-lines* is omitted or 0, no maximum number is enforced.

max-pages input

INT(32):value

is the maximum number of pages to allow for the job. The current range is 0 through 999999. This parameter is a 32-bit integer to allow for future expansion of the upper limit. If *max-pages* is omitted or 0, no maximum number is enforced.

file-name input, output

INT:ref:12

specifies the file name of the spooler collector or spooler job file to be opened for the spooling session.

filenum input, output

INT:ref:1

is an alternative parameter to *filenum-of-collector*. See Considerations on page 4-67 for more information.

extended-level-3-buff output

INT:.EXT.ref.*

indicates that the spooler interface procedures are used to send data to the collector. The data is put into a buffer. The address of this buffer is returned by the SPOOLSTART procedure and must be passed to other interface procedures. The buffer is initialized and its address returned as a result of this call. The buffer is allocated in an extended data segment.

Use either *level-3-buff* or *extended-level-3-buff,* but not both.

Considerations

The following considerations apply to the use of the SPOOLSTART procedure:

- Only level-3 spooler data can be directed to a spooler job file, an unstructured disk file with a file code of 129. Use *file-name* to specify the spooler job file. If *file-name* does not exist, a spooler job file is created with primary and secondary extent sizes of 50 pages (2048 bytes per page) and maximum extents of 1000.
- If a blank-filled volume name is passed in *file-name*, a temporary spooler job file is created. The complete *file-name* is returned in *file-name*. If *file-name* exists, new data is appended to the data already in the file unless *flags*.<12> is equal to 1, in which case an error message is generated.
- If SPOOLSTART is appending data to a previously written spooler job file and either *max-lines* or *max-pages* is specified, the current number of lines or pages already in the file is added to the maximums so that the maximums then refer to the total amount of data the file can hold. If either maximum value is greater than 65,534, the value is reset to 0 (no maximum enforced).
- The *filenum-of-collector* and *filenum* parameters are two different parameters for the file number of the collector or spooler job file. The *filenum-of-collector* is a value parameter and the *filenum* is a reference parameter. You should pass one or the other of these parameters to SPOOLSTART if you use them, but not both.

Pass the three parameters *filenum-of-collector*, *file-name*, and *filenum* in one of the following ways:

- If the file is already open, pass the file number as either *filenum-of-collector* or *filenum*. You can pass *file-name* but it is ignored.
- If the file is not open and the caller does not need to know the file number, pass *file-name*. The *filenum-of-collector* and *filenum* parameters can be omitted, or one of them can be passed containing the value -1.
- If neither the *filenum-of-collector* nor the *filenum* is passed, the *file-name* must be passed.
- If the file is not open and the caller wants to know the file number, pass *file-name* and *filenum* set to -1. Do not pass *filenum-of-collector*. The number of the file opened is returned in *filenum*. The value -1 is returned if the file cannot be opened.

- Some file-system errors have special significance to a process sending data to a collector; these errors are listed in the *Guardian Procedure Errors and Messages Manual*.

  A program using level-1 or level-2 spooling gets these errors from the WRITE[X] or OPEN procedure, while a program spooling at level 3 obtains these errors in bits <8:15> of a spooler error code in the %1000 range.

- ASCII compression (specified by *flags*.<1>) is meaningful only when writing to spooler job files. The compression results in a file-size savings of about 33 percent.

SPOOLWRITE Procedure

The SPOOLWRITE procedure is used to write to a collector when the application process is spooling at level 3.

The SPOOLWRITE procedure compresses and blocks data into the level-3 buffer and, when the buffer is full, writes the buffer to the collector or a spooler job file. This procedure must be used in place of the WRITE[X] procedure if a level-3 buffer is specified in a call to SPOOLSTART.

```
error-code := SPOOLWRITE ( level-3-buff                    ! i,o
                         ,print-line                       ! i
                         ,write-count                      ! i
                         ,[ bytes-written-to-buff ] )      ! o
                         ,[ extended-level-3-buff ] );     ! i,o
```

error-code returned value

INT

returns one of the following spooler error codes:

0	Successful operation
%1000-%1377	Error on file to collector (<8:15> contains a file-system error number; see Considerations on page 4-70)
%10000	Missing parameter
%10001	Parameter is present, but its content is wrong
%11000	Checkpoint exit
%11001	Attempted to write to the collector without first opening the file

level-3-buff input, output

INT:ref:512

is the *level-3-buff* specified in the call to SPOOLSTART.

print-line input

INT:ref:*

is an array containing the line of data to be sent to the collector. The size of *print-line* must not exceed 900 bytes.

write-count input

INT:value

is the number of bytes of *print-line* to be written; it must not exceed 900 bytes.

bytes-written-to-buff output

INT:ref:*

returns the number of bytes in the *level-3-buff* to be checkpointed.

extended-level-3-buff input,output

INT:.EXT.ref.*

is the *extended-level-3-buff* specified in the SPOOLSTART procedure.

Considerations

The following considerations apply to the use of the SPOOLWRITE procedure:

- Each call to SPOOLWRITE causes *print-line* to be written to the *level-3-buff*. When a call to SPOOLWRITE causes the *level-3-buff* to overflow, the buffer is written to the collector.

  The blocking and compression of data into the *level-3-buff* are invisible to the application process.

- If bit 11 of the *flags* parameter of SPOOLSTART is set to 1, SPOOLWRITE exits with a spooler error code of %11000 prior to writing the *level-3-buff* to the collector. Applications running as a NonStop process pair can then perform a checkpoint before the buffer is written to the collector. SPOOLWRITE should be called again after checkpointing.

- Some file-system errors have special significance to a process sending data to a collector; these errors are listed in the *Guardian Procedure Errors and Messages Manual*.

  A program using level-1 or level-2 spooling gets these errors from the WRITE[X] or OPEN procedures, while a program spooling at level 3 obtains these errors in bits <8:15> of a spooler error code in the %1000 range.

Example

```
SPERRNUM := CALL SPOOLWRITE ( COLL^BUFFER , PRINT^LINE ,
LENGTH );
```

A Sample Print Process

This appendix shows an example of a print process. It includes a description of the program and the code.

The example has been kept as simple as possible in the interest of clarity. Simplifications include the following:

- Only one job at a time can be printed.
- No form alignment is performed.
- The record size of all devices is assumed to be 132.
- The truncation flag is ignored.
- The print process parameter and device parameter are not used.

The structure of the sample print process is such that these features can easily be implemented.

Note. The program presented in this appendix can be compiled and run exactly as presented. However, it is *not* a supported software product of HP and has not undergone the rigorous testing given to an officially released product. Please keep this in mind when adapting the code for your needs.

```
INT .p^buf[0:63],                   ! print control buffer
    .job^buffer[0:559],             ! job buffer
    .data^line[0:65],               ! next line of data
    count^read,                     ! number of bytes in "data^line"
    .out^buf[0:65],                 ! buffer written to device
    .time^array [0:6],              ! date & time for header pages
    recv^fnum,                      ! $RECEIVE file number
    supv^fnum,                      ! supervisor file number
    dev^fnum,                       ! device file number
    data^file^fnum,                 ! collector data file number
    busy^flag := 0,                 ! true when a job is printing
    suspend   := 0,                 ! true when a job is suspended
    suspended := 0,                 ! true when a job is suspended due
                                    !   to a write error
    end^of^job^flag := 0,           ! true = indicates end of job,
                                    !        start job clean-up tasks
    header^index :=0,               ! true = print job banner page
    msg^type,                       ! used to send error to supervisor
    successful^op := 1;             ! used to determine which message
                                    !   type is sent on device error
                                    !   information returned by
                                    !   PRINTREADCOMMAND

    STRUCT DEV;
      BEGIN
        INT flags;                      ! indicates the state of the
                                        ! device's header and
                                        ! truncation flags
        INT param;                      ! parameter specified in the
                                        ! "DEV, PARAM" SPOOLCOM
                                        ! command
        INT width;                      ! value specified in the "DEV,
                                        ! WIDTH" SPOOLCOM command
        INT job^num;                    ! spooler-assigned job number
        INT locationname[0:7];          ! location name
        INT formname[0:7];              ! form name
        INT reportname[0:7];            ! report name
        INT page^size;                  ! number of lines per page
      END;
```

```
  STRING
      .s^out^buf := @out^buf '<<' 1,     ! string pointer to "out^buf"
      .s^data^line := @data^line '<<' 1; ! string pointer to
                                         ! "data^line"

  STRUCT .header[0:14];             ! header array
    BEGIN
      INT line[0:59];
    END;

  LITERAL
      sending^status = 0,           ! PRINTSTATUS message types
      dev^error^1 = 1,              !
      end^job = 2,                  !
      cant^open^device = 3,         !
      invalid^operation = 4,        !
      dev^error^5 = 5,              !
      max^read^count = 132,         ! maximum PRINTREAD count
      data^file^error = %2000,      ! spooler error codes
      device^file^error = %4000,    !
      no^job^printing = %13002,     !
      job^running = %13003,         !
      tables^full = %13004;         !

!SOURCE $SYSTEM.SYSTEM.EXTDECS ( ABEND, AWAITIO, CANCEL, CONTROL,
!       CLOSE, FILEINFO, NUMOUT, OPEN, READ, SETMODE, STOP, TIME,
!       WRITE, PRINTINIT, PRINTCOMPLETE, PRINTINFO, PRINTSTATUS,
!       PRINTREAD, PRINTREADCOMMAND, PRINTSTART )
?NOLIST
?SOURCE $SYSTEM.SYSTEM.EXTDECS ( ABEND, AWAITIO, CANCEL, CONTROL,
?       CLOSE, FILEINFO, NUMOUT, OPEN, READ, SETMODE, STOP, TIME,
?       WRITE, PRINTINIT, PRINTCOMPLETE, PRINTINFO, PRINTSTATUS,
?       PRINTREAD, PRINTREADCOMMAND, PRINTSTART )
?LIST
INT PROC openfile (fname, fnum, flags) VARIABLE;
  INT .fname,
      .fnum,
      flags;
  FORWARD;
INT PROC writefile (fnum, buf, write^count);
  INT fnum,
      .buf,
      write^count;
  FORWARD;
PROC tell^super (type, device, error, page, line, total^lines,
                 num^copies) VARIABLE;
  INT type,                 ! required parameter
      .device,              ! required parameter, name of the device
      error,                ! required parameter for type 1, 2, 4, and 5
      page,                 ! required parameter for type 0; current
                            ! page number
      line,                 ! required parameter for type 0; current
                            ! line number
      total^lines,          ! required parameter for type 0; total
                            ! number of lines printed
      num^copies;           ! required parameter for type 0; number of
                            ! copies of the job remaining to be printed

  FORWARD;
PROC build^header;
  FORWARD;
PROC read^and^print (page^num, device);
  INT page^num,
      .device;
  FORWARD;
PROC open^dev (device);
```

```
  INT .device;
  FORWARD;
PROC close^dev (device);
  INT .device;
  FORWARD;
PROC start^job (device, data^file^name, job^num, location, formname,
                reportname, flags, params, dev^width, page^size);
  INT .device,
      .data^file^name,
      job^num,
      .location,
      .formname,
      .reportname,
      flags,
      params,
      dev^width,
      page^size;
  FORWARD;
PROC init^job;
  FORWARD;
PROC stop^job (device, error);
  INT .device,
      error;
  FORWARD;
PROC req^stop^job (device);
  INT .device;
  FORWARD;
PROC resume^job (device);
  INT .device;
  FORWARD;
PROC suspend^job (device);
  INT .device;
  FORWARD;
PROC align^form (device);
  INT .device;
  FORWARD;
PROC skip^to^page (skip^num, device);
  INT skip^num,
      .device;
  FORWARD;
PROC skip^page (skip^num, device);
  INT skip^num,
      .device;
  FORWARD;
PROC send^status (device);
  INT .device;
  FORWARD;
PROC main^loop;
  FORWARD;

! *===================================================================*
!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
! procedure name: OPENFILE
! parameters: FNAME - name of the file to open
!             FNUM - file number returned to the calling routine
!             FLAGS - open flags
! description: This procedure is the same as the Guardian file-
!              system procedure "OPEN", plus a check of the returned
!              condition code. Returns a Guardian file-system error
!              number if condition code < 0.
!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

INT PROC openfile (fname, fnum, flags) VARIABLE;
  INT .fname,
      .fnum,
      flags;
```

```
  BEGIN
    LITERAL missing^parameter = 29;

    INT err := 0,
        flags^ := 0;

!---Beginning of code-----------------------------------------------------
    IF NOT ($PARAM (fname) AND $PARAM (fnum)) THEN
      RETURN missing^parameter;
    IF $PARAM (flags) THEN
      flags^ := flags;
    CALL OPEN (fname, fnum, flags^);
    IF < THEN
      CALL FILEINFO (-1, err);
    RETURN err;
  END;

! *===================================================================*

!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
! procedure name: WRITEFILE
! parameters:  FNUM - file number
!              BUF - buffer that contains the text to be written
!              WRITE^COUNT - number of bytes to be written
! description: This procedure is the same as the Guardian file-
!              system procedure "WRITE", plus a check of the returned
!              condition code. Returns a Guardian file-system error
!              number if condition code < 0.
!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

INT PROC writefile (fnum, buf, write^count);
  INT fnum,
      .buf,
      write^count;
  BEGIN

    INT err := 0;

!---Beginning of code-----------------------------------------------------

    CALL WRITE (fnum, buf, write^count);
    IF < THEN
      CALL FILEINFO (fnum, err);
    RETURN err;
  END;

! *===================================================================*

!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
! procedure name: TELL^SUPER
! parameters: TYPE - type of message sent
!             DEVICE - name of the device on which the error occurred
!             ERROR - the error number that caused this call to
!                     TELL^SUPER
!             PAGE - current page number, required parameter for type
!                    0
!             LINE - current line number, required parameter for type
!                    0
!             TOTAL^LINES - total number of lines printed, required
!                           parameter for type 0
!             NUM^COPIES - number of copies of the job remaining to be
!                          printed, required parameter for type 0
! description: This procedure informs the spooler supervisor that an
!              error occurred (Guardian file-system error number) or
!              responds to a request for the status of a job by
!              calling PRINTSTATUS. The parameters to this procedure
```

```
!              are exactly the same as those of PRINTSTATUS, except
!              that the supervisor file number and print control
!              buffer are not required. Instead, this procedure simply
!              assumes that these are "supv^fnum" and "p^buf",
!              respectively.
!
!              Like PRINTSTATUS, some parameters of this procedure are
!              either required or optional depending on the message
!               type.  However, rather than checking for required
!              parameters, it is assumed that these parameters have
!              been correctly supplied by the print process.
!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

PROC tell^super (type, device, error, page, line, total^lines,
                 num^copies) VARIABLE;
  INT type,                 ! required parameter
      .device,              ! required parameter, name of the device
      error,                ! required parameter for type 1, 2, 4, and 5
      page,                 ! required parameter for type 0; current
                            ! page number
      line,                 ! required parameter for type 0; current
                            ! line number
      total^lines,          ! required parameter for type 0; total
                            ! number of lines printed
      num^copies;           ! required parameter for type 0; number of
                            ! copies of the job remaining to be printed
  BEGIN

!---Beginning of code-------------------------------------------------

    CASE type OF
      BEGIN
!-0- sending status of a job
        error := PRINTSTATUS (supv^fnum, p^buf, type, device,
                              num^copies, page, line, total^lines);

!-1- error occurred on print device, previous operation was unsuccessful
        error := PRINTSTATUS (supv^fnum, p^buf, type, device, error);

!-2- end of job
        error := PRINTSTATUS (supv^fnum, p^buf, type, device, error);

!-3- unable to open job
        error := PRINTSTATUS (supv^fnum, p^buf, type, device, error);

!-4- invalid operation in current state
        error := PRINTSTATUS (supv^fnum, p^buf, type, device, error);

!-5- error occurred on print device, previous operation was successful
        error := PRINTSTATUS (supv^fnum, p^buf, type, device, error);
      END;
    IF error THEN
      CALL ABEND;
    RETURN;
  END;
```

```
! *===================================================================*

!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
! procedure name: BUILD^HEADER
! parameters: none
! description: This procedure builds the job page banner, which in
!              general is entirely application-defined. The page banner
!              printed by this sample process is:
!
!              ****************************************
!              *                                      *
!              *  DATE:  today's date                 *
!              *  TIME:  current time                 *
!              *  JOB NUMBER:  spooler job number     *
!              *  LOCATION NAME:  spooler location name   *
!              *  REPORTNAME:  name of report         *
!              *  FORM NAME:  name of form            *
!              *                                      *
!              ****************************************
!
!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

PROC build^header;
  BEGIN
    INT i,
        error;
    STRING .locations := @dev.locationname '<<' 1,
           .formnames := @dev.formname '<<' 1,
           .reportnames := @dev.reportname '<<' 1;
    STRING months [0:35] = 'P' :=
                           "JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC";
    STRING .ptr;

!---Beginning of code-------------------------------------------------

    FOR i := 0 to 14 DO
      BEGIN
        header[i].line := "  ";
        header[i].line[1] ':=' header[i].line FOR 59;
      END;
!   Load the header array
    IF header^index = 2 THEN
      BEGIN
        @ptr := @header[0].line '<<' 1;
        ptr[20] := "*";
        ptr[21] ':=' ptr[20] FOR 60;
        error := writefile (dev^fnum,header[0].line,60);
        IF error THEN
          CALL ABEND;
        header^index := header^index + 1;
        RETURN;
      END;
    IF header^index = 3 THEN
      BEGIN
        @ptr := @header[1].line '<<' 1;
        ptr[20] := ptr[80] := "*";
        CALL TIME (time^array );
        error := writefile (dev^fnum,header[1].line,60);
        IF error THEN
          CALL ABEND;
        header^index := header^index + 1;
        RETURN;
      END;
    IF header^index = 4 THEN
      BEGIN
        @ptr := @header[2].line '<<' 1;
```

```
    ptr[20] := ptr[80] := "*";
    ptr[30] ':=' "DATE:  ";
    CALL NUMOUT ( ptr[37], time^array[2], 10, 2 );
    ptr[40] ':='  months [3 * ( time^array[1] - 1 )] FOR 3;
    CALL NUMOUT ( ptr[44], time^array, 10, 2 );
    error := writefile (dev^fnum,header[2].line,60);
    IF error THEN
      CALL ABEND;
    header^index := header^index + 1;
    RETURN;
  END;
IF header^index = 5 THEN
  BEGIN
    @ptr := @header[3].line '<<' 1;
    ptr[20] := ptr[80] := "*";
    error := writefile (dev^fnum,header[3].line,60);
    IF error THEN
      CALL ABEND;
    header^index := header^index + 1;
    RETURN;
  END;
IF header^index = 6 THEN
  BEGIN
    @ptr := @header[4].line '<<' 1;
    ptr[20] := ptr[80] := "*";
    ptr[30] ':=' "TIME:  ";
    ptr[39] := ptr[42] := ":";
    CALL NUMOUT ( ptr[37], time^array[3], 10, 2 );
    CALL NUMOUT ( ptr[40], time^array[4], 10, 2 );
    CALL NUMOUT ( ptr[43], time^array[5], 10, 2 );
    error := writefile (dev^fnum,header[4].line,60);
    IF error THEN
      CALL ABEND;
    header^index := header^index + 1;
    RETURN;
  END;
IF header^index = 7 THEN
  BEGIN
    @ptr := @header[5].line '<<' 1;
    ptr[20] := ptr[80] := "*";
    error := writefile (dev^fnum,header[5].line,60);
    IF error THEN
      CALL ABEND;
    header^index := header^index + 1;
    RETURN;
  END;
IF header^index = 8 THEN
  BEGIN
    @ptr := @header[6].line '<<' 1;
    ptr[20] := ptr[80] := "*";
    ptr[30] ':=' "JOB NUMBER:  ";
    CALL NUMOUT ( ptr[43], dev.job^num, 10, 2 );
    error := writefile (dev^fnum,header[6].line,60);
    IF error THEN
      CALL ABEND;
    header^index := header^index + 1;
    RETURN;
  END;
IF header^index = 9 THEN
  BEGIN
    @ptr := @header[7].line '<<' 1;
    ptr[20] := ptr[80] := "*";
    error := writefile (dev^fnum,header[7].line,60);
    IF error THEN
      CALL ABEND;
    header^index := header^index + 1;
```

```
    RETURN;
  END;
IF header^index = 10 THEN
  BEGIN
    @ptr := @header[8].line '<<' 1;
    ptr[20] := ptr[80] := "*";
    ptr[30] ':=' "LOCATION NAME:  " & locations FOR 8;
    ptr[54] := ".";
    ptr[55] ':=' locations[8] FOR 8;
    error := writefile (dev^fnum,header[8].line,60);
    IF error THEN
      CALL ABEND;
    header^index := header^index + 1;
    RETURN;
  END;
IF header^index = 11 THEN
  BEGIN
    @ptr := @header[9].line '<<' 1;
    ptr[20] := ptr[80] := "*";
    error := writefile (dev^fnum,header[9].line,60);
    IF error THEN
      CALL ABEND;
    header^index := header^index + 1;
    RETURN;
  END;
IF header^index = 12 THEN
  BEGIN
    @ptr := @header[10].line '<<' 1;
    ptr[20] := ptr[80] := "*";
    ptr[30] ':=' "FORM NAME:  " & formnames FOR 8;
    error := writefile (dev^fnum,header[10].line,60);
    IF error THEN
      CALL ABEND;
    header^index := header^index + 1;
    RETURN;
  END;
IF header^index = 13 THEN
  BEGIN
    @ptr := @header[11].line '<<' 1;
    ptr[20] := ptr[80] := "*";
    error := writefile (dev^fnum,header[11].line,60);
    IF error THEN
      CALL ABEND;
    header^index := header^index + 1;
    RETURN;
  END;
IF header^index = 14 THEN
  BEGIN
    @ptr := @header[12].line '<<' 1;
    ptr[20] := ptr[80] := "*";
    ptr[30] ':=' "REPORT NAME:  " & reportnames FOR 16;
    error := writefile (dev^fnum,header[12].line,60);
    IF error THEN
      CALL ABEND;
    header^index := header^index + 1;
    RETURN;
  END;
IF header^index = 15 THEN
  BEGIN
    @ptr := @header[13].line '<<' 1;
    ptr[20] := ptr[80] := "*";
    error := writefile (dev^fnum,header[13].line,60);
    IF error THEN
      CALL ABEND;
    header^index := header^index + 1;
    RETURN;
```

```
      END;
    IF header^index = 16 THEN
      BEGIN
        @ptr := @header[14].line '<<' 1;
        ptr[20] := "*";
        ptr[21] ':=' ptr[20] FOR 60;
        error := writefile (dev^fnum,header[14].line,60);
        IF error THEN
          CALL ABEND;
        header^index := 0;
        RETURN;
      END;
  END;

! *===================================================================*

!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
! procedure name: READ^AND^PRINT
! parameters:  PAGE^NUM - indicates page that will be printed
!              DEVICE - name of the device where the job is printing
! description: This procedure gets and prints one line of the job.  If
!              the global flag HEADER^FLAG is true, then this
!              procedure will print the job banner (header) by a
!              succession of calls, one call for each line of the job
!              banner, 15 lines in all.
!
!              When HEADER^FLAG is false, the spooler procedure
!              PRINTREAD is called to return a line of spooled data.
!              The line of spooled data is written to the device when
!              the error code returned from PRINTREAD is zero.
!
!              There are six valid error codes that can be returned by
!              PRINTREAD:
!
!                12000 = end of file found. The procedure STOP^JOB is
!                        called for job termination (normal is
!                        indicated).
!                12001 = end of copy found. The variable ERROR is
!                        reset to 0, HEADER^FLAG is set to its
!                        original start-of-job value, and control is
!                        passed back to the beginning of this
!                        procedure.
!                12002 = data file is bad. The procedure STOP^JOB is
!                        called for job termination (abnormal is
!                        indicated).
!                12003 = CONTROL found. CONTROL is issued to the IOP.
!                        If the CONTROL was successful, the next
!                        spooled data line is read. If unsuccessful,
!                        the spooler supervisor is notified of the
!                        error and the print job will be terminated.
!                12004 = SETMODE found. SETMODE is issued to the IOP.
!                        If the SETMODE was successful, the next
!                        spooled data line is read. If unsuccessful,
!                        the spooler supervisor is notified of the
!                        error and the print job will be terminated.
!                12005 = CONTROLBUF found. CONTROLBUF is issued to the
!                        IOP.  If the CONTROLBUF was successful, the
!                        next spooled data line is read. If
!                        unsuccessful, the spooler supervisor is
!                        notified of the error and the print job will
!                        be terminated.
!              > 12005 = invalid returned error code; the print
!                        process will abend.
!
!              When the return code (variable ERROR) value from the
!              procedure WRITE^DEV is 0 (positive return code),
```

```
!              control is passed back to MAIN^LOOP. When the return
!              code value is <> 0 (negative return code), the spooler
!              supervisor is notified of the error and the job will be
!              terminated.
!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

PROC read^and^print (page^num, device);
  INT page^num,
      .device;

  BEGIN

    LITERAL end^of^file   = %12000,
            end^of^copy   = %12001,
            control^found = %12003,
            setmode^found = %12004,
            next^line     = 0;

    INT write^count,              ! number of bytes to be written
        error,
        wait^needed;              ! Set true when we need to return.

!---Beginning of code-----------------------------------------------

next^copy:

    IF header^index THEN
      BEGIN
        IF header^index = 1 THEN
          BEGIN
    !       Issue CONTROL to IOP print process
            CALL CONTROL (dev^fnum, 1, 0);
            IF <> THEN  ! check CONTROL condition code
              BEGIN
    !           CONTROL was unsuccessful
                suspend   := 1;
                suspended := 1;
                CALL FILEINFO (dev^fnum, error);
                msg^type := IF successful^op THEN
                              dev^error^1
                            ELSE
                              dev^error^5;
                CALL tell^super ( msg^type, device,
                                  device^file^error + error);
                successful^op := 0;
              END
             ELSE
    !         CONTROL was successful
              successful^op := 1;
            header^index := header^index + 1;
          END
         ELSE
          CALL build^header;
      END
     ELSE
! Get either first or next line of spooled data
      DO
        BEGIN
          wait^needed := 1;
          error := PRINTREAD (job^buffer, data^line, max^read^count,
                              count^read, page^num);
          IF error THEN
            BEGIN
              CASE error - end^of^file OF
                BEGIN
!-12000- end of file found
```

```
                  BEGIN
                    CALL stop^job (device, 0);
                    RETURN;
                  END;
!-12001- end of copy found
                  BEGIN
                    header^index := dev.flags.<13>;
                    error := 0;
                    CALL SETMODE (dev^fnum, 28, 0);
                    goto next^copy;
                  END;
!-12002- data file is bad
                  BEGIN
                    CALL stop^job (device, error);
                  END;
!-12003- CONTROL found
                  BEGIN
!                   Issue CONTROL to IOP print process--issue only
!                   forms control operations - any others are ignored.
                    IF data^line <> 1 THEN
                      wait^needed := 0    ! No IO so no need to wait
                    ELSE
                      BEGIN
                        CALL CONTROL (dev^fnum, data^line, data^line[1]);
                        IF <> THEN  ! check CONTROL condition code
                          BEGIN
!                           CONTROL was unsuccessful
                            suspend   := 1;
                            suspended := 1;
                            CALL FILEINFO (dev^fnum, error);
                            msg^type := IF successful^op THEN
                                          dev^error^1
                                        ELSE
                                          dev^error^5;
                            CALL tell^super ( msg^type, device,
                                              device^file^error + error);
                            successful^op := 0;
                          END
                        ELSE
!                         CONTROL was successful
                          successful^op := 1;
                      END;
                  END;
!-12004- SETMODE found
                  BEGIN
!                   Issue SETMODE to IOP print process
                    CALL SETMODE (dev^fnum, data^line, data^line[1],
                                  data^line[2]);
                    IF <> THEN  ! check SETMODE condition code
                      BEGIN
!                       SETMODE was unsuccessful
                        suspend   := 1;
                        suspended := 1;
                        CALL FILEINFO (dev^fnum, error);
                        msg^type := IF successful^op THEN
                                      dev^error^1
                                    ELSE
                                      dev^error^5;
                        CALL tell^super ( msg^type, device,
                                          device^file^error + error);
                        successful^op := 0;
                      END
                     ELSE
!                     SETMODE was successful
                      BEGIN
                        successful^op := 1;
```

```
                        wait^needed := 0;    ! SETMODE is a waited
                                             ! operation
                      END;
                  END;
                OTHERWISE
                  BEGIN
                  END;
!       End of error case
                  END;
             END
           ELSE
!       Successful PRINTREAD, write data to device
             BEGIN
               ! If the DEV WIDTH is <0, allow any size of IO
               ! up to a limit set by the literal max^read^count.
               ! Otherwise truncate the IO so that we don't
               ! send more than WIDTH chars to the device
               IF dev.width < 0 THEN
                  write^count := count^read
               ELSE
                  write^count := $MIN ( dev.width , count^read );
               out^buf ':=' data^line FOR write^count;
               error := writefile (dev^fnum, out^buf, write^count);
               IF error THEN
                 BEGIN
                   suspend   := 1;
                   suspended := 1;
                   msg^type := IF successful^op THEN
                                 dev^error^1
                               ELSE
                                dev^error^5;
                   CALL tell^super (msg^type, device,
                                    device^file^error + error);
                   successful^op := 0;
                 END;
             END;
!       End of DO loop
           END
       UNTIL wait^needed;
    RETURN;
  END;

! *===================================================================*

!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
! procedure name: STOP^JOB
! parameters: DEVICE - name of device job is printing on
!             ERROR - error number
! description: Closes the collector data file if still open,
!              initializes device and job variables/flags, and reports
!              "end of job" status (thru TELL^SUPER) to the spooler
!              supervisor
!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

PROC stop^job (device, error);
  INT .device,
      error;
  BEGIN

!---Beginning of code-------------------------------------------------

    IF data^file^fnum THEN
      CALL CLOSE (data^file^fnum);
    CALL init^job;
    IF error AND error < 1000 THEN
      CALL tell^super (invalid^operation, device, data^file^error +
```

```
                        error)
      ELSE
       CALL tell^super (end^job, device, error);
     RETURN;
  END;

! *===================================================================*

PROC init^job;
  BEGIN

!---Beginning of code-----------------------------------------------

    data^file^fnum := 0;
    busy^flag := 0;
    suspend   := 0;
    suspended := 0;
    msg^type := 0;
    successful^op := 0;
    end^of^job^flag := 0;
    header^index := 0;
  END;

! *===================================================================*

! CONTROL NUMBER = 0: OPEN DEVICE

!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
! procedure name: OPEN^DEV
! parameters:  DEVICE - name of device to be opened
! description: This procedure services the spooler supervisor command
!              to open a device (exclusive and nowaited). If the open
!              is unsuccessful, a PRINTSTATUS message, type 3 (unable
!              to open device), will be sent to the spooler supervisor.
!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

PROC open^dev (device);
  INT .device;
  BEGIN
    INT exclusive^and^nowait := %21,
        error;

!---Beginning of code-----------------------------------------------

    error := openfile (device, dev^fnum, exclusive^and^nowait);
    IF error THEN
      BEGIN
        CALL tell^super (cant^open^device, device, device^file^error +
                         error);
      END;
    RETURN;
  END;

! *===================================================================*

! CONTROL NUMBER = 1: CLOSE DEVICE

!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
! procedure name: CLOSE^DEV
! parameters:  DEVICE - name of device to be closed
! description: This procedure services the spooler supervisor command
!              to close a device. If device is currently printing a
!              job, the spooler supervisor is notified that the close
!              command is invalid.
!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
PROC close^dev (device);
  INT .device;

  BEGIN

!---Beginning of code-------------------------------------------------

    IF busy^flag THEN
!     Cannot close device; currently printing a job
      CALL tell^super (invalid^operation, device, job^running)
     ELSE
      BEGIN
        IF NOT dev^fnum THEN
!         Device is already closed
        ELSE
          BEGIN
            CALL CLOSE (dev^fnum);
            IF < THEN
              CALL ABEND;
            dev^fnum := 0;
          END;
      END;
    RETURN;
  END;

! *===================================================================*

! CONTROL NUMBER = 2: START JOB

!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
! procedure name: START^JOB
! parameters: DEVICE - name of device where the job is to be printed
!             DATA^FILE^NAME - name of the collector data file
!             JOB^NUM - job number of the current job
!             LOCATION - location name
!             FORMNAME - name of form
!             REPORTNAME - name of report
!             FLAGS - flags that indicate items such as header on
!             PARAMS - user parameter word defined through SPOOLCOM
!             DEV^WIDTH - device width in bytes, defined through
!                         SPOOLCOM
!             PAGE^SIZE - number of lines in a page
! description: This procedure services the spooler supervisor command
!              to start a job.  The collector data file is opened (read
!              only). If the open is unsuccessful, the spooler
!              supervisor is notified that the job has ended, and the
!              file-system error (plus spooler print process error
!              base number) that caused the job to end (in this case
!              terminated) is returned.
!
!              The spooler procedure PRINTSTART is called to format
!              the job buffer for the job being started. The job
!              buffer will be used in subsequent calls to the spooler
!              procedure PRINTREAD when reading data from the
!              collector file.
!
!              SETMODE 28 is issued to the IOP to reset the line
!              parameters back to the SYSGEN/OSBUILDER or default values.
!
!              Control is then passed to the procedure READ^AND^PRINT
!              to fetch data from the collector data file and write it
!              to the device.
!
!              ***NOTE*** As stated in the beginning of this print
!              process, it is designed to support only one device. If
!              it is desired to support more than one device, this
```

```
!              procedure should move information contained in the
!              calling parameters into a device table.
!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

PROC start^job (device, data^file^name, job^num, location, formname,
                reportname, flags, params, dev^width, page^size);
  INT .device,
      .data^file^name,
      job^num,
      .location,
      .formname,
      .reportname,
      flags,
      params,
      dev^width,
      page^size;
  BEGIN

    LITERAL first^line = 0;

    INT error,
        read^only := 1 '<<' 10;  ! used for opening data file

!---Beginning of code-------------------------------------------------

    IF NOT dev^fnum THEN        ! The spooler supervisor can ask us to
      CALL open^dev (device);   ! start a job without asking for the
                                ! device to be opened.
    IF NOT dev^fnum THEN
      RETURN;                   ! The OPEN failed
    error := openfile (data^file^name, data^file^fnum, read^only);
    IF error THEN
      BEGIN
        CALL tell^super (end^job, device, data^file^error + error);
        CALL STOP;
      END
     ELSE
      IF PRINTSTART (job^buffer, p^buf, data^file^fnum) THEN
        CALL ABEND;
!**** NOTE***************************************************
! this would be a good location for the device table code
!*********************************************************
    CALL SETMODE (dev^fnum, 28, 0);
    busy^flag := 1;
    dev.flags := flags;
    dev.param := params;
    dev.width := dev^width;
    dev.job^num := job^num;
    dev.page^size := page^size;
    dev.locationname ':=' location for 8;
    dev.formname ':=' formname for 8;
    dev.reportname ':=' reportname for 8;
    header^index := flags.<13>;
    CALL read^and^print (first^line, device);
    RETURN;
  END;

! *=================================================================*

! CONTROL NUMBER = 3: STOP JOB

!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
! procedure name: REQ^STOP^JOB
! parameters:  DEVICE - name of device where the job is to be printed
! description: This procedure services the spooler supervisor command
!              to stop a job. If the print process is not busy
```

```
!              printing a job, the spooler supervisor is notified that
!              the stop job command is invalid.
!
!              If the print process is currently printing a job, two
!              CANCEL requests are issued to cancel any outstanding
!              requests.  The second request is most likely not needed;
!              it is added as extra insurance.
!
!              Control is passed to the procedure STOP^JOB to complete
!              the tasks of stopping the current job.
!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

PROC req^stop^job (device);
  INT .device;
  BEGIN

!---Beginning of code-------------------------------------------------

    IF NOT busy^flag THEN
      CALL tell^super (invalid^operation, device, no^job^printing)
     ELSE
      BEGIN
        CALL CANCEL (dev^fnum);
        CALL CANCEL (dev^fnum);
        CALL stop^job (device, 0);
      END;
    RETURN;
  END;

! *===================================================================*

! CONTROL NUMBER = 4: RESUME JOB

!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
! procedure name: RESUME^JOB
! parameters:  DEVICE - name of device where the job is to be printed
! description: This procedure services the spooler supervisor command
!              to resume printing a job that had previously been
!              suspended. If the current print job is not in a
!              suspended state, the spooler supervisor is notified
!              that the resume job command is invalid.
!
!              When "suspended" flag is set to true, printing will
!              resume with the record whose write previously failed.
!              Otherwise, printing will resume with the next
!              sequential record.
!
!              Control is passed to the procedure READ^AND^PRINT, where
!              the job resumes printing.
!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

PROC resume^job (device);
  INT .device;
  BEGIN
    INT error,
        next^line := 0;

!---Beginning of code-------------------------------------------------

    IF NOT suspend THEN
      CALL tell^super (invalid^operation, device, no^job^printing)
     ELSE
      BEGIN
        IF suspended THEN
          next^line := -1;
        suspend   := 0;
```

```
        suspended := 0;
        CALL read^and^print (next^line, device);
      END;
  END;

! *===================================================================*

! CONTROL NUMBER = 5: SUSPEND JOB

!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
! procedure name: SUSPEND^JOB
! parameters: DEVICE - name of device where the job is currently
!                      printing.
! description: This procedure services the spooler supervisor command
!              to suspend the printing of the current job.  If the
!              print process is not currently printing a job, the
!              spooler supervisor is notified that the suspend job
!              command is invalid.
!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

PROC suspend^job (device);
  INT .device;
  BEGIN

!---Beginning of code-----------------------------------------------

    IF NOT busy^flag THEN
      CALL tell^super (invalid^operation, device, no^job^printing)
     ELSE
      suspend := 1;
    RETURN;
  END;

! *===================================================================*

! CONTROL NUMBER = 6: ALIGN FORM

!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
! procedure name: ALIGN^FORM
! parameters:  DEVICE - name of device where the alignment template is
!                       to be printed.
! description: This procedure services the spooler supervisor command
!              to print the alignment template. If the print process
!              is currently printing a job, the spooler supervisor is
!              notified that the form alignment command is invalid.
!
!              The write of the form alignment template can fail in
!              two areas: at the Guardian 90 procedure WRITE call (in
!              the procedure WRITEFILE) and at the Guardian 90
!              procedure AWAITIO call (in the procedure MAIN^LOOP). The
!              spooler supervisor will be notified if the form
!              alignment template write was unsuccessful.
!
!              When the form alignment template write is successful,
!              end-of-job status is sent to the spooler supervisor
!              through a call to the procedure STOP^JOB in the
!              procedure MAIN^LOOP after the AWAITIO completion.
!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

PROC align^form (device);
  INT .device;
  BEGIN
    INT error;

!---Beginning of code-----------------------------------------------
```

```
    IF busy^flag THEN
      CALL tell^super (invalid^operation, device, job^running)
     ELSE
      BEGIN
        IF not dev^fnum THEN
           CALL open^dev (device);
        IF not dev^fnum THEN   ! Open failed
           RETURN;
        data^line ':='
          "....+....1....+....2....+....3....+....4....+....5....+....6"
         &"....+....7....+....8....+....9....+....0....+....1....+....2";
         error := writefile(dev^fnum, data^line, 120);
         IF error THEN
           BEGIN
             msg^type := IF successful^op THEN
                           dev^error^1
                         ELSE
                           dev^error^5;
             CALL tell^super (msg^type, device,
                              device^file^error + error);
             successful^op := 0;
           END;
        successful^op := 1;
        end^of^job^flag := 1;
      END;
    RETURN;
  END;

! *===================================================================*

! CONTROL NUMBER = 7: SKIP TO PAGE

!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
! procedure name: SKIP^TO^PAGE
! parameters: SKIP^NUM - Specific page number where printing is to
!                        resume.
!             DEVICE - name of device where job is printing.
! description: This procedure services the spooler supervisor command
!              to skip to a specific page and resume printing.  If the
!              print process has not started a job (is not busy), the
!              spooler supervisor is notified that the skip-to-page
!              command is invalid.
!
!              A call to CANCEL is issued to cancel any outstanding
!              I/O request to prevent error 28 (too many outstanding
!              no-wait requests) from occurring.
!
!              The header flag is set to off so that the job page
!              header will not be repeated.  Control is then passed to
!              the procedure READ^AND^PRINT, where the skip-to page
!              number is passed in the PRINTREAD call. PRINTREAD will
!              return the first record on the specified page, which is
!              the skip-to page.
!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

PROC skip^to^page (skip^num, device);
  INT skip^num,
      .device;
  BEGIN
    INT error;

!---Beginning of code-------------------------------------------------

    IF NOT busy^flag THEN
      CALL tell^super (invalid^operation, device, no^job^printing)
     ELSE
```

```
      BEGIN
        CALL CANCEL (dev^fnum);
        header^index := 0;
        CALL read^and^print (skip^num, device);
      END;
    RETURN;
  END;

! *===================================================================*

! CONTROL NUMBER = 8: SKIP <skip^num> PAGES

!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

! procedure name: SKIP^PAGE
! parameters: SKIP^NUM - number of pages to be skipped.
!             DEVICE - name of device where job is printing.
! description: This procedure services the spooler supervisor command
!              to skip a specific number of pages and resume printing.
!              The spooler procedure PRINTINFO is called to fetch the
!              current page number of the current print job. If
!              PRINTINFO returns an error, the print process will
!              terminate.
!
!              The current page number plus the number of pages to
!              be skipped is passed to the procedure SKIP^TO^PAGE,
!              where printing will resume at the desired page (current
!              page number + skip^num + 1).
!
!              *** NOTE*** This sample print process does not issue an
!              error message prior to calling the Guardian 90 procedure
!              ABEND.  When creating a print process, it would be best
!              to issue an error message prior to abnormal termination.
!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

PROC skip^page (skip^num, device);
  INT skip^num,
      .device;
  BEGIN
    INT error,
        page;

!---Beginning of code-------------------------------------------------

    error := PRINTINFO (job^buffer, , page);
    IF error THEN
      CALL ABEND;
    page := page + skip^num;
    IF page < 0 THEN
      page := 0;
    header^index := 0;
    CALL skip^to^page (page, device);
    RETURN;
  END;

! *===================================================================*

! CONTROL NUMBER = 9: SEND STATUS

!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
! procedure name: SEND^STATUS
! parameters: DEVICE - name of device where job is printing.
! description: This procedure services the spooler supervisor command
!              for status of the current print job.
!
!              If there is no current print job, the spooler supervisor
```

```
!              is notified that the status command is invalid.
!
!              The spooler procedure PRINTINFO is called to fetch
!              information about the current print job.
!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

PROC send^status (device);
  INT .device;
  BEGIN
    INT error,
        page,
        line,
        total^lines,
        num^copies;

!---Beginning of code-------------------------------------------------

    IF NOT busy^flag THEN
      CALL tell^super (invalid^operation, device, no^job^printing)
     ELSE
       BEGIN
         error := PRINTINFO (job^buffer, num^copies, page, line,
                             total^lines);
         IF error THEN
           CALL ABEND;
         CALL tell^super (sending^status, device, page, line,
                          total^lines, num^copies);
       END;
    RETURN;
  END;

! *===================================================================*

! outer program loop

!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

! procedure name: MAIN^LOOP
! parameters: none
! description: This procedure is the nucleus of the print process. It
!              issues the call to AWAITIO and services its
!              completion.  The AWAITIO completion is divided into two
!              sections, completion of spooler supervisor requests
!              and completion of writes to the device. The writes to
!              the device are further divided into two areas,
!              successful writes and unsuccessful writes.
!
!              SUPERVISOR REQUEST COMPLETIONS: The spooler procedure
!              PRINTCOMPLETE is called to obtain a message (command)
!              from the spooler supervisor. The message is contained
!              in the buffer P^BUF. If an error is returned by
!              PRINTCOMPLETE, the print process will terminate
!              abnormally and no error message is issued.  NOTE: there
!              are other options than abending the print process (for
!              example, retry the PRINTCOMPLETE call).  Also, issuing
!              an error message should be considered.
!
!              Upon successful completion of PRINTCOMPLETE, the
!              spooler procedure PRINTREADCOMMAND is called to
!              interpret the message returned by PRINTCOMPLETE. Here
!              again, if PRINTREADCOMMAND returns an error, the print
!              process will terminate abnormally and no error message
!              is issued.
!
!              Upon successful completion of PRINTREADCOMMAND, the
!              command value stored in "control" is used to select the
```

```
!              procedure to process the spooler supervisor command.
!
!              DEVICE WRITE COMPLETIONS:
!              ERROR:
!              When the device write completes in error, the spooler
!              supervisor is notified. This notification also
!              indicates whether the previous device write was
!              successful.  This information is used for retryable type
!              errors.  The "suspend" flag is set to true to indicate
!              that the current print job has been suspended. The
!              "suspended" flag is also set to true to indicate that
!              the current print job was suspended due to an error.
!              When the "suspended" flag is true, the record that
!              resulted in the write error is returned by
!              PRINTREAD, versus PRINTREAD returning the next record
!              ("suspend" flag = true and "suspended" flag = false).
!
!              When the error is retryable, the spooler supervisor
!              sends a RESUME command to the print process, where
!              the job resumes printing with the record whose
!              write previously failed.
!              SUCCESSFUL:
!              When the device write successfully completes, the
!              "successful^op" flag is set to true, indicating a
!              successful I/O occurred.  If there is an active print
!              job (not suspended), control is passed to the procedure
!              READ^AND^PRINT, where the printing of the job is
!              continued.
!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

PROC main^loop;
  BEGIN
    INT control,             ! Parameter returned by PRINTREADCOMMAND
        .device[0:11] := 12 * ["  "],
        dev^flags,
        dev^param,
        dev^width,
        skip^num,
        .data^file^name[0:11] := 12 * ["  "],
        job^num,
        .location[0:7] := 8 * ["  "],
        .formname[0:7] := 8 * ["  "],
        .reportname[0:7] := 8 * ["  "],
        page^size,
        fnum,
        error;
    INT(32) timeout;
    LITERAL next^line = 0;

!---Beginning of code-------------------------------------------------

    WHILE 1 DO
      BEGIN
        fnum := -1;
        ! If the device is not open, wait 2 minutes before stopping.
        ! During this time the spooler can ask the user to start a new
        ! job without having to fire up a new process.
        IF dev^fnum = 0 THEN
           timeout := 12000D  ! 2 minutes
        ELSE
           timeout := -1D;    ! forever
        CALL AWAITIO (fnum,,,, timeout);
        CALL FILEINFO (fnum, error);
        IF error = 40 then    ! No completion within time limit,
           CALL STOP;         ! must be time to stop
        IF fnum = dev^fnum THEN
```

```
            BEGIN
              IF error THEN
                BEGIN
                  suspend   := 1;
                  suspended := 1;
                  msg^type := IF successful^op THEN
                                dev^error^5
                              ELSE
                               dev^error^1;
                  call tell^super (msg^type, device, device^file^error +
                                   error);
                  successful^op := 0;
                END
               ELSE
                BEGIN
                  successful^op := 1;
                  IF busy^flag AND NOT suspended THEN
                    CALL read^and^print (next^line, device);
                  IF end^of^job^flag THEN
                    CALL stop^job (device, error);
                END;
            END;
          IF fnum = supv^fnum THEN
            BEGIN
              IF PRINTCOMPLETE (supv^fnum, p^buf) THEN
                CALL ABEND;
              IF PRINTREADCOMMAND (p^buf,
                                   control,
                                   device,
                                   dev^flags,
                                   dev^param,
                                   dev^width,
                                   skip^num,
                                   data^file^name,
                                   job^num,
                                   location,
                                   formname,
                                   reportname,
                                   page^size) THEN
                CALL ABEND;
              CASE control OF
                BEGIN
! control = 0: OPEN DEVICE
                  CALL open^dev (device);
! control = 1: CLOSE DEVICE
                  CALL close^dev (device);
! control = 2: START JOB
                  CALL start^job (device, data^file^name, job^num,
                                  location, formname, reportname,
                                  dev^flags, dev^param, dev^width,
                                  page^size);
! control = 3: STOP JOB
                  BEGIN
                    CALL req^stop^job (device);
                  END;
! control = 4: RESUME JOB
                  CALL resume^job (device);
! control = 5: SUSPEND JOB
                  CALL suspend^job (DEVICE);
! control = 6: FORM ALIGNMENT
                  CALL align^form (DEVICE);
! control = 7: SKIP TO PAGE
                  CALL skip^to^page (skip^num, device);
! control = 8: SKIP PAGES
                  CALL skip^page (skip^num, device);
! control = 9: SEND STATUS
```

```
                CALL send^status (device);
              END;  ! End of control case
          END;
      END;     ! End of WHILE 1 loop
  END;

! *===================================================================*

! main procedure

!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
! procedure name: SAMPLE^PRINT^PROCESS
! parameters: none
! description: This procedure opens $RECEIVE and reads the Startup
!              message.  The spooler supervisor process name is
!              extracted from the Startup message and opened. The
!              spooler process PRINTINIT is called to initialize the
!              print control buffer (P^BUF), which will be used in
!              other calls to spooler print procedures.  Control is
!              then passed to the procedure MAIN^LOOP.
!
!              If the $RECEIVE open and read fails, if the spooler
!              supervisor open fails, or if PRINTINIT returns an error,
!              the print process will abnormally terminate. No error
!              message indicating abnormal termination is issued.
!- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

PROC sample^print^process MAIN;
  BEGIN
    INT .recv^buf[0:65] := ["$RECEIVE", 62 * ["  "]],
        .supv^name[0:11] := 12 * ["  "];

!---Beginning of code-------------------------------------------------

    IF openfile (recv^buf, recv^fnum) THEN
      CALL ABEND;
    CALL READ (recv^fnum, recv^buf, 132);
    IF <> THEN
      CALL ABEND;
    CALL CLOSE (RECV^FNUM);
    supv^name ':=' recv^buf[21] for 12;
    IF openfile (supv^name, supv^fnum, 1) THEN
      CALL ABEND;
    IF PRINTINIT (supv^fnum, p^buf) THEN
      CALL ABEND;
    CALL main^loop;
  END;

! *===================================================================*
```

B Sample Perusal Process

This appendix shows a sample perusal process. This sample process functions in much the same way as the listing section of the PERUSE program. The sections of code concerned with taking commands from the terminal and listing errors are not included because they are not actually related to the perusal process operation.

Note. The program presented in this appendix can be compiled and run as presented. However, it is *not* a supported software product of HP and has not undergone the rigorous testing given to an officially released product. Please keep this in mind when adapting the code for your needs.

```
! This program performs some of the operations executed by PERUSE.
! The purpose of this program is to illustrate the use of the
! spooler utility procedures and print procedures in a perusal
! process.  The commands recognized have been reduced to a subset
! of the PERUSE commands.  They include LOC, HOLD, JOB, DEV, LIST,
! and EXIT.  The DEV command has been enhanced to return the status
! of all devices when an argument is not supplied.

! All input is taken from the home terminal, and all output is returned
! to the home terminal (IN and OUT files are not used).

! This program can be copied from this guide and compiled.

?page
   STRING command  [8:104] = 'P' := ["LOC     ",
                                     "HOLD    ",
                                     "JOB     ",
                                     "DEV     ",
                                     "LIST    ",
                                     "EXIT    ",
                                     "LO      ",
                                     "H       ",
                                     "J       ",
                                     "D       ",
                                     "L       ",
                                     "E       "];

! This structure is used to get the status of devices

STRUCT .dev^stat;
   BEGIN
   INT name[0:15],
       state,
       last^error,
       flags;

   STRING form^name [0:15];

   INT retry^interval,
       time^out,
       speed;

   STRING print^process [0:5];

   INT job^number,
       parameter,
       width,
       retries,
       busy^time[0:1];
   END;
```

```
?page

! This structure is used to get the status of jobs in a device queue

STRUCT .job^status;
BEGIN
   INT state,
       device^name [0:15],
       sequence^number,
       job^number;

   STRUCT location;
      BEGIN
         STRING group [0:7],
                destination [0:7];
      END;

   INT copies,
       pages,
       reserved,
       total^lines;

   STRING form^name [0:15];

   INT owner^id;
END;
?page

! This structure is used to get the status of jobs in a device queue

STRUCT .job ;
BEGIN
   INT number,
       state;

   STRUCT location;
   BEGIN
      STRING group [0:7],
             destination [0:7];
   END;

   STRING form^name [0:15],
          report^name [0:15];

   INT flags,
       page^size,
       owner^id,
       copies,
       pages,
       lines,
       time^opened [0:2],
       time^closed [0:2];

   STRUCT data^file;
   BEGIN
      INT volume [0:3],
          subvolume [0:3],
          filename [0:3];
   END;

   STRING collector^process^name [0:5];
END;

?page
! This structure receives the TACL Startup message from $RECEIVE
```

```
STRUCT .ci^Startup;
   BEGIN
   INT msgcode;
   STRUCT default;
      BEGIN
      INT volume[0:3],
          subvolume[0:3];
      END;

   STRUCT infile;
      BEGIN
      INT volume[0:3],
          subvolume[0:3],
          dname [0:3];
      END;

   STRUCT outfile;
      BEGIN
      INT volume[0:3],
          subvolume[0:3],
          dname [0:3];
      END;

   STRING param [0:24];
   END;

?page
! Global declarations


INT .termname [0:11] := [ 12 * ["  "] ],           ! contains the name of
                                                   ! the user's terminal
    .recname [0:11] := "$RECEIVE                ", ! for reading Startup
                                                   ! message
    .supername [0:11] := "$SPLS                   ", ! file name of
                                                     ! supervisor
    .datafile[0:11],                            ! file name of collector's
                                                ! data file
    termnum,                    ! file number for terminal
    recnum,                     ! file number for $RECEIVE
    supernum,                   ! file number for supervisor
    datanum,                    ! file number for collector's data file
    userid,                     ! executor's user ID
    current^job,                ! number of the current job
    list^job,                   ! number of the job currently ready for
                                ! listing

    .message[0:63],             ! these arrays are used by the
    .jobbuff[0:559],            ! print procedures to get a
    .dline[0:449];              ! job's data


INT .iline [0:39] := 40 * ["  "];
STRING .line :=  @iline '<<' 1;   ! this is the string address of iline.

! The following lines show the external declarations used in this
! program.  The actual source command has not been listed for brevity.
! The command itself is shown in the comment lines that follow.

!source $system.system.extdecs( spoolerequest, printinfo, printread,
! printstart, printreadcommand, fileinfo, numout, close, numin,
! spoolercommand, spoolerstatus, open, write, read, writeread,
! creatoraccessid, stop, myterm)

?nolist
```

```
?source $system.system.extdecs( spoolerequest, printinfo, printread,
? printstart,  printreadcommand, fileinfo, numout, close, numin,
? spoolercommand, spoolerstatus, open, write, read, writeread,
? creatoraccessid, stop, myterm)
?list

?page
! This procedure handles errors from the spooler utility procedures

! No error recovery or diagnostic message is displayed; only the error
! number is listed.  A finished program would have error recovery, and
! a meaningful diagnostic message would be displayed.

 PROC err (errcode);
   INT errcode;
   BEGIN
   STRING err^line [0:39] := ["error^code = ", 26 * [" "]];
   INT .ierr^line := @err^line '>>' 1; ! int address of err^line
   CALL NUMOUT ( err^line[14], errcode, 10, 10);
   CALL WRITE( termnum, ierr^line,30);
   END;

! This procedure handles errors from the file I/O procedures

! No error recovery or diagnostic message is displayed; only the error
! number is listed.  A finished program would have error recovery, and
! a meaningful diagnostic message would be displayed.

 PROC filerr (fnum);
   INT fnum;
   BEGIN
   STRING err^line [0:39] := ["error^code=", 28 * [" "]];
   INT .ierr^line := @err^line '>>' 1; ! int address of err^line
   CALL NUMOUT ( err^line[14], fnum, 10, 10);
   CALL WRITE( termnum, ierr^line,30);
   END;
?page

! hold puts the current job in the hold state

PROC hold (holdp);
   INT holdp;
   BEGIN
   INT error^code;

   IF holdp THEN
      BEGIN
      ! Put the job in the hold state
      error^code := SPOOLERCOMMAND(supernum,2,current^job,122);
      IF error^code <> 0 THEN
         BEGIN
         CALL err(error^code);
         ! Error handling
         END;
      END
   ELSE
      BEGIN
      ! Start the job
      error^code := SPOOLERCOMMAND(supernum,2,current^job,115);
      IF error^code <> 0 THEN
         BEGIN
         CALL err(error^code);
         ! Error handling
         END
      END;
   END;
```

```
?page

! This procedure breaks a string at a period or a space and puts
! the characters up to a period or a space in the second string

INT PROC breakstr (str, index, dest);
   STRING .str,
          .dest;
   INT index;
   BEGIN
   INT i,ret;
   i := 0;
! Check for invalid characters or a string longer than 8 characters.
   WHILE ($ALPHA(str[i+index])   OR
          $NUMERIC(str[i+index]) OR
          str[i+index] = "#"     OR
          str[i+index] = "$"     OR
          str[i+index] = "\"      )
          AND i < 8                    DO
      BEGIN
      dest[i] := str[i+index];
      i := i+1;
      END;
   IF i >= 8 OR i <= 0 THEN ret := -1
   ELSE IF str[i+index] = " " OR str[i+index] = %15 THEN ret := 0
   ELSE IF str[i+index] = "." THEN ret := i + index
   ELSE ret := -1;
   RETURN (ret)
   END;

?page
! This procedure returns an integer corresponding to the command
! received in its parameter

INT PROC getcom(comstr);
    STRING .comstr;
   BEGIN
   INT i, j, match;

   FOR i := 0 TO 7 DO IF $ALPHA(comstr[i]) THEN
                      comstr[i] := comstr[i] LAND %737;
   match := 1;
   FOR i := 1 TO 12 DO
      BEGIN
      FOR j := 0 to 7 DO
         BEGIN
         IF match THEN
         IF command[(i*8)+j] <> comstr[j] THEN match := 0;
         END;
      IF match THEN RETURN (IF i > 6 THEN i-6 ELSE i);
      match := 1;
      END;
   RETURN (0);
   END;

?page
! comint reads a command from the terminal, decides which command
! it is, and separates the parameters.

PROC comint(command, jobn, location, page, number,lines, hold, device);
   INT .command,
       .jobn,
       .location,
       .page,
       .number,
```

```
        .lines,
        .hold,
        .device;
    BEGIN
    STRING str [0:79] := 80*[" "],
           comstr [0:7] := 8 * [" "];
    INT temp, ptr, count, status,devcnt;

    SUBPROC error (errcode);
    INT errcode;
       BEGIN
       CALL err(errcode);
       command := 0;
       END;

    str[0] := "_";
    CALL WRITEREAD (termnum, str, 1, 80,count);
    IF <> THEN CALL filerr(termnum)
    ELSE
       BEGIN
 ! Remove leading spaces from command line
          WHILE str [0] = " " AND count > 0 DO
             BEGIN
             str [0] ':=' str[1] FOR count;
             count := count-1;
             END;
          temp := breakstr (str, 0, comstr);
          IF temp <> 0 THEN CALL error(1)
          ELSE
             BEGIN
 ! Remove command from command line
             WHILE str [0] <> " " AND str [0] <> %15 AND count > 0 DO
                BEGIN
                str [0] ':=' str[1] FOR count;
                count := count-1;
                END;
 ! Remove spaces between command and parameters
             WHILE str [0] = " " AND count > 0 DO
                BEGIN
                str [0] ':=' str[1] FOR count;
                count := count-1;
                END;
             command := getcom(comstr);
! Remove parameters from str
             CASE command OF
                BEGIN

 ! 0 Bad Command !
                CALL error(1);

 !1 LOC!        BEGIN
                location[0] := "  ";
                location[1] ':=' location[0] FOR 7;
                IF count = 0 THEN location ':=' "#DEFAULT"
                ELSE
                   BEGIN
                   temp := breakstr(str, 0, location);
                   IF temp < 0 THEN CALL error (2)
                   ELSE IF temp > 0 THEN
                      BEGIN
                      temp := temp + 1;
                      temp := breakstr(str, temp, location[4]);
                      IF temp <> 0 THEN CALL error (2);
                      END;
                   END;
                END;
```

```
!2 HOLD!        BEGIN
                IF count = 0 THEN hold := 1
                ELSE IF str[0] = "O" OR str[0] = "o" THEN
                   BEGIN
                   IF str[1] = "N" OR str[1] = "n" THEN hold := 1
                   ELSE
                   IF str[1] = "F" OR str[1] = "f" AND
                      str[2] = "F" OR str[2] = "f" THEN hold := 0
                      ELSE CALL error (2);
                   END
                END;

!3 JOB!         BEGIN
                IF count = 0 THEN jobn := 0
                ELSE
                   BEGIN
                   CALL NUMIN (str,jobn, 10, status);
                   IF status <> 0 THEN CALL error(2)
                   ELSE IF jobn > 4095 OR jobn < 1 THEN CALL error (2);
                   END;
                END;

!4 DEV!         BEGIN
                devcnt:= 0;
                device[0] := "  ";
                device[1] ':=' device[0] FOR 15;
                IF count <> 0 THEN
                   BEGIN
                   ! Test for local or remote device
                   IF str[0] = "$" THEN devcnt := 4;
                   temp := breakstr(str, 0, device[devcnt]);
                   IF temp < 0 THEN CALL error (2)
                   ELSE IF temp > 0 THEN
                      BEGIN
                      temp := temp + 1;
                      devcnt := devcnt + 4;
                      temp := breakstr(str, temp, device[devcnt]);
                      IF temp < 0 THEN CALL error (2)
                      ELSE IF temp > 0 THEN
                         BEGIN
                         temp := temp + 1;
                         devcnt := devcnt + 4;
                         temp := breakstr(str, temp, device[devcnt]);
                         IF temp < 0 THEN CALL error (2)
                         ELSE IF temp > 0 AND devcnt <= 12 THEN
                            BEGIN
                            temp := temp + 1;
                            devcnt := devcnt + 4;
                            temp := breakstr(str, temp, device[devcnt]);
                            IF temp <> 0 THEN CALL error (2);
                            END;
                         END;
                      END;
                   END;
                END;

!5 LIST!        BEGIN
                CALL NUMIN (str, page, 10, status);
                IF status <> 0 THEN CALL error (2)
                ELSE
                   BEGIN
                   WHILE $NUMERIC(str[0]) DO str[0] ':=' str[1] FOR count;
                   IF str[0] = "/" THEN
                      BEGIN
                      str[0] ':=' str[1] FOR count;
```

```
                        CALL NUMIN (str,number, 10, status);
                        IF status <> 0 THEN CALL error(2)
                        ELSE
                           BEGIN
                           number := (number - page) + 1;
                           IF number < 1 THEN CALL error(2);
                           END;
                        END
                     ELSE number := 1;
                     END;
                  END;
 !6 EXIT!         OTHERWISE     ;
            END;
         END;
      END;
   END;

?page
! loc changes the location of the current job

PROC loc (location);
   INT .location;
   BEGIN
   INT error^code;

   job.number := current^job;
   ! Get current state of job
   error^code := SPOOLERSTATUS(supernum,2,0,job);
   IF error^code <> 0 THEN
      BEGIN
      ! Error handling
      CALL err(error^code);
      END
   ELSE
      BEGIN
      ! If job is in ready state, put it in the hold state first
      IF job.state = 2 THEN
        BEGIN
        CALL hold(1);
        ! Change the location of the job
        error^code := SPOOLERCOMMAND(supernum, 2, current^job,
                                                 125, location);
        IF error^code <> 0 THEN
           BEGIN
           ! Error handling
           CALL err(error^code);
           END
        ! Restart job
        ELSE CALL hold(0);
        END
      ELSE ! Or else it is already in hold, open, or print state
         BEGIN
         IF job.state = 4 THEN CALL err(7)  !job is printing now
         ELSE
            BEGIN
            ! Change the location of the job
            error^code :=
SPOOLERCOMMAND(supernum,2,current^job,125,location);
            IF error^code <> 0 THEN
               BEGIN
               ! Error handling
               CALL err(error^code);
               END;
            END;
         END;
      END;
```

```
   END;

?page

! jobchange changes the current job to the one specified in the
! parameter.  If jobnum is not owned by the current user, then no change
! is made.

PROC jobchange (jobnum);
   INT jobnum;
   BEGIN
   INT error^code,
       temp;
   ! Get the status of the specified job
   job.number := jobnum;
   error^code := SPOOLERSTATUS(supernum, 2, 0, job);
   IF error^code <> 0 THEN
      BEGIN
      CALL err(error^code);
      ! Error handling
      END
   ELSE
      BEGIN
      ! If owner ID matches, then make the specified job the current job.
      IF job.owner^id = userid THEN
            current^job := jobnum
      ELSE
         BEGIN
         CALL err(6156);
         ! 6156 = security violation
         END;
      END;
   END;
?page

! dev displays the status of a device and the jobs in its queue.

PROC dev(device);
   INT .device;
   BEGIN
   INT error^code,
       temp,
       line^total := 0;
   ! Prepare the header line of the display
   line[0] := " ";
   line[1] ':=' line[0] FOR 79;
   line[0] ':=' "DEVICE STATE: ";
   line[28] ':=' "FORM:";
   dev^stat.name ':=' device FOR 16;
   ! Get the status of the specified device
   error^code := SPOOLERSTATUS(supernum, 1, 0, dev^stat);
   IF error^code <> 0 THEN call err(error^code)
   ELSE
   ! Produce the display
      BEGIN
      CASE dev^stat.state OF
         BEGIN
  !0!    call err(dev^stat.state);
  !1!    line[14] ':=' "WAITING";
  !2!    line[14] ':=' "PRINTING";
  !3!    line[14] ':=' "SUSPENDED";
  !4!    line[14] ':=' "DEVERROR";
  !5!    line[14] ':=' "OFFLINE";
  !6!    line[14] ':=' "PROCERROR";
  OTHERWISE CALL err (dev^stat.state);
         END;
```

```
      line[34] ':=' dev^stat.form^name FOR 8;
      CALL WRITE (termnum, iline, 80);
      line[0] := " ";
      line[1] ':=' line[0] FOR 79;
      line[2] ':=' "JOB";
      line[8] ':=' "OWNER";
      line[17] ':=' "PAGES";
      line[24] ':=' "WAIT";
      line[35] ':=' "FORM";
      CALL WRITE (termnum, iline, 0);
      CALL WRITE (termnum, iline, 80);
      ! Get and display the status of the jobs waiting to print on the
      ! device
      job^status.device^name ':=' device FOR 16;
      job^status.sequence^number := 0;
      error^code := SPOOLERSTATUS(supernum, 7, 1, job^status);
      WHILE error^code = 0 DO
         BEGIN
         line[0] := " ";
         line[1] ':=' line[0] FOR 79;
         CALL NUMOUT(line[2], job^status.job^number,10, 4);
         temp := job^status.owner^id.<0:7>;
         CALL NUMOUT(line[8], temp, 10, 3);
         temp := job^status.owner^id.<8:15>;
         line[11] := ",";
         CALL NUMOUT(line[12], temp, 10, 3);
         CALL NUMOUT(line[17], job^status.pages,10, 4);
         line^total := line^total +
                           (job^status.total^lines/dev^stat.speed);
         CALL NUMOUT(line[24], line^total, 10, 6);
         line[35] ':=' job^status.form^name FOR 16;
         CALL WRITE( termnum, iline, 60);
         error^code := SPOOLERSTATUS(supernum, 7, 1, job^status);
         END;
   IF error^code <> %14006 THEN CALL err(error^code);
      END;
   END;

?page
! devall displays the status of all devices in the system.

PROC devall;
   BEGIN
   INT error^code,
       i, j, temp,
       line^total := 0;
! Prepare the header line of the display
   line[0] := " ";
   line[1] ':=' line[0] FOR 79;
   line[0] ':=' "DEVICE";
   line[36] ':=' "STATE";
   line[51] ':=' "FLAGS";
   line[58] ':=' "PROC";
   line[66] ':=' "FORM";
   CALL WRITE (termnum, iline, 80);
   dev^stat.name ':=' [ 16 * ["  "] ];
! Get and display the status of all devices in the spooler
   error^code := SPOOLERSTATUS(supernum, 1, 1, dev^stat);
   WHILE error^code = 0 DO
      BEGIN
      line[0] := " ";
      line[1] ':=' line[0] FOR 79;
      j := 0;
      FOR i := 0 to 16 DO
         BEGIN
         IF dev^stat.name[i].<0:7> <> " " THEN
```

```
            BEGIN
            IF j <> 0 AND  (i = 4 OR i = 8 OR i = 12) THEN
               BEGIN
               line[j] := ".";
               j := j + 1;
               END;
            line [j] := dev^stat.name[i].<0:7>;
            j := j +1;
            END;
         IF dev^stat.name[i].<8:15> <> " " THEN
            BEGIN
            line [j] := dev^stat.name[i].<8:15>;
            j := j+1;
            END;
         END;
      CASE dev^stat.state OF
         BEGIN
  !0!    call err(dev^stat.state);
  !1!    line[36] ':=' "WAITING";
  !2!    line[36] ':=' "PRINTING";
  !3!    line[36] ':=' "SUSPENDED";
  !4!    line[36] ':=' "DEVERROR";
  !5!    line[36] ':=' "OFFLINE";
  !6!    line[36] ':=' "PROCERROR";
  OTHERWISE CALL err (dev^stat.state);
         END;
      IF dev^stat.flags.<10> THEN line[51] :="T";
      IF dev^stat.flags.<12> THEN line[52] :="D";
      IF dev^stat.flags.<13> THEN line[53] :="H";
      IF dev^stat.flags.<14> THEN line[54] :="E";
      IF dev^stat.flags.<15> THEN line[51] :="F";
      line[58] ':=' dev^stat.print^process FOR 6;
      line[66] ':=' dev^stat.form^name FOR 16;
      CALL WRITE (termnum, iline, 80);
      error^code := SPOOLERSTATUS(supernum, 1, 1, dev^stat);
      END;
   IF error^code <> %14006 THEN CALL err(error^code) ;
   END;

?page
! jobstat displays the status of all jobs owned by the current user.

PROC jobstat;
   BEGIN
   INT error^code,
       temp;
! Prepare the header line of the display
   line[0] := " ";
   line[1] ':=' line[0] FOR 79;
   line[2] ':=' "JOB";
   line[8] ':=' "STATE";
   line[15] ':=' "PAGES";
   line[22] ':=' "COPIES";
   line[30] ':=' "PRI";
   line[35] ':=' "HOLD";
   line[41] ':=' "LOCATION";
   line[57] ':=' "REPORT";
   CALL WRITE( termnum, iline, 0);
   CALL WRITE( termnum, iline, 79);
   job.number := 0;
   job.number.<0> := 1;
   job.owner^id := CREATORACCESSID;
   ! Get the status of each job and display it
   error^code := SPOOLERSTATUS(supernum, 2, 1, job);
   WHILE error^code = 0 DO
      BEGIN
```

```
      IF job.owner^id = CREATORACCESSID THEN
         BEGIN
         line[0] := " ";
         line[1] ':=' line[0] FOR 79;
         IF job.number = current^job THEN line[1] := "J";
         CALL NUMOUT(line[2], job.number, 10, 4);
         CASE job.state OF
            BEGIN
     !0!    ;
     !1!    line[8] ':=' "OPEN";
     !2!    line[8] ':=' "READY";
     !3!    line[8] ':=' "HOLD";
     !4!    line[8] ':=' "PRINT";
            END;
         CALL NUMOUT(line[15], job.pages, 10, 5);
         CALL NUMOUT(line[22], job.copies, 10, 5);
         temp := 0;
         temp := job.flags.<13:15>;
         CALL NUMOUT(line[30], temp, 10, 1);
         IF job.state = 1 and job.flags.<9> = 1 THEN
         line[35] := "B";
         IF job.flags.<10> = 1 THEN line[36] := "A";
         line[41] ':=' job.location.group FOR 16;
         line[57] ':=' job.report^name FOR 16;
         CALL WRITE (termnum, iline, 79);
         IF <> THEN CALL filerr(termnum);
         END;
      job.owner^id := CREATORACCESSID;
      job.number.<0> := 1;
      error^code := SPOOLERSTATUS(supernum, 2, 1, job);
      END;
   CALL WRITE( termnum, iline, 0);
   IF <> THEN CALL filerr(termnum);
END;

?page
! This procedure calls PRINTREAD and strips off all SETMODE,
! CONTROL, and CONTROLBUF lines.  It then returns the data line.

INT PROC STRIPLINES(buff, line, count, page);
   INT .buff,
       .line,
       count,
       page;
   BEGIN
   INT err^code;
   err^code := PRINTREAD(buff, line, count, page);
   WHILE (err^code = %12003 OR
         err^code = %12004 OR
         err^code = %12005 ) DO
      err^code := PRINTREAD(buff, line, count,, 0);
   RETURN (err^code)
   END;

?page
! This procedure closes the job currently open for listing and opens
! a new job.  It is called when the current job has been changed since
! the last list command.

PROC open^list^job;
      BEGIN
      INT error^code;
      CALL CLOSE(datanum);
      error^code := SPOOLEREQUEST( supernum,current^job,message);
      IF error^code <> 0 THEN
         BEGIN
```

```
            ! Error handling
            CALL err(error^code);
            END
         ELSE
            BEGIN
            error^code := PRINTREADCOMMAND(message, ,,,,,,datafile);
            IF error^code <> 0 THEN
               BEGIN
               CALL err(error^code);
               ! Error handling
               END
            ELSE
               BEGIN
               CALL OPEN (datafile,datanum,%2000,1);
               IF  <>  THEN
                  BEGIN
                  ! Error handling
                  CALL filerr(-1);
                  END
               ELSE
                  BEGIN
                  error^code := PRINTSTART(jobbuff, message,datanum);
                  IF error^code <> 0 THEN
                     BEGIN
                     CALL err(error^code);
                     ! Error handling
                     END
                  ELSE list^job := current^job;
                  END;
               END;
            END;
         END;
?page
! LIST performs the listing operation. It first decides whether the
! correct job is open, then it decides whether a set of pages or lines
! should be printed, and then it lists the requested data.

PROC LIST (page, number, lines);
   INT page,
       number,
       lines;
   BEGIN
   INT error^code,
       current^page,
       I;

! If the job currently open does not match the current job, then
! call open^list^job to close the one job and open the other.

   IF list^job <> current^job THEN CALL open^list^job;

   IF page > 0 THEN
      BEGIN
      error^code := STRIPLINES(jobbuff, dline, 900, page);
      IF error^code <> 0 THEN
         BEGIN
         CALL err(error^code);
         ! Error handling
         END
      ELSE
         BEGIN
         error^code := PRINTINFO(jobbuff, , current^page);
         IF error^code <> 0 THEN
            BEGIN
            CALL err(error^code);
            ! Error handling
```

```
            END
         ELSE
            BEGIN
            WHILE ((page + number) > current^page)
            AND (error^code = 0) DO
               BEGIN
               CALL WRITE(termnum, dline, 80);
               error^code := STRIPLINES(jobbuff, dline, 900, 0);
               IF error^code <> 0 AND error^code <> %14002 THEN
                  BEGIN
                  ! Error handling
                  CALL err(error^code);
                  END;
               error^code := PRINTINFO(jobbuff, , current^page);
               IF error^code <> 0 AND error^code <> %14002 THEN
                  BEGIN
                  ! Error handling
                  CALL err(error^code);
                  END
               END;
            END;
         END;
      END
   ELSE ! request for lines not pages
      BEGIN
      FOR i := 1 to lines DO
         BEGIN
         error^code := STRIPLINES(jobbuff, dline, 900, page);
         IF error^code <> 0 THEN
            BEGIN
            ! Error handling
            CALL err(error^code);
            END
         ELSE CALL WRITE(termnum, dline, 80);
         END;
      END;
   END; ! LIST

?page
! INIT initializes global values and opens the terminal and supervisor.
! It also opens $RECEIVE and reads the Startup message.
! Current^job is set to the job most recently opened.
PROC INIT;
   BEGIN
   INT timestamp[0:2] := 3*[0],
       error^code := 0,
       i;
   userid := CREATORACCESSID;
   CALL OPEN(recname, recnum);
   IF <> THEN CALL filerr(-1);
   CALL READ(recnum,ci^id,64);
   IF <> THEN CALL filerr(recnum);
   CALL MYTERM(termname);
   CALL OPEN(termname, termnum);
   IF <> THEN CALL filerr(-1);
   CALL OPEN(supername, supernum);
   IF <> THEN CALL filerr(-1);

   job.number := current^job := 0;

! Find job most recently opened

   job.number.<0> := 1;
   job.owner^id := userid;
   error^code := SPOOLERSTATUS(supernum, 2, 1, job);
   IF error^code <> 0 THEN CALL err(error^code)
```

```
   ELSE
      BEGIN
      WHILE error^code = 0 DO
         BEGIN
         IF userid = job.owner^id THEN
            BEGIN
            I :=0;
            WHILE job.time^opened[i] = timestamp[i] DO I:=i+1;
            IF job.time^opened[i] > timestamp[i] THEN
               BEGIN
               current^job := job.number;
               timestamp ':=' job.time^opened FOR 3;
               END;
            END;
         job.number.<0> := 1;
         job.owner^id := userid;
         error^code := SPOOLERSTATUS(supernum, 2, 1, job);
         END;
      END;
   CALL jobstat;
   END;
?page
PROC ROOT MAIN;
   BEGIN
   INT count := 0,
       command := 0,
       jobn := 0,
       page := 0,
       number := 0,
       lines := 0,
       holdp := 0,
       location [0:7] := 8 * ["  "],
       device [0:15] := 16 * [" "];

   STRING formn [0:15] := 16 * [" "],
          reportn [0:15] := 16 * [" "];
   CALL INIT;
   WHILE 1 DO
      BEGIN
      CALL comint( command,
                   jobn,
                   location,
                   page,
                   number,
                   lines,
                   holdp,
                   device);

      CASE command OF
         BEGIN
!0 no command! ;
!1 LOC !       CALL loc (location);
!2 HOLD!       CALL hold(holdp);
!3 JOB !       IF jobn = 0 THEN CALL jobstat ELSE CALL jobchange(jobn);
!4 DEV !       IF device = [ 16 * ["  "] ] THEN CALL devall
                  ELSE CALL dev(device);
!5 LIST!       CALL list(page, number, lines);
!6 EXIT!       CALL STOP;
OTHERWISE      BEGIN
               CALL err(27);
               CALL STOP;
               END;
         END;
      END;
   END;
```

C Spooler-Related Errors

This appendix explains the error codes returned by the spooler interface, print, and utility procedures. These codes are listed numerically by type.

Each spooler procedure is a type INT function that returns an octal spooler error code indicating its outcome. The leftmost part of the error number indicates the source of the error:

%1000 = File-system errors
%2000 = Collector file errors
%3000 = Spool control file errors
%4000 = Device errors
%5000 = Print errors
%100000 = NEWPROCESS errors or PROCESS_CREATE_ errors

You can extract the relevant file-system error number from most of these error codes by examining bits <8:15>. For example, if the error code returned is %1074, bits <8:15> contain file-system error %074 (device has been downed).

Some of the common file-system error codes are described in this appendix. All other file-system error codes are described in the *System Procedure Errors and Messages Manual*.

The NEWPROCESS error codes and the PROCESS_CREATE_ errors are described in the *Guardian Procedure Errors and Messages Manual*. The NEWPROCESS error codes should not be confused with the PROCESS_CREATE_ errors. For example, when a NEWPROCESS error is encountered, the cause is often included in bits <8:15> of the error code. With PROCESS_CREATE_ errors, the cause of the error is often returned in the *error-detail* parameter.

Interface Errors

The following error codes are returned by the spooler interface procedures.

```
0     (%0)
```

Cause. The operation was completed.

Effect. None.

Recovery. None.

```
512-767     (%1000 - %1377)
```

Cause. The collector encountered a file-system error.

Effect. The spooler ignores the request.

Recovery. Refer to the information on file-system errors in the *Guardian Procedure Errors and Messages Manual* for corrective action for the error number indicated in bits <8:15>.

4096 (%10000)

Cause. A parameter is missing.

Effect. The spooler ignores the request.

Recovery. Correct the syntax and reenter the command.

4097 (%10001)

Cause. The content of a parameter is wrong, or both *filenum* and *filenum-to-collector* were specified (only one can be specified).

Effect. The spooler ignores the request.

Recovery. Correct the syntax and reenter the command.

4098 (%10002)

Cause. The format of a spooler job file (file code 129) is invalid. A job from a previous session is not correctly formatted.

Effect. The spooler ignores the request.

Recovery. Purge the spooler job file and build a new one.

4608 (%11000)

Cause. A checkpoint exit occurred. This error code has special significance for an application running as a NonStop process pair. If bit 11 of the *flags* parameter of SPOOLSTART was set to 1, the procedures SPOOLWRITE, SPOOLCONTROL, SPOOLCONTROLBUF, and SPOOLSETMODE return a value of %11000 when the level-3 buffer is about to be written to the collector.

Effect. The block is complete and ready to be written to the data file.

Recovery. Checkpoint the buffer and repeat the call.

4609 (%11001)

Cause. A process attempted to write to the collector without first opening the file. This error occurs if the collector was not opened successfully.

Effect. The HP NonStop Kernel operating system returns a -1 as the file number on a failed open.

Recovery. Correct your program to verify that the collector has been opened successfully.

File-System Errors

The following file-system error codes have special significance to a process sending data to a collector. An application spooling at level 1 or 2 gets these errors from the write or the open procedure, while an application spooling at level 3 obtains these errors in bits <8:15> of a spooler error code in the range %1000 through %1377.

0	(%0)	OPERATION SUCCESSFUL

Cause. The operation completed successfully.

Effect. None.

Recovery. None.

2	(%2)	OPERATION NOT ALLOWED ON THIS TYPE FILE

Cause. The collector received an invalid message. For example, SPOOLSTART was called for a job that is already open, SPOOLEND was called for a job that is not open, or the file-system WRITE[X] procedure was called for a job being spooled at level 3.

Effect. The collector ignores the invalid message.

Recovery. Correct the error in your program.

17	(%21)	ATTEMPTED CHECKOPEN, PARAMETERS DO NOT MATCH PRIMARY OPEN

Cause. Illegal parameters were specified in a call to CHECKOPEN.

Effect. The spooler ignores requests from the backup process.

Recovery. Correct your call to the CHECKOPEN procedure for the collector file.

21	(%25)	ILLEGAL COUNT SPECIFIED

Cause. A data line written to the spooler was too long.

Effect. The write is not performed.

Recovery. Decrease the byte count and reenter the command.

28	(%34)	NUMBER OF OUTSTANDING NOWAIT OPERATIONS WOULD EXCEED THAT SPECIFIED

Cause. The collector was opened with an illegal sync or nowait depth.

Effect. The open is ignored.

Recovery. Make sure that the sync depth is less than or equal to 32 and that bits <12:15> of the OPEN *flags* parameter contain a value less than or equal to 1.

44	(%54)	DISK DIRECTORY IS FULL; DCT IS FULL

Cause. The collector could not obtain enough of some resource (internal buffer space, job numbers, and so on) to accept a job. For example, more than 32 jobs at level 1 or 2 are opened to a collector.

Effect. The spooler ignores the request.

Recovery. Use Spoolcom to check the collector status. Check your program for the number of jobs and reduce that number where possible.

For process files, the system might not create any newly named process until at least one existing named process has stopped.

If the problem is a spooler problem (caused by too many opens for the collector), consider starting another collector. See the *Spooler Utilities Reference Manual* for more information.

45	(%55)	FILE IS FULL

Cause. The collector does not have room in its data file for user data.

Effect. The spooler ignores your request.

Recovery. Periodically retry the operation that returned this error. Also contact your system operator. You might want to increase the size of your collector data file.

60	(%74)	DEVICE DOWNED OR NOT OPENED, OR PROCESS HAS FAILED SINCE IT OPENED FILE

Cause. Something stopped the collector after a process opened the collector's file.

Effect. None.

Recovery. Try opening the file again. If this action is unsuccessful, try another collector.

```
66    (%102)   DEVICE DOWNED, LIU NOT YET DOWNLOADED,
               OR HARD FAILURE OCCURRED ON CONTROLLER
```

Cause. The collector is not accepting open requests. This error can occur for the following reasons:

- The system operator brought down the specified device.
- A hard error occurred on the device controller.
- The spooler collector process sent a spooled job to a nonmirrored disk.
- Both halves of a mirrored disk are down.

Effect. A Spoolcom user has issued a SPOOLER DRAIN command. The collector finishes accepting any jobs that were open when the command was issued, but no new jobs can start.

Recovery. The same Spoolcom user must issue a SPOOLER START command.

```
100     (%144)    DEVICE NOT READY OR CONTROLLER NOT
                  OPERATIONAL
                  (device type:  any except 2)
```

Cause. This error can occur for the following reasons:

- The device was not powered up or is not online.
- A printer is out of paper or is not working.
- A card reader is out of cards.
- A tape drive is accessed while rewinding.
- A tape drive is at a load point but is not online.
- A heavily loaded processor receives a call to open a server process but cannot respond.

Effect. The procedure sets the error code and returns without performing the requested operation.

Recovery. Make the device ready. In the case of a heavily loaded processor, repeat the call to open the process.

```
102     (%146)    PAPER OUT, BAIL OPEN, OR END OF RIBBON
```

Cause. A printer could not continue because it was out of paper or because the paper bail was not in place.

Effect. The procedure sets the error code and returns without performing the requested operation.

Recovery. Load more paper, close the bail, or replace the ribbon as needed.

140	(%214)	MODEM ERROR (COMMUNICATION LINK NOT YET ESTABLISHED, MODEM FAILURE, MOMENTARY LOSS OF CARRIER, OR DISCONNECT)

Cause. A modem error occurred. For example, the communications link is not yet established, a modem failure occurred, a momentary loss of carrier occurred, the modem or link is disconnected, the interprocessor bus monitor process ($IPB) reported that the FOX link to an Expand line-handler process is down, or a subunit or logical unit is not in the started condition.

Effect. The procedure sets the error code and returns without performing the requested operation.

Recovery. Corrective action is device-dependent. If the problem is still not apparent, submit the trace, OPRLOG, CONFLIST, and subunit and line configuration to your HP representative.

201	(%311)	CURRENT PATH TO DEVICE IS DOWN ATTEMPT WAS MADE TO WRITE TO A NONEXISTENT PROCESS, OR ERROR IN MESSAGE SYSTEM INTERFACE

Cause. A processor probably failed.

Effect. The collector is opened with a sync depth of 0, and its backup process takes over.

Recovery. You can retry a write, but lines of data might be lost.

Spooler Utility Errors

The spooler utility procedures return the following error codes:

0	(%0)

Cause. The operation completed successfully.

Effect. None.

Recovery. None.

4096	(%10000)

Cause. A parameter is missing.

Effect. The spooler ignores your request.

Recovery. Correct the syntax and reenter the command.

```
6144    (%14000)
```

Cause. An invalid command was issued.

Effect. The spooler ignores your request.

Recovery. Correct the command code and reenter the command.

```
6145    (%14001)
```

Cause. A command parameter is missing.

Effect. The spooler ignores your request.

Recovery. Correct the command syntax and reenter the command.

```
6146    (%14002)
```

Cause. A command parameter is in error.

Effect. The spooler ignores your request.

Recovery. Correct the parameter value and reenter the command.

```
6147    (%14003)
```

Cause. An invalid subcommand was issued.

Effect. The spooler ignores your request.

Recovery. Correct the subcommand code and reenter your command.

```
6148    (%14004)
```

Cause. A subcommand is missing.

Effect. The spooler ignores your request.

Recovery. Correct the syntax and reenter the command.

```
6149    (%14005)
```

Cause. A subcommand parameter is in error.

Effect. The spooler ignores your request.

Recovery. Correct the parameter value and reenter the command.

```
6150    (%14006)
```

Cause. The program reached the end of the SPOOLERSTATUS entries.

Effect. The spooler ignores your request.

Recovery. If this value is expected, no action is needed. Otherwise, correct the program so it handles end-of-entries.

```
6151    (%14007)
```

Cause. SPOOLERSTATUS or SPOOLEREQUEST could not find an entry.

Effect. The spooler ignores your request.

Recovery. If this value is expected, no action is needed. If not, issue a command for a valid entry or check the application to determine why the error occurred.

```
6152    (%14010)
```

Cause. SPOOLERSTATUS could not add an entry to the tables.

Effect. The spooler ignores your request.

Recovery. If necessary, expand the spooler configuration during the next coldstart.

```
6153    (%14011)
```

Cause. The requested entry could not be found.

Effect. The spooler ignores your request.

Recovery. If this value is expected, no action is needed. If not, issue a command for a valid entry or check the application to determine why the error occurred.

```
6154    (%14012)
```

Cause. The entry is not in the proper state for the requested operation.

Effect. The spooler ignores your request.

Recovery. Either change the entry state or wait until the state changes.

```
6155    (%14013)
```

Cause. Because the entry is in use, it cannot be deleted.

Effect. The spooler ignores your request.

Recovery. Wait until the entry becomes available and then delete it.

```
6156    (%14014)
```

Cause. The attempted request failed because of a security violation.

Effect. The spooler ignores your request.

Recovery. To execute the request, log on as an authorized user with execute access to the Spoolcom program.

```
6157    (%14015)
```

Cause. The requested process was not a spooler supervisor.

Effect. The command fails to establish communication with the spooler.

Recovery. Use a correct supervisor name and retry the request.

```
6158    (%14016)
```

Cause. A SPOOLERSTATUS request was in progress.

Effect. The spooler ignores your request.

Recovery. Call SPOOLERSTATUS again.

```
6159    (%14017)
```

Cause. There was an attempt to delete a job that was associated with a font. This is not allowed.

Effect. The operation is ignored.

Recovery. Delete the font associated with the job in question and retry the operation.

```
6160    (%14020)
```

Cause. A JOB command issued through the SPOOLERCOMMAND procedure cannot be performed on a portion of a spooler batch job.

Effect. The operation is ignored.

Recovery. Perform the operation on the spooler batch job, or unlink the spooler job from the spooler batch job and retry the operation.

```
6161    (%14021)
```

Cause. An attempt was made to link a spooler job whose attributes (owner, form, device, and gmom-crtpid-jobid) do not match those of the spooler batch job.

Effect. The operation is ignored.

Recovery. Change the key attributes of the spooler job to match those of the spooler batch job.

```
6162    (%14022)
```

Cause. An attempt was made to link a spooler job to a spooler batch job, but the spooler job already belonged to the spooler batch job.

Effect. The operation is ignored.

Recovery. None.

```
6163    (%14023)
```

Cause. There was an attempt to unlink a spooler job from a spooler batch job although they were not linked.

Effect. The operation is ignored.

Recovery. This is an informative message only; no action is needed.

```
32768-36771    (%100000 - %107477)
```

Cause. An error occurred while the system was creating a collector or print process.

Effect. The spooler ignores your request.

Recovery. Refer to the information about the PROCESS_CREATE_ errors *Guardian Procedure Errors and Messages Manual* for corrective action concerning error information returned in the *error-detail* parameter.

Print Procedure Errors

Consider the following when you handle codes returned by print procedures:

- Error codes returned by PRINTREAD that are not listed below are fatal errors. In such cases, terminate the job and send the error to the supervisor process.
- Use a negative *pagenum* parameter in the call to PRINTREAD if you want PRINTREAD to return the number of the line where a WRITE[X] to a device failed.
- When the spooler supervisor process receives a job-end message with an error other than 0 or %12000 (end of file), the supervisor process writes the error to its error log file and puts the job on hold.

The following spooler error codes are relevant to print procedures.

```
0     (%0)
```

Cause. The operation was completed successfully.

Effect. None.

Recovery. None.

```
1024-1279     (%2000-%2377)
```

Cause. The data file has a file-system error.

Effect. The print procedure returns no data.

Recovery. Abort the print process. Refer to the information on file-system errors in the *Guardian Procedure Errors and Messages Manual* for corrective action for the error number indicated in bits <8:15>.

```
1536-1791     (%3000-%3377)
```

Cause. The print procedure encountered a file-system error while attempting to open the supervisor file.

Effect. There is a communications problem with the supervisor.

Recovery. The print procedure can either call ABEND, retry the operation a number of times, or continue reading and printing jobs without any further communication with the supervisor process. Refer to the information on file-system errors in the *Guardian Procedure Errors and Messages Manual* for corrective action for the error number indicated in bits <8:15>.

```
2048-2303     (%4000 - %4377)
```

Cause. There was a file-system error on a spooler device.

Effect. The spooler ignores your request.

Recovery. Refer to the information on file-system errors in the *Guardian Procedure Errors and Messages Manual* for corrective action for the error number indicated in bits <8:15>.

2560-2815 (%5000 - %5377)

Cause. There was a file-system error on a file to the print process.

Effect. The spooler ignores your request.

Recovery. Refer to the information on file-system errors in the *Guardian Procedure Errors and Messages Manual* for corrective action for the error number indicated in bits <8:15>.

4096 (%10000)

Cause. A parameter is missing.

Effect. The print procedure ignores your command.

Recovery. Correct the syntax and reenter the command.

4097 (%10001)

Cause. A parameter is in error.

Effect. None.

Recovery. Correct the parameter value and reenter the command.

5120 (%12000)

Cause. A print procedure reached the end of the file.

Effect. If this error code is returned by the PRINTREAD procedure, all lines in the job have been transferred.

Recovery. Send an “end job” message to the supervisor, using the PRINTSTATUS procedure, to delete the job.

5121 (%12001)

Cause. A print procedure reached the end of the copy.

Effect. If this error code is returned by the PRINTREAD procedure, the next line returned is the first line of the next copy of the job. The data line does not contain valid data when PRINTREAD returns an end-of-copy indication.

Recovery. A print procedure can choose to either print a header or do a top-of-form when an end-of-copy indication is returned.

5122 (%12002)

Cause. A print procedure encountered an invalid data file. The data file from which PRINTREAD is attempting to read data does not contain the correct job.

A perusal process can receive this error from the PRINTREAD procedure if the job it is reading is deleted during the read.

Effect. There is no effect. PRINTREAD did not return data.

Recovery. If a print procedure receives this error, it should terminate the job and send the error to the supervisor using the PRINTSTATUS procedure.

If a perusal process receives this error, refer to the *Guardian Procedure Calls Reference Manual* for information about the SPOOLEREQUEST procedure.

5123 (%12003)

Cause. The data line contained a file-system CONTROL operation.

Effect. If this error code is returned by the PRINTREAD procedure, the data line contains a file-system CONTROL message for the print device. The format of the CONTROL message is

data-line [0] = operation
data-line [1] = parameter

Recovery. The print procedure should pass the CONTROL message to the device by way of the file-system CONTROL procedure. The CONTROL procedure is documented in the *Guardian Procedure Calls Reference Manual*.

5124 (%12004)

Cause. The data line contained a file-system SETMODE operation.

Effect. If this error code is returned by the PRINTREAD procedure, the data line contains a file-system SETMODE instruction. The SETMODE instruction format is

data-line [0] = SETMODE function
data-line [1] = *param-1*
data-line [2] = *param-2*

Recovery. The print procedure should issue a SETMODE operation to the device. You can find the syntax for the SETMODE procedure in the *Guardian Procedure Calls Reference Manual*.

```
5125    (%12005)
```

Cause. The data line contained a file-system CONTROLBUF operation.

Effect. If this error code is returned by the PRINTREAD procedure, the data line contains a file-system CONTROLBUF message for the print device. The format of the CONTROLBUF message is

data-line [0] = operation
data-line [1] = buffer address
data-line [2] = count operation

Recovery. The print procedure should pass the CONTROLBUF message to the device by way of the file-system CONTROLBUF procedure. The CONTROLBUF procedure is documented in the *Guardian Procedure Calls Reference Manual*.

```
5632    (%13000)
```

Cause. The print process does not recognize a device name that was received in a supervisor request.

Effect. The spooler takes the device in question offline. If the error occurs before the job has finished printing, the job remains in the spooler print queue. If the job has finished printing but the device has not been closed, the job is deleted from the print queue.

Recovery. Using Spoolcom, restart the device (for example, issue the DEV $LP, START command).

If the error persists, stop and then restart the print process.

If the error occurs after you have stopped and restarted the print process, shut down the spooler and then warmstart it.

```
5633    (%13001)
```

Cause. The print process received a request from the supervisor to open a device that is already open.

Effect. The spooler takes the device in question offline. If the error occurs before the job has finished printing, the job remains in the spooler print queue. If the job has finished printing but the device has not been closed, the job is deleted from the print queue.

Recovery. Using Spoolcom, restart the device (for example, issue the DEV $LP, START command).

If the error persists, stop and then restart the print process.

If the error occurs after you have stopped and restarted the print process, shut down the spooler and then warmstart it.

5635 (%13003)

Cause. The print process received a request from the supervisor to perform an action on a device that is busy.

Effect. The spooler takes the device in question offline. If the error occurs before the job has finished printing, the job remains in the spooler print queue. If the job has finished printing but the device has not been closed, the job is deleted from the print queue.

Recovery. Using Spoolcom, restart the device (for example, issue the DEV $LP, START command).

If the error persists, stop and then restart the print process.

If the error occurs after you have stopped and restarted the print process, shut down the spooler and then warmstart it.

5636 (%13004)

Cause. The print process received a request from the supervisor to start a job, but the print process has run out of device table space.

Effect. The spooler takes the device in question offline. If the error occurs before the job has finished printing, the job remains in the spooler print queue. If the job has finished printing but the device has not been closed, the job is deleted from the print queue.

Recovery. Using Spoolcom, restart the device (for example, issue the DEV $LP, START command).

If the error persists, stop and then restart the print process.

If the error occurs after you have stopped and restarted the print process, shut down the spooler and then warmstart it.

Index

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

W

Special Characters