VMS

digital

VMS RTL Parallel Processing (PPL$) Manual

Order Number: AA–LA74B–TE

VMS RTL Parallel Processing (PPL$) Manual

Order Number: AA–LA74B–TE

June 1990

This manual documents the parallel processing routines contained in the PPL$ facility of the VMS Run-Time Library.

Revision/Update Information:	This manual supersedes the *VMS RTL Parallel Processing (PPL$) Manual*, Version 5.2.
Software Version:	VMS Version 5.4

digital equipment corporation
maynard, massachusetts

June 1990

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

CDA	DEQNA	MicroVAX	VAX RMS
DDIF	Desktop–VMS	PrintServer 40	VAXserver
DEC	DIGITAL	Q-bus	VAXstation
DECdtm	GIGI	ReGIS	VMS
DECnet	HSC	ULTRIX	VT
DECUS	LiveLink	UNIBUS	XUI
DECwindows	LN03	VAX	
DECwriter	MASSBUS	VAXcluster	digital™

The following is a third-party trademark:

PostScript is a registered trademark of Adobe Systems Incorporated.

ZK4375

Production Note

This book was produced with the VAX DOCUMENT electronic publishing system, a software tool developed and sold by Digital. In this system, writers use an ASCII text editor to create source files containing text and English-like code; this code labels the structural elements of the document, such as chapters, paragraphs, and tables. The VAX DOCUMENT software, which runs on the VMS operating system, interprets the code to format the text, generate a table of contents and index, and paginate the entire document. Writers can print the document on the terminal or line printer, or they can use Digital-supported devices, such as the LN03 laser printer and PostScript printers (PrintServer 40 or LN03R ScriptPrinter), to produce a typeset-quality copy containing integrated graphics.

Contents

INDEX

EXAMPLES

FIGURES

Preface

This manual provides users of the VMS operating system with detailed usage and reference information on parallel processing routines supplied in the PPL$ facility of the Run-Time Library.

Run-Time Library routines can only be used in programs written in languages that produce native code for the VAX hardware. At present, these languages include VAX MACRO and the following compiled high-level languages:

VAX Ada
VAX BASIC
BLISS-32
VAX C
VAX COBOL
VAX COBOL-74
VAX CORAL
VAX DIBOL
VAX FORTRAN
VAX Pascal
VAX PL/I
VAX RPG
VAX SCAN

Interpreted languages that can also access Run-Time Library routines include VAX DSM and DATATRIEVE.

Intended Audience

This manual is intended for system and application programmers who want to call Run-Time Library routines.

Document Structure

This manual is organized into two parts as follows:

- Part I includes chapters that provide guidelines and reference material on PPL$ routines.

  Chapter 1 provides a brief overview of parallel processing.

  Chapter 2 discusses process management and naming operations.

  Chapter 3 describes shared memory operations.

  Chapter 4 discusses synchronization operations.

  Chapter 5 discusses some recommended methods for using the Parallel Processing Facility for developing new programs.

  Chapter 6 contains examples demonstrating how to call some PPL$ routines from major VAX languages.

- Part II provides detailed reference information on each routine contained in the PPL$ facility of the Run-Time Library. This information is presented using the documentation format described in the *Introduction to the VMS Run-Time Library*. Routine descriptions appear in alphabetical order by routine name.

Associated Documents

The Run-Time Library routines are documented in a series of reference manuals. A general overview of the Run-Time Library and a description of how the Run-Time Library routines are accessed are presented in the *Introduction to the VMS Run-Time Library*. Descriptions of the other RTL facilities and their corresponding routines are presented in the following books:

- The *VMS RTL DECtalk (DTK$) Manual*
- The *VMS RTL Library (LIB$) Manual*
- The *VMS RTL Mathematics (MTH$) Manual*
- The *VMS RTL General Purpose (OTS$) Manual*
- The *VMS RTL Screen Management (SMG$) Manual*
- The *VMS RTL String Manipulation (STR$) Manual*

The VAX Procedure Calling and Condition Handling Standard, which is documented in the *Introduction to VMS System Routines*, contains useful information for anyone who wants to call Run-Time Library routines.

Application programmers in any language may refer to the *Guide to Creating VMS Modular Procedures* for the Modular Programming Standard and other guidelines.

High-level language programmers will find additional information on calling Run-Time Library routines in their language reference manuals. Additional information may also be found in the language user's guide provided with your VAX language.

The *Guide to Using VMS Command Procedures* may also be useful.

For a complete list and description of the manuals in the VMS documentation set, see the *Overview of VMS Documentation*.

Conventions

The following conventions are used in this manual:

Ctrl/x	A sequence such as Ctrl/x indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 x	A sequence such as PF1 x indicates that you must first press and release the key labeled PF1, then press and release another key or a pointing device button.
Return	A key name is shown enclosed to indicate that you press a key on the keyboard.
. . .	In examples, a horizontal ellipsis indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
()	In format descriptions, parentheses indicate that, if you choose more than one option, you must enclose the choices in parentheses.
[]	In format descriptions, brackets indicate that whatever is enclosed is optional; you can select none, one, or all of the choices.
{}	In format descriptions, braces surround a required choice of options; you must choose one of the options listed.
red ink	Red ink indicates information that you must enter from the keyboard or a screen object that you must choose or click on. For online versions, user input is shown in **bold**.
boldface text	Boldface text represents the introduction of a new term or the name of an argument, an attribute, or a reason.
italic text	Italic text represents information that can vary in system messages (for example, Internal error *number*).

UPPERCASE TEXT	Uppercase letters indicate one of the following possibilities: • The name of a routine, the name of a file, the name of a file protection code, or the abbreviation for a system privilege. • You must enter a command (for example, enter OPEN/READ). Hyphens in coding examples indicate that additional arguments to the request are provided on the line that follows.
numbers	Unless otherwise noted, all numbers in the text are assumed to be decimal. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

Other conventions used in the documentation of Run-Time Library routines are described in the *Introduction to the VMS Run-Time Library*.

1 Overview of Parallel Processing

Parallel processing occurs when a section of an application is divided into multiple tasks, and those multiple tasks are executed simultaneously on multiple processors. You should not confuse parallel processing with the more widely known computing method called multiprogramming.

Briefly, **multiprogramming** is a mode of operation that lets you share hardware resources among multiple, independent software processes. Timesharing is a common form of multiprogramming. In a timesharing system, each process is given a specific amount of execution time on a processor. When the time for one process has run out, that process is put into a wait state and the next process begins execution for its allotted time, and so on.

You can use parallel processing techniques to implement fault-tolerant systems, to decrease the amount of elapsed time required to execute an application, and to express the inherent logical parallelism in an algorithm. While the term **parallel processing** usually implies a number of processors working together on a particular problem, you can apply the same techniques to a variety of applications, including those that run on a single CPU.

The PPL$ facility offers routines to help you implement concurrent programs on both single-CPU and multiprocessor systems, using VMS processes for parallelism.

1.1 Advantages of Parallel Processing

The PPL$ facility provides routines to simplify many of the tasks commonly required to implement a parallel processing application. The PPL$ routines are designed to work together to help you create and maintain parallel applications. Instead of using all of the common event flag system services, for example, to implement a semaphore, you can use the PPL$ routines that create, read, decrement, increment, and delete a semaphore.

The parallel processing techniques implemented by PPL$ show the greatest performance improvements in applications that are CPU intensive. Areas such as computer-aided design, image processing, high-energy physics, and geophysical research, among others, see a significant lessening of elapsed time when using PPL$ routines. Applications that are I/O intensive will most often not realize any significant decrease in elapsed time, and may even suffer in system performance when executing in parallel. In other words, you must examine every application individually to determine whether or not using the PPL$ routines, or parallel processing in general, is appropriate.

1.2 Definition of Terms

A **process** is the basic entity that is scheduled by the system software. This system software provides the context in which an image executes; for example, the process's quota, privilege, and file context. A process, therefore, consists of an address space and both hardware and software context.

On a VMS system there are two possible types of processes: detached processes and subprocesses. A **detached process** is an independent entity on the system. A **subprocess**, or **subordinate** process, is spawned from another process; therefore a subordinate shares some system resources with its parent process, and it is deleted either when the parent is deleted or when the image that it is executing exits. In the PPL$ facility, a subordinate is defined as a VMS subprocess.

The term **participant** is used to refer to any one of an arbitrary number of independent processes (parent or subordinate) that performs an application-defined piece of work.

One method of ensuring consistent access to program data is to synchronize cooperating processes. **Synchronization** can be described as a set of constraints that affects or controls the ordering of events in your decomposed application. You can use synchronization mechanisms to delay execution of a particular process in order to satisfy any such constraints.

A **synchronization element** is a part of the PPL$ facility that controls the order of processing in a parallel application. A synchronization element can be a barrier, event, semaphore, spin lock, or work queue. An **object** can be a synchronization element or shared memory zone.

Mutual exclusion describes the situation where only one participant at a time is allowed access to a critical section of a parallel task or a critical physical resource. (A physical resource can be a printer or an I/O device, for example.) Mutual exclusion can be implemented using either a spin lock or a semaphore. (Refer to Chapter 4 for more information about spin lock and semaphore synchronization.)

Within VMS, a global section (or shared memory) is a data structure or shareable image section potentially available to all processes in the system. See the *VMS System Services Reference Manual* for more information on global sections.

When a participant is **blocked**, a synchronization element is preventing that participant from executing. A participant can be blocked by a barrier, semaphore, event, spin lock, or work queue. When you specify blocking of a participant, the participant is usually blocked by a PPL$ call to the system service $HIBER, so that ASTs can be delivered. (In the case of spin locks, however, a blocked participant executes a tight loop instead of hibernating.)

The term **critical section** refers to any segment of your program that must be executed only by a single process at a time.

1.3 Characteristics of a Parallel Processing Application

Applications that can benefit from using PPL$ routines will likely be described by at least one of the following characteristics:

- The application runs on a multiprocessing system that consists of two or more processors (CPUs) that can use shared memory (global sections).
- The application represents a single application program that can have several tasks or instructions executing simultaneously across multiple processors.
- The application uses communication and synchronization mechanisms for controlling access to shared variables.

1.4 Software Models for Parallel Processing

The routines provided by the PPL$ facility are based on several software and performance models. This section discusses the models to consider when you design your own parallel applications.

When you begin designing an application for parallel execution, you should structure your program after the parallel processing model that best fits your application. You will find that, in general, your application does not exactly match one particular model, but instead more closely resembles a collection or combination of these models, including

- The master/slave model
- The pipelining model
- The work queue processing model

1.4.1 Master/Slave

The general **master/slave** model of parallel processing has the following characteristics:

- One participant is selected as the master, and that participant is responsible for creating and deleting any subordinates (slaves) required for your application.
- When you separate your application into single-stream and multiple-stream tasks, the master is responsible for executing all of the single-stream tasks and notifying the slave subordinates when multiple-stream tasks are available for execution. Note that the master can also execute some of the parallel code, but is always responsible for the execution of the single-stream code.

All of the characteristics mentioned above hold true for any master /slave software model. However, within this general model there are two different forms of the master/slave model: the true master/slave model, and a self-scheduling master/slave model, sometimes called the queuing model.

1.4.1.1 True Master/Slave Model

In the **true** master/slave model of parallel processing, the master executes all the single-stream tasks and then specifically assigns a multiple-stream task to each slave subordinate. In other words, the master is not only responsible for executing all of the single-stream tasks and notifying the subordinates that multiple-stream tasks are available, but also for assigning a task to each subordinate for execution; the subordinates cannot assign work to themselves.

1.4.1.2 Self-Scheduling Master/Slave Model

In the **self-scheduling** master/slave model, the master is again responsible for executing all the single-stream tasks and notifying the slave subordinates that multiple-stream tasks are available for execution. However, in the self-scheduling master/slave model, the master does not assign tasks to the subordinates. Instead, the master informs the slaves which multiple-stream tasks are available, and each slave subordinate takes a task and executes it. That is, the slave subordinates assign tasks to themselves, although the master is still responsible for the creation of these subordinates as well as the execution of the single-stream code.

1.4.1.3 Synchronization Method

The most common synchronization method to use in the master/slave parallel processing model is barrier synchronization. That is, once the master notifies the slave subordinates that multiple-stream tasks are available for execution, the master waits until all the slaves reach the designated barrier, which is generally at the completion of a set of work items. At that point, the master resumes control and continues to execute the single-stream code. (Refer to Chapter 4 for more information about barrier synchronization.)

1.4.2 Pipelining

The **pipelining** parallel processing model is task oriented. That is, each processor in the system is assigned a specific task, and the data moves from task to task. At each time step, each processor performs its assigned task and then passes the information on to the next task, meanwhile receiving data from the previous task.

You can compare the pipelining model of parallel processing to an assembly line, where the work performed at each station in the line is a task in the pipe, and the piece moving through the assembly line is the piece of data moving through the pipe.

In the ideal situation, all of the stations in an assembly line have equal processing speed, so that once the assembly line is fully loaded it outputs one completed product per clock period. The same is true for a pipelining parallel processing model. Ideally, each task requires the same amount of execution time so that, once fully loaded, the pipe outputs one completed product per clock period. If this is not the case, then the slowest task becomes the bottleneck for the entire pipe. There is, however, a time overhead associated with the initial filling of the pipe, before the first output item appears. This overhead is a function of the number of tasks and the completion time for each task.

Because a pipelining model is task oriented, there are not many synchronization and communication requirements, and those that exist can be satisfied with a message-passing technique such as a mailbox.

1.4.3 Work Queue Processing

The work queue parallel processing model consists of a queue of work items and processes to complete these work items. Each participant can take a work item off the queue, and if necessary, each participant can add newly generated work items to the queue. As each participant completes its work item, it does not wait for some participant to assign it a new task, but instead takes the next item off the work queue and begins execution. The work queue parallel processing model can be combined with other models. For example, work queues can be used in pipelining to carry data between tasks.

The work queue parallel processing model is similar to the self-scheduling master/slave model in that the participants can assign themselves tasks from the queue and execute them to completion. However, there are two major differences. In the self-scheduling master/slave model, the predesignated master participant always executes the single-stream code; the work queue model has a "floating" master, which means that any participant that assigns itself the single-stream code can execute it. The other difference is that, in the self-scheduling master/slave model, if a slave subordinate generates an additional piece of work, it must pass that information back to the master. In the work queue model, any participant that generates additional work items can simply add them to the queue.

A common example of the work queue model of parallel processing is a typing pool. The work that must be done is stored in a bin in the middle of the room, and each typist takes one of these work items and completes it. If that work item in turn generates additional work, the typist puts the additional work items back into the bin and completes execution of the current work item. When the typist has completed the current work item, the typist simply takes the next work item from the bin and performs that task. Again, if that task generates additional work items, those items are placed in the bin for later execution.

1.4.3.1 Synchronization Method

Use the PPL$ routines that manage work queues and work items to achieve synchronization in a work queue model. Refer to Section 4.6 for more information about work queue synchronization.

1.5 System Requirements

This section discusses the privileges and process quotas required by the PPL$ facility.

1.5.1 Privileges

Privileges are not required to use most of the routines in the PPL$ facility. However, four routines require privileges. The following table shows those PPL$ routines and the privileges required to use them.

Routine	Required Privilege
PPL$CREATE_APPLICATION with PPL$M_PERM flag	PRMGBL
PPL$CREATE_APPLICATION with PPL$M_SYSTEM flag	SYSGBL and SYSLCK
PPL$CREATE_SHARED_MEMORY with PPL$M_PERM flag	PRMGBL
PPL$CREATE_SHARED_MEMORY with PPL$M_SYSTEM flag	SYSGBL
PPL$DELETE_APPLICATION	PRMGBL
PPL$DELETE_SHARED_MEMORY with PPL$M_PERM flag	PRMGBL

GROUP/WORLD privileges are required for process control (for example, $WAKE) in applications comprised of processes with different user/group UICs.

1.5.2 Quotas

Before you begin using PPL$, check your process quotas by using the following DCL command:

```
$ SHOW PROCESS/QUOTA
```

The following sections discuss some process quotas that PPL$ may require you to increase.

1.5.2.1 Subprocess Quota

Each user process has a quota that determines the maximum number of subprocesses that process can create, thereby limiting the number of processes spawned by a participant in a PPL$ application. Check your subprocess quota to be sure that the quota is greater than or equal to the number of subprocesses you plan to create in your parallel application.

1.5.2.2 AST Limit

Because PPL$ uses ASTs (asynchronous system traps) internally, a PPL$ application that uses other AST system services extensively may need to increase its ASTLM quota. Under most conditions, adding 2 per participant to your current value is sufficient. For each application, you must calculate the possible extent of your AST use.

1.5.2.3 Enqueue Quota

PPL$ uses the $ENQ system service internally. If you also use the locking system services independently of PPL$, you may have to increase your ENQLM quota. The largest possible increase that PPL$ may need is 5 per participant.

1.5.2.4 Global Section Quota

If your application uses a large amount of shared memory, you may want to request that your system manager increase the following SYSGEN parameters:

GBLSECTIONS
GBLPAGES
GBLPAGFIL

2 Process Management and Naming Operations

The PPL$ facility provides routines to help you manage your application's processes. These management routines include those that create and delete an application, terminate a process' access to an application, create and delete subordinates, retrieve subordinate information, and produce a name for the application that is unique but consistent throughout the application.

2.1 Accessing the PPL$ Facility

The PPL$ facility provides the following routines to create and delete an application and terminate a process' access to an application:

PPL$CREATE_APPLICATION	Informs the PPL$ facility that the caller is forming or joining the parallel application
PPL$DELETE_APPLICATION	Marks the PPL$ internal data area for deletion and prevents additional processes from joining the application
PPL$TERMINATE	Ends the caller's participation in the parallel application

2.1.1 Initializing PPL$

PPL$CREATE_APPLICATION informs the PPL$ facility that the caller is forming or joining the parallel application. You are not required to call this routine. You need only call PPL$CREATE_APPLICATION if you want to specify a value other than the supplied defaults. If you do not call it explicitly, PPL$CREATE_APPLICATION is called automatically when you call one of the routines listed in the following table. Note that PPL$ does not automatically initialize when you call routines that require a previously created element. (In other words, PPL$ does not automatically initialize when you call a routine listed in the following table for the second and subsequent times.) This keeps the overhead of these routines—requests for barriers, semaphores, events, spin locks, and work queues—at a minimum.

The routines that perform automatic initialization when first called are:

PPL$CREATE_BARRIER	PPL$FIND_OBJECT_ID
PPL$CREATE_EVENT	PPL$GET_INDEX
PPL$CREATE_SEMAPHORE	PPL$INDEX_TO_PID
PPL$CREATE_SHARED_MEMORY	PPL$PID_TO_INDEX
PPL$CREATE_SPIN_LOCK	PPL$SPAWN

PPL$CREATE_VM_ZONE	PPL$STOP
PPL$CREATE_WORK_QUEUE	PPL$UNIQUE_NAME

If you do not call PPL$CREATE_APPLICATION, PPL$ allocates the default (link time) constant PPL$K_INIT_SIZE pages for its internal data structures. This initial allocation accommodates a minimum of 32 processes, 8 semaphores, 4 events, 4 spin locks, 4 barriers, 4 work queues, and 16 global sections. (These numbers represent a rough guideline for combinations of PPL$ components. If you have less than 32 processes, for example, you can have more than 8 semaphores, and so forth.) You can specify another value for the **size** argument in PPL$CREATE_APPLICATION if these defaults are not appropriate for your application. If you intend to use more PPL$ resources than PPL$K_INIT_SIZE pages allows, you should specify a larger value for the **size** argument.

2.1.2 Deleting an Application

PPL$DELETE_APPLICATION marks all shared memory in an application for deletion. This includes the PPL$ internal data area, all shared memory sections, and shared zone sections. Because the shared memory is not actually deallocated until the last process exits, this routine has no effect on processes that are already members of the application. However, after you call this routine, no new processes are allowed to join the application. The process calling this routine requires the PRMGBL privilege.

If a process attempts to join an application that has been deleted, PPL$ instead forms a new application with the same name (subject to the options specified in PPL$CREATE_APPLICATION). This prevents completely separate instances of an application with the same name from interfering with each other.

Calling PPL$DELETE_APPLICATION is the only way to remove a permanent application (one which was formed with the PPL$M_PERM flag set in PPL$CREATE_APPLICATION). After calling PPL$DELETE_APPLICATION, the application is no longer permanent. When the last process leaves the application, all shared memory sections are deallocated, and the application is deleted.

2.1.3 Terminating Access to the PPL$ Facility

The PPL$TERMINATE routine lets you "prematurely" terminate the caller's participation in the application, that is, before the caller has actually completed its execution. Normally, you do not need to call this routine because the PPL$ facility automatically performs cleanup operations when the participating process completes its execution. Optionally, this routine forces the exit of all of the caller's descendants.

2.2 Participant Management

The PPL$ facility provides several routines to simplify the tasks involved in creating, deleting, and retrieving information about a participant. These routines are as follows:

PPL$SPAWN	Creates one or more subordinates to execute code in parallel with the caller
PPL$STOP	Terminates the execution of a participant in the application
PPL$GET_INDEX	Returns a unique index for the specified participant
PPL$INDEX_TO_PID	Returns the process identifier of the participant associated with the specified index
PPL$PID_TO_INDEX	Returns the index of the participant with the specified process identifier

These routines are discussed in the following sections.

2.2.1 Creating a Subordinate

The PPL$SPAWN routine lets you create one or more subordinates that can execute code in parallel with the caller. Any subordinate created executes the specified code in parallel on the same node as the caller. After calling PPL$SPAWN, typically the parent (caller) immediately continues processing in its own context, and each subordinate begins executing immediately after it is created. Optionally, you can specify that the caller and all the subordinates being created only continue after each and every subordinate has performed its PPL$ initialization, that is, performed a call to PPL$CREATE_APPLICATION. You can also specify the PPL$M_NODEBUG value for the **flags** argument. Specifying this value prevents the startup of the VMS Debugger, even if the debugger was linked with the image. You can therefore selectively turn the Debugger on and off for each subordinate process.

It is important to note that if you want to be notified when a subordinate terminates execution abnormally, you must call PPL$ENABLE_EVENT_SIGNAL or PPL$ENABLE_EVENT_AST. PPL$ENABLE_EVENT_SIGNAL and PPL$ENABLE_EVENT_AST are discussed in Chapter 4. In the following example, the call to PPL$ENABLE_EVENT_SIGNAL indicates that the user wants to be notified if any of the created subordinates terminates abnormally.

```
desired_condition = PPL$K_ABNORMAL_EXIT
status = PPL$ENABLE_EVENT_SIGNAL (desired_condition)
status = PPL$SPAWN (num_of_procs,,id_array)
```

2.2.2 Deleting a Subordinate

The PPL$STOP routine terminates the execution of the specified participant in the parallel application. If you call PPL$STOP for a process that has spawned subordinates, VMS forces the termination of the "descendants" of the specified process. You should call this routine only if you want to stop a participant before it completes execution.

2.2.3 Retrieving Participant Information

The PPL$ facility provides three routines that supply information about a particular participant. These routines are as follows:

PPL$GET_INDEX
PPL$INDEX_TO_PID
PPL$PID_TO_INDEX

The PPL$GET_INDEX routine returns an index that is unique within the parallel application. An index with a zero value indicates the **top** or **main** process, that is, the participant executing first in the application. The index of each participant is assigned in order as it joins the application, so that all the participants in the application always return an index greater than zero.

You can use PPL$GET_INDEX to retrieve the PPL$ identifier (participant index) of the caller.

```
status = PPL$GET_INDEX (my_index)
```

The PPL$INDEX_TO_PID routine returns the VMS process identifier of the participant associated with the index you specify. Similarly, the PPL$PID_TO_INDEX routine takes a VMS process identifier and returns the PPL$ index of the associated participant.

To continue the previous example, the caller can subsequently call PPL$INDEX_TO_PID to retrieve its own process identifier.

```
status = PPL$INDEX_TO_PID (my_index, my_pid)
```

2.3 Application-Wide Naming

The PPL$UNIQUE_NAME routine returns an application-unique name. This name consists of a system-unique string that is specific to the calling application; this string is appended to the string that you specify. The resulting name will be identical for all participants in the application, but different from all other applications on that system.

This unique name is useful, for example, if your application creates a scratch file that must not interfere with other users who are also running their own copies of the same application at the same time.

For example, two users running the same application in different jobs call PPL$UNIQUE_NAME and supply the same value for the **name-string** argument ("x"). The name that PPL$UNIQUE_NAME returns to the first user is different from the name returned to the second user.

3 Shared Memory Operations

When you execute a program in a sequential processing environment, all the instructions in your program are executed in order. However, when you execute your applications in a parallel processing environment, the operating system controls such things as the availability of processors and the order of execution and completion of participants. While the instructions within a single task are still executed sequentially, you cannot predict the order in which tasks will execute.

Because of this unpredictability, you often require some form of interprocess communication for tasks that are executed in parallel. The PPL$ facility provides several routines that facilitate interprocess communication by creating and controlling shared memory. **Shared memory** is a generic term that refers to any memory that can be accessed by two or more processes running concurrently. Applications calling PPL$ routines use shared memory (known as global sections in VMS) to share information among participants. Shared memory contains shareable code or data that can be read, or read and written, by more than one process. For more information about global sections, refer to the *VMS System Services Reference Manual*.

3.1 Shared Memory Routines

The shared memory routines provided by the PPL$ facility are as follows:

PPL$CREATE_SHARED_MEMORY	Create (if necessary) and map a section of memory that can be shared by multiple participants
PPL$FLUSH_SHARED_MEMORY	Write (flush) the contents of a global section to disk
PPL$DELETE_SHARED_MEMORY	Delete or unmap from a global section
PPL$CREATE_VM_ZONE	Create a new storage zone that is available to all participants in the application
PPL$DELETE_VM_ZONE	Delete a storage zone

These routines are discussed in more detail in the following sections.

3.1.1 Creating Shared Memory

The PPL$CREATE_SHARED_MEMORY routine creates (if another participant has not already created) and maps a section of memory that can be shared by multiple participants. By default, PPL$CREATE_SHARED_MEMORY gives the shared memory a name unique to the application, initializes the section to zero, and maps the section with read /write access. If you want to change any of these defaults, you can do so using the **flags** argument.

In addition, PPL$ tries to share the memory at the same address with all other participants in the application, if possible. This operation merely attempts to "reserve" that address range, and it is only mapped in other participants at the time they issue calls to this routine. If PPL$CREATE_SHARED_MEMORY cannot map the shared memory to the same addresses for all participants, the error condition value PPL$_NONPIC is returned and the shared memory is not created. (This might occur when the application executes more than one different program image.)

Optionally, this routine opens a backup storage file for the shared memory with a specified file name.

The PPL$ facility offers two distinct memory sharing services through PPL$CREATE_SHARED_MEMORY. The first mechanism lets you request an unspecified range of addresses, and the PPL$ facility arranges to allocate the same set of addresses in each participant in the application. In other words, you let the PPL$ facility determine the address of the shared memory being created. You request this service by specifying the starting address as zero. If you allow the PPL$ facility to select the virtual addresses for a section of shared memory, PPL$ selects the virtual addresses so that each process already in the application can map the section to the same address range. A participant that joins the application after the shared memory is created may not be able to access the shared memory if the new participant's image size is significantly larger than the image size of the participant(s) that created the shared memory. If you have difficulty creating shared memory, be sure that all participants that will use the section have joined the application *before* the shared memory is created. This applies to memory allocated from shared VM zones as well, because they are created using PPL$ shared memory sections.

The second mechanism lets you specify a particular range of addresses to be shared. This allows the sharing of an arbitrary collection of variables, such as a FORTRAN common block, that appear at a certain address. Since VMS maps memory in pages (512 bytes), you must take care to share exactly the data intended for sharing—no more and no less. When the data does not fall exactly on page boundaries, extra effort is required to prevent accidental sharing of local data while guaranteeing that all participants can access the shared memory at the expected addresses. You can accomplish this by allocating a 512-byte array at both the beginning and the end of such a data area (common block). The request to this routine then specifies the starting address to be that of the front "guard" array. The length is calculated by subtracting the last address of the last "guard page" from the starting address of the front guard. PPL$ maps the requested memory so that the lower address is rounded up to the nearest page boundary, and the higher address is rounded down to the nearest page boundary. This guarantees that no data is shared unexpectedly, and that all important data in the common area (that is, everything but the two guard pages) is fully shared.

In the following example, a section of shared memory contains a variable named *front_guard* (the first variable in the section), as well as *last_guard* (the last variable in the section). The *lenadr* array contains the length of the desired section (including guard pages) and the starting location of the section.

```
parameter (one_page = 512)
.
.
.
lenadr(1) = %LOC(last_guard) + one_page - %LOC(front_guard)
lenadr(2) = %LOC(front_guard)
status = PPL$CREATE_SHARED_MEMORY ('pgm_shared_data', lenadr)
IF (.NOT. STATUS) GO TO 999
```

3.1.2 Flushing Shared Memory to Disk

The PPL$FLUSH_SHARED_MEMORY routine writes (flushes the contents of) a global section that has a disk file backing store to disk. This global section must have been created by a call to PPL$CREATE_SHARED_MEMORY.

Only the pages that have been modified are flushed to disk. This is useful, for example, if you want to store intermediate values of the variables stored in the global section. The flush is performed by each of the participants.

To continue the previous example, you use the following statement to flush the *pgm_shared_data* section to disk:

```
status = PPL$FLUSH_SHARED_MEMORY ('pgm_shared_data')
```

3.1.3 Deleting Shared Memory

The PPL$DELETE_SHARED_MEMORY routine deletes or unmaps from a global section. This global section must have been created through a call to PPL$CREATE_SHARED_MEMORY. If you specify PPL$M_FLUSH as an argument to this routine, the contents of the global section are written to disk before the section is deleted. Note that if another participant is using the global section when you call this routine, PPL$DELETE_SHARED_MEMORY unmaps the global section from the calling process' memory. When all participants have unmapped from the section or have been deleted, PPL$DELETE_SHARED_MEMORY deletes the global section.

In the following example, the section *pgm_shared_data* is deleted after its contents are written to disk. (This is specified by the PPL$M_FLUSH flag.)

```
flag = PPL$M_FLUSH
status = PPL$DELETE_SHARED_MEMORY ('pgm_shared_data',,flag)
```

3.2 Creating a Virtual Memory Zone

The PPL$CREATE_VM_ZONE routine creates a new storage zone that is available to all participants in the application. You can use the zone identifier returned by this routine in calls to the following RTL LIB$ routines:

LIB$FREE_VM	LIB$RESET_VM_ZONE
LIB$GET_VM	LIB$SHOW_VM_ZONE
LIB$DELETE_VM_ZONE	LIB$VERIFY_VM_ZONE

The arguments for PPL$CREATE_VM_ZONE are identical to those of LIB$CREATE_VM_ZONE, except for the last two arguments; PPL$CREATE_VM_ZONE does not accept the *get-page* and *free-page* arguments provided by LIB$CREATE_VM_ZONE. It is the caller's responsibility to ensure that the caller has exclusive access to the zone while the reset operation is being performed.

All participants in the application share the memory allocated by calls to LIB$GET_VM. Therefore, memory allocated by one participant can be read and freed by another participant.

3.3 Deleting a Virtual Memory Zone

PPL$DELETE_VM_ZONE deletes a specified storage zone and returns all pages owned by the zone to the application-wide page pool. For more information on deleting virtual memory zones, refer to the description of LIB$DELETE_VM_ZONE in the *VMS RTL Library (LIB$) Manual*.

You must ensure that all processes in the application are no longer using any of the memory in the zone before you call PPL$DELETE_VM_ZONE. None of the processes in the application can perform any further operations on the zone after you call PPL$DELETE_VM_ZONE.

4 Synchronization Operations

One method of ensuring consistent access to program data is to synchronize cooperating processes. **Synchronization** can be described as a set of constraints that affects or controls the ordering of events in your decomposed application. You can use synchronization mechanisms to delay execution of a particular process in order to satisfy any such constraints.

The PPL$ facility provides routines to create and manipulate **synchronization elements**, which control the order of processing in a parallel application. These routines implement the following synchronization elements:

- Barriers
- Events
- Semaphores
- Spin locks
- Work queues

The PPL$ facility also provides a routine that can be used to retrieve the identifier of any named object. An object can be a synchronization element or shared memory zone. All of the synchronization routines are discussed in the following sections.

4.1 Retrieving an Object Identifier

Given the name of a barrier, event, semaphore, spin lock, work queue, or shared memory zone, the PPL$FIND_OBJECT_ID routine returns the identifier of the object associated with the name you specify. (An object can be a synchronization element or shared memory zone.) This routine is useful when you are trying to ensure that a particular object's identifier is available to all the participants that need access to that object. By naming the object when you actually create it, you can then use PPL$FIND_OBJECT_ID to let other participants retrieve the identifier of the object of that name. Object names are case sensitive.

You can also retrieve the identifier of an object by naming that object and "re-creating" it. That is, after you have created an object, all participants that need to access that object's identifier can call the appropriate "create" routine, specifying the same name for the object. This returns the identifier of the existing object and a status of PPL$_ELEALREXI. One benefit of using this method is that all participants can share the same code, with each one calling the same create routine with the same parameter values.

4.2 Barrier Synchronization

Barrier synchronization lets you establish a barrier, or a point that a specific number of participants must reach before continuing their work. This method of synchronization is useful if you have multiple execution paths that need to synchronize at a particular point (generally, at the completion of a set of work items). To implement barrier synchronization, the PPL$ facility supplies the following routines:

PPL$CREATE_BARRIER	Creates a barrier synchronization element
PPL$DELETE_BARRIER	Deletes a barrier synchronization element
PPL$READ_BARRIER	Returns a barrier's current quorum and number of participants waiting at the barrier
PPL$WAIT_AT_BARRIER	Waits until the quorum is reached for that barrier
PPL$SET_QUORUM	Establishes the initial quorum for an inactive barrier
PPL$ADJUST_QUORUM	Increments or decrements an active barrier's quorum

Using all of these routines, you can implement barrier synchronization in your parallel application.

4.2.1 Creating a Barrier

The PPL$CREATE_BARRIER routine creates and initializes a barrier synchronization element, and returns the identifier of that barrier. This identifier is used as an argument to the other barrier synchronization routines.

When you create a barrier using PPL$CREATE_BARRIER, you can optionally specify a *quorum*. The quorum specifies the number of participants that are required to terminate a wait for the barrier. For example, if the quorum value is set to 3, the first two callers of PPL$WAIT_AT_BARRIER that specify this barrier will be blocked until a third caller issues that request. At that point, all three participants will be released for further processing. If you omit the quorum parameter, a default value of 1 is assigned.

For example, the following call to PPL$CREATE_BARRIER creates a barrier named *my_barrier* with a quorum value of 3.

```
status = PPL$CREATE_BARRIER (my_barrier_id, 'my_barrier', %REF(3))
```

PPL$CREATE_BARRIER returns a barrier identifier that you must use in all subsequent operations on that barrier. It is your responsibility to make this identifier available to all the participants that need to access that barrier. To do so, you can place the barrier identifier in shared memory. However, there is another option. When you create the barrier initially, specify a barrier name. Then use either of the two following methods to let any participant retrieve the barrier identifier:

- Call PPL$FIND_OBJECT_ID to retrieve the identifier of the barrier with that name.

- Call PPL$CREATE_BARRIER again, specifying the same barrier name, to retrieve the identifier of that barrier.

4.2.2 Deleting a Barrier

PPL$DELETE_BARRIER deletes a barrier and releases any storage associated with it. A barrier may be specified by either its name or its identifier.

You cannot delete a barrier if participants are waiting at the barrier. If you attempt to delete a barrier at which participants are waiting, PPL$ returns the PPL$_ELEINUSE error. (Call PPL$ADJUST_QUORUM to release the waiting participants before deleting the barrier.) None of the participants in the application can perform any further operations on the barrier after you call PPL$DELETE_BARRIER.

4.2.3 Reading a Barrier

The PPL$READ_BARRIER routine returns the specified barrier's current quorum and the number of participants currently waiting (blocked) at the barrier. This routine is useful if, for example, you want to adjust the barrier's quorum with the PPL$ADJUST_QUORUM routine, but you want to first determine how many participants have reached the barrier.

Calls by other participants to the PPL$ barrier routines may affect the values returned by PPL$READ_BARRIER. In effect, the values returned by this routine may be outdated before you receive them.

4.2.4 Waiting at a Barrier

The PPL$WAIT_AT_BARRIER routine causes the caller to wait at the specified barrier until the specified number (quorum) of participants have arrived at the barrier. Once the quorum is reached, all waiting participants are released for further execution. The barrier is in effect from the time the first participant calls PPL$WAIT_AT_BARRIER until each member of the quorum has issued the call.

The number of participants required to constitute a quorum can be defined by calls to the PPL$CREATE_BARRIER, PPL$SET_QUORUM, and PPL$ADJUST_QUORUM services. Note that a call to PPL$ADJUST_QUORUM can result in the conclusion of a barrier wait.

In the following example, a barrier is created with the name *synch_barrier* and an identifier named *barrier_id*. The quorum for this barrier is set to 1 greater than the number of subordinates, meaning that all participants in the application (including the parent) are required to terminate barrier *synch_barrier*. Later in the application, every participant must perform the same call to PPL$WAIT_AT_BARRIER, specifying the same barrier identifer (*barrier_id*) in order to reach the quorum and terminate the barrier.

```
status = PPL$CREATE_BARRIER (barrier_id, 'synch_barrier',
1                            %REF(subordinates+1))
.
.
.
status = PPL$WAIT_AT_BARRIER (barrier_id)
```

4.2.5 Setting a Barrier Quorum

The PPL$SET_QUORUM routine lets you establish an initial value for the specified barrier's quorum. That is, you can use PPL$SET_QUORUM to change the value of the quorum for any barrier at which participants are not currently waiting. For example, you might want to use PPL$SET_QUORUM to set the initial barrier quorum if you did not supply a value in your call to PPL$CREATE_BARRIER. You can also use PPL$SET_QUORUM to change the quorum of a barrier once the previous quorum has been reached and all waiting participants have continued execution.

To illustrate, the previous example could also have been accomplished using the PPL$SET_QUORUM routine instead of specifying the quorum value in the call to PPL$CREATE_BARRIER.

```
status = PPL$CREATE_BARRIER (barrier_id, 'synch_barrier')
status = PPL$SET_QUORUM (barrier_id, %REF(subordinates+1))
.
.
.
status = PPL$WAIT_AT_BARRIER (barrier_id)
```

Note that PPL$SET_QUORUM must be called while no participants have called PPL$WAIT_AT_BARRIER (in other words, while there are no participants waiting at the barrier).

4.2.6 Adjusting a Barrier Quorum

The PPL$ADJUST_QUORUM routine lets you increment or decrement the quorum of a barrier that is currently active. That is, using PPL$ADJUST_QUORUM, you can dynamically alter the number of participants that are required to conclude a wait on a particular barrier.

For example, if an expected barrier participant terminates without calling PPL$WAIT_AT_BARRIER, the quorum will never be reached and the waiting participants will wait forever. By using PPL$ADJUST_QUORUM, any participant that discovers the unexpected termination of a barrier participant could then decrement the quorum value by 1 to accommodate this situation. Note that if you dynamically alter the quorum value to match the number of participants already waiting at a barrier, the barrier will be concluded and the participants will continue their execution. Be sure that your application properly accounts for the number of participants expected to wait at a specified barrier.

4.3 Event Synchronization

An **event** is a synchronization element that has an associated state; this state may take on a value of *occurred* or *not_occurred*. You can enable notification when an event occurs, and you can trigger that notification as desired. You can also enable event notification for two predefined events, PPL$K_NORMAL_EXIT and PPL$K_ABNORMAL_EXIT. One of those events, as appropriate, is triggered automatically by PPL$ when a process terminates. (See PPL$CREATE_EVENT for more information.) The PPL$ facility provides nine routines that help you implement event synchronization.

PPL$CREATE_EVENT	Creates a user-defined event
PPL$DELETE_EVENT	Deletes a user-defined event
PPL$ENABLE_EVENT_AST	Delivers an AST when the event has *occurred*
PPL$ENABLE_EVENT_SIGNAL	Delivers a signal condition when the event has *occurred*
PPL$DISABLE_EVENT	Disables delivery of event notification to the caller by AST or signal, or both
PPL$AWAIT_EVENT	Blocks the caller until the event state becomes *occurred*
PPL$TRIGGER_EVENT	Sets the event state to *occurred*
PPL$READ_EVENT	Returns the current state of the event
PPL$RESET_EVENT	Resets the event state to *not_occurred*

The following sections discuss each of the event synchronization routines in more detail.

4.3.1 Creating an Event

The PPL$CREATE_EVENT routine creates an arbitrary user-defined event and returns the event's identifier. When you first create an event, the state of the event is set to *not_occurred*. All other operations on an event hinge on the operation of setting the state of an event to *occurred*.

In the following example, an event is created called *synch_event*.

```
status = PPL$CREATE_EVENT (my_event_id, 'synch_event')
```

PPL$CREATE_EVENT returns an event identifier that you must use in all subsequent operations on that event. It is your responsibility to make this identifier available to all the participants that need to access that event. To do so, you could place the event identifier in shared memory. However, there is another option. When you create the event initially, specify an event name. Then use either one of the two following methods to let any participant retrieve the event identifier:

- Call PPL$FIND_OBJECT_ID to retrieve the identifier of the event with that name.
- Call PPL$CREATE_EVENT again, specifying the same event name, to retrieve the identifier of that event.

4.3.2 Deleting an Event

PPL$DELETE_EVENT deletes a specified event and releases any storage associated with it. You cannot delete an event if participants are waiting for the event to occur. However, an event can be deleted if other participants have enabled notification of the event, or if outstanding triggers are queued for the event. None of the participants in the application can perform any further operations on the event after you call PPL$DELETE_EVENT.

4.3.3 Enabling an Event AST

The PPL$ENABLE_EVENT_AST routine lets you establish an AST routine (and optionally an argument to that routine) that will deliver an AST when a specified event occurs, that is, when the state of the event becomes *occurred*. If the state of the event is already *occurred* when you call this routine, the AST is delivered immediately, and the event state is reset to *not_occurred*. If the state of the event is *not_occurred* when you call this routine, your request for an AST to notify the caller of an event's occurrence is placed in a queue, and is processed once the event actually occurs (when a corresponding trigger is issued). Generally, a trigger is issued when a participant calls PPL$TRIGGER_EVENT. However, the PPL$ facility triggers predefined events automatically. Note that the caller continues execution immediately after the AST request is placed in the queue.

The **astprm** parameter has special requirements when used in conjunction with the PPL$ facility.

- For user-defined events, the **astprm** should point to a vector of two unsigned longwords. The first longword is a "context" reserved for the user; it is not read or modified by PPL$. The second longword receives the value specified in the call to PPL$TRIGGER_EVENT that results in the delivery of this AST.
- For PPL$-defined events (those not created by the user), the **astprm** parameter should point to a vector of four unsigned longwords that accommodates the following:
  - The user's "context" longword
  - The longword to receive the event's distinguishing condition-value
  - The parameters to the PPL$-defined event (the "trigger" parameter)

  For example, the events corresponding to PPL$_ABNORMAL_EXIT and PPL$_NORMAL_EXIT require an array of four longwords, since each of these events has two additional parameters—the **participant-index** and the **exit-status** of the terminating participant. A condition value of PPL$_ABNORMAL_EXIT or PPL$_NORMAL_EXIT is passed as the **astprm** for the corresponding PPL$-defined event.

4.3.4 Enabling an Event Signal

The PPL$ENABLE_EVENT_SIGNAL routine lets you specify a condition value to be signaled when the specified event occurs, that is, when the state of the event becomes *occurred*. If the state of the event is already *occurred* when you call this routine, the signal is delivered immediately, and the event state is reset to *not_occurred*. Otherwise, your request for a signal to notify the caller of an event's occurrence is placed in the queue, and is processed once the event actually occurs (when a corresponding trigger is issued). Generally, a trigger is issued when a participant calls PPL$TRIGGER_EVENT. However, the PPL$ facility triggers predefined events automatically. Note that the caller continues execution immediately after the signal request is placed in the queue.

The following example illustrates a simple call to PPL$ENABLE_EVENT_SIGNAL. Once the event *my_event_id* is triggered, the value *user_arg* is signaled at the time the event occurs.

```
user_arg = my_cond_value + STS$K_SEVERE
status = PPL$ENABLE_EVENT_SIGNAL (my_event_id, user_arg)
```

Note that two values are signaled if you also supply a value to PPL$TRIGGER_EVENT. The parameter you pass to PPL$ENABLE_EVENT_SIGNAL is the first condition value, and the parameter you pass to PPL$TRIGGER_EVENT is the second condition value.

Refer to the *VMS System Services Reference Manual* for more information on condition values.

4.3.5 Disabling an Event

PPL$DISABLE_EVENT disables delivery of event notification to the calling participant by AST or signal, or both. (This routine has no effect on other participants that called PPL$AWAIT_EVENT and are waiting for an event to occur.)

There may be some delay between the time that this routine is called and the time that the event is actually disabled. The calling program should be prepared to handle event notification until the time that this routine *returns*.

4.3.6 Awaiting an Event

The PPL$AWAIT_EVENT routine lets you specify that the caller should be blocked until the specified event occurs, that is, until the state of the event becomes *occurred*. If the state of the event is already *occurred* when you call this routine, the caller proceeds immediately (without being blocked), and the event state is reset to *not_occurred*. Otherwise, the caller's request to be awakened when the event occurs is queued, and the caller is blocked.

In this example, the caller specifies that it should be blocked until the event specified by *my_event_id* has occurred.

```
user_arg = my_cond_value
status = PPL$AWAIT_EVENT (my_event_id, user_arg)
```

4.3.7 Triggering an Event

The PPL$TRIGGER_EVENT routine lets you set an event's state to *occurred*. At that point, all requests that are queued for event notification are processed, so that any enabled ASTs or signals, or both, are delivered, and any participant blocked awaiting an event is awakened. You may also specify notification of only one of the queued requests. Once any signals and ASTs have been processed and any blocked participants have been awakened, the state of the event is reset to *not_occurred*. All of these actions occur atomically with respect to the event (in other words, once these actions begin, they complete without interruption from other event operations).

If no requests are queued for the event at the time of the trigger, the event's state becomes *occurred*, and the first call to PPL$ENABLE_EVENT_AST or PPL$ENABLE_EVENT_SIGNAL receives the requested notification.

Note: An arbitrary number of triggers may be queued for an event before any participant enables event notification. The presence of another queued trigger at the completion of processing one trigger forces the state to again become *occurred*. That is, processing of a queued trigger occurs immediately after processing the previous trigger.

In the following example, the event *my_event_id* is triggered, thereby releasing all participants awaiting the change of that event's state to *occurred*. Because a value is not specified for the signal value in this call, or in the previous call to PPL$ENABLE_EVENT_SIGNAL, the status signaled is PPL$_EVENT_OCCURRED.

```
user_arg = my_cond_value + STS$K_SEVERE
status = PPL$TRIGGER_EVENT (my_event_id, user_arg)
```

4.3.8 Reading an Event

The PPL$READ_EVENT routine returns the current state of the specified event. The state can be *occurred* or *not_occurred*.

Calls by other participants to the PPL$ event routines may affect the values returned by PPL$READ_EVENT. In effect, the values returned by this routine may be outdated before you receive them.

4.3.9 Resetting an Event

PPL$RESET_EVENT resets the event state associated with a specified event to *not_occurred*. Any triggers queued by calls to PPL$TRIGGER_EVENT are removed from the queue.

4.3.10 Predefined Events

The PPL$ facility creates and predefines the events PPL$K_NORMAL_EXIT and PPL$K_ABNORMAL_EXIT. You need not create these events. (These events are described in the following sections.) When a normal or abnormal exit occurs, PPL$ triggers the event automatically. Note that you can ignore these predefined events at no cost. However, DIGITAL recommends that you enable event notification of PPL$K_ABNORMAL_EXIT, because that condition usually indicates a severe error. Notification is delivered only if you explicitly request it by specifying the predefined event as the **event-id** in a call to PPL$ENABLE_EVENT_SIGNAL, PPL$ENABLE_EVENT_AST, or PPL$AWAIT_EVENT.

1 PPL$K_NORMAL_EXIT—This event is triggered by PPL$ when an application participant exits normally. Normal exits include the following:

- The participant returns a success status
- The participant calls PPL$TERMINATE
- The subordinate's parent calls PPL$TERMINATE, specifying PPL$M_STOP_CHILDREN
- Some other participant calls PPL$STOP to terminate this participant or its parent

If you enabled a signal for this event through a call to PPL$ENABLE_EVENT_SIGNAL, the condition signaled as the trigger parameter is PPL$_NORMAL_EXIT.

2 PPL$K_ABNORMAL_EXIT—This event is triggered by PPL$ when an application participant exits abnormally. Abnormal exits include the following:

- The participant returns an error status
- A mechanism outside PPL$ forces termination and prevents the execution of exit handlers (for example, the DCL command STOP /ID)

If you enabled a signal for this event through a call to PPL$ENABLE_EVENT_SIGNAL, the condition signaled as the trigger parameter is PPL$_ABNORMAL_EXIT.

There are some special usage considerations for the PPL$ predefined events if delivery of an AST is requested. Refer to the description section of PPL$ENABLE_EVENT_AST for more information.

4.4 Semaphore Synchronization

A semaphore is a special common variable that you can use to implement mutual exclusion. That is, a semaphore lets you control the availability of a particular critical region or resource. The value of the semaphore variable represents the number of resources available, so that by decrementing and incrementing the semaphore you can control the access to some critical section of your application.

The semaphore, referred to here as *s*, is usually initialized to 1. If *s* can only assume the values 0 and 1, it is called a **binary semaphore**. A binary semaphore acts like a lock bit; it allows only one process at a time to execute a critical section or access a resource, and all other processes are blocked. If a **fairness queue** is included in the implementation of the semaphore, then any blocked process is put into the queue. When the critical region or resource becomes available, the blocked processes are granted access in some fair order. If there is no fairness queue, then when the critical section is free, any waiting process can be granted access randomly.

If you want to implement a semaphore that represents multiple resources, you can use a **counting semaphore**. A counting semaphore can take any nonnegative integer value; this value again represents the number of available resources. By decrementing and incrementing this semaphore, you can control access to these multiple resources. The semaphores provided by the PPL$ facility are counting semaphores. Counting semaphores have associated waiting queues, so that semaphore requests are not lost but are placed in the appropriate waiting queue until they can be processed. (If you want to implement a binary semaphore, specify a maximum semaphore value of 1.)

Consider the following situation. You have a system with three available line printers. By creating a counting semaphore with an initial value of 3, you can control access to these resources. Process A requests a single line printer; you decrement the semaphore value and grant process A access. Process B then requests access to a line printer. Because there are two printers still available, you decrement the semaphore again and grant process B access. If, however, process C requests access to two line printers, you cannot grant access to process C because there is only one printer available. At that point, process C must wait until one of the other printers becomes available, and then access can be granted.

You control the values of both binary and counting semaphore variables by means of the **Wait** and **Signal** primitive operations. (See Section 4.4.3 for the routine, PPL$DECREMENT_SEMAPHORE, that corresponds to the Wait operation. See Section 4.4.4 for the routine, PPL$INCREMENT_SEMAPHORE, that corresponds to the Signal operation.) The Wait operation lets you acquire permission to enter a critical section. If the process cannot be granted access at that time, the Wait operation causes the requesting process to wait. The Signal operation records the termination of another process's access to a critical section and signals the next process to get permission to enter. The following example shows a general implementation of the Wait and Signal operations.

```
Wait(s):    IF s > 0
            THEN
              s := s - 1
            ELSE
              Suspend the execution of the process that
              called Wait(s).
```

```
Signal(s):    IF some process A has been suspended by a previous
              Wait(s) on this semaphore
            THEN
              Wake up process A
            ELSE
              s := s + 1
```

The Wait operation tries to grant a requesting process access to some resource. If, at the start of the Wait operation, the value of the semaphore *s* is greater than zero, then access is granted and the semaphore is decremented. If the value of *s* is zero, then no resources are available and the requesting process is forced to wait.

The Signal operation notifies a waiting process that a resource is now available. If there is a suspended process waiting for that resource, then that process is immediately woken and granted access. If there is no waiting process, then the Signal operation increments the value of *s* by the appropriate number so that the released resources are marked as available.

The PPL$ facility supports the use of semaphores as a synchronization technique. You can implement semaphore synchronization using the following PPL$ routines:

PPL$CREATE_SEMAPHORE	Creates and initializes a semaphore with a waiting queue
PPL$DELETE_SEMAPHORE	Deletes a semaphore and releases any storage associated with it
PPL$DECREMENT_SEMAPHORE	Waits for the semaphore to have a value greater than zero and then decrements the semaphore
PPL$INCREMENT_SEMAPHORE	Increments the value of a semaphore by 1
PPL$READ_SEMAPHORE	Returns the current and/or maximum values of the specified semaphore or the number of waiting participants
PPL$ADJUST_SEMAPHORE_MAXIMUM	Increments or decrements the maximum value of a semaphore
PPL$SET_SEMAPHORE_MAXIMUM	Sets the maximum value of a semaphore

The routines supporting semaphore synchronization are discussed in the following sections.

4.4.1 Creating a Semaphore

The PPL$CREATE_SEMAPHORE routine creates and initializes a semaphore with a waiting queue. The waiting queue stores any caller of PPL$DECREMENT_SEMAPHORE that must be blocked because the resource is unavailable. In your call to PPL$CREATE_SEMAPHORE, you can specify a semaphore name, a maximum value for the semaphore, and an initial value for the semaphore, all of which are optional. The identifier parameter is required.

In the following example, PPL$CREATE_SEMAPHORE is used to create a binary semaphore (maximum value of 1).

```
max_value = 1
init_value = 1
.
.
.
status = PPL$CREATE_SEMAPHORE (my_sem_id, 'my_sem', max_value,
                               init_value)
```

PPL$CREATE_SEMAPHORE returns a semaphore identifier that you must use in all subsequent operations on that semaphore. It is your responsibility to make this identifier available to all the participants that need to access that semaphore. To do so, you could place the semaphore identifier in shared memory. However, there is another option. When you create the semaphore initially, specify a semaphore name. Then use either one of the two following methods to let any participant retrieve the semaphore identifier:

- Call PPL$FIND_OBJECT_ID to retrieve the identifier of the semaphore with that name.
- Call PPL$CREATE_SEMAPHORE again, specifying the same semaphore name, to retrieve the identifier of that semaphore.

4.4.2 Deleting a Semaphore

PPL$DELETE_SEMAPHORE deletes a specified semaphore and releases any storage associated with it. You cannot delete a semaphore if participants are waiting for the semaphore. None of the participants in the application can perform any further operations on the semaphore after you call PPL$DELETE_SEMAPHORE.

4.4.3 Decrementing a Semaphore

The PPL$DECREMENT_SEMAPHORE routine waits for a semaphore to have a value greater than zero, and then decrements the value by 1 to indicate the allocation of a resource. If the value of the semaphore is zero at the time of the call, the caller is put in the associated waiting queue and is suspended. This operation is analogous to the **wait** protocol.

You can modify the behavior of this routine by specifying a flag parameter that indicates that you do not want the caller to be blocked for this operation. That is, you can request that the semaphore be decremented if and only if it can be done without causing the caller to be blocked. You might want to do this, for example, in situations where the cost of waiting for a resource is not desirable, or if you merely intend to request immediate access to any one of a number of resources.

In the following example, the caller requests access to a resource by decrementing the semaphore *my_sem_id*. However, the caller does not want to be blocked if the resource is not available, so the flag PPL$M_NON_BLOCKING is specified.

```
flag = PPL$M_NON_BLOCKING
.
.
.
status = PPL$DECREMENT_SEMAPHORE (my_sem_id, flag)
```

4.4.4 Incrementing a Semaphore

The PPL$INCREMENT_SEMAPHORE routine increments the value of a semaphore by 1. This is analogous to the **signal** protocol. If any participants are blocked on a call to PPL$DECREMENT_SEMAPHORE for this particular semaphore, one of these participants is removed from the waiting queue and awakened.

If the caller in the previous example gains access to the semaphore *my_sem_id*, a subsequent call to PPL$INCREMENT_SEMAPHORE is required once the caller completes its required access to the resource.

```
status = PPL$INCREMENT_SEMAPHORE (my_sem_id)
```

4.4.5 Reading a Semaphore Value

PPL$READ_SEMAPHORE returns the following information about a semaphore:

- the current value of the specified semaphore
- the number of participants that are currently waiting at the semaphore
- the maximum value of the specified semaphore

You can use this routine to determine how many resources are currently available, for example, or the maximum number of resources that can be allocated.

Calls by other participants to the PPL$ semaphore routines may affect the values returned by PPL$READ_SEMAPHORE. In effect, the values returned by this routine may be outdated before you receive them.

4.4.6 Adjusting a Semaphore Maximum

PPL$ADJUST_SEMAPHORE_MAXIMUM dynamically increases or decreases the maximum value of a semaphore, therefore allowing you to dynamically alter the number of resources protected by the semaphore. The semaphore's current value is adjusted by a value you specify to reflect the new maximum. A semaphore maximum cannot be decreased by a value that is greater than the current value of the semaphore.

4.4.7 Setting a Semaphore Maximum

PPL$SET_SEMAPHORE_MAXIMUM allows you to dynamically set the maximum value of a semaphore. This allows semaphores to be reused for different purposes with various numbers of participants.

Calling PPL$SET_SEMAPHORE_MAXIMUM changes the semaphore's current value to the new maximum value you specify.

You can call PPL$SET_SEMAPHORE_MAXIMUM only when the semaphore's current value is equal to its maximum (in other words, there are no participants using resources that are protected by the semaphore).

4.5 Spin Lock Synchronization

A spin lock is a lock on a critical section that constantly tests to see whether or not access to the critical section is available. (Any segment of your program that must be executed by only a single process at a time is called a critical section.) Because this method of mutual exclusion is constantly testing the lock and is therefore CPU intensive, it should only be used on dedicated parallel processing systems. A spin lock is not recommended for use in a general time-sharing environment, or when fairness in obtaining the lock is important.

The PPL$ facility provides routines to implement spin lock synchronization. You can implement spin locks using the following PPL$ routines:

PPL$CREATE_SPIN_LOCK	Creates and initializes a simple (spin) lock
PPL$DELETE_SPIN_LOCK	Deletes a spin lock and releases any storage associated with it
PPL$SEIZE_SPIN_LOCK	Retrieves a simple (spin) lock by waiting in a spin loop until the lock is free
PPL$RELEASE_SPIN_LOCK	Relinquishes a spin lock
PPL$READ_SPIN_LOCK	Returns the current state of a spin lock

The routines implementing spin locks are discussed in the following sections.

4.5.1 Creating a Spin Lock

The PPL$CREATE_SPIN_LOCK routine creates and initializes a simple (spin) lock and returns the identifier. The newly created lock is initialized to zero, indicating that the lock is not set.

In the following example, a spin lock is created named *my_spin_lock*. This lock is initialized to zero at creation.

```
status = PPL$CREATE_SPIN_LOCK (my_lock_id, 'my_spin_lock')
```

PPL$CREATE_SPIN_LOCK returns a spin lock identifier that you must use in all subsequent operations on that spin lock. It is your responsibility to make this identifier available to all the participants that need to access that spin lock. To do so, you could place the spin lock identifier in shared memory. However, there is another option. When you create the spin lock

initially, specify a spin lock name. Then use either one of the two following methods to let any participant retrieve the spin lock identifier:

- Call PPL$FIND_OBJECT_ID to retrieve the identifier of the spin lock with that name.
- Call PPL$CREATE_SPIN_LOCK again, specifying the same spin lock name, to retrieve the identifier of that spin lock.

4.5.2 Deleting a Spin Lock

PPL$DELETE_SPIN_LOCK deletes a spin lock and releases any storage associated with it. You cannot delete a spin lock if it is held by any process in the application. None of the participants in the application can perform any further operations on the spin lock after you call PPL$DELETE_SPIN_LOCK.

4.5.3 Seizing a Spin Lock

The PPL$SEIZE_SPIN_LOCK routine acquires a spin lock by waiting in a spin loop until the lock is free. If you specify the PPL$M_NON_BLOCKING flag in your call to PPL$SEIZE_SPIN_LOCK, the caller does not wait in the spin loop if it cannot immediately obtain the lock.

Once you acquire the spin lock, you have exclusive access to it until you call PPL$RELEASE_SPIN_LOCK to free the lock.

In the following example, the caller puts itself in a spin loop waiting for the spin lock to be available. If the caller had specified the PPL$M_NON_BLOCKING flag, the caller would not have been blocked if the spin lock was not available.

```
status = PPL$SEIZE_SPIN_LOCK (my_lock_id)
```

4.5.4 Releasing a Spin Lock

The PPL$RELEASE_SPIN_LOCK routine relinquishes your control over the spin lock. If there are other participants waiting in a spin loop to obtain this lock, this routine allows one of those participants to get the lock, thereby terminating that spin loop. Note that the participant that then gets the lock is not necessarily the one that has been waiting the longest.

Continuing the previous example, once the caller gains access to the spin loop, it continues processing and must call PPL$RELEASE_SPIN_LOCK once the caller is finished with the lock. Otherwise, any other participants blocked in spin loops can never resume execution.

```
status = PPL$RELEASE_SPIN_LOCK (my_lock_id)
```

4.5.5 Reading a Spin Lock

PPL$READ_SPIN_LOCK returns the current state of the specified spin lock. The state can be *seized* or *not_seized*. Calls by other participants to the PPL$ spin lock routines can affect the state returned by this routine. In effect, the state returned by this routine may be outdated before you receive it.

4.6 Work Queue Synchronization

A parallel application that uses a work queue consists of a queue of work items and participants to complete the work items. In this method of parallel processing, one or more participants serve as task "dispatchers." These participants place work items that identify a task to be performed into a work queue. For example, the work items can be obtained from a user or a transaction file. Other participants ("servers") remove the work items from the queue and execute the indicated task. When there is no work to be done, the dispatchers await input from the user, and the servers block on the empty queue.

Work queues are similar to semaphores—inserting an item into a work queue is analogous to incrementing a semaphore, and removing an item from a work queue is analogous to decrementing a semaphore.

The PPL$ work queue routines implement a simple priority queue. You can attach a priority to a work item, but by default the priority is zero. Therefore, if an application accepts the default when inserting work items into the queue, the queue behaves like a simple FIFO (first in, first out) model.

The PPL$ facility provides the following routines to implement work queue synchronization.

PPL$CREATE_WORK_QUEUE	Creates a work queue
PPL$DELETE_WORK_QUEUE	Deletes a work queue
PPL$READ_WORK_QUEUE	Retrieves the number of items in a work queue or the number of waiting participants
PPL$DELETE_WORK_ITEM	Deletes a specified item from a work queue
PPL$INSERT_WORK_ITEM	Inserts an item into a work queue
PPL$REMOVE_WORK_ITEM	Removes the next item in order from a work queue

The routines implementing work queues are discussed in the following sections.

4.6.1 Creating a Work Queue

PPL$CREATE_WORK_QUEUE creates and initializes a work queue, and returns the identifier of the queue. The work queue stores work items, which identify tasks to be performed by other participants.

PPL$CREATE_WORK_QUEUE returns a work queue identifier that you must use in all subsequent operations on that queue. It is your responsibility to make this identifier available to all the participants that need to access the queue. To do so, you can place the work queue identifier in shared memory. However, there is another option. When you create the work queue initially, specify a queue name. Then use either one of the two following methods to let any participant retrieve the work queue identifier:

- Call PPL$FIND_OBJECT_ID to retrieve the identifier of the work queue with that name.
- Call PPL$CREATE_WORK_QUEUE again, specifying the same queue name, to retrieve the identifier of that work queue.

4.6.2 Deleting a Work Queue

PPL$DELETE_WORK_QUEUE deletes a work queue and releases any internal storage associated with that queue. If another participant is waiting for a work item to be placed in the queue, it is awakened. None of the participants in the application can do any further operations on the work queue after you call PPL$DELETE_WORK_QUEUE.

A work queue must be empty before it can be deleted (unless you specify the PPL$M_FORCEDEL flag). You can force deletion of a queue that is not empty by specifying the PPL$M_FORCEDEL flag. If you force a queue to be deleted, the PPL$ facility makes no assumptions about the contents of the work items. If your items consist of pointers to pieces of shared memory, it is your responsibility to deallocate all work items in the queue before deleting the queue.

4.6.3 Reading a Work Queue

PPL$READ_WORK_QUEUE returns the following information about a work queue:

- The number of items that are presently in the specified work queue
- The number of participants that are currently waiting for items to be inserted into the work queue

Calls by other participants to the PPL$ work queue routines may affect the values returned by PPL$READ_WORK_QUEUE. In effect, the values returned by this routine may be outdated before you receive them.

4.6.4 Inserting a Work Item

PPL$INSERT_WORK_ITEM inserts a value into a work queue. If another participant is waiting for an item to be placed into the queue, that participant is awakened and will remove the newly inserted item after the call to PPL$INSERT_WORK_ITEM.

By default, the item is inserted into the queue *after* any items with a higher or equal numerical priority and *before* any items with a lower priority. If you specify the flag PPL$M_ATHEAD, the item is inserted before any other items of an equal priority.

If an application always uses the default (zero) for the **priority** argument, the result is a simple FIFO (first in, first out) queue. PPL$ inserts new items at the end of the queue by default, or at the beginning of the queue if you specify the PPL$M_ATHEAD value for the **flags** argument.

The content of the **work-item** argument is completely arbitrary. You may want to place single longword values into **work-item** (for example, the number of a function or task to be performed). You can also use **work-item** to pass a pointer to a data block. (This data block must reside in memory created by PPL$CREATE_SHARED_MEMORY or allocated from a shared memory zone created by PPL$CREATE_VM_ZONE.)

4.6.5 Removing a Work Item

PPL$REMOVE_WORK_ITEM removes the next item with the highest priority from the specified queue. If the queue is empty, the participant hibernates until an item is placed in the queue by another participant. When an item is placed in the queue, the participant awakens and proceeds normally. If the queue is empty and PPL$REMOVE_WORK_ITEM is called with the PPL$M_NON_BLOCKING flag set, the routine returns immediately with the PPL$_NOT_AVAILABLE status, indicating that an item was not removed from the queue.

If a participant is hibernating (awaiting an item to be placed into the queue) and the queue is deleted, the participant is awakened and returns immediately with the PPL$_DELETED status, indicating that the queue was deleted and no item was removed.

4.6.6 Deleting a Work Item

PPL$DELETE_WORK_ITEM searches a work queue for an item whose value matches the one you specify. By default, this routine searches the queue from beginning to end. If you specify PPL$M_TAILFIRST for the **flags** argument, the queue is searched from the end to the beginning. When the first matching work item is found, the item is deleted and the routine returns with a success status. However, if the PPL$M_DELETEALL flag is set, PPL$DELETE_WORK_ITEM continues searching and deleting matching items until it reaches the end of the queue.

5 Developing Parallel Processing Applications

The Parallel Processing Facility (PPL$) is a process-oriented library of routines that simplifies development of a parallel application. This chapter discusses some recommended methods for using the Parallel Processing Facility for developing new programs.

5.1 Programming Considerations

This section describes some items you should consider when you develop a parallel processing application.

5.1.1 Granularity and Decomposition

When you initially design an application, or redesign an existing application for parallel execution, you must first **decompose** the application into separate tasks that can be executed simultaneously. Decomposition, then, is the act of modifying a single-stream program into a parallel program by creating parallel sections that can be executed concurrently.

The first step in decomposing an application is to determine the maximum number of work items that can be executed simultaneously. You should ideally be able to carry out these work items independently of

- The number of processors available
- The order in which the work items will begin execution
- Which work items will finish first

In addition to determining the maximum number of work items for simultaneous execution, you should also analyze the code to determine

- Which portions of the application are compute intensive
- What dependencies exist between the data used by these work items
- How the data structures are accessed by the application during parallel execution

By decomposing your application, you have defined which tasks you can execute in parallel. The amount of work performed by these separate tasks defines the level of **granularity** of your decomposed application. The levels of granularity are as follows:

Coarse	Level 1 tasks are those tasks that are actually separate programs.
	Level 2 tasks are procedures or subroutines within a program.

	Level 3 tasks are loop iterations or code sequences within a single routine of a program.
	Level 4 tasks are the individual statements within a routine of a program.
Fine	Level 5 tasks are individual machine instructions.

In general, a finer level of granularity means that less work is performed by each task, thereby causing a more granular decomposition. It is important to note that, in most cases, very fine granular decomposition causes a decrease in performance because the amount of work included in each task does not warrant the oversynchronization and communication for those tasks.

5.1.2 Data Dependence

The term **data dependence** describes a situation in which information produced by one part of your program is needed for another part to produce accurate results. By examining data usage in your application, you can determine any data dependencies that you must maintain in order for your application to achieve the correct results. Therefore, the goal in analyzing and defining the data dependencies in a program is to identify which variables and constructs need synchronization before parallel processing can be implemented correctly.

There are four types of data dependence:

- True dependence
- Antidependence
- Output dependence
- Control or conditional dependence

These types of data dependence are described as follows:

Where S represents a statement, if S_1 precedes S_2, then S_2 depends on S_1 if

- S_2 uses the output of S_1. This situation is defined as **true dependence** and is illustrated by the following statements:

  $S_1 : x = y$

  $S_2 : z = x$

  In this example, the value of z in S_2 depends on the value x is assigned in S_1, thus creating a true dependence.

- S_2 might incorrectly use the output of S_1 if the statements were reversed. This is defined as **antidependence**, and is shown in the following example.

  $S_1 : z = x$

  $S_2 : x = y$

In the preceding example, no dependence exists between S_1 and S_2 when the statements are executed sequentially. The value of x in S_2 does not depend on the execution of S_1. However, if the order of execution of these two statements is reversed, then a dependence is created. If S_2 is executed before S_1, then the value of y again depends on the previous value of x. Therefore, this example illustrates antidependence.

- S_2 resets the output of S_1. This defines **output dependence**, which is illustrated in the following statements.

  $S_1 : x = 3$

  $S_2 : x = 2$

  Output dependence defines the situation in which two values are assigned to the same variable. In this situation, you must preserve the order of these assignments in case these values are output in some significant order.

- S_1 is a conditional statement upon which the execution of S_2 depends. This situation is defined as **conditional** or **control dependence**, and can be seen in the following example.

  $S_1 : IF\ x = 1\ THEN...$

  $S_2 : y = 4$

  The execution of statement S_2 depends on the conditional statement in S_1. If the conditional statement does not test as expected, required code may not be executed; hence you have a conditional dependence.

A common method of avoiding data dependence is to make sure that the statements exhibiting dependence are coded in the same task. This ensures that they will be executed in the proper order. However, for some dependencies you can derive solutions that still allow for the parallel execution of operations. Consider the following example:

```
do i = 6,1000
   a(i) = a(i-5) * 5
end do
```

In the above situation, every value of $a(i)$ depends on the previously calculated value of $a(i-5)$, thereby exhibiting a true dependence. Although it seems that this loop cannot be decomposed into parallel tasks because of the data dependency, there is a possible solution. Since each value of $a(i)$ depends only on the value of $a(i-5)$, you can implement parallel do loops that calculate a series of $a(i)$ values. For example, the first loop could calculate $a(i)$ for values of i including 6, 11, 16, 21, and so on. The second loop could concurrently calculate $a(i)$ for values of i including 7, 12, 17, and so forth. All of these loops could then be executed concurrently.

```
LOOP 1:

    do i=6,1000 step 5
       a(i) = a(i-5) * 5
    end do

LOOP 2:
```

```
    do i=7,1000 step 5
       a(i) = a(i-5) * 5
    end do
LOOP 3:

    do i=8,1000 step 5
       a(i) = a(i-5) * 5
    end do
LOOP 4:

    do i=9,1000 step 5
       a(i) = a(i-5) * 5
    end do

LOOP 5:

    do i=10,1000 step 5
       a(i) = a(i-5) * 5
    end do
```

You should generally try to avoid situations involving data dependence in parallel tasks. However, if data dependence is unavoidable, you must use the correct synchronization and communication mechanisms to maintain the integrity of the data while executing paths in parallel.

5.1.3 Deadlock

A process is said to be in a state of **deadlock** if it is waiting for a particular lock that it can never get. Deadlock can occur whenever processes compete for resources or whenever processes wait for each other to complete certain actions. A simple example of a deadlock situation is as follows: process A is granted resource X and then requests resource Y. Process B is granted resource Y and then requests resource X. If both of these resources are unshareable and neither process will release the resource it holds, then deadlock occurs.

In general, you can create a deadlock situation if one or more of the following conditions is in effect:

- Mutual exclusion — Each task claims exclusive control of the resources allocated to it.
- Nonpreemption — A task cannot release the resources it holds until they are no longer required.
- Wait for — Tasks hold resources already allocated to them while waiting for additional resources.
- Circular wait — A circular chain of tasks exists such that each task holds one or more resources that is being requested by the next task in the chain.

Although the first three conditions listed above are desirable for parallel programming, their existence can lead to deadlock situations in your parallel applications. There are three ways to deal with deadlock:

- Prevention

In general, to implement a prevention method, you must somehow constrain your processes so that requests leading to a deadlock never occur. In other words, you must design your application so that none of the conditions outlined above can occur.

- Avoidance

  For deadlock avoidance, you must create some sort of "scheduler" that controls resource allocation on the basis of advance information about resource usage so that deadlock is avoided. That is, you must create an environment where the criteria for deadlock can never occur.

- Detection and Recovery

  With deadlock detection and recovery, your "scheduler" gives a resource to a requesting process as soon as it becomes available. If a deadlock is detected, the "scheduler" preempts some resource in order to recover from the deadlock situation.

The difference between deadlock avoidance and detection lies in the fact that, to implement deadlock avoidance, you must create a parallel processing environment that never allows the criteria for deadlock to be satisfied. This need not be true for deadlock detection to be implemented. Deadlock may still occur in a detection implementation; however, your system will detect the deadlock situation and recover from it.

5.1.4 Naming Components

DIGITAL recommends that you do not name any user-defined component of PPL$ with a name that includes a dollar sign ($). A name that includes a dollar sign may coincide with VMS facility names.

5.1.5 Using SYS$HIBER

PPL$ uses the $HIBER system service internally. If you intend to use $HIBER in an environment of layered software, you must consider possible interactions with underlying layers. Two possible interactions are of particular concern here. The $WAKE system service does not maintain a count of the number of calls to $WAKE issued for a given process, nor does it provide for any association between a particular $WAKE and a particular $HIBER. A program using these services in a multiprocess environment must ensure that it responds only to those $WAKEs intended for that program. The program must also guarantee that it does not unintentionally "use up" a $WAKE required by some other component.

Therefore, any call to $HIBER should be enclosed in a loop that checks for the validity of a received $WAKE request. This helps ensure correct behavior, at the expense of some overhead for instances in which no $WAKE was issued to another facility.

```
loop
exit when (this_op.hiber_ended);
        -- implying the waker sets this
$HIBER;
endloop;
$WAKE;  -- in case someone else needs it,
        -- expecting that they will similarly
        -- check validity
```

Note, however, that in a multithread (for example, Ada tasking) environment, this approach still does not allow for the immediate resumption of threads that are blocked by a call to $HIBER other than the current thread, because the current thread is effectively queuing all those resumption requests. This scenario can be repeated to an arbitrary depth, thus defeating the parallelism. In addition, threads that rely on other threads for resumption may be arbitrarily delayed as a result of the deferral of calls to $WAKE.

In sum, use of a $HIBER/$WAKE scheme can increase response latency and program overhead. You can avoid the need for those services by using only the PPL$ routines. Use of the $HIBER and $WAKE system services is discouraged in conjunction with the PPL$ facility.

Therefore, do not use $HIBER in your parallel application because doing so can cause unpredictable results.

5.1.6 Disabling ASTs

Because of the potential impact on PPL$, user disabling of ASTs is not recommended. Use the PPL$ routines for synchronization and notification rather than AST synchronization and notification techniques.

5.1.7 VAX Ada and VAX FORTRAN Considerations

If you call PPL$ from a VAX Ada application, be sure that only one task issues calls to PPL$ routines. This is necessary because PPL$ is not multi-thread (Ada task) reentrant. You may want to implement a monitor task that performs all PPL$ operations.

If you use PPL$ in conjunction with VAX FORTRAN Version 5.0, be sure that you do not explicitly share memory that FORTRAN is already sharing. The remapping can make it impossible for FORTRAN code to reference these addresses. DIGITAL recommends that you do not use PPL$ and FORTRAN to operate on the same shared data. However, you can use PPL$ and FORTRAN facilities within the same application, and all other features are compatible. Since FORTRAN's parallel support is fine-grained, you may want to use FORTRAN for fine-grained tasks and PPL$ for medium- to coarse-grained tasks.

If you use the /PARALLEL qualifier when compiling a FORTRAN program that calls PPL$ routines, be aware that when the FORTRAN parallel processing application completes, the subprocesses that are created during its execution are not properly deleted and are left in a spin loop, using CPU time. These subprocesses must be stopped by the user. The easiest

way to stop them is to either log out of the account that executed the parallel application or explicitly stop them.

5.2 Comparing the Use of Synchronization Elements

When you begin developing a parallel processing application, you can choose from five types of synchronization elements: barriers, events, semaphores, spin locks, and work queues. The following sections discuss in general terms each type of synchronization element. (Refer to Chapter 4 for more information on synchronization elements.)

5.2.1 Barriers

A barrier achieves synchronization by actually controlling the execution of the participant. Barrier synchronization is useful if you have multiple execution paths (participants) that need to synchronize at a particular point (generally, at the completion of a set of work items). A barrier synchronizes participants or tasks rather than resources.

A barrier has an associated **quorum** of participants that are required to reach the barrier before the blocked participants are released for further execution. You can dynamically alter the value of the quorum after you create the barrier and even after participants are waiting at the barrier. This is useful if, for example, a participant terminates prematurely so that the quorum you initially established is never reached.

5.2.2 Events

Event synchronization is different from barrier synchronization in that a participant reacts to an outside event rather than simply reaching normally some point in its code. The event routines are useful if you want to search a tree in parallel, for example. In that case, you could define an event to indicate the completion of the search. All of the participants call PPL$ENABLE_EVENT_SIGNAL, and when one participant successfully completes the search, that participant calls PPL$TRIGGER_EVENT. The other participants are notified that the search is complete and they then stop their own searches. Using events in this case allows you to search the tree in parallel while preventing participants from needlessly searching after the desired item is located.

You can use the PPL$ predefined events (described in PPL$CREATE_EVENT) to be notified when a participant exits. You do this by passing the predefined event name (for example, PPL$K_ABNORMAL_EXIT) as the event's identifier in a call to PPL$ENABLE_EVENT_AST or PPL$ENABLE_EVENT_SIGNAL. For example, if you call PPL$ENABLE_EVENT_AST specifying PPL$K_ABNORMAL_EXIT as the event identifier, you could supply an AST routine that checks to see if the terminated participant is a member of a barrier. If so, the AST routine could then call PPL$ADJUST_QUORUM to decrement the quorum by 1 so that the other waiting participants do not hang. (Note that PPL$K_ABNORMAL_EXIT refers to the event identifier, while PPL$_ABNORMAL_EXIT refers to the corresponding condition value.)

5.2.2.1 Asynchronous Signal

PPL$ENABLE_EVENT_SIGNAL provides for cross-process asynchronous signaling. This is a powerful mechanism, and it must be used only in carefully controlled environments.

Asynchronous exceptions are those that are not a direct result of the execution of the code, but rather are caused by some concurrent and not directly related event. For example, an AST interrupts a MOVC instruction and the AST routine attempts to reference an invalid address, resulting in an access violation. The signaled exception is an ACCVIO, and it is not related to the interrupted MOVC instruction. Occurrences of asynchronous exceptions have previously been quite uncommon, and the majority of existing code expects to terminate upon receipt of such an exception. The PPL$ENABLE_EVENT_SIGNAL service introduces the means for use of asynchronous signals as a communications mechanism.

Delivery of an asynchronous signal to an arbitrary layered environment can result in unwinding code that is totally unprepared for it, resulting in corrupted data. For example, any RTL routine or the code of a layered product might be interrupted by such an exception. Code that executes in multiple threads under one process context is particularly vulnerable—for example, Ada tasking. Delivery of an asynchronous exception interrupts the task that is executing at the time, and will result in task termination. Do not use this routine in environments that support multitasking within a process.

To avoid the potential program data corruptions and unintended alterations of control flow implied by unexpected unwinding of an unprepared code section, use this asynchronous signaling capability only when the code that can be interrupted is your own. Also note that you can accomplish the same tasks in a less dangerous way—using the standard AST facilities—by using the PPL$ENABLE_EVENT_AST routine.

5.2.3 Semaphores

A semaphore lets you control the availability of a particular critical region or resource; in other words, it implements mutual exclusion. You can also use a semaphore as a communication tool. The value of the semaphore variable represents number of processes waiting or the number of resources available, so that by decrementing and incrementing the semaphore you can control the access to a critical section of your application.

5.2.4 Spin Locks

Spin locks are one of the fastest forms of synchronization. This form of synchronization can improve the performance of applications using fine-grained parallelism. (Fine-grained parallelism means that each task performs a very small amount of work, such as an individual machine instruction or a few program statements.)

However, there are some negative consequences of using a spin lock. First, the spinning action consumes a large amount of CPU resources. Second, when the lock is released, it is given at random to a participant waiting in the spin loop. In other words, it does not take into consideration how long a participant has been waiting for the lock. In general, you should not use a spin lock in a time-sharing environment or when fairness is important.

5.2.5 Work Queues

Work queues are a higher-level construct than simple synchronization mechanisms, such as spin locks. The work queue parallel processing model consists of a queue of work items and processes to complete these work items. Each participant removes a work item from the queue, and if necessary, each participant can insert newly generated work items into the queue. As each participant completes its work item, it does not wait for some participant to assign it a new task, but instead removes the next item from the work queue and begins execution.

Work queues are similar to semaphores—inserting an item into a work queue is analogous to incrementing a semaphore, and removing an item from a work queue is analogous to decrementing a semaphore. Unlike semaphores, however, work queues allow you to prioritize each work item inserted into the queue. If you do not attach a priority to a work item, by default the priority is zero. Therefore, if an application accepts the default when inserting work items into the queue, the queue behaves like a simple FIFO (first in, first out) model.

The work queue routines also provide process synchronization. If there are no items in a work queue, a participant attempting to remove a work item from the queue by default will hibernate until an item is inserted into the queue.

5.2.6 Sharing an Element Identifier

To use the PPL$ synchronization elements (barriers, events, semaphores, and spin locks), you first create the element with the appropriate "create" routine (for example, PPL$CREATE_BARRIER). You then use the identifier returned by that routine in all calls to other routines that use the element you created (for example, PPL$WAIT_AT_BARRIER). The following list shows four ways that you can share the element identifier among the routines that need to access it.

1. Call PPL$FIND_OBJECT_ID, supplying the name of the element.
2. "Re-create" each element by calling the PPL$CREATE routine again, supplying the existing element name.
3. Place the element identifier in shared memory.
4. Use a facility outside of PPL$ (such as a VMS mailbox) to communicate the identifiers among participants.

Note that when you first create an element, you must give it a name if you want to use the name in other calls to retrieve the element identifier (as described in **1** and **2** in the preceding list).

You can also create unnamed synchronization elements that you do not have to name. However, this forces you to arrange for shared access to the identifier (as in **3** or **4** in the preceding list). The advantage of unnamed objects is that they are unique, and cannot be inadvertently re-created by another part of an application.

5.3 Performance Measurements

It is difficult to accurately predict and measure the performance of a parallel application. Performance, after all, depends not only on the optimal decomposition of the application, but also on the multiprocessing system running the application. Performance also depends on your goals. Although the generally accepted goal of parallel processing is to increase system throughput, there are situations in which response time or fault tolerance may be more significant than increased throughput.

Theoretically, the maximum speedup that you can attain using a multiprocessing system with n identical processors working concurrently on a single problem is at most n times faster than a single processor system working on the same problem. In practice, the speedup is much less, since at a given time some processors are idle because of memory or bus conflicts or inefficient decomposition of the program.

There are numerous methods that you can use to measure the performance of a parallel application. The following list mentions some of the theoretical models that you can use to try to predict the performance of your parallel application:

- Digital simulations
- Analytic models
- Probability functions
- Geometric models

To measure performance you can use simple test programs, or hardware or software monitors. The following section describes the geometric model of performance.

5.3.1 Geometric Model of Performance

The geometric model of performance lets you approximate the parallelism ratio and efficiency ratio for a system with W parallel tasks, N processors, and a single job stream. Based on this model, the following figures illustrate the amount of time required for N processors to complete W tasks. In these figures, t_1 represents the time required to execute any initialization or single-stream code, and t_2 is the time required to execute any task W.

In Figure 5–1, a parallel application having W tasks and a single processor will require some initialization time (t_1) plus the amount of time required to execute each task multiplied by the number of tasks. Therefore, for a system with W tasks and one processor, the total processing time T_1 is as follows:

$$T_1 = t_1 + Wt_2$$

Figure 5–1 Time-Processor Product for a System with No Parallelism

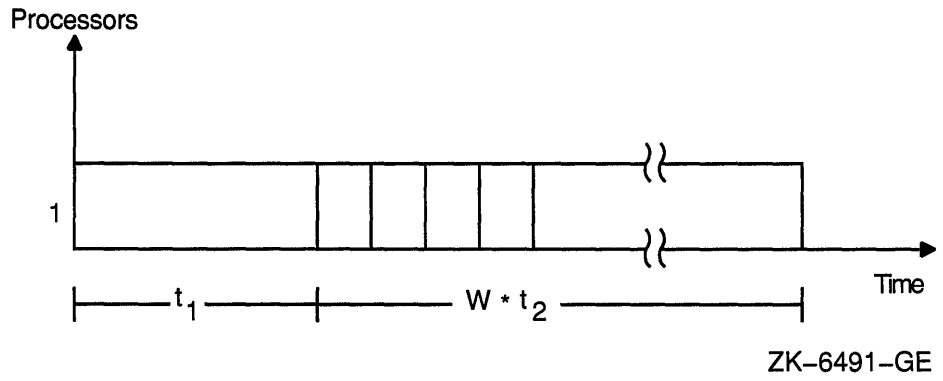


Figure 5–2 illustrates an application with W parallel tasks being run on a system with an infinite number of processors. Because there are an unlimited number of processors available, no matter how many parallel tasks there are, the total processing time T_∞ is as follows:

$T_\infty = t_1 + t_2$

Figure 5–2 Time-Processor Product for a System with Unlimited Parallelism

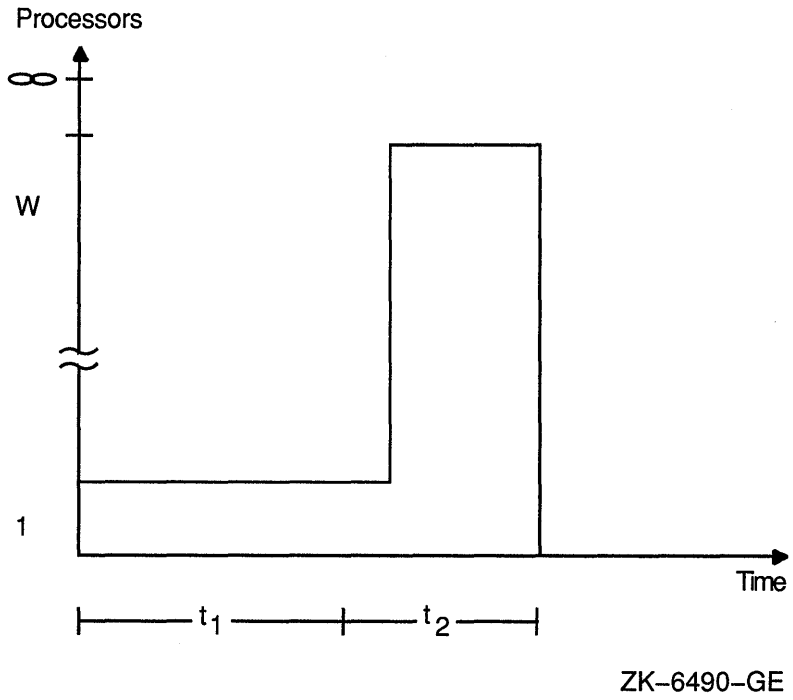


Figure 5–3 shows a typical multiprocessing system, where you have W parallel tasks running on N processors. For this type of configuration, the total processing time T_N is as follows:

$$T_N = t_1 + \frac{W}{N} t_2$$

Using the geometric model described above, you can determine several performance ratios for a given parallel system. You can use the following equation to determine the parallelism ratio p:

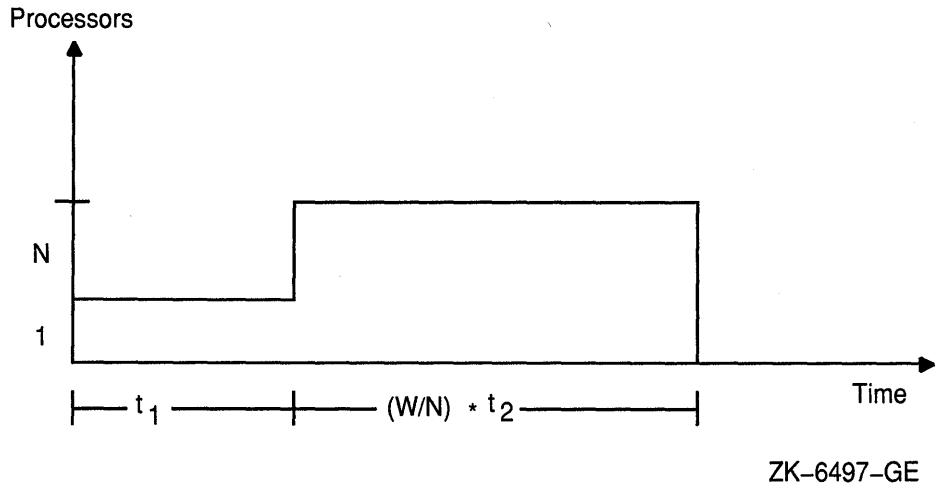
$$p = \frac{W t_2}{t_1 + W t_2}$$

Another performance ratio that you can derive using this geometric model is the efficiency ratio E:

$$E = \frac{t_1 + W t_2}{T_N * N}$$

All of the above equations can be used to predict the performance of a given multiprocessing system. Although the models are theoretical, tests have shown that these models yield fairly accurate results.

Figure 5–3 Time-Processor Product for a System with Limited Parallelism



6 Examples of Calling PPL$ Routines

This chapter contains examples demonstrating possible uses of PPL$ routines in the implementation of concurrent programming problems from BLISS-32, VAX FORTRAN, and VAX C. The example programs in this chapter include I/O statements to aid you in tracing program execution. Prime considerations for program decomposition include a complete understanding of the data to be processed and the data to be output, and mechanisms for controlling access to this data by the various participants in the application. PPL$ routines provide mechanisms to support access to the data by all participating processes and to control that access. This chapter is not meant to provide a comprehensive methodology for the development of concurrent algorithms.

6.1 BLISS-32 Example

Example 6–1 shows the use of PPL$ routines in BLISS-32 to perform a predefined event test.

Example 6–1 Using PPL$ Routines in BLISS-32

```
module example2 (main=main, addressing_mode (external=general)) =
begin

!+
! This example program shows event handling using PPL$ routines.
!
! This program demonstrates the need for recognizing the termination of a
! subordinate and its potential impact on the use of synchronization
! mechanisms - in this case, a barrier.
!
!-

library 'sys$library:starlet';

forward routine
    main,
    handler : novalue,
    print;

external routine
    PPL$ADJUST_QUORUM,
    PPL$CREATE_BARRIER,
    PPL$ENABLE_EVENT_AST,
    PPL$GET_INDEX,
    PPL$SPAWN,
    PPL$WAIT_AT_BARRIER;
```

(continued on next page)

Example 6–1 (Cont.) Using PPL$ Routines in BLISS-32

```
macro
    fail_badly_ =
        begin
        local i;

        i = .0;                 ! Cause an Access Violation
        end
    %;

own
    index   :   unsigned long,  ! This participant's role
    barrier :   unsigned long;

routine main =
begin
    local
        status  :       unsigned long;
    literal
        num_children = 1;

    status = PPL$GET_INDEX (index);
    if not .status then return signal (.status);
    print (%ascid'!/I am #!UL', .index);

    status = PPL$CREATE_BARRIER (barrier, %ascid'barrier',%ref(num_children+1));
    if not .status then return signal (.status);
    print (%ascid'!/!UL:!_created barrier = !XL!_stat = !XL',
        .index, .barrier, .status);

    case .index from 0 to num_children of
        set
        [0] :   ! Parent code
            begin
            ! Set up for termination event notification
            status = PPL$ENABLE_EVENT_AST (%ref(ppl$k_abnormal_exit), handler);
            if not .status then return signal (.status);

            ! Create the child(ren)
            status = PPL$SPAWN (%ref(num_children),     ! Number of children
                                0,                      ! Run the same image
                                0,                      ! Don't need the ID list
                                %ref(ppl$m_init_synch   ! Wait for child to init
                                    or ppl$m_nodebug)); ! Don't run the debugger
            print (%ascid'!/!UL:!_spawn status = !XL', .index, .status);

            ! It's safe for the parent to wait here, since the condition
            ! handler will correct the barrier quorum when a participant exits
            ! prematurely.  Otherwise, this wait would hang the application.
            status = PPL$WAIT_AT_BARRIER (barrier);
            print(%ascid'!/!UL:!_wait status = !XL', .index, .status);
            end;    ! End of parent code

            [1] :   ! Child code
            begin
            ! This child would normally be doing some work for the application,
            ! but it breaks, never reaching the barrer as the parent expects.

            fail_badly_;         ! Terminate

            ! The remaining code never executes...
```

(continued on next page)

Example 6–1 (Cont.) Using PPL$ Routines in BLISS-32

```
            status = PPL$WAIT_AT_BARRIER (barrier);
            print(%ascid'!/!UL:!_wait status = !XL', .index, .status);
            end;    ! End of child code

        [inrange]   :   begin
                         status = PPL$WAIT_AT_BARRIER (barrier);
                         print(%ascid'!/!UL:!_wait status = !XL', .index, .status);
                         end;
        [outrange]  :   0;  ! Do nothing
        tes;

    return ss$_normal;

end;    ! End of Main Routine

routine handler : NOVALUE =
begin

    !+
    ! This handler is invoked as notification of the predefined event,
    ! PPL$K_ABNORMAL_EXIT.  It fixes up the barrer quorum to account for the
    ! lost "body", and prevents the application-wide hang.
    !-

    local status;

    ! Avoid application-wide hang by completing the outstanding barrer wait
    status = PPL$ADJUST_QUORUM (barrier, %ref(-1));
    print (%ascid'!/!UL:!_adjust_quorum status = !XL', .index, .status);

end;    ! End of handler


routine print (ctrstr, p1) =
begin

    !+
    ! This formats a string with $fao and writes it.
    !-

    external routine
        LIB$PUT_OUTPUT;

    local
        buffer  :   $bblock[132],
        desc    :   vector[2];

    desc[0] = %allocation(buffer);
    desc[1] = buffer;

    $faol  (ctrstr = .ctrstr,
            outlen = desc[0],
            outbuf = desc[0],
            prmlst = p1);
    LIB$PUT_OUTPUT (desc[0])

end;    ! End of print

end                             ! End of module
eludom
```

6.1 BLISS-32 Example

The preceding BLISS example illustrates the potential for application-wide problems when a participant terminates. A participant may terminate without performing its expected synchronization functions.

In this example, the parent and child plan to synchronize at the completion of the work by waiting at a common barrier. The parent handles possible failure of a subordinate by requesting notification of the PPL$K_ABNORMAL_EXIT event. (Note that this artificial example merely demonstrates the principle, and that a typical application might have a more difficult time determining whether the child had reached the barrier. For example, the PPL$READ_BARRIER routine might be useful in order to determine the current number of participants waiting at the barrier.)

Tracing the execution of this program, the parent spawns the child, and then waits for completion of the work at the barrier. The child terminates prematurely, which triggers the PPL$K_ABNORMAL_EXIT event that delivers the AST as requested by the parent. The parent's AST service routine adjusts the barrier quorum to account for this termination, the hang is prevented, and the application completes.

6.2 VAX FORTRAN Example

Example 6–2 shows the use of PPL$ routines in VAX FORTRAN to decompose a loop.

Example 6–2 Using PPL$ Routines in VAX FORTRAN

```
0001            PROGRAM EXAMPLE1
0002
0003    C
0004    C PROGRAM DESCRIPTION:
0005    C
0006    C       This example program demonstrates master/slave loop
0007    C       decomposition using PPL$ routines.
0008    C
0009    C*****************************************************************
0010    C DATA DECLARATIONS
0011    C*****************************************************************
0012
0013    C EXTERNAL DEFINITIONS
0014            EXTERNAL    sts$k_info, sts$k_severe
0015            EXTERNAL    PPL$K_ABNORMAL_EXIT, PPL$M_NODEBUG
0016            INTEGER*4   PPL$CREATE_APPLICATION, PPL$CREATE_SHARED_MEMORY
0017            INTEGER*4   PPL$SPAWN, PPL$GET_INDEX
0018            INTEGER*4   PPL$CREATE_BARRIER, PPL$WAIT_AT_BARRIER
0019            INTEGER*4   PPL$CREATE_SEMAPHORE
0020            INTEGER*4   PPL$INCREMENT_SEMAPHORE, PPL$DECREMENT_SEMAPHORE
0021            INTEGER*4   PPL$ENABLE_EVENT_SIGNAL
0022            INTEGER*4   LIB$PUT_OUTPUT
0023
0024    C DEFINE ITEMS FOR USE WITH PPL$
0025            INTEGER*4   spawn_flags
0026            INTEGER*4   fatal_signal
0027                            !for event handling
0028            INTEGER*4   sem_max_val, sem_init_val
```

(continued on next page)

Example 6–2 (Cont.) Using PPL$ Routines in VAX FORTRAN

```
0029              PARAMETER (sem_max_val = 1, sem_init_val = 1)
0030                        !for a binary semaphore
0031
0032      C DEFINE APPLICATION DATA NEEDS
0033              INTEGER*4   stride
0034                  PARAMETER (stride = 5)
0035                        !number of consecutive array indices each party processes
0036              INTEGER*4   subordinates
0037                  PARAMETER (subordinates = 2)
0038                        !number of slaves
0039              INTEGER*4   array_size
0040                  PARAMETER (array_size = 50)
0041                        !a small array for demonstrative purposes
0042              INTEGER*4   one_page
0043                  parameter (one_page =  512)
0044
0045      C DEFINE DATA TO BE USED LOCALLY BY EACH PARTICIPANT
0046              INTEGER*4   copies                  !for parent's spawn call
0047              INTEGER*4   id_list(subordinates)   !"
0048              INTEGER*4   index                   !each participant's PPL-index
0049              INTEGER*4   lenadr(2)               !for creating shared memory
0050              INTEGER*4   status                  !info on each call
0051              INTEGER*4   mygroup, row, col, offset  !for doing the real work
0052              CHARACTER*12   my_id, group         !just for execution trace
0053
0054      C DEFINE DATA FOR SHARING
0055              byte        front_guard(one_page) !memory is mapped on page boundaries
0056              INTEGER*4   array1(array_size, array_size)      !input array
0057              INTEGER*4   array2(array_size, array_size)      !input array
0058              INTEGER*4   final_array(array_size, array_size) !output array
0059              INTEGER*4   next_task_number                    !work item info
0060              INTEGER*4   semaphore_id                        !synchronization
0061              INTEGER*4   barrier_id                          !"
0062              byte        rear_guard(one_page)
0063
0064      C PUT ALL THE SHARED DATA IN A COMMON BLOCK, WHICH WILL GET SHARED
0065              COMMON /pgm_shared_data/ front_guard,
0066              1                        array1,
0067              1                        array2,
0068              1                        final_array,
0069              1                        next_task_number,
0070              1                        semaphore_id,
0071              1                        barrier_id,
0072              1                        rear_guard
0073
0074
0892      C*********************************************************************
0893      C       CODE FOR ALL PARTICIPANTS STARTS HERE
0894      C*********************************************************************
0895
0896              type *,'Initializing'
0897              status = PPL$CREATE_APPLICATION ()
0898              if (.not. status) call LIB$STOP (%val(status))
0899
0900
0901      C MAP SHARED ADDRESS SPACE - enough for the shared variables + the spacers
0902
0903              lenadr(1) = %loc(rear_guard) + one_page - %loc(front_guard)
```

(continued on next page)

Example 6–2 (Cont.) Using PPL$ Routines in VAX FORTRAN

```
0904            lenadr(2) = %loc(front_guard)
0905            status = PPL$CREATE_SHARED_MEMORY ('pgm_shared_data', lenadr)
0906            if (.not. status) call LIB$STOP (%val(status))
0907
0908
0909    C DISPATCH TO ROLE-SPECIFIC CODE - PARENT @100, CHILD @200
0910
0911            status = PPL$GET_INDEX (index)
0912            if (.not. status) call LIB$STOP (%val(status))
0913
0914            if (index .ne. 0) go to 200     !act like a child
0915            go to 100                       !act like a parent
0916
0917
0918    C*********************************************************************
0919    C               PARENT CODE HERE
0920    C*********************************************************************
0921
0922    C The master performs all set-up functions, initializing both the
0923    C application's data and the synchronization support.
0924
0925
0926    100     type *, 'Parent Init'
0927
0928
0929    C INIT A SEMAPHORE AND BARRIER FOR SYNCHRONIZATION
0930
0931            status = PPL$CREATE_BARRIER (barrier_id, 'synch_barrier',
0932            1                           %ref (subordinates + 1))
0933                        !slaves and master all wait at this barrier
0934            if (.not. status) call LIB$STOP (%val(status))
0935
0936            status = PPL$CREATE_SEMAPHORE (semaphore_id, 'mutex',
0937            1                               sem_max_val, sem_init_val)
0938            if (.not. status) call LIB$STOP (%val(status))
0939
0940
0941    C REQUEST A SIGNAL IF SOMETHING UNUSUAL OCCURS IN A SUBORDINATE
0942
0943            fatal_signal = %loc(sts$k_severe)       !severe means we stop
0944            status = PPL$ENABLE_EVENT_SIGNAL (%loc(ppl$k_abnormal_exit),
0945            1                                 %val(fatal_signal))
0946            if (.not. status) call LIB$STOP (%val(status))
0947
0948
0949    C CREATE THE SUBORDINATES
0950
0951            copies = subordinates
0952            spawn_flags = %loc(ppl$m_nodebug)       !disable child debug
0953            status = PPL$SPAWN (copies,             !how many children
0954            1                       ,               !use current image name
0955            1                       id_list,        !children IDs
0956            1                       spawn_flags)    !special D
0957            if (.not. status) call LIB$STOP (%val(status))
0958
0959
0960    C PREPARE FOR TASK (WORK ITEM) ALLOCATION
0961
```

(continued on next page)

Example 6–2 (Cont.) Using PPL$ Routines in VAX FORTRAN

```
0962            next_task_number = 1
0963
0964
0965    C INIT THE DATA TO BE PROCESSED
0966
0967            do 11 i = 1,array_size          !clear the space for results
0968                do 12 j = 1,array_size
0969                    final_array(i,j) = 0
0970    12      continue
0971    11      continue
0972
0973            do 13 i = 1,array_size          !arbitrarily init array 1
0974                do 14 j = 1,array_size
0975                    array1(i,j) = i
0976    14      continue
0977    13      continue
0978
0979            do 15 i = 1,array_size          !likewise init array 2
0980                do 16 j = 1,array_size
0981                    array2(i,j) = 1
0982    16      continue
0983    15      continue
0984
0985
0986
0987    C At this point, all initialization functions have been performed by
0988    C the master, which must now wait for the subordinates to catch up.
0989    C Each of the subordinates waits at this barrier, guaranteeing that
0990    C they all proceed in unison.
0991
0992
0993    C WAIT FOR THE CHILDREN TO INIT
0994
0995            type *,'Parent waiting for children to init'
0996            status = PPL$WAIT_AT_BARRIER (barrier_id)
0997            if (.not. status) call LIB$STOP (%val(status))
0998
0999
1000    C A master might also want to participate in the parallel code sections,
1001    C which would happen right here.
1002    C In this example, the master waits.
1003
1004
1005    C WAIT FOR THE CHILDREN TO COMPLETE
1006
1007            type *,'Parent waiting for work completion'
1008            status = PPL$WAIT_AT_BARRIER (barrier_id)
1009            if (.not. status) call LIB$STOP (%val(status))
1010
1011
1012    C VERIFY RESULTS - LEFT AS AN EXERCISE FOR THE READER
1013
1014    C       call verify_results
1015            write (*,2) ( ( (final_array(i,j), i=1,array_size), j=1,array_size) )
1016    2       format (z12.8)
1017
1018
1019    C WRITE TERMINATION MESSAGE
```

(continued on next page)

Example 6–2 (Cont.) Using PPL$ Routines in VAX FORTRAN

```
1020
1021            type *, 'Parent Terminating'
1022
1023            go to 999
1024
1025
1026
1027    C***************************************************************************
1028    C           CHILD CODE HERE
1029    C***************************************************************************
1030
1031
1032    C PREPARE MY INDEX FOR OUTPUT - EXECUTION TRACE
1033
1034    200     write (unit=my_id, fmt='(I12)') index
1035            status = LIB$PUT_OUTPUT ('child init' // my_id(10:12))
1036
1037
1038    C GET READY, SLAVES
1039
1040            status = PPL$WAIT_AT_BARRIER (barrier_id)   !parent has to say go
1041            if (.not. status) call LIB$STOP (%val(status))
1042
1043
1044    C PROCESSING LOOP - PERFORM THE INTENDED PROGRAM FUNCTION
1045
1046            do while (.true.)
1047
1048                ! FIND OUT WHAT TO DO - IN A CRITICAL REGION
1049
1050                status = PPL$DECREMENT_SEMAPHORE (semaphore_id)
1051                if (.not. status) call LIB$STOP (%val(status))
1052
1053                mygroup = next_task_number
1054                next_task_number = next_task_number + stride
1055
1056                status = PPL$INCREMENT_SEMAPHORE (semaphore_id)
1057                if (.not. status) call LIB$STOP (%val(status))
1058
1059
1060                ! MAYBE THERE'S NO WORK TO DO
1061
1062                if (mygroup .gt. array_size) go to 888
1063
1064
1065                ! EXECUTION TRACE
1066
1067                write (unit=group, fmt='(I12)') mygroup
1068                status =
1069            1   LIB$PUT_OUTPUT ('child/grp:' // my_id(10:12) // group(10:12))
1070
1071
1072                ! DO THE WORK
1073
1074                do 333 offset = 0,(stride-1)
1075                    row = mygroup + offset
1076                    do 344 col = 1,array_size
1077                        final_array(row, col) = 0
```

(continued on next page)

Example 6–2 (Cont.) Using PPL$ Routines in VAX FORTRAN

```
1078                        do 355 i = 1,array_size
1079                            final_array(row,col) = final_array(row,col) +
1080           1                                    (array1(row,i) * array2(i,col))
1081    355                 continue
1082    344               continue
1083    333           continue
1084
1085            end do
1086
1087
1088    C CHILD TERMINATION POINT - get here when all work is done
1089
1090    888     status = PPL$WAIT_AT_BARRIER (barrier_id)
1091            if (.not. status) call LIB$STOP (%val(status))
1092
1093            status = LIB$PUT_OUTPUT ('termination: child ' // my_id(10:12))
1094
1095            go to 999
1096
1097
1098
1099    C*********************************************************************
1100    C          EXIT
1101    C*********************************************************************
1102
1103    C WRITE STATUS
1104
1105    999     print 1,status
1106    1       format(t8, 'final status= ',z12.8)
1107
1108            END
```

The preceding FORTRAN example shows a PPL$ implementation of a loop decomposition problem. It performs a matrix multiplication of two input arrays, resulting in an output array containing the matrix product.

Note that this example illustrates a simple master/slave model, just one of many approaches to solve this problem. Although this application is particularly suited to a fine-grained parallel approach, it is useful to illustrate some parallel processing techniques. This example also includes nonessential I/O calls to help you understand the flow of control. Similarly, for purposes of demonstration, the contents of the arrays is irrelevant, and their size has been diminished considerably from what might normally be expected for an effective (beneficial) parallel implementation.

Data dependencies are not of concern in this example, and several participants in this application work on the calculations at the same time in a straightforward manner. Each participant calculates the results for a different subset of the array indexes. This requires that each participant can access the data (that it be shared), and that each participant abide by a common set of conventions for maintaining data integrity (by use of standard synchronization mechanisms).

Each participant in the application executes the same program image. (This is not a requirement, but is convenient in this example.) This is accomplished by calling PPL$SPAWN and specifying a null value for the **image-name** argument. The differentiation of the master and slave roles is achieved by interpreting the **participant-index** for each participant (returned by the call to PPL$GET_INDEX). The value 0 means "master". All other values are used to indicate "slave" roles. (Such conventions for use of the **participant-index** are entirely at the discretion of the application designer.) All necessary common functions, such as setting up access to the shared data, are performed in code executed by all participants. Then, role-specific code is executed by the master or slave, as appropriate.

The required data is shared by placing both input arrays (*array1* and *array2*) and the output array (*final_array*) in a single common block, named *pgm_shared_data*. This common block is shared by all participants through their calls to PPL$CREATE_SHARED_MEMORY. Note that you must guarantee that all shared data is actually shared, and that no local data is accidentally shared. This example demonstrates the use of guard pages at the front and rear of the shared data. (See the Description section of PPL$CREATE_SHARED_MEMORY for more information.) A set of array indexes is allocated to each participant upon its request for a work item. To assure that this task assignment phase is not confused by concurrent access to the controlling data, these actions are performed in an atomic fashion by use of a (binary) semaphore. (Other synchronization elements such as spin locks can be used similarly.) Examine the following code sequence:

```
PPL$INCREMENT_SEMAPHORE (semaphore_id)
mygroup = next_task_number
next_task_number = next_task_number + stride
PPL$DECREMENT_SEMAPHORE (semaphore_id)
```

Calls to the semaphore routines establish a critical region around the use of the *next_task_number*, which provides an application-wide mechanism for guaranteeing that all array indexes are considered in the calculations. That is, *next_task_number* indicates the starting array index to be processed by a participant requesting a work item. *Next_task_number* must also be included in the data to be shared, but it is there for functional reasons quite different from the need to share the input and output arrrays. The variable *mygroup* obtains the identification of the work item (the starting array index) for use locally by a given participant. This requires that *mygroup* is not a shared data item. *Stride* is the number of array indexes that each participant processes. All must agree on this range to avoid miscalculation.

The **semaphore-id** used in implementing this critical region must be the same in all participants. There are several ways to do this, but the method used here places that **semaphore-id** in shared memory. Again, it is there for reasons of common access entirely separate from the need to access the actual data being manipulated by the algorithm.

This FORTRAN program arranges for orderly initiation and completion of the application. The master (which has a **participant-index** of 0) creates the subordinates and performs all single-stream actions. These actions include preparing the data initially and doing any required cleanup (both of which are only touched upon lightly in this example). The slaves wait for the master to say "go". They do this by waiting at a (common) barrier. As each participant calls PPL$WAIT_AT_BARRIER, it is blocked until all participants have reached that barrier. Then they all proceed. Once the master has freed the participants to do their work, it waits until the work is done, and then does cleanup. This completion is also indicated by waiting at the barrier. This **barrier-id** must also be in shared memory so that they wait at the same barrier.

Finally, this example enables notification of the predefined event PPL$K_ABNORMAL_EXIT. This event is triggered if any process in the application exits with a failure status. It is recommended that you always enable this event, since PPL$K_ABNORMAL_EXIT usually indicates a severe problem with the application. Notice that the value STS$K_SEVERE is specified as the enable parameter, which forces termination of the process that receives the notification (in the absence of a condition handler).

6.3 VAX C Example

Example 6–3 consists of two programs that show the use of PPL$ routines in VAX C to create an intercom between two processes running separate program images. This example demonstrates the following:

- Simple pipelining to perform tasks in parallel
- Allocating shared virtual memory zones, passing the address of the memory between processes, and freeing the shared memory
- Using PPL$FIND_OBJECT_ID
- Using work queues to synchronize processes and pass information between processes

In this C example, the first program, named SEND, creates a PPL$ application and PPL$ objects. It then waits for an outside (second) process to join. When the second process joins the application, the first process acquires shared memory, deposits a user input string into the shared memory, and then passes the memory address to the second process, via a work queue. The second process then displays the string. The process continues until "quit" is input.

The second program, named RECEIVE, joins the application created by the first process, finds the identifiers of all PPL$ objects needed to perform its tasks, synchronizes with the first process at a barrier, and then waits on a work queue to receive the address of a character string in shared memory. When it receives the address, the character string is displayed and the associated memory is freed.

6.3 VAX C Example

You must run the SEND program first because it creates the PPL$ application. If SEND is not run first, the second program will not find an existing PPL$ application and will exit with an error. Programs SEND and RECEIVE must be run under the same user UIC. (Running the programs on separate terminals is one way to accomplish this.)

Example 6–3 Using PPL$ Routines in VAX C

```
/*                  PROGRAM SEND                                        */

#include    <ppl$def.h>
#include    <ppl$routines.h>
#include    <stdio.h>
#include    <descrip.h>
#define     MAXLINE 256

/* PPL$ object names */
$DESCRIPTOR( barr1_name , "Synch1_barr" );
$DESCRIPTOR( barr2_name , "Synch2_barr" );
$DESCRIPTOR( app_name   , "Intercom"    );
$DESCRIPTOR( workq_name , "Task_queue"  );
$DESCRIPTOR( shared_mem , "Wire"        );

globalvalue
    PPL$K_INIT_SIZE;              /* Default PPL$ size        */

int size;           /* Application size         */
int prot;           /* Application protection   */
int flag;           /* Flag for application     */
int status;         /* Return status            */
int barrier1;       /* Barrier1 ID              */
int barrier2;       /* Barrier2 ID              */
int sendq;          /* Work Queue ID - send     */
int mem_id;         /* Shared VM zone ID        */
int num_bytes;      /* Message buffer size      */
int base_address;   /* Address returned for VM  */
short quorum;       /* Barrier quorum           */
char *message;      /* Pointer to shared memory message buffer */

int ppl$create_application();

main ( )
{

    size = 2 * PPL$K_INIT_SIZE; /* Application size twice the default       */
    prot = 0XFF0D;              /* Protection allows User and System access */
    flag = PPL$M_FORMONLY;      /* Form an application, do not join one     */

    status = ppl$create_application ( &size, &app_name, &prot, &flag );
    if (!(status & 1)) return status;

    /* Create work queue to pass message buffer to second process */
    status = ppl$create_work_queue ( &sendq , &workq_name );
    if (!(status & 1)) return status;

    quorum = 2;  /* Need two processes at barrier to pass */

    /* Create first barrier to synchronize processes */
    status = ppl$create_barrier ( &barrier1, &barr1_name, &quorum );
    if (!(status & 1)) return status;

    /* Create second barrier to synchronize processes */
    status = ppl$create_barrier ( &barrier2, &barr2_name, &quorum );
    if (!(status & 1)) return status;
```

(continued on next page)

Example 6–3 (Cont.) Using PPL$ Routines in VAX C

```
    /* Wait at the barrier until second process joins the application */
    status = ppl$wait_at_barrier ( &barrier1 );
    if (!(status & 1)) return status;

    /* Create shared memory zone, obtain an ID, give a specific name */
    status = ppl$create_vm_zone(&mem_id, 0,0,0,0,0,0,0,0,0, &shared_mem);
    if (!(status & 1)) return status;

    /* Wait for joining process to find VM zone */
    status = ppl$wait_at_barrier ( &barrier2 );
    if (!(status & 1)) return status;

    num_bytes = MAXLINE + 1;      /* Size of VM to be created */

    do
    {
        /* Get memory for message */
        status = lib$get_vm(&num_bytes, &message, &mem_id);
        if (!(status & 1)) return status;

        printf("Input>  ");
        gets(message);              /* Get user message */

        /* Put address of user message into workq */
        status = ppl$insert_work_item ( &sendq , message );
        if (!(status & 1)) return status;
    }
    while (strcmp(message, "quit") != 0);

}   /* End of program SEND */

/*                  PROGRAM RECEIVE                                          */

#include    <ppl$def.h>
#include    <ppl$routines.h>
#include    <stdio.h>
#include    <descrip.h>
#define     MAXLINE 256

/* PPL$ object names */
$DESCRIPTOR( barr1_name , "Synch1_barr" );
$DESCRIPTOR( barr2_name , "Synch2_barr" );
$DESCRIPTOR( app_name   , "Intercom"    );
$DESCRIPTOR( workq_name , "Task_queue"  );
$DESCRIPTOR( shared_mem , "Wire"        );

int flag;           /* Flag for applicaton      */
int status;         /* Return status            */
int barrier1;       /* Barrier1 ID              */
int barrier2;       /* Barrier2 ID              */
int receiveq;       /* Work Queue ID - send     */
int mem_id;         /* Shared VM zone ID        */
int procede;        /* Flag used for exiting    */
int num_bytes;      /* Message buffer size      */
char *message;      /* Pointer to shared memory */

int ppl$create_application();

main ( )
{
    flag = PPL$M_JOINONLY;       /* Join, do not form application */
```

(continued on next page)

Example 6–3 (Cont.) Using PPL$ Routines in VAX C

```
    status = ppl$create_application ( 0, &app_name, 0, &flag );
    if (!(status & 1)) return status;

    /* Find the ID for synchronizing barrier1 */
    status = ppl$find_object_id ( &barrier1, &barr1_name );
    if (!(status & 1)) return status;

    /* Synchronize with the other process */
    status = ppl$wait_at_barrier ( &barrier1 );
    if (!(status & 1)) return status;

    /* Find the ID for work queue */
    status = ppl$find_object_id ( &receiveq , &workq_name );
    if (!(status & 1)) return status;

    /* Find the ID for synchronizing barrier2 */
    status = ppl$find_object_id ( &barrier2, &barr2_name );
    if (!(status & 1)) return status;

    /* Allow original process to create VM zone */
    status = ppl$wait_at_barrier ( &barrier2 );
    if (!(status & 1)) return status;

    /* Find the ID for shared memory zone */
    status = ppl$find_object_id( &mem_id, &shared_mem);
    if (!(status & 1)) return status;

    procede = 1;                    /* Used to signal 'quit' */
    num_bytes = MAXLINE + 1;        /* Size of VM zone       */

    do
    {
        /* Get message address from work queue */
        status = ppl$remove_work_item ( &receiveq , &message );
        if (!(status & 1)) return status;

        /* If message is to quit, set procede to signal this */
        if (strcmp(message, "quit") == 0)
            procede = 0;
        else                                /* Print message */
            printf("Message>  %s\n", message);

        /* Free VM in which message was contained */
        status = lib$free_vm(&num_bytes, &message, &mem_id);
        if (!(status & 1)) return status;
    }
    while (procede == 1);
}   /* End of program RECEIVE */
```

In the preceding C example, program SEND is run first so that it can set up the PPL$ environment for program RECEIVE to join. The environment set up by program SEND has twice the default (PPL$K_INIT_SIZE) memory available for PPL$ objects, and allows user and system access to the PPL$ application. An application name is specified in the call to PPL$CREATE_APPLICATION so that program RECEIVE can join the correct application. (If an application name is not specified, an external program cannot join an application.)

When the application is created, program SEND creates a work queue and a barrier. It then waits at the created barrier for RECEIVE to join. The shared memory zone is created after both processes have joined the application.

RECEIVE will only join an existing PPL$ application—it will not create a new application because the PPL$M_JOINONLY flag has been specified in its call to PPL$CREATE_APPLICATION. When RECEIVE joins the application, it uses the PPL$FIND_OBJECT_ID routine to obtain the identifiers of the work queue and barrier that were created by SEND. When the identifiers are obtained, RECEIVE waits at the barrier, allowing both RECEIVE and SEND to proceed.

Communication begins when both programs have joined the PPL$ application, the underlying communication and synchronization objects are in place, and both programs possess those objects' identifiers.

First, SEND obtains a block of virtual memory from the shared memory zone. This is done by passing the shared memory zone identifier to LIB$GET_VM as a parameter. (The identifier was originally retrieved from a call to PPL$CREATE_VM_ZONE.)

SEND prompts the user for one line of input and places it in the newly created shared memory. SEND then passes the address of this memory to RECEIVE. This is done by inserting the address of the memory into a work queue using the PPL$INSERT_WORK_ITEM routine.

RECEIVE calls PPL$REMOVE_WORK_ITEM to obtain the address of the shared memory (specifying the same work queue identifier as SEND specified in its call PPL$INSERT_WORK_QUEUE). If no work items are in the work queue when RECEIVE calls PPL$REMOVE_WORK_ITEM, RECEIVE is blocked until an item (in this case, the address of the shared memory) is inserted into the queue. When RECEIVE removes the address of the shared memory, it displays the contents of the memory location (the character string entered by the user) to the screen and then frees the memory using LIB$FREE_VM. This communication continues until the user inputs the string "quit".

PPL$ Reference Section

This section provides detailed descriptions of the routines provided by the VMS RTL Parallel Processing (PPL$) Facility.

PPL$ADJUST_QUORUM Adjust Barrier Quorum

The Adjust Barrier Quorum routine increments or decrements the quorum associated with a barrier.

FORMAT

PPL$ADJUST_QUORUM *barrier-id ,amount*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

barrier-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the barrier. The **barrier-id** argument is the address of an unsigned longword containing the barrier identifier.

Barrier-id is returned by PPL$CREATE_BARRIER.

amount

VMS usage: **word_signed**
type: **word (signed)**
access: **read only**
mechanism: **by reference**

Value to add to the barrier quorum. The **amount** argument is the address of a signed word containing the amount. You can specify a negative value to decrement the quorum.

DESCRIPTION

PPL$ADJUST_QUORUM allows you to dynamically alter the number of participants expected to wait at a barrier. A quorum is the number of participants required to call PPL$WAIT_AT_BARRIER (and thereby be blocked) before all blocked participants are unblocked and allowed to pass the barrier. The barrier must have been created by PPL$CREATE_BARRIER. (See PPL$CREATE_BARRIER for more information about quorums.)

A barrier's quorum can be dynamically increased or decreased to allow more participants in the quorum. This can be useful when a process that was an expected barrier participant terminates without calling PPL$WAIT_AT_BARRIER. The process that discovers the termination of an expected participant can then call this routine, specifying a value of –1 for the **amount** argument. This adjustment of the barrier quorum results in the conclusion of a barrier wait when sufficient participants are already blocked at the barrier.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INVARG	Invalid argument.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$ADJUST_SEMAPHORE_MAXIMUM Adjust a Semaphore Maximum

The Adjust a Semaphore Maximum routine increments or decrements the maximum associated with a semaphore.

FORMAT

PPL$ADJUST_SEMAPHORE_MAXIMUM *semaphore-id ,amount*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

semaphore-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the semaphore. The **semaphore-id** argument is the address of an unsigned longword containing the identifier.

amount

VMS usage: **word_signed**
type: **word (signed)**
access: **read only**
mechanism: **by reference**

Value to add to the semaphore maximum. The **amount** argument is the address of a signed word containing the amount. Specify a positive value for **amount** to increase the maximum; specify a negative value to decrease the maximum.

DESCRIPTION

PPL$ADJUST_SEMAPHORE_MAXIMUM dynamically increases or decreases the maximum value of a semaphore, therefore allowing you to dynamically alter the number of resources protected by the semaphore. The semaphore's current value is adjusted by the same value you specify for **amount** to reflect the new maximum. A semaphore maximum cannot be decreased by a value that is greater than the current value of the semaphore. The semaphore must have been created by PPL$CREATE_SEMAPHORE.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INVARG	Invalid argument.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$AWAIT_EVENT Await Event Occurrence

The Await Event Occurrence routine blocks the caller until an event occurs.

FORMAT

PPL$AWAIT_EVENT *event-id [,output]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

event-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the event. The **event-id** argument is the address of an unsigned longword containing the identifier.

The **event-id** is returned by PPL$CREATE_EVENT.

output

VMS usage: **user_arg**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the **event-param** argument from PPL$TRIGGER_EVENT. The **output** argument is the address of an unsigned longword that receives the value of **event-param**. The value of **event-param** is copied to **output** when an event is triggered.

DESCRIPTION

PPL$AWAIT_EVENT blocks the caller until a corresponding trigger sets the event's state to *occurred*. (Generally, a trigger is issued when a participant calls PPL$TRIGGER_EVENT. However, the PPL$ facility triggers predefined events automatically.) The caller is blocked by the PPL$ facility's call to the system service $HIBER.

If the event state is *occurred* when this routine is called, the caller continues execution immediately (without blocking), and the event state is reset to *not_occurred*. If the event state is *not_occurred* when this routine is called, the caller is blocked and a request for a wakeup is queued. The caller is awakened when a corresponding trigger is issued for this event.

Refer to Section 4.3.7 for more information about triggering an event.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INSVIRMEM	Insufficient virtual memory available.
PPL$_INVARG	Invalid argument.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$CREATE_APPLICATION Form or Join a PPL$ Application

The Form or Join a PPL$ Application routine informs the PPL$ facility that the calling process is forming or joining a parallel application.

FORMAT

PPL$CREATE_APPLICATION *[size] [,application-name] [,protection] [,flags]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

size

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Number of pages that PPL$ allocates for its internal data structures. The optional **size** argument is the address of an unsigned longword containing this size value. See the Description section for information about the default value.

application-name

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by descriptor**

The name of the application that the calling process will form or join. The optional **application-name** argument is the address of a descriptor pointing to a character string containing the name of the application. The **application-name** argument can contain up to 11 characters.

protection

VMS usage: **file_protection**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Numeric value representing the protection mask to be applied to the application. The optional **protection** argument is the address of an unsigned longword containing this numeric value. For more information, see the description of the $CRMPSC system service in the *VMS System Services Reference Manual.*

flags

VMS usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Specifies options for forming or joining a PPL$ application. The **flags** argument is a longword bit mask containing the flags. Valid values for **flags** are as follows:

PPL$M_FORMONLY	Form a new application only—do not join an existing application. If this flag is not specified, a process will join an application if it already exists.
PPL$M_JOINONLY	Join an existing application only—do not form a new application. If this flag is not specified, a process will form an application if it does not already exist.
PPL$M_PERM	Form a permanent application in which data is maintained even though there are no active processes. By default, application data is lost when the last process in the application exits. Use of this flag requires PRMGBL privilege.
PPL$M_SYSTEM	Form or join a system-wide application. By default, the application is available only to processes running under the same group UIC. Use of this flag requires SYSGBL and SYSLCK privileges.

DESCRIPTION

PPL$CREATE_APPLICATION informs the PPL$ facility that the calling process is forming or joining a parallel application. This routine initializes internal data structures that provide the caller with all of the PPL$ functions. You need only call PPL$CREATE_APPLICATION if you want to specify a value other than the supplied defaults. If you do not call it explicitly, PPL$CREATE_APPLICATION is called automatically when you call one of the routines listed in the following table. Note that PPL$ does not automatically initialize when you call routines that require a previously created element. (PPL$ does not automatically initialize when you call a routine listed in the following table for the second and subsequent times.) This keeps the overhead of these routines—requests for barriers, semaphores, events, spin locks, and work queues—at a minimum.

The routines that perform automatic initialization when first called are:

PPL$CREATE_BARRIER	PPL$FIND_OBJECT_ID
PPL$CREATE_EVENT	PPL$GET_INDEX
PPL$CREATE_SEMAPHORE	PPL$INDEX_TO_PID
PPL$CREATE_SHARED_MEMORY	PPL$PID_TO_INDEX
PPL$CREATE_SPIN_LOCK	PPL$SPAWN
PPL$CREATE_VM_ZONE	PPL$STOP
PPL$CREATE_WORK_QUEUE	PPL$UNIQUE_NAME

The **size** argument determines the amount of space allocated for the supporting PPL$ data structures. If your application terminates with the fatal error PPL$_INSVIRMEM when you call a PPL$ routine, you do not have enough space for the PPL$ routines to perform the requested operation. The lack of space can occur because of the following:

- Your system quotas are not sufficient for the amount of memory requested by the application.
- You have requested PPL$ routines for which the default allocation cannot accommodate the necessary data structures. In this case, you should carefully consider your use of PPL$ routines. You can increase the PPL$ allocation of space for internal data structures by specifying a larger value for the **size** parameter.

By default, PPL$ allocates PPL$K_INIT_SIZE pages for its internal data structures. (PPL$K_INIT_SIZE is available to user programs as a link-time, in other words, external, constant.) This initial allocation provided by PPL$ accommodates a minimum of 32 processes, 8 semaphores, 4 barriers, 4 events, 4 spin locks, 4 work queues, and 16 sections of shared memory. (These numbers represent a rough guideline for combinations of PPL$ components. If you have fewer than 32 processes, for example, you can have more than 8 semaphores, and so forth.) You can increase this allocation by specifying another value, as in the following example:

```
status = PPL$CREATE_APPLICATION (2*PPL$K_INIT_SIZE)
```

If your process was spawned using PPL$SPAWN, and you do not call PPL$CREATE_APPLICATION explicitly (or if you call it explicitly but do not specify an application name), your process joins the same application to which the spawning process is joined. If your process was not spawned using PPL$SPAWN, and you do not call PPL$CREATE_APPLICATION explicitly (or if you call it explicitly but do not specify an application name), PPL$ checks the spawning process to determine if it is a member of a PPL$ application. If it is a member, your process joins that application. Otherwise, the process forms a private (unnamed) application. In a private application, only processes that were spawned by a member of the application can join it. Because the application has no name, no other process may specify it in a call to PPL$CREATE_APPLICATION and therefore cannot join it.

By default, only processes with the same group UIC may participate in the same application. Therefore, if two users with different group UICs run the same parallel application (in other words, a PPL$ application with the same name), two separate applications will run. However, if two users with the same group UIC run the same parallel application, their processes will attempt to form a single application. The same problem results with two invocations of the same system-wide parallel application (one that is initialized with the PPL$M_SYSTEM flag set). It is important that each invocation of an application have a unique name to keep it from interfering with, or being interfered by, another application or invocation of the same application.

The PPL$M_FORMONLY and PPL$M_JOINONLY flags can be used to keep different instances of the same application from interfering with each other. Use the PPL$M_FORMONLY flag in the initialization of a process that expects to form a new PPL$ application. Its initialization will fail with the PPL$_APPALREXI error if an application with that name is already running. Similarly, use the PPL$M_JOINONLY flag in the initialization of a process that expects only to join an existing application. Its initialization will fail with the PPL$_NOSUCHAPP error if the specified application is not currently in existence. Note that PPL$M_JOINONLY and PPL$M_FORMONLY are conflicting options and you cannot specify both in a single call to PPL$CREATE_APPLICATION.

The application protection mask can be used to control the ability of different processes to access a PPL$ application. Access to an application can be granted to the following (similar to file access):

- System processes
- Processes with the same owner as the process that formed the application
- Processes in the same UIC group as the forming process (GROUP privilege is required)
- Processes with different group UICs (WORLD privilege is required)

To participate in a PPL$ application, a process needs read and write access to the application (execute and delete access are not used). Processes that are not in the same group as the forming process may not join the application, regardless of the protection mask, unless the application is initialized with the PPL$M_SYSTEM flag set. (This requires SYSLCK and SYSGBL privileges as well as WORLD.)

By default, PPL$ internal data structures are deallocated when the last process in an application terminates. However, an application's data structures can be preserved when no processes are participating in the application, provided that the application and all shared memory and zones are created with the PPL$M_PERM flag set. (This requires PRMGBL privilege.)

The **size** and **protection** arguments, and the PPL$M_PERM flag are meaningful only at application *formation* and should not be specified by a process that is *joining* an application. If values are specified for these arguments that are incompatible with the existing application, the process's initialization will fail, and PPL$CREATE_APPLICATION will return the PPL$_INCOMPARG error.

CONDITION VALUES RETURNED

PPL$_FORMEDAPP	Successful completion. Formed a new application.
PPL$_JOINEDAPP	Successful completion. Joined an existing application.
PPL$_APPALREXI	The specified application already exists.
PPL$_INCOMPARG	Specified arguments are incompatible with the existing application.

PPL$_INSVIRMEM	Insufficient virtual memory available.
PPL$_INVAPPNAM	Invalid application name or illegal character string.
PPL$_INVARG	Invalid argument.
PPL$_NONPIC	Cannot map shared memory to same addresses as other processes have mapped section.
PPL$_NOSUCHAPP	The specified application does not exist.
PPL$_WRONUMARG	Wrong number or arguments.

Any condition value returned by the system service $CRMPSC.

PPL$CREATE_BARRIER Create a Barrier

The Create a Barrier routine creates and initializes a barrier, and returns the barrier identifier. You use the barrier identifier to perform all operations on that barrier.

FORMAT

PPL$CREATE_BARRIER *barrier-id [,barrier-name] [,quorum]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

barrier-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Identifier of the barrier. The **barrier-id** argument is the address of an unsigned longword containing the identifier. **Barrier-id** must be used in calls to the other barrier routines (listed in the Description section) to identify the barrier.

barrier-name

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by descriptor**

Name of the barrier. The optional **barrier-name** argument is the address of a descriptor pointing to a character string containing the barrier name. The name of the barrier is arbitrary. If you do not specify this argument, or if you specify 0, an unnamed barrier is created. An arbitrary number of unnamed barriers may be created by a given application.

quorum

VMS usage: **word_signed**
type: **word (signed)**
access: **read only**
mechanism: **by reference**

Number of participants required to terminate an active wait for this barrier. The **quorum** argument is the address of a signed word containing the quorum number. For example, a **quorum** value of 3 indicates that the first two callers of PPL$WAIT_AT_BARRIER specifying this **barrier-id** are blocked until a third caller calls PPL$WAIT_AT_BARRIER. At that point, all three participants are released for further processing. If you do not specify a value for **quorum**, a default value of 1 is assigned.

DESCRIPTION

PPL$CREATE_BARRIER creates and initializes a barrier, and returns the barrier identifier. A barrier is a synchronization mechanism that allows an arbitrary number of participants to cooperate by blocking at a given point (generally at the conclusion of a set of work items), until all have reached the barrier.

If an element having the specified **barrier-name** already exists, then the current request must be for the same type of synchronization element. If the types are different, the error PPL$_INCOMPEXI is returned. For example, if a lock of a given name exists, you cannot create a barrier by that name. (The name is case sensitive.) If the elements are of the same type, this routine returns the **barrier-id** of the existing element. A new barrier is created each time a null name is supplied.

It is your responsibility to ensure that the **barrier-id** returned is made available to any other participant in the application using the barrier. You can retrieve the **barrier-id** by naming the barrier and "re-creating" it. That is, after you have created the barrier, all participants that need to access that barrier's identifier call this routine, specifying the same name for the element. This returns the **barrier-id** of the existing barrier and a status of PPL$_ELEALREXI. (Note that this method does not work for unnamed barriers.) Another method is to store the returned **barrier-id** in shared memory.

The value you specify for **quorum** indicates exactly how many participants are required to conclude a wait at that barrier. If you do not specify a value, a default of 1 is assigned for the quorum.

Related routines that implement barrier synchronization are as follows:

PPL$DELETE_BARRIER	Deletes the barrier and release any storage associated with it.
PPL$WAIT_AT_BARRIER	Waits until the quorum reaches the barrier.
PPL$READ_BARRIER	Returns the barrier's quorum and number of waiting participants.
PPL$SET_QUORUM	Establishes the initial quorum for the barrier.
PPL$ADJUST_QUORUM	Increments or decrements a barrier's quorum.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_ELEALREXI	Successful completion. An element of the same name already exists.
PPL$_INCOMPEXI	Incompatible type of element with the same name already exists.
PPL$_INSVIRMEM	Insufficient virtual memory available.
PPL$_INVARG	Invalid argument.
PPL$_INVELENAM	Invalid element name or illegal character.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$CREATE_EVENT Create an Event

The Create an Event routine creates an arbitrary user-defined event and returns the event identifier. You use the event identifier to perform all operations on that event.

FORMAT

PPL$CREATE_EVENT *event-id [,event-name]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

event-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Identifier of the event. The **event-id** argument is the address of an unsigned longword containing the identifier. **Event-id** must be used in other calls to identify the event.

event-name

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by descriptor**

Name of the event. The **event-name** argument is the address of a descriptor pointing to a character string containing the event name. The name of the event is entirely arbitrary. If you do not specify a value for **event-name**, or if you specify 0, a new unnamed event is created, which can be referenced only by its identifier. An arbitrary number of unnamed events can be created by a given application.

DESCRIPTION

PPL$CREATE_EVENT creates an arbitrary user-defined event and returns its identifier, which is used in subsequent calls to other PPL$ event routines.

If an element having the specified **event-name** already exists, then the current request must be for the same type of synchronization element. If the types are different, the error PPL$_INCOMPEXI is returned. For example, if a lock of a given name exists, you cannot create an event by that name. (The names are case sensitive.) If the elements are of the same type, this routine returns the **event-id** of the existing element. A new event is created each time a null name is supplied.

It is your responsibility to ensure that the **event-id** returned is made available to any other participant in the application using the event. You can retrieve the **event-id** by naming the event and "re-creating" it. That is, after you have created the event, all participants that need to access that event's identifier call this routine, specifying the same name for the element. This returns the **event-id** of the existing event and a status of PPL$_ELEALREXI. (Note that this method does not work for anonymous events.) Another method is to store the returned **event-id** in shared memory.

An event is a synchronization mechanism having an associated state that may be either *occurred* or *not_occurred.* (A call to this routine initializes the state to *not_occurred.*) A participant can trigger an event (by calling PPL$TRIGGER_EVENT), as well as enable an action to be taken when an event is triggered. When a participant triggers an event, it may request that either exactly one pending action is processed, or that all pending actions are processed. An action is either an AST, a signal (condition), or a wakeup.

Refer to Section 4.3.7 for more information about triggering an event.

Related routines that implement event operations are as follows:

PPL$AWAIT_EVENT	Blocks the caller until the event state becomes *occurred.* If the state is already *occurred* when this routine is called, the state is reset to *not_occurred* and the caller continues processing without being blocked. (If there is a queued trigger for the event when this routine is called, then once again the state immediately becomes *occurred.*) If the event is *not_occurred* when this routine is called, the caller is blocked, to be awakened by a corresponding trigger for this event.
PPL$DELETE_EVENT	Deletes the event and releases any storage associated with it.
PPL$DISABLE_EVENT	Disables delivery of event notification to the calling process by AST or signal, or both.
PPL$ENABLE_EVENT_AST	Requests that a specified AST be delivered when the event has occurred. If the state is already *occurred* when this routine is called, the AST is immediately delivered and the state is reset to *not_occurred.* (If there is a queued trigger for the event when this routine is called, then once again the state immediately becomes *occurred.*) If the state is *not_occurred* when this routine is called, the request is queued to the event, and the AST is delivered as a result of a corresponding trigger for this event.

PPL$ENABLE_EVENT_SIGNAL	Requests that a specified signal condition be delivered when the event is *occurred.* If the state is already *occurred* when this routine is called, the signal is immediately delivered and the state is reset to *not_occurred.* (If there is a queued trigger for the event when this routine is called, then once again the state immediately becomes *occurred.*) Otherwise, the request is queued to the event, and the signal will be delivered as a result of a corresponding trigger for this event.
PPL$READ_EVENT	Returns the current state of the event. The state can be *occurred* or *not_occurred.*
PPL$RESET_EVENT	Resets the event state to *not_occurred.* Any queued calls to PPL$TRIGGER_EVENT are removed from the queue.
PPL$TRIGGER_EVENT	Sets the event state to *occurred* and examines the queue of requested operations. If any signals or ASTs have been enabled for the event, or if any participant is waiting for the event, the appropriate action is taken and the event state is reset to *not_occurred.* If the event state is already *occurred,* then the trigger is queued for later processing.

The PPL$ facility creates and predefines the events PPL$K_NORMAL_EXIT and PPL$K_ABNORMAL_EXIT. You need not create these events. (These events are described in the following sections.) When a normal or abnormal exit occurs, PPL$ triggers the event automatically. Note that you can ignore these predefined events at no cost. However, DIGITAL recommends that you enable event notification of PPL$K_ABNORMAL_EXIT, because that condition usually indicates a severe error. Notification is delivered only if you explicitly request it by specifying the predefined event as the **event-id** in a call to PPL$ENABLE_EVENT_SIGNAL, PPL$ENABLE_EVENT_AST, or PPL$AWAIT_EVENT.

1 PPL$K_NORMAL_EXIT—PPL$ triggers this event when an application participant exits normally. Normal exits include the following:

- The participant returns a success status
- The participant calls PPL$TERMINATE
- The subordinate's parent calls PPL$TERMINATE specifying PPL$M_STOP_CHILDREN
- Some other participant calls PPL$STOP to terminate this participant

If you enabled a signal for this event through a call to PPL$ENABLE_EVENT_SIGNAL, the condition signaled as the trigger parameter is PPL$_NORMAL_EXIT.

2 PPL$K_ABNORMAL_EXIT—PPL$ triggers this event when an application participant exits abnormally. Abnormal exits include the following:

- The participant returns an error status
- A mechanism outside of PPL$ forces termination and prevents the execution of exit handlers (for example, the DCL command STOP/ID)

If you enabled a signal for this event through a call to PPL$ENABLE_EVENT_SIGNAL, the condition signaled as the trigger parameter is PPL$_ABNORMAL_EXIT.

There are some special usage considerations for the PPL$ predefined events if delivery of a signal is requested. Refer to the Description section of PPL$ENABLE_EVENT_SIGNAL for more information.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_ELEALREXI	Successful completion. An element of the same name already exists.
PPL$_INCOMPEXI	Incompatible type of element with the same name already exists.
PPL$_INSVIRMEM	Insufficient virtual memory available.
PPL$_INVARG	Invalid argument.
PPL$_INVELENAM	Invalid element name or illegal character.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$CREATE_SEMAPHORE Create a Semaphore

The Create a Semaphore routine creates and initializes a semaphore with a waiting queue, and returns the semaphore identifier. You use the semaphore identifier to perform all operations on that semaphore.

FORMAT

PPL$CREATE_SEMAPHORE *semaphore-id*
[,semaphore-name]
[,semaphore-maximum]
[,semaphore-initial]

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

semaphore-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Identifier of the semaphore. The **semaphore-id** argument is the address of an unsigned longword containing the identifier. **Semaphore-id** must be used in other calls to identify the semaphore.

semaphore-name

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by descriptor**

Name of the semaphore. The **semaphore-name** argument is the address of a descriptor pointing to a character string containing the semaphore name. The name of the semaphore is entirely arbitrary. If you do not specify a value for **semaphore-name**, or if you specify 0, a new unnamed semaphore is created. An arbitrary number of unnamed semaphores may be created by a given application.

semaphore-maximum

VMS usage: **word_signed**
type: **word (signed)**
access: **read only**
mechanism: **by reference**

Maximum value of the semaphore. The **semaphore-maximum** argument is the address of a signed word containing the maximum value. This value must be nonnegative. If you do not supply a value for **semaphore-maximum**, a default value of 1 is used, thereby making it a binary semaphore.

semaphore-initial

VMS usage: **word_signed**
type: **word (signed)**
access: **read only**
mechanism: **by reference**

Initial value of the semaphore. The **semaphore-initial** argument is the address of a signed word containing the initial value. This value must be less than or equal to the **semaphore-maximum** value. If you do not supply a value for **semaphore-initial**, a default value equal to **semaphore-maximum** is used.

DESCRIPTION

PPL$CREATE_SEMAPHORE creates and initializes a semaphore and a waiting queue, and returns the identifier of the semaphore. The semaphore created may be used to control access to any user-defined resource.

If an element having the specified **semaphore-name** already exists, then the current request must be for the same type of synchronization element. If the types are different, the error PPL$_INCOMPEXI is returned. For example, if a lock of a given name exists, you cannot create a semaphore by that name. (The name is case sensitive.) If the elements are of the same type, this routine returns the **semaphore-id** of the existing element. A new semaphore is created each time a null name is supplied.

It is your responsibility to ensure that the **semaphore-id** returned is made available to any other participant in the application using the semaphore. You can retrieve the **semaphore-id** by naming the semaphore and "re-creating" it. That is, after you have created the semaphore, all participants that need to access that semaphore's identifier call this routine, specifying the same name for the element. This returns the **semaphore-id** of the existing semaphore and a status of PPL$_ELEALREXI. (Note that this method does not work for unnamed semaphores.) Another method is to store the returned **semaphore-id** in shared memory. Refer to Section 5.2.6 for more information.

Depending on the value specified for **semaphore-maximum**, you can create either a binary semaphore (**semaphore-maximum** = 1) or a counting semaphore (**semaphore-maximum** > 1).

Related routines that implement semaphore synchronization are as follows:

PPL$ADJUST_SEMAPHORE_MAXIMUM	Increments or decrements the maximum value of a semaphore.
PPL$DECREMENT_SEMAPHORE	Waits for the semaphore to have a value greater than zero, then decrements the semaphore.
PPL$DELETE_SEMAPHORE	Deletes a semaphore and releases any storage associated with it.
PPL$INCREMENT_SEMAPHORE	Increments the semaphore and wakes a participant blocked by the semaphore, if any exists.

PPL$READ_SEMAPHORE	Returns the current and/or maximum values of a semaphore.
PPL$SET_SEMAPHORE_MAXIMUM	Dynamically sets the maximum value of a semaphore.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_ELEALREXI	Successful completion. An element of the same name already exists.
PPL$_INCOMPEXI	Incompatible type of element with the same name already exists.
PPL$_INSVIRMEM	Insufficient virtual memory available.
PPL$_INVARG	Invalid argument.
PPL$_INVELENAM	Invalid element name or illegal character.
PPL$_INVSEMINI	Invalid semaphore initial value; cannot be greater than the maximum value.
PPL$_INVSEMMAX	Invalid semaphore maximum value; must be greater than zero.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$CREATE_SHARED_MEMORY Create Shared Memory

The Create Shared Memory routine creates (if necessary) and maps a section of memory that can be shared by multiple processes.

FORMAT

PPL$CREATE_SHARED_MEMORY *section-name*
,memory-area
[,flags]
[,file-name]
[,protection]

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

section-name

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by descriptor**

Name of the shared memory section you want to create. The **section-name** argument is the address of a descriptor pointing to the shared memory section name.

memory-area

VMS usage: **vector_longword_unsigned**
type: **longword (unsigned)**
access: **modify**
mechanism: **by reference, array reference**

The area of memory into which the shared memory is mapped. The **memory-area** argument is the address of a two-longword array containing, in order, the length (in bytes) and the starting virtual address for the area of memory.

If you specify the starting address as zero, the PPL$ facility selects the virtual address space so that each current process in the application can map the section to the same set of virtual addresses.

PPL$CREATE_SHARED_MEMORY returns to this argument the actual length and starting virtual address of the shared memory created or mapped.

flags

VMS usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Specifies options for creating and mapping shared memory. The **flags** argument is the address of a longword bit mask containing the flag. Valid values are as follows:

PPL$M_NOZERO	Does not initialize the shared memory to zero. By default, PPL$CREATE_SHARED_MEMORY initializes the shared memory to zero.
PPL$M_NOWRT	Maps the shared memory with no write access (in other words, read only). By default, the shared memory is available with read/write access.
PPL$M_NOUNI	Names the shared memory a nonunique name. By default, PPL$CREATE_SHARED_MEMORY gives the specified shared memory a name unique to the application by using PPL$UNIQUE_NAME.
PPL$M_PERM	Creates permanent shared memory in which data is maintained even though there are no active processes. The default is determined by your call to PPL$CREATE_APPLICATION: if you specify the PPL$M_PERM flag in your call to PPL$CREATE_APPLICATION, this behavior is the default and you do not need to specify PPL$M_PERM in your call to PPL$CREATE_SHARED_MEMORY. If you do not specify the PPL$M_PERM flag in your calls to PPL$CREATE_APPLICATION and PPL$CREATE_SHARED_MEMORY, application data is lost when the last process in the application exits. Use of this flag requires PRMGBL privilege.
PPL$M_SYSTEM	Creates system-wide shared memory. The default is determined by your call to PPL$CREATE_APPLICATION: if you specify the PPL$M_SYSTEM flag in your call to PPL$CREATE_APPLICATION, this behavior is the default and you do not need to specify PPL$M_SYSTEM in your call to PPL$CREATE_SHARED_MEMORY. If you do not specify the PPL$M_SYSTEM flag in your calls to PPL$CREATE_APPLICATION and PPL$CREATE_SHARED_MEMORY, the application is available only to processes running under the same group UIC. Use of this flag requires the SYSGBL privilege.

file-name

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by descriptor**

Name of the file used for backup storage of the shared memory. The **filename** argument is the address of a descriptor pointing to the file name. The size of the resulting address space is the smaller of the following:

the specified section size
the size of the file being mapped

If you do not specify a file name, PPL$CREATE_SHARED_MEMORY creates for backup storage a page file section instead of a disk file section.

If you specify a file that does not exist, PPL$CREATE_SHARED_MEMORY creates it.

protection

VMS usage: **file_protection**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Numeric value representing the protection mask to be applied to the shared memory. The optional **protection** argument is the address of an unsigned longword containing this numeric value. If you do not specify a value, the default is the value for **protection** specified in the call to PPL$CREATE_APPLICATION. For more information, see the description of the $CRMPSC system service in the *VMS System Services Reference Manual*.

DESCRIPTION

PPL$CREATE_SHARED_MEMORY creates (if necessary) and maps a section of memory that can be shared by multiple processes. Within VMS, a global section (or shared memory) is a data structure or shareable image section potentially available to all processes in the system. See the *VMS System Services Reference Manual* for more information on global sections.

By default, PPL$CREATE_SHARED_MEMORY gives the shared memory a name unique to the application, initializes the section to zero, and maps the section with read/write access. You use the **flags** argument to change any or all of those defaults. In addition, all other participants share the same memory addresses if possible. This operation merely attempts to "reserve" that address range, and it is only mapped in other participants at the time they issue calls to this routine. If PPL$CREATE_SHARED_MEMORY cannot map the shared memory to the same addresses in all participants, the memory is not mapped and PPL$_NONPIC is returned. (This might occur when the application executes more than one different program image.)

Optionally, this routine opens a backup storage file for the shared memory with a specified file name.

The PPL$ facility offers two distinct memory sharing services through this routine. The first mechanism lets you request an unspecified range of addresses, and the PPL$ facility arranges to allocate the same set of addresses in each participant in the application. You request this service by specifying the starting address as zero. If you allow the PPL$ facility to select the virtual addresses for a section of shared memory, PPL$ selects the virtual addresses so that each process already in the application can map the section to the same address range. A participant that joins the

application after the shared memory is created may not be able to access the shared memory if the new participant's image size is significantly larger than the image size of the participant(s) that created the shared memory. If you have difficulty creating shared memory, be sure that all participants that will use the section have joined the application *before* the shared memory is created.

The second mechanism lets you specify a particular range of addresses to be shared. This allows the sharing of an arbitrary collection of variables that appears at a certain address, such as a FORTRAN common block. Because VMS maps memory in pages (512 bytes), you must take care to share exactly the data intended for sharing—no more and no less. When the data does not fall exactly on page boundaries, extra effort is required to prevent accidental sharing of local data while guaranteeing that all participants can access the shared memory at the expected addresses. You can accomplish this by allocating a 512-byte array at both the beginning and the end of such a data area (common block). The request to this routine then specifies the starting address to be that of the front "guard" array. The length is calculated by subtracting the last address of the end "guard page" from the starting address of the front guard. PPL$ maps the requested memory so that the lower address is rounded up to the nearest page boundary, and the higher address is rounded down to the nearest page boundary. This guarantees that no data is shared unexpectedly, and that all important data in the common area (that is, everything but the two guard pages) is fully shared.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_CREATED	Successful completion. Shared memory created.
PPL$_INVARG	Invalid argument.
PPL$_NONPIC	Cannot map shared memory to same addresses as other processes have mapped section.
PPL$_WRONUMARG	Wrong number of arguments.
RMS$_xxx	Miscellaneous RMS errors pertaining to file name.

Any error returned by the system service $CRMPSC.

PPL$CREATE_SPIN_LOCK Create Spin Lock

The Create Spin Lock routine creates and initializes a simple (spin) lock, and returns the lock identifier. You use that lock identifier to get and free the lock.

FORMAT

PPL$CREATE_SPIN_LOCK *lock-id [,lock-name]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

lock-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Identifier of the newly created lock. The **lock-id** argument is the address of an unsigned longword containing the lock identifier. You must use **lock-id** when getting or freeing the lock.

lock-name

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by descriptor**

Name of the lock. The **lock-name** argument is the address of a descriptor pointing to a character string containing the name. The name of the lock is entirely arbitrary. If you do not specify this argument, or if you specify 0, an unnamed lock is created. An arbitrary number of unnamed locks can be created by a given application.

DESCRIPTION

PPL$CREATE_SPIN_LOCK creates and initializes a simple lock, and returns the lock identifier. The lock is initialized to zero (not set).

If an element having the specified **lock-name** already exists, then the current request must be for the same type of synchronization element. If the types are different, the error PPL$_INCOMPEXI is returned. For example, if a barrier of a given name exists, you cannot create a lock by that name. (The name is case sensitive.) If the elements are of the same type, this routine returns the **lock-id** of the existing element. A new lock is created each time a null name is supplied.

It is your responsibility to ensure that the **lock-id** returned is made available to any other participant in the application using the lock. You can retrieve the **lock-id** by naming the lock and "re-creating" it. That is, after you have created the lock, all participants that need to access

that lock's identifier call this routine, specifying the same name for the element. This returns the **lock-id** of the existing lock and a status of PPL$_ELEALREXI. (Note that this method does not work for unnamed anonymous locks.) Another method is to store the returned **lock-id** in shared memory.

Related routines that implement spin lock synchronization are as follows:

PPL$DELETE_SPIN_LOCK	Deletes a spin lock and releases any storage associated with it.
PPL$READ_SPIN_LOCK	Returns the current state of the spin lock. The state can be *seized* or *not_seized*.
PPL$RELEASE_SPIN_LOCK	Releases the lock.
PPL$SEIZE_SPIN_LOCK	Obtains the lock for exclusive access.

This form of lock is recommended for use only in a dedicated parallel processing environment, and only when fairness is not important. This lock is not recommended for use in a general time-sharing environment because in that environment a spin lock consumes CPU resources.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_ELEALREXI	Successful completion. An element of the same name already exists.
PPL$_INCOMPEXI	Incompatible type of element with the same name already exists.
PPL$_INVARG	Invalid argument.
PPL$_INVELENAM	Invalid element name or illegal character string.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$CREATE_VM_ZONE Create a New Virtual Memory Zone

The Create a New Virtual Memory Zone routine creates a new storage zone, according to specified arguments, which is available to all participants in the application.

FORMAT

PPL$CREATE_VM_ZONE *zone-id [,algorithm] [,algorithm-argument] [,flags] [,extend-size] [,initial-size] [,block-size] [,alignment] [,page-limit] [,smallest-block-size] [,zone-name]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

zone-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Zone identifier. The **zone-id** argument is the address of a longword that is set to the zone identifier of the newly created zone.

algorithm

VMS usage: **longword_signed**
type: **longword (signed)**
access: **read only**
mechanism: **by reference**

Algorithm. The **algorithm** argument is the address of a signed longword that represents the code for one of the LIB$VM algorithms:

1	LIB$K_VM_FIRST_FIT	First fit
2	LIB$K_VM_QUICK_FIT	Quick fit, lookaside list
3	LIB$K_VM_FREQ_SIZES	Frequent sizes, lookaside list
4	LIB$K_VM_FIXED	Fixed size blocks

If **algorithm** is not specified, a default of 1 (first fit) is used.

algorithm-argument

VMS usage: **longword_signed**
type: **longword (signed)**
access: **read only**
mechanism: **by reference**

Algorithm argument. The **algorithm-argument** argument is the address of a signed longword that contains a value that is specific to the particular allocation algorithm.

Algorithm	Value
QUICK_FIT	The number of queues used. The number of queues must be between 1 and 128.
FREQ_SIZES	The number of cache slots used. The number of cache slots must be between 1 and 16.
FIXED	The fixed request size (in bytes) for each get or free. The request size must be greater than 0.
FIRST_FIT	Not used, may be omitted.

The **algorithm-argument** argument must be specified if you are using the quick-fit, frequent-sizes or fixed-size-blocks algorithms. However, this argument is optional if you are using the first-fit algorithm.

flags

VMS usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Flags. The **flags** argument is the address of an unsigned longword that contains flag bits that control various options:

Bit	Value	Description
Bit 0	LIB$M_VM_BOUNDARY_TAGS	Boundary tags for faster freeing Adds a minimum of eight bytes to each block
Bit 1	LIB$M_VM_GET_FILL0	LIB$GET_VM; fill with bytes of 0
Bit 2	LIB$M_VM_GET_FILL1	LIB$GET_VM; fill with bytes of FF (hexadecimal)
Bit 3	LIB$M_VM_FREE_FILL0	LIB$FREE_VM; fill with bytes of 0
Bit 4	LIB$M_VM_FREE_FILL1	LIB$FREE_VM; fill with bytes of FF (hexadecimal)
Bit 5	LIB$M_VM_EXTEND_AREA	Add extents to existing areas if possible

Bits 6 through 31 are reserved and must be 0.

This is an optional argument. If **flags** is omitted, the default of 0 (no fill and no boundary tags) is used.

extend-size

VMS usage: **longword_signed**
type: **longword (signed)**
access: **read only**
mechanism: **by reference**

Zone extend size. The **extend-size** argument is the address of a signed longword that contains the number of (512-byte) pages to be added to the zone each time it is extended.

The value of **extend-size** must be between 1 and 1024.

This is an optional argument. If **extend-size** is not specified, a default of 16 pages is used.

Note: *Extend-size* does not limit the number of blocks that can be allocated from the zone. The actual extension size is the greater of extend-size and the number of pages needed to satisfy the LIB$GET_VM call that caused the extend.

initial-size

VMS usage: **longword_signed**
type: **longword (signed)**
access: **read only**
mechanism: **by reference**

Initial size for the zone. The **initial-size** argument is the address of a signed longword that contains the number of (512-byte) pages to be allocated for the zone as the zone is created.

This is an optional argument. If **initial-size** is not specified or is specified as 0, no pages are allocated when the zone is created. The first call to LIB$GET_VM for the zone allocates **extend-size** pages.

block-size

VMS usage: **longword_signed**
type: **longword (signed)**
access: **read only**
mechanism: **by reference**

Block size of the zone. The **block-size** argument is the address of a signed longword specifying the allocation quantum (in bytes) for the zone. All blocks allocated are rounded up to a multiple of **block-size**.

The value of **block-size** must be a power of 2 between 8 and 512. This is an optional argument. If **block-size** is not specified, a default of 8 is used.

alignment

VMS usage: **longword_signed**
type: **longword (signed)**
access: **read only**
mechanism: **by reference**

Block alignment. The **alignment** argument is the address of a signed longword that specifies the required address alignment (in bytes) for each block allocated.

The value of **alignment** must be a power of 2 between 4 and 512. This is an optional argument. If **alignment** is not specified, a default of 8 (quadword alignment) is used.

page-limit

VMS usage: **longword_signed**
type: **longword (signed)**
access: **read only**
mechanism: **by reference**

Maximum page limit. The **page-limit** argument is the address of a signed longword that specifies the maximum number of (512-byte) pages that can be allocated for the zone. The value of **page-limit** must be between 0 and 32,767. Note that part of the zone is used for header information.

This is an optional argument. If **page-limit** is not specified or is specified as 0, the only limit is the total process virtual address space limit imposed by VMS. If **page-limit** is specified, then **initial-size** must also be specified.

smallest-block-size

VMS usage: **longword_signed**
type: **longword (signed)**
access: **read only**
mechanism: **by reference**

Smallest block size. The **smallest-block-size** argument is the address of a signed longword that specifies the smallest block size (in bytes) that has a queue for the quick fit algorithm.

If **smallest-block-size** is not specified, the default of **block-size** is used. That is, queues are provided for the first *n* multiples of **block-size**.

zone-name

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by descriptor**

Name to be associated with the zone being created. The optional **zone-name** argument is the address of a descriptor pointing to a character string containing the zone name. If **zone-name** is not specified, the zone does not have an associated name.

DESCRIPTION

PPL$CREATE_VM_ZONE creates a new storage zone. The zone identifier value that is returned can be used in calls to the following LIB$ routines:

LIB$FREE_VM	LIB$RESET_VM_ZONE
LIB$GET_VM	LIB$SHOW_VM_ZONE
LIB$DELETE_VM_ZONE	LIB$VERIFY_VM_ZONE

The arguments for PPL$CREATE_VM_ZONE are identical to those for LIB$CREATE_VM_ZONE, except for the last two arguments; PPL$CREATE_VM_ZONE does not accept the **get-page** and **free-page** arguments provided by LIB$CREATE_VM_ZONE. For more information about the RTL LIB$ virtual memory zone routines, refer to the *VMS RTL Library (LIB$) Manual*.

The restrictions for LIB$RESET_VM_ZONE also apply to shared zones. That is, it is the caller's responsibility to ensure that the called program has exclusive access to the zone while the reset operation is being performed.

All participants in the application share the memory allocated by calls to LIB$GET_VM. Memory allocated by one process may be freed by another process.

It is your responsibility to ensure that the **zone-id** returned is made available to any other participant in the application using the zone. You can retrieve the **zone-id** by naming the zone and "recreating" it. That is, after you have created the zone, all participants that need to access that zone's identifier call this routine, specifying the same name for the element. This returns the **zone-id** of the existing zone and a status of PPL$_ELEALREXI. (Note that this method does not work for unnamed zones.) Another method is to store the returned **zone-id** in shared memory.

If an error status is returned, the zone is not created.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_ELEALREXI	Successful completion. An element of the same name already exists.
PPL$_INCOMPEXI	Incompatible type of element with the same name already exists.
PPL$_INSVIRMEM	Insufficient virtual memory available.
PPL$_INVARG	Invalid argument.

Any error returned by LIB$CREATE_VM_ZONE.

PPL$CREATE_WORK_QUEUE Create a Work Queue

The Create a Work Queue routine creates and initializes a work queue, and returns the work queue identifier.

FORMAT

PPL$CREATE_WORK_QUEUE *queue-id [,queue-name]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

queue-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

The work queue identifier. The **queue-id** argument is the address of an unsigned longword containing the identifier. **Queue-id** must be used in calls to the other work queue routines to identify the work queue.

queue-name

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by descriptor**

Name of the work queue. The optional **queue-name** argument is the address of a descriptor pointing to a character string containing the work queue name. The work queue name is case sensitive. If you do not specify this argument, or if you specify 0, an unnamed work queue is created. An arbitrary number of unnamed work queues may be created by a given application.

DESCRIPTION

PPL$CREATE_WORK_QUEUE creates and initializes a work queue, and returns the identifier of the work queue.

A parallel application that uses a work queue consists of a work queue of work items and participants to complete the work items. One or more participants serve as task dispatchers. These participants place work items that identify a task to be performed into a work queue. Other participants (servers) remove the work items from the work queue and execute the indicated task. When there is no work to be done, the dispatchers await input, and the servers block on the empty work queue.

If a PPL$ element having the specified **queue-name** already exists, then the current request must be for the same type of element. For example, if a semaphore of a given name exists, you cannot create a work queue by that name. If the types are different, the error PPL$_INCOMPEXI is returned. If the elements are of the same type, this routine returns the **queue-id** of the existing element. A new work queue is created every time a null name is specified.

It is your responsibility to ensure that the **queue-id** returned is made available to any other participant in the application using the work queue. You can retrieve the **queue-id** by naming the work queue and "recreating" it. That is, after you have created the work queue, all participants that need to access that work queue's identifier call this routine, specifying the same name for the element. This returns the **queue-id** of the existing work queue and a status of PPL$_ELEALREXI. (Note that this method does not work for unnamed work queues.) Another method is to store the returned **queue-id** in shared memory.

Other routines that manipulate work queues are:

PPL$DELETE_WORK_QUEUE	Deletes a work queue.
PPL$READ_WORK_QUEUE	Retrieves the number of items in a work queue or the number of waiting processes.
PPL$DELETE_WORK_ITEM	Deletes a specified item from a work queue.
PPL$INSERT_WORK_ITEM	Inserts an item into a work queue.
PPL$REMOVE_WORK_ITEM	Removes the next item in order from a work queue.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_ELEALREXI	Successful completion. An element of the same name already exists.
PPL$_INCOMPEXI	Incompatible type of element with the same name already exists.
PPL$_INSVIRMEM	Insufficient virtual memory available.
PPL$_INVARG	Invalid argument.
PPL$_INVELENAM	Invalid element name or illegal character string.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$DECREMENT_SEMAPHORE Decrement a Semaphore

The Decrement a Semaphore routine waits for a semaphore to have a value greater than 0, then decrements the value by 1 to indicate the allocation of a resource.

FORMAT

PPL$DECREMENT_SEMAPHORE *semaphore-id [,flags]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

semaphore-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the semaphore. The **semaphore-id** argument is the address of an unsigned longword containing the identifier.

Semaphore-id is returned by PPL$CREATE_SEMAPHORE.

flags

VMS usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Bit mask specifying options for decrementing the semaphore. The **flags** argument is a longword bit mask containing the flag. The valid value for **flags** is as follows:

PPL$M_NON_BLOCKING	The semaphore is decremented if and only if it can be decremented without causing the caller to be blocked. (This can be useful in situations where the cost of waiting for a resource is not desirable, or if the caller merely intends to request immediate access to any one of a number of resources.)

DESCRIPTION

PPL$DECREMENT_SEMAPHORE waits for a semaphore to have a value greater than 0, then decrements the value by 1 to indicate the allocation of a resource. If the value of the semaphore is 0 at the time of the call, the caller is put in the queue and suspended, unless the PPL$M_NON_BLOCKING value for the **flags** argument is specified. If you specify PPL$M_NON_BLOCKING, the caller is not blocked, the semaphore is

not decremented, and the routine returns the status code PPL$_NOT_AVAILABLE. The semaphore must have been created by PPL$CREATE_SEMAPHORE.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INVARG	Invalid argument.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_NOT_AVAILABLE	Operation cannot be performed immediately; therefore it is not performed.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$DELETE_APPLICATION Delete a PPL$ Application

The Delete a PPL$ Application routine marks all shared memory for deletion and prevents additional processes from joining the application.

FORMAT

PPL$DELETE_APPLICATION

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

None.

DESCRIPTION

PPL$DELETE_APPLICATION marks all shared memory in an application for deletion. This includes the PPL$ internal data area, all shared memory sections, and shared zone sections. Because the shared memory is not actually deallocated until the last process exits, this routine has no effect on processes that are already members of the application. However, after you call this routine, no new processes are allowed to join the application. The process calling this routine requires the PRMGBL privilege.

If a process attempts to join an application that has been deleted, PPL$ instead forms a new application with the same name (subject to the options specified in PPL$CREATE_APPLICATION). This prevents completely separate instances of an application with the same name from interfering with each other.

Calling PPL$DELETE_APPLICATION is the only way to remove a permanent application (one which was formed with the PPL$M_PERM flag set in PPL$CREATE_APPLICATION). After calling PPL$DELETE_APPLICATION, the application is no longer permanent. When the last process leaves the application, all shared memory sections are deallocated, and the application is deleted.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.

Any condition value returned by the system service $DGBLSC.

PPL$DELETE_BARRIER Delete a Barrier

The Delete a Barrier routine deletes a barrier and releases any storage associated with it.

FORMAT

PPL$DELETE_BARRIER *[barrier-id] [,barrier-name]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

barrier-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the barrier. The optional **barrier-id** argument is the address of an unsigned longword containing the barrier identifier.

barrier-name

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by descriptor**

Name of the barrier. The optional **barrier-name** argument is the address of a descriptor pointing to a character string containing the barrier name.

DESCRIPTION

PPL$DELETE_BARRIER deletes a specified barrier and releases any storage associated with it. A barrier may be specified by either its name or by its identifier. Unnamed barriers must be deleted by specifying the **barrier-id**.

You cannot delete a barrier if there are participants waiting at the barrier. If you attempt to delete a barrier at which participants are waiting, PPL$ returns the PPL$_ELEINUSE error. (Call PPL$ADJUST_QUORUM to release the waiting participants before deleting the barrier.) None of the participants in the application can perform any further operations on the barrier after you call PPL$DELETE_BARRIER.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_ELEINUSE	The specified element is currently in use and cannot be deleted.

PPL$_INVARG	Invalid argument.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_NOSUCHELE	The element you specified does not exist.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$DELETE_EVENT Delete an Event

The Delete an Event routine deletes an event and releases any storage associated with it.

FORMAT

PPL$DELETE_EVENT *[event-id] [,event-name]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

event-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the event. The optional **event-id** argument is the address of an unsigned longword containing the event identifier.

event-name

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by descriptor**

Name of the event. The optional **event-name** argument is the address of a descriptor pointing to a character string containing the event name.

DESCRIPTION

PPL$DELETE_EVENT deletes a specified event and releases any storage associated with it. An event can be specified either by its name or by its identifier. Unnamed events must be deleted by specifying the **event-id**.

You cannot delete an event if there are participants waiting for the event to occur. If you attempt to delete such an event, PPL$ returns the PPL$_ELEINUSE error. (Call PPL$TRIGGER_EVENT to release the waiting participants before deleting an event.) However, an event can be deleted if other participants have enabled notification of the event, or if there are outstanding triggers queued for the event. None of the participants in the application can perform any further operations on the event after you call PPL$DELETE_EVENT.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_ELEINUSE	The specified element is currently in use and cannot be deleted.
PPL$_INVARG	Invalid argument.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_NOSUCHELE	The element you specified does not exist.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$DELETE_SEMAPHORE Delete a Semaphore

The Delete a Semaphore routine deletes a semaphore and releases any storage associated with it.

FORMAT

PPL$DELETE_SEMAPHORE *[semaphore-id] [,semaphore-name]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

semaphore-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the semaphore. The optional **semaphore-id** argument is the address of an unsigned longword containing the semaphore identifier.

semaphore-name

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by descriptor**

Name of the semaphore. The optional **semaphore-name** argument is the address of a descriptor pointing to a character string containing the semaphore-name.

DESCRIPTION

PPL$DELETE_SEMAPHORE deletes a specified semaphore and releases any storage associated with it. A semaphore can be specified either by its name or by its identifier. Unnamed semaphores must be deleted by specifying **semaphore-id**.

You cannot delete a semaphore if there are participants waiting for the semaphore. If you attempt to delete a semaphore for which participants are waiting, PPL$ returns the PPL$_ELEINUSE error. (Call PPL$INCREMENT_SEMAPHORE to release waiting participants before deleting the semaphore.) None of the participants in the application can perform any further operations on the semaphore after you call PPL$DELETE_SEMAPHORE.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_ELEINUSE	The specified element is currently in use and cannot be deleted.
PPL$_INVARG	Invalid argument.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_NOSUCHELE	The element you specified does not exist.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$DELETE_SHARED_MEMORY Delete Shared Memory

The Delete Shared Memory routine deletes or unmaps from a global section that you created using the PPL$CREATE_SHARED_MEMORY routine. Optionally, this routine writes the contents of the global section to disk before deleting the section.

FORMAT

PPL$DELETE_SHARED_MEMORY *section-name [,memory-area] [,flags]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

section-name

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by descriptor**

Name of the global section you want to delete. The **section-name** argument is the address of a descriptor pointing to a character string containing the global section name.

memory-area

VMS usage: **vector_longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference, array reference**

The area of memory into which the global section that you want to delete is mapped. The **memory-area** argument is the address of a two-longword array containing, in order, the length in bytes and the starting virtual address of the area of memory.

flags

VMS usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Bit mask specifying actions to be performed before deleting the global section. The **flags** argument is the address of a longword bit mask containing the flag. Valid values for **flags** are as follows:

PPL$M_FLUSH	Writes the global section to disk before deleting it.
PPL$M_NOUNI	Identifies the global section as having a nonunique name. By default, PPL$CREATE_SHARED_MEMORY gives the specified global section a name unique to the application by using PPL$UNIQUE_NAME. If you specified this value to give the global section a nonunique name when you called PPL$CREATE_SHARED_MEMORY, you must also specify it when you call PPL$DELETE_SHARED_MEMORY.

DESCRIPTION

PPL$DELETE_SHARED_MEMORY unmaps the calling process from a global section that you created using the PPL$CREATE_SHARED_MEMORY routine. A VMS global section is a section of memory potentially available to all processes in the system.

A temporary global section is implicitly deleted when the last process unmaps from it. Permanent global sections must be explicitly deleted by calling this routine; however, a permanent global section is not actually deleted until the last process unmaps from it.

After a process calls this routine to delete a permanent global section, no other processes can map that global section. If a process subsequently specifies the global section name in a call to PPL$CREATE_SHARED_MEMORY, that routine creates a *new* global section with the same name. The new global section is not shared with processes that mapped the old global section of the same name before PPL$DELETE_SHARED_MEMORY was called.

You can use the **flags** argument to specify that the contents of the global section are written to disk before the section is deleted, or to identify the global section as having a nonunique name, or both.

If the global section is mapped in another process when you call this routine, PPL$DELETE_SHARED_MEMORY unmaps from the global section. When all processes have unmapped from the section or have been deleted, PPL$DELETE_SHARED_MEMORY deletes the global section.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INVARG	Invalid argument.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_WRONUMARG	Wrong number of arguments.

Any error returned by the system service $DELTVA.

PPL$DELETE_SPIN_LOCK Delete a Spin Lock

The Delete a Spin Lock routine deletes a spin lock and releases any storage associated with it.

FORMAT

PPL$DELETE_SPIN_LOCK *[lock-id] [,lock-name]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

lock-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the lock. The optional **lock-id** argument is the address of an unsigned longword containing the lock identifier.

lock-name

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by descriptor**

Name of the lock. The optional **lock-name** argument is the address of a descriptor pointing to a character string containing the lock name.

DESCRIPTION

PPL$DELETE_SPIN_LOCK deletes a specified spin lock and releases any storage associated with it. You can specify a spin lock by its name or by its identifier. Unnamed spin locks must be deleted by specifying **lock-id**.

You cannot delete a spin lock if it is currently held by any process in the application. If you attempt to delete a spin lock that is currently held, PPL$ returns the PPL$_ELEINUSE error. None of the participants in the application can perform any further operations on the spin lock after you call PPL$DELETE_SPIN_LOCK.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_ELEINUSE	The specified element is currently in use and cannot be deleted.
PPL$_INVARG	Invalid argument.

PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_NOSUCHELE	The element you specified does not exist.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$DELETE_VM_ZONE Delete a Virtual Memory Zone

The Delete a Virtual Memory Zone routine deletes a storage zone and returns all pages owned by the zone to the application-wide page pool.

FORMAT

PPL$DELETE_VM_ZONE *[zone-id] [,zone-name]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

zone-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the zone. The optional **zone-id** argument is the address of an unsigned longword containing the zone identifier.

zone-name

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by descriptor**

Name of the zone. The optional **zone-name** argument is the address of a descriptor pointing to a character string containing the zone name.

DESCRIPTION

PPL$DELETE_VM_ZONE deletes a specified storage zone and returns all pages owned by the zone to the application-wide page pool. The zone can be specified by its name or identifier. If the zone does not have a name associated with it, specify **zone-id** to delete the zone. For more information on deleting virtual memory zones, refer to the description of LIB$DELETE_VM_ZONE in the *VMS RTL Library (LIB$) Manual*.

You must ensure that all participants in the application are no longer using any of the memory in the zone before you call PPL$DELETE_VM_ZONE. None of the participants in the application can perform any further operations on the zone after you call PPL$DELETE_VM_ZONE.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INVELENAM	Invalid element name or illegal character string.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_NOSUCHELE	The element you specified does not exist.
PPL$_WRONUMARG	Wrong number of arguments.

Any condition value returned by LIB$DELETE_VM_ZONE.

PPL$DELETE_WORK_ITEM Delete a Work Queue Item

The Delete a Work Queue Item routine deletes a work item from a work queue.

FORMAT

PPL$DELETE_WORK_ITEM *queue-id ,work-item [,flags]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

queue-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

The queue identifier. The **queue-id** argument is the address of an unsigned longword containing the identifier.

work-item

VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

The value of the item to be deleted from the queue. The **work-item** argument is the address of an unsigned longword containing this value.

flags

VMS usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Specifies options for deleting items from a work queue. The optional **flags** argument is the address of a longword bit mask containing the flag. Valid values are as follows:

PPL$M_DELETEALL	Delete all items in the specified queue whose value matches **work-item**. By default, only the first item encountered is deleted.

PPL$M_TAILFIRST	Begin searching at the end of the queue and move toward the beginning. By default, the search begins at the beginning of the queue and moves toward the end.

DESCRIPTION

PPL$DELETE_WORK_ITEM searches a specified work queue for an item whose value matches **work-item**. By default, this routine searches the queue from beginning to end. If the flag PPL$M_TAILFIRST is specified, the queue is searched from the end to the beginning. When the first matching work item is found, it is deleted and the routine returns with a success status. However, if the PPL$M_DELETEALL flag is set, PPL$DELETE_WORK_ITEM continues searching and deleting matching items until it reaches the opposite end of the queue.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INVARG	Invalid argument.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_NOMATCH	No match for the specified element found.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$DELETE_WORK_QUEUE Delete a Work Queue

The Delete a Work Queue routine deletes the specified work queue, and releases any storage associated with it.

FORMAT

PPL$DELETE_WORK_QUEUE *[queue-id] [,queue-name] [,flags]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

queue-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

The work queue identifier. The optional **queue-id** argument is the address of an unsigned longword containing the identifier.

queue-name

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by descriptor**

Name of the work queue. The optional **queue-name** argument is the address of a descriptor pointing to a character string containing the work queue name.

flags

VMS usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Specifies options for deleting a work queue. The optional **flags** argument is the address of a longword bit mask containing the flag. The valid value is as follows:

PPL$M_FORCEDEL	Delete the work queue regardless of whether it is empty. By default, PPL$ returns the error PPL$_ELEINUSE if you attempt to delete a work queue at which proceses are blocked or that contains work items.

PPL$DELETE_WORK_QUEUE

DESCRIPTION

PPL$DELETE_WORK_QUEUE deletes the specified work queue and releases any internal storage associated with that queue. A work queue may be specified either by **queue-name** or **queue-id**. Unnamed queues must be deleted by specifying the **queue-id**.

If another participant is waiting for a work item to be placed in the work queue, it is awakened. None of the participants in the application can do any further operations on the work queue after you call PPL$DELETE_WORK_QUEUE.

A work queue must be empty before it can be deleted (unless you specify the PPL$M_FORCEDEL flag). If you attempt to delete a work queue at which processes are blocked or that contains work items, PPL$ will return the PPL$_ELEINUSE error. You can force deletion of a work queue that is not empty by specifying the PPL$M_FORCEDEL flag. The PPL$_DELETED status is then returned, indicating that the work queue was deleted.

If you force a work queue to be deleted, the PPL$ facility makes no assumptions about the contents of the work items. If your items consist of pointers to pieces of shared memory, it is your responsibility to deallocate all work items in the work queue before deleting the work queue.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_DELETED	Successful completion. The specified element was forcibly deleted.
PPL$_ELEINUSE	The specified element is currently in use and may not be deleted.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELENAM	Invalid element name or illegal character string.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_NOSUCHELE	The element you specified does not exist.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$DISABLE_EVENT Disable Asynchronous Notification of an Event

The Disable Asynchronous Notification of an Event routine disables delivery to a process of notification of an event by either AST or signal.

FORMAT

PPL$DISABLE_EVENT *event-id*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

event-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the event. The **event-id** argument is the address of an unsigned longword containing the identifier.

DESCRIPTION

PPL$DISABLE_EVENT disables delivery of event notification to the calling process by AST or signal, or both. This routine has no effect on other processes that have called PPL$AWAIT_EVENT and are waiting for an event to occur.

There may be some delay between the time that this routine is called and the time that the event is actually disabled. The calling program should be prepared to handle event notification up until the time that this routine *returns*.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$ENABLE_EVENT_AST Enable AST Notification of an Event

The Enable AST Notification of an Event routine specifies the address of an AST routine (and optionally an argument to that routine) to be delivered when an event occurs.

FORMAT

PPL$ENABLE_EVENT_AST *event-id ,astadr [,astprm]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

event-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the event. The **event-id** argument is the address of an unsigned longword containing the identifier.

Event-id is returned by PPL$CREATE_EVENT.

astadr

VMS usage: **ast_procedure**
type: **procedure entry mask**
access: **call without stack unwinding**
mechanism: **by reference**

AST routine. The **astadr** argument is the address of the procedure entry mask for the user's AST routine. This routine is called on the user's behalf when the event state becomes *occurred.*

astprm

VMS usage: **user_arg**
type: **unspecified**
access: **read only**
mechanism: **by value**

AST value passed as the argument to the specified AST routine. The **astprm** argument is the address of a vector of unsigned longwords containing this optional value. If this argument is not specified, PPL$_EVENT_OCCURRED is the **astprm** for a user-created event. The **astprm** argument has special restrictions when used in conjunction with the PPL$ event routines.

- For user-defined events, the **AST-argument** must point to a vector of two unsigned longwords. The first longword is a "context" reserved for the user; it is not read or modified by PPL$. The second longword

receives the value specified by the **event-param** argument in the call to PPL$TRIGGER_EVENT that results in the delivery of this AST.

- For PPL$-defined events (those not created by the user), the **astprm** argument must point to a vector of four unsigned longwords. The vector accommodates the following:
  - The user's "context" longword
  - The longword to receive the event's distinguishing condition value
  - The parameters to the PPL$-defined event (the "trigger" parameter)

  Because each of the predefined events takes two arguments, the vector that **astprm** points to must be four longwords in length.

DESCRIPTION

PPL$ENABLE_EVENT_AST requests the delivery of a specified AST when a corresponding trigger sets the event state to *occurred*. (Generally, a trigger is issued when a participant calls PPL$TRIGGER_EVENT. However, the PPL$ facility triggers predefined events automatically.) Refer to Section 4.3.7 for more information about triggering an event.

An asynchronous system trap (AST) is a VMS mechanism for providing a software interrupt when an external event occurs. When you call this routine, follow all standard VMS conventions for using ASTs.

If the event state is already *occurred* when you call this routine, the AST is delivered immediately and, if there are no other pending triggers, the event state is reset to *not_occurred*. If the state of the event is *not_occurred* when you call this routine, your request for an AST to notify the caller of an event's occurrence is placed in a queue and is processed once the event actually occurs. Note that the caller continues execution immediately after the AST request is placed in the queue. (Event notification is a one-time occurrence. You must call this routine each time you want to re-enable event notification after an event occurs.)

If you do not specify a value for the **astprm** argument, PPL$_EVENT_OCCURRED is passed as the **astprm** argument when the event occurs. If **astprm** is specified, it must conform to the requirements described in the **astprm** argument description.

For user-defined events, you can supply a value for the **event-param** argument in the call to PPL$TRIGGER_EVENT that causes the delivery of this AST. If you specify an **event-param**, it appears in this routine as the second longword in the **astprm** array.

PPL$ predefines the conditions PPL$_ABNORMAL_EXIT and PPL$_NORMAL_EXIT, corresponding to the PPL$-defined event constants PPL$K_ABNORMAL_EXIT and PPL$K_NORMAL_EXIT. You can use one of these event constants as the **event-id** in a call to PPL$ENABLE_EVENT_AST if you want to be notified when a participant exits. Each predefined event has two additional parameters: the **participant-index** and the **exit-status** of the terminating participant. When a normal or abnormal exit occurs, PPL$ triggers the corresponding event automatically. Refer to PPL$CREATE_EVENT for more information about predefined events.

For a given event, any calls to this routine from a given participant after the first call overwrite the information previously specified. In general, you should only call it once for each event for each participant.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INSVIRMEM	Insufficient virtual memory available.
PPL$_INVARG	Invalid argument.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$ENABLE_EVENT_SIGNAL Enable Signal Notification of an Event

The Enable Signal Notification of an Event routine specifies a condition value to be signaled when the event occurs.

FORMAT

PPL$ENABLE_EVENT_SIGNAL *event-id [,signal-value]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

event-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the event. The **event-id** argument is the address of an unsigned longword containing the identifier.

Event-id is returned by PPL$CREATE_EVENT.

signal-value

VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Optional user-defined value to be signaled when the event occurs. The **signal-value** argument is an unsigned longword containing this value.

DESCRIPTION

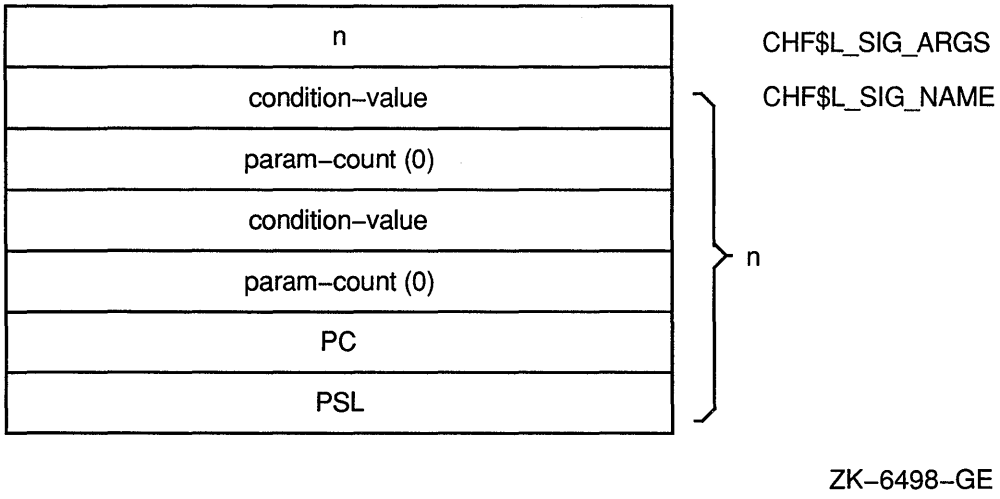
PPL$ENABLE_EVENT_SIGNAL requests the delivery of a specified condition value when a trigger sets the event state to *occurred.* (Generally, a trigger is issued when a participant calls PPL$TRIGGER_EVENT. However, the PPL$ facility triggers predefined events automatically.) Refer to Section 4.3.7 for more information about triggering an event.

If the event state is already *occurred* when you call this routine, the signal is delivered immediately and, if there are no other pending triggers, the event state is reset to *not_occurred.* If the state of the event is *not_occurred* when you call this routine, your request for a signal to notify the caller of an event's occurrence is placed in a queue, and is processed once the corresponding event is triggered. Note that the caller continues

execution immediately after the signal request is placed in the queue. (Event notification is a one-time occurrence. You must call this routine each time you want to re-enable event notification after an event occurs.)

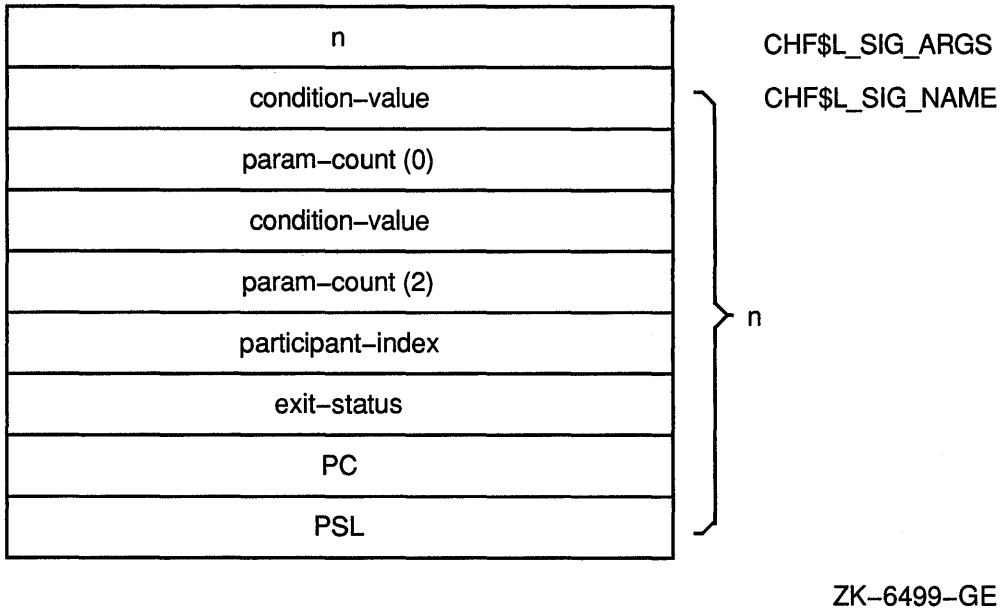
If you specify the **signal-value** argument, that value is the first condition signaled in the signal vector when the event occurs. If you do not specify **signal-value**, PPL$_EVENT_OCCURRED is signaled. If the **event-param** argument is specified in the call to PPL$TRIGGER_EVENT that causes the delivery of this signal, that argument appears as the second condition value in the signal vector. Figure PPL–1 illustrates the structure of a signal vector for a user-defined event.

Figure PPL–1 Signal Vector for a User-Defined Event



PPL$ predefines the conditions PPL$_ABNORMAL_EXIT and PPL$_NORMAL_EXIT, corresponding to the PPL$-defined event constants, PPL$K_ABNORMAL_EXIT and PPL$K_NORMAL_EXIT. You use one of these event constants as the **event-id** in a call to PPL$_ENABLE_EVENT_SIGNAL if you want to be notified when a participant exits. Each predefined event has two additional parameters: the **participant-index** and the **exit-status** of the terminating participant. When a normal or abnormal exit occurs, PPL$ triggers the corresponding event automatically. Refer to PPL$CREATE_EVENT for more information about predefined events. Figure PPL–2 illustrates the structure of a signal vector for a PPL$-defined event.

Figure PPL–2 Signal Vector for a PPL$-Defined Event



For more information about signal vectors, refer to the VAX Procedure Calling and Condition Handling Standard in the *Introduction to VMS System Routines*.

For a given event, any calls to this routine from a given participant after the first call overwrite the information previously specified. You should only call this routine once for each event for each participant.

PPL$ENABLE_EVENT_SIGNAL provides for cross-process asynchronous signaling. This is a powerful mechanism, and it must be used only in carefully controlled environments.

Asynchronous exceptions are those that are not a direct result of the execution of the code, but rather are caused by some concurrent and not directly related event. For example, an AST interrupts a MOVC instruction and the AST routine attempts to reference an invalid address, resulting in an access violation. The signaled exception is an ACCVIO, and it is not related to the interrupted MOVC instruction. Occurrences of asynchronous exceptions have previously been quite uncommon, and the majority of existing code expects to terminate upon receipt of such an exception. The PPL$ENABLE_EVENT_SIGNAL service introduces the means for use of asynchronous signals as a communications mechanism.

Delivery of an asynchronous signal to an arbitrary layered environment can result in unwinding code that is totally unprepared for it, resulting in corrupted data. For example, any RTL routine or the code of a layered product might be interrupted by such an exception. Code that executes in multiple threads under one process context is particularly vulnerable—for example, Ada tasking. Delivery of an asynchronous exception interrupts the task that is executing at the time, and will result in task termination.

Do not use this routine in environments that support multitasking within a process.

To avoid the potential program data corruptions and unintended alterations of control flow implied by unexpected unwinding of an unprepared code section, use this asynchronous signaling capability only when the code that can be interrupted is your own. Also note that you can accomplish the same tasks in a less dangerous way—using the standard AST facilities—by using the PPL$ENABLE_EVENT_AST routine.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INSVIRMEM	Insufficient virtual memory available.
PPL$_INVARG	Invalid argument.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$FIND_OBJECT_ID Find Object Identification

Given the name of a spin lock, semaphore, barrier, event, work queue, or shared memory zone, the Find Object Identification routine returns the identifier of the object associated with the name you specify.

FORMAT

PPL$FIND_OBJECT_ID *object-id ,object-name*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

object-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **modify**
mechanism: **by reference**

Object identifier to be returned. The **object-id** argument is the address of an unsigned longword that receives the associated identifier.

object-name

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by descriptor**

Name of the object for which to return the associated identifier. The **object-name** argument is the address of a descriptor pointing to a character string containing the (user-defined) name of the object.

DESCRIPTION

Given the name of a spin lock, semaphore, barrier, event, work queue, or shared memory zone, PPL$FIND_OBJECT_ID returns the identifier of the object associated with the name you specify. An object is any synchronization element (spin lock, semaphore, barrier, event, or work queue) or shared memory zone previously created and named in a call to one of the following routines:

PPL$CREATE_BARRIER
PPL$CREATE_EVENT
PPL$CREATE_SEMAPHORE
PPL$CREATE_SPIN_LOCK
PPL$CREATE_VM_ZONE
PPL$CREATE_WORK_QUEUE

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INVARG	Invalid argument.
PPL$_INVELENAM	Invalid element name, or illegal character string.
PPL$_NOSUCHELE	The element you specified does not exist.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$FLUSH_SHARED_MEMORY Flush Shared Memory

The Flush Shared Memory routine writes (flushes) to disk the contents of a global section that you created using the PPL$CREATE_SHARED_MEMORY routine. Only pages that have been modified are flushed to disk.

FORMAT

PPL$FLUSH_SHARED_MEMORY *section-name [,memory-area] [,flags]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

section-name

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by descriptor**

Name of the global section whose contents are to be written to disk. The **section-name** argument is the address of a descriptor pointing to a character string containing the global section name.

memory-area

VMS usage: **vector_longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference, array reference**

The area of memory into which the specified global section is mapped. The **memory-area** argument is the address of a two-longword array containing, in order, the length (in bytes) and the starting virtual address for the area of memory.

flags

VMS usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Bit mask specifying actions to perform before flushing the global section. The **flags** argument is the address of a longword bit mask containing the flag. The valid value for **flags** is as follows:

PPL$M_NOUNI	Identifies the global section as having a nonunique name. By default, PPL$CREATE_SHARED_MEMORY gives the specified global section a name unique to the application by using PPL$UNIQUE_NAME. If you specified this value to give the global section a nonunique name when you called PPL$CREATE_SHARED_MEMORY, you must also specify it when you call PPL$FLUSH_SHARED_MEMORY.

DESCRIPTION

PPL$FLUSH_SHARED_MEMORY writes (flushes) to disk the contents of a global section that was created using the PPL$CREATE_SHARED_MEMORY routine. (A VMS global section is a data structure or shareable image section potentially available to all processes in the system.) If you specified a file name in the call to PPL$CREATE_SHARED_MEMORY, the shared memory is written to that file when you call PPL$FLUSH_SHARED_MEMORY. The shared memory name is used as a related file name. Only pages that have been modified are flushed to disk. When one participant calls this routine, all other participants flush their modified pages as well.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INVARG	Invalid argument.
PPL$_INVDESC	Invalid descriptor.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_NOSECEX	The section that you specified does not exist.
PPL$_WRONUMARG	Wrong number of arguments.

Any error returned by the system service $UPDSEC.

PPL$GET_INDEX Get Index of a Participant

The Get Index of a Participant routine returns an index that is unique within the application. A value of zero signifies the participant that formed the application. The other participants in the application always return an index greater than zero.

FORMAT

PPL$GET_INDEX *participant-index*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

participant-index

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

The index of the caller within this application. The **participant-index** argument is the address of an unsigned longword that contains this index. This index is assigned at process creation time and is unique for each participant.

DESCRIPTION

PPL$GET_INDEX returns the unique index of the calling participant within the application. The index of the participant that formed the application is always zero. The index of each subordinate is assigned in the order in which it is spawned or joins the application (by a call to PPL$CREATE_APPLICATION). For example, the first subordinate spawned by or joining the application is assigned an index of 1, the second 2, and so on.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.

PPL$INCREMENT_SEMAPHORE Increment a Semaphore

The Increment a Semaphore routine increments the value of the semaphore by 1, analogous to the signal protocol. If any other participants are blocked on a call to PPL$DECREMENT_SEMAPHORE for this semaphore, one is removed from the queue and awakened. The semaphore must have been created by PPL$CREATE_SEMAPHORE.

FORMAT

PPL$INCREMENT_SEMAPHORE *semaphore-id*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

semaphore-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the semaphore. The **semaphore-id** argument is the address of an unsigned longword containing the identifier.

Semaphore-id is returned by PPL$CREATE_SEMAPHORE.

DESCRIPTION

PPL$INCREMENT_SEMAPHORE increments the value of the semaphore by 1, analogous to the signal protocol. In addition, if any participants are blocked on a call to PPL$DECREMENT_SEMAPHORE for this semaphore, one is removed from the queue and awakened.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_SEMALRMAX	The semaphore is already at its maximum value.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$INDEX_TO_PID Convert Participant Index to VMS PID

The Convert Participant Index to VMS PID routine returns the VMS PID of the process associated with the specified index.

FORMAT

PPL$INDEX_TO_PID *participant-index ,pid*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

participant-index

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Index of the caller within this application. The **participant-index** argument is the address of an unsigned longword that contains this index. **Participant-index** is assigned at process creation time and is unique for each participant.

pid

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

PID (process identifier) of the VMS process associated with the specified participant-index. The **pid** argument is the address of an unsigned longword that receives this PID.

DESCRIPTION

PPL$INDEX_TO_PID returns the VMS PID of the process associated with the specified participant index.

The return status PPL$_NO_SUCH_PARTY indicates that the specified process participated in the current application but presently is not a member (because it called PPL$TERMINATE or exited). The value returned in **pid** is the PID of the process when it was a participant. If PPL$_NO_SUCH_PARTY is returned, this PID may be no longer valid.

The return status PPL$_INVARG indicates that the process with the specified PID was never a participant in the current application.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INVARG	Invalid argument.
PPL$_NO_SUCH_PARTY	The participant specified does not exist in this application.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$INSERT_WORK_ITEM Insert a Work Queue Item

The Insert a Work Queue Item routine inserts a work item into the specified work queue.

FORMAT

PPL$INSERT_WORK_ITEM *queue-id ,work-item [,flags] [,priority]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

queue-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

The queue identifier. The **queue-id** argument is the address of an unsigned longword containing the identifier.

work-item

VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

A value to be entered into the queue. The **work-item** argument is an unsigned longword containing this value. The content of **work-item** is completely arbitrary. You may want to place single longword values into **work-item** (for example, the number of a function or task to be performed). You can also use **work-item** to pass a pointer to a data block. (This data block must reside in memory created by PPL$CREATE_SHARED_MEMORY or allocated from a shared memory zone created by PPL$CREATE_VM_ZONE.)

flags

VMS usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Specifies options for inserting a work item into a work queue. The optional **flags** argument is the address of a longword bit mask containing the flag. The valid value is as follows:

PPL$M_ATHEAD	Insert item as the first of those items with the same priority (in other words, at the head of the priority). By default, items are inserted after other items of the same priority.

priority

VMS usage: **longword_signed**
type: **longword (signed)**
access: **read only**
mechanism: **by reference**

Specifies the priority of the item being inserted. The optional **priority** argument is an unsigned longword containing the priority value for the item to be inserted. If unspecified, the default value is zero. A high numerical value indicates a high priority.

DESCRIPTION

PPL$INSERT_WORK_ITEM inserts the value specified by **work-item** into the specified work queue. If another process is waiting for an item to be placed into the queue, that process is awakened and will remove the newly inserted item after the call to PPL$INSERT_WORK_ITEM.

By default, the item is inserted into the queue *after* any items with a higher or equal numerical priority and *before* any items with a lower priority. If you specify the flag PPL$M_ATHEAD, the item is inserted before any other items of an equal priority.

If an application always uses the default (zero) for **priority**, the result is a simple FIFO (first in, first out) queue. PPL$ inserts new items at the end of the queue by default, or at the beginning of the queue if PPL$M_ATHEAD is specified.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INSVIRMEM	Insufficient virtual memory available.
PPL$_INVARG	Invalid argument.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$PID_TO_INDEX Convert VMS PID to Participant Index

The Convert VMS PID to Participant Index routine returns the PPL$-defined participant index of the process associated with the specified VMS PID.

FORMAT

PPL$PID_TO_INDEX *pid ,participant-index*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

pid

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

PID (process identifier) of the VMS process or subprocess whose participant index is to be obtained. The **pid** argument is the address of an unsigned longword that contains this PID.

participant-index

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Participant index of the process or subprocess associated with the specified VMS PID. The **participant-index** argument is the address of an unsigned longword that receives this index. **Participant-index** is assigned by the PPL$ facility at process creation time and is unique for each participant.

DESCRIPTION

PPL$PID_TO_INDEX returns the participant index of the VMS process specified by the input VMS PID.

The return status PPL$_NO_SUCH_PARTY indicates that the specified process participated in the current application but presently is not a member (because it called PPL$TERMINATE or exited). The value returned in **participant-index** is the index of the process when it was a participant.

The return status PPL$_INVARG indicates that the process with the specified PID was never a participant in the current application.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INVARG	Invalid argument.
PPL$_NO_SUCH_PARTY	The participant specified does not exist in this application.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$READ_BARRIER Read a Barrier

The Read a Barrier routine returns the specified barrier's current quorum and the number of participants currently waiting (blocked) at the barrier. The barrier must have been created by PPL$CREATE_BARRIER.

FORMAT

PPL$READ_BARRIER *barrier-id ,quorum ,waiters*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

barrier-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the specified event. The **barrier-id** argument is the address of an unsigned longword containing the identifier.

Barrier-id is returned by PPL$CREATE_BARRIER.

quorum

VMS usage: **word_signed**
type: **word (signed)**
access: **write only**
mechanism: **by reference**

Number of participants required to terminate a wait for this barrier. The **quorum** argument is the address of a signed word containing the quorum value. This argument returns the current **quorum** value that you set with PPL$CREATE_BARRIER, PPL$SET_QUORUM, or PPL$ADJUST_QUORUM.

waiters

VMS usage: **word_signed**
type: **word (signed)**
access: **write only**
mechanism: **by reference**

Number of participants currently waiting at this barrier. The **waiters** argument is the address of a signed word containing the number of waiting participants.

PPL$READ_BARRIER

DESCRIPTION

PPL$READ_BARRIER returns the specified barrier's current quorum and the number of participants currenty waiting (blocked) at the barrier. (Note that calls by other participants to the PPL$ barrier routines may affect the values returned by this routine. In effect, the values you receive for this routine may be outdated before you receive them.)

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INVARG	Invalid argument.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_NOSUCHELE	The element you specified does not exist.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$READ_EVENT Read an Event State

The Read an Event State routine returns the current state of the specified event. The state can be *occurred* or *not_occurred.*

FORMAT

PPL$READ_EVENT *event-id ,occurred*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

event-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the specified event. The **event-id** argument is the address of an unsigned longword containing the identifier.

Event-id is returned by PPL$CREATE_EVENT.

occurred

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the state of the specified event. The **occurred** argument is the address of an unsigned longword that receives the event state. This argument returns a value of **true** if the current state of the event is *occurred,* and returns **false** if the current state of the event is *not_occurred.*

DESCRIPTION

PPL$READ_EVENT returns the current state of the specified event. The state can be *occurred* or *not_occurred.* (Note that calls by other participants to the PPL$ event routines may affect the state returned by this routine. In effect, the state returned by this routine may be outdated before you receive it.)

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INVARG	Invalid argument.
PPL$_INVELEID	Invalid element identifier.

PPL$_INVELETYP	Invalid element type.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_NOSUCHELE	The element you specified does not exist.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$READ_SEMAPHORE Read Semaphore Values

The Read Semaphore Values routine returns the current or maximum values, or both, of the specified counting semaphore. The semaphore must have been created by PPL$CREATE_SEMAPHORE.

FORMAT

PPL$READ_SEMAPHORE
semaphore-id [,semaphore-value] [,semaphore-maximum]

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

semaphore-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the specified semaphore. The **semaphore-id** argument is the address of an unsigned longword containing the identifier.

Semaphore-id is returned by PPL$CREATE_SEMAPHORE.

semaphore-value

VMS usage: **word_signed**
type: **word (signed)**
access: **write only**
mechanism: **by reference**

Receives information about the specified semaphore. The optional **semaphore-value** argument is the address of a signed word containing the current value of the semaphore or the number of blocked processes. If positive, **semaphore-value** contains the number of available resources associated with this semaphore; if negative, it contains the number of waiting processes. If the value returned is zero, there are no available resources and no waiting processes.

semaphore-maximum

VMS usage: **word_signed**
type: **word (signed)**
access: **write only**
mechanism: **by reference**

Maximum value of the semaphore. The **semaphore-maximum** argument is the address of a signed word containing the maximum value of the semaphore specified by **semaphore-id**.

PPL$READ_SEMAPHORE

DESCRIPTION

PPL$READ_SEMAPHORE returns the current value of the specified semaphore or the number of processes waiting for the semaphore. PPL$READ_SEMAPHORE also returns the maximum value of the semaphore. If no values are requested, a status code of PPL$_NORMAL is returned. (Note that calls by other participants to the PPL$ semaphore routines may affect the values returned by this routine. In effect, the values returned by this routine may be outdated before you receive them.)

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INVARG	Invalid argument.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_NOSUCHELE	The element you specified does not exist.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$READ_SPIN_LOCK Read a Spin Lock State

The Read a Spin Lock State routine returns the current state of a spin lock. The state can be *seized* or *not_seized.*

FORMAT

PPL$READ_SPIN_LOCK *lock-id ,seized*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

lock-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the specified spin lock. The **lock-id** argument is the address of an unsigned longword containing the identifier.

seized

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the state of the specified spin lock. The **seized** argument is the address of an unsigned longword that receives the spin lock state. This argument returns a value of true if the current state of the spin lock is *seized,* and it returns a value of false if the current state of the spin lock is *not_seized.*

DESCRIPTION

PPL$READ_SPIN_LOCK returns the current state of the specified spin lock. The state can be *seized* or *not_seized.* Calls by other participants to the PPL$ spin lock routines can affect the state returned by this routine. In effect, the state returned by this routine may be outdated before you receive it.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$READ_WORK_QUEUE Read a Work Queue

The Read a Work Queue routine returns information about a work queue.

FORMAT

PPL$READ_WORK_QUEUE *queue-id [,queue-value]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

queue-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

The queue identifier. The **queue-id** argument is the address of an unsigned longword containing the identifier.

queue-value

VMS usage: **longword_signed**
type: **longword (signed)**
access: **write only**
mechanism: **by reference**

Receives information about the specified work queue. If positive, **queue-value** contains the number of items currently in the work queue; if negative, it contains the number of processes currently blocked (waiting for an item to be placed in the queue). If the value returned is zero, there are no work items in the queue and no blocked processes. The optional **queue-value** argument is the address of a signed longword that receives the number of work items or blocked processes.

DESCRIPTION

For a specified **queue-id**, PPL$READ_WORK_QUEUE returns one of the following:

- The number of items that are presently in the specified work queue
- The number of processes that are currently waiting for items to be inserted into the work queue

Calls by other participants to the PPL$ work queue routines may affect the values returned by this routine. In effect, the values returned by this routine may be outdated before you receive them.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INVARG	Invalid argument.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$RELEASE_SPIN_LOCK Release Spin Lock

The Release Spin Lock routine relinquishes the spin lock by clearing the bit representing the lock. The lock must have been created by PPL$CREATE_SPIN_LOCK.

FORMAT

PPL$RELEASE_SPIN_LOCK *lock-id*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

lock-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the specified lock. The **lock-id** argument is the address of an unsigned longword containing the lock identifier.

Lock-id is returned by PPL$CREATE_SPIN_LOCK.

DESCRIPTION

PPL$RELEASE_SPIN_LOCK relinquishes the spin lock by clearing the bit representing the lock.

If there are other participants waiting in a spin loop to obtain this lock, this routine allows one of the waiting participants in the spin loop to get the lock.

This form of lock is recommended for use only in a dedicated parallel processing environment, and only when fairness is not important. A spin lock is not recommended for use in a general time-sharing environment because the spin consumes CPU resources.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type.
PPL$_LOCNOTEST	The lock was not established.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_NOSUCHELE	The element you specified does not exist.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$REMOVE_WORK_ITEM Remove a Work Queue Item

The Remove a Work Queue Item routine removes the next item in order from a work queue.

FORMAT

PPL$REMOVE_WORK_ITEM *queue-id ,work-item [,flags]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

queue-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

The queue identifier. The **queue-id** argument is the address of an unsigned longword containing the identifier.

work-item

VMS usage: **user_arg**
type: **longword (unsigned)**
access: **write only**
mechanism: **by reference**

Receives the value of the item that is removed from the work queue. The **work-item** argument is the address of an unsigned longword that receives the value of the item that is removed from the work queue.

flags

VMS usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Specifies options for removing an item from the work queue. The optional **flags** argument is the address of a longword bit mask containing the flag. Valid values are as follows:

PPL$M_NON_BLOCKING	If the specified work queue is empty, return immediately with the PPL$_NOT_AVAILABLE status indicating that no items are available to be removed from the work queue. By default, if the work queue is empty the process hibernates until there is an item available to be removed from the work queue.
PPL$M_FROMTAIL	Remove item from the end (or tail) of the work queue. By default, this routine removes an item from the beginning (or head) of the work queue.

DESCRIPTION

PPL$REMOVE_WORK_ITEM removes the next item from the beginning of the specified queue. Because the queue is sorted by priority, this is the item with the highest priority. If the queue is empty, the process hibernates until an item is placed in the queue by another process. When an item is placed in the queue, the process awakens and proceeds normally. If the queue is empty and PPL$REMOVE_WORK_ITEM is called with the PPL$M_NON_BLOCKING flag set, the routine returns immediately with the PPL$_NOT_AVAILABLE status, indicating that an item was not removed from the queue.

If a process is hibernating (awaiting an item to be placed into the queue) and the queue is deleted, the process is awakened and PPL$REMOVE_WORK_ITEM returns the PPL$_DELETED status indicating that the queue was deleted and no item was removed.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_DELETED	Successful completion. The specified element was forcibly deleted.
PPL$_INVARG	Invalid argument.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_NOT_AVAILABLE	Operation cannot be performed immediately; therefore, it is not performed.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$RESET_EVENT Reset an Event

The Reset an Event routine resets an event's state to *not_occurred.*

FORMAT

PPL$RESET_EVENT *event-id*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

event-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the event. The **event-id** argument is the address of an unsigned longword containing the identifier.

DESCRIPTION

PPL$RESET_EVENT resets the event state associated with a specified event to *not_occurred.* Any pending triggers are removed from the queue.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$SEIZE_SPIN_LOCK Seize Spin Lock

The Seize Spin Lock routine retrieves a simple (spin) lock by waiting in a spin loop until the lock is free. The lock must have been created by PPL$CREATE_SPIN_LOCK.

FORMAT

PPL$SEIZE_SPIN_LOCK *lock-id [,flags]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

lock-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the lock to be seized. The **lock-id** argument is the address of an unsigned longword containing the lock identifier.

Lock-id is returned by PPL$CREATE_SPIN_LOCK.

flags

VMS usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Bit mask specifying options for seizing the lock. The **flags** argument is a longword bit mask containing the flag. The valid value for **flags** is as follows:

PPL$M_NON_BLOCKING	The lock is seized if and only if it can be done without causing the caller to wait (spin). (This can be useful in situations where the cost of waiting for a resource is not desirable, or if the caller merely intends to request immediate access to any one of a number of resources.)

DESCRIPTION

PPL$SEIZE_SPIN_LOCK acquires a spin lock by waiting in a spin loop until the lock is free. If you specify PPL$M_NON_BLOCKING for the **flags** argument, the caller does not wait in the spin loop if the lock cannot be immediately obtained. In that case the status code PPL$_NOT_AVAILABLE is returned.

You have exclusive access to the spin lock after you acquire it by calling this routine. Call PPL$RELEASE_SPIN_LOCK to free the lock when you no longer need it.

This form of lock is recommended for use only in a dedicated parallel processing environment, and only when fairness is not important. A spin lock is not recommended for use in a general time-sharing environment because the spin consumes CPU resources.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INVELETYP	Invalid element type for requested operation.
PPL$_NOSUCHLOC	A lock with the specified ID does not exist.
PPL$_NOT_AVAILABLE	Operation cannot be performed immediately; therefore it is not performed.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$SET_QUORUM Set Barrier Quorum

The Set Barrier Quorum routine dynamically sets a value for the specified barrier's quorum.

FORMAT

PPL$SET_QUORUM *barrier-id ,quorum*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

barrier-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the barrier. The **barrier-id** argument is the address of the barrier identifier.

Barrier-id is returned by PPL$CREATE_BARRIER.

quorum

VMS usage: **word_signed**
type: **word (signed)**
access: **read only**
mechanism: **by reference**

The number of participants required to terminate an active wait for this barrier. The **quorum** argument is the address of a signed word containing the quorum number. For example, a **quorum** value of 3 indicates that the first two callers of PPL$WAIT_AT_BARRIER specifying this **barrier-id** are blocked until a third participant calls PPL$WAIT_AT_BARRIER. At that point, all three are released for further processing. If you specify zero for **quorum**, the quorum is set to the number of processes currently in the application. The value of **quorum** must be positive or zero.

DESCRIPTION

PPL$SET_QUORUM allows the user to dynamically set the value of a barrier's quorum. A barrier's quorum is the number of participants required to call PPL$WAIT_AT_BARRIER (and thereby be blocked) before all blocked participants are unblocked to pass the barrier and continue processing. This allows you to reuse a barrier for different work items with various numbers of participants. The barrier must have been created by PPL$CREATE_BARRIER.

Note that PPL$SET_QUORUM must be called while no participants have called PPL$WAIT_AT_BARRIER (in other words, while there are no participants waiting at the barrier).

CONDITION VALUES RETURNED

PPL$_NORMAL	Routine successfully completed.
PPL$_INVARG	Invalid argument.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_IN_BARRIER_WAIT	One or more participants is waiting at the barrier; therefore, the quorum is not modified.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$SET_SEMAPHORE_MAXIMUM Set a Semaphore Maximum

The Set a Semaphore Maximum routine dynamically sets the maximum value of a semaphore.

FORMAT

PPL$SET_SEMAPHORE_MAXIMUM *semaphore-id*
,semaphore-maximum

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

semaphore-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the semaphore. The **semaphore-id** argument is the address of an unsigned longword containing the identifier.

semaphore-maximum

VMS usage: **word_signed**
type: **word (signed)**
access: **read only**
mechanism: **by reference**

New maximum value of the semaphore. The **semaphore-maximum** argument is the address of a signed word containing the maximum value. This value must be nonnegative.

DESCRIPTION

PPL$SET_SEMAPHORE_MAXIMUM allows you to dynamically set the maximum value of a specified semaphore. This allows semaphores to be reused easily for different purposes with various numbers of resources. The semaphore must have been created by PPL$CREATE_SEMAPHORE.

Call this routine only when the semaphore's current value is equal to its maximum (in other words, there are no participants using resources that are protected by the semaphore). If this routine is called when the current value is not equal to the maximum, PPL$ returns the PPL$_ELEINUSE error.

Calling PPL$SET_SEMAPHORE_MAXIMUM changes the semaphore's current value to the new maximum value you specify in **semaphore-maximum**.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_ELEINUSE	The specified element is currently in use and cannot be deleted.
PPL$_INVARG	Invalid argument.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$SPAWN Initiate Parallel Execution

The Initiate Parallel Execution routine executes code in parallel with the caller by creating one or more subordinate threads of execution (VMS subprocesses).

FORMAT

PPL$SPAWN *copies [,program-name] [,children-ids] [,flags] [,std-input-file] [,std-output-file]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

copies

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **modify**
mechanism: **by reference**

Number of subordinates of the specified program to be executed concurrently. The **copies** argument is the address of an unsigned longword containing this number. Its value must be positive. If you specify a value greater than 1 for the **copies** argument, each copy created will have the same subprocess information (for example, standard input and output files). If you want to specify different information for each subprocess, call PPL$SPAWN once for each subprocess.

On output, this parameter contains the number of subordinates actually created. This value differs from the requested number if an individual spawn attempt fails, for example, because of insufficient quotas.

program-name

VMS usage: **logical_name**
type: **character string**
access: **read only**
mechanism: **by descriptor, fixed-length**

Name of the program (image) to be invoked. The **program-name** argument is the address of a descriptor pointing to a character string containing the file specification of the image. **Program-name** must have no more than 63 characters. If **program-name** contains a logical name, the equivalence name must be in a logical name table that the created subordinate can access. If you do not specify a **program-name**, the default is to execute in parallel the image being run by the caller.

children-ids

VMS usage: **vector_longword_unsigned**
type: **longword (unsigned)**

access: **write only**
mechanism: **by reference, array reference**
Identifiers of each of the newly created subordinates. The **children-ids** argument is the address of a vector of longwords into which is written the index within the executing application of each subordinate successfully initiated by this call.

flags

VMS usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**
Bit mask specifying options for creating processes. The **flags** argument is a longword bit mask containing the flags. Valid values for **flags** are as follows:

PPL$M_INIT_SYNCH	If set, the caller of this routine and all subordinates created by this call are synchronized to continue processing only after each and every subordinate created by this call has called PPL$CREATE_APPLICATION. (See the Description section for more information.) A failure of the created subordinate after it successfully starts but before its call to PPL$CREATE_APPLICATION can cause difficulties with the use of this flag value.
PPL$M_NOCLISYM	If set, the created processes do not inherit CLI symbols from the calling process. The default action is for created processes to inherit all currently defined CLI symbols.
PPL$M_NOCONTROL	If set, prompt strings are not prefixed by carriage return/line feeds. The default action is to prefix any prompt string specified with a carriage return/line feed.
PPL$M_NODEBUG	Prevents the startup of the VMS Debugger, even if the debugger was linked with the image.
PPL$M_NOKEYPAD	If set, created processes inherit the current keypad symbols and state from the calling process. The default action is that created processes do not inherit keypad symbols and state.
PPL$M_NOLOGNAM	If set, created processes do not inherit process logical names from the calling process. The default is for created processes to inherit all currently defined process logical names.
PPL$M_NOTIFY	If set, a message is broadcast to SYS$OUTPUT as each process terminates. This flag is ignored if the process is not interactive (for example, run in batch).

std-input-file

VMS usage: **logical-name**
type: **character string**
access: **read only**
mechanism: **by descriptor**
File name of the file to serve as the standard input file in the created subordinates. The **std-input-file** argument is the address of a descriptor pointing to a character string containing the file name. If you do not

specify a value for this argument, the subordinate inherits the creating participant's standard input file (SYS$INPUT).

std-output-file

VMS usage: **logical-name**
type: **character string**
access: **read only**
mechanism: **by descriptor**

File name of the file to serve as the standard output file in the created subordinates. The **std-output-file** argument is the address of a descriptor pointing to a character string containing the file name. If you do not specify a value for this argument, the subordinate inherits the creating participant's standard output file (SYS$OUTPUT).

DESCRIPTION

PPL$SPAWN executes code in parallel with the caller by creating one or more subordinate threads of execution (VMS subprocesses). This routine initiates the parallel execution of the specified code on the same node as the caller.

By default, the parent (caller) immediately continues processing in its own context, and each child (subordinate) proceeds immediately following its creation. (Note that here "immediately" means "subject only to systemwide scheduling constraints.") The PPL$M_INIT_SYNCH flag arranges that processing in the parent and the subordinates continues only when each and every child created by this operation has called PPL$CREATE_APPLICATION. (Note that this initialization is also performed automatically by PPL$ at the first call to a PPL$ routine; see PPL$CREATE_APPLICATION for more information.) This synchronization is achieved by blocking the parent in the call to PPL$SPAWN, and blocking each child in its PPL$CREATE_APPLICATION call, until the last child executes this call. Then all participants are released for further execution.

The subordinates created by this call execute the code you specify in the **program-name** argument. If you do not specify an image name in this argument, the image being executed by the current process is used in the creation of the subordinate. Subordinates do not inherit any process logical names if PPL$M_NOLOGNAM is specified for the **flags** argument. If you specify PPL$M_NOLOGNAM, subordinates should not depend upon process logical names defined in the parent.

This routine creates one or more VMS subprocesses, each of which is related to its creator in a tree-like fashion. Each has the same UIC as the parent. Each receives a portion of the creator's resource quotas. If subprocesses exist when their creator is deleted, they are automatically deleted, and resources are reclaimed according to VMS-defined semantics. In addition, this routine arranges that process logical names are inherited from parent to (each) subordinate.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_CREATED_SOME	Not all of the requested processes were spawned.
PPL$_INVNUMCHI	Invalid number of processes; cannot be less than one.
PPL$_WRONUMARG	Wrong number of arguments.

Any error returned by LIB$SPAWN.

PPL$STOP Stop a Participant

The Stop a Participant routine terminates the execution of the specified participant in this application.

FORMAT

PPL$STOP *participant-index*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

participant-index

VMS usage: **longword_unsigned**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

PPL$-defined index of the participant to be terminated. The **participant-index** argument is the address of an unsigned longword containing the index.

Participant-index is obtained by a call to PPL$SPAWN or PPL$GET_INDEX.

DESCRIPTION

PPL$STOP terminates the execution of the specified participant in this application. This will also result in the termination of all subordinates of the specified participant.

Call this routine only if you want to stop a participant before it completes its execution.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.

Any error returned by $FORCEX.

PPL$TERMINATE Abort PPL$ Participation

The Abort PPL$ Participation routine ends the caller's participation in the application "prematurely"—that is, at some time before the caller actually completes its execution.

FORMAT

PPL$TERMINATE *[flags]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

flags

VMS usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Bit mask specifying options for terminating access to PPL$. The **flags** argument is the address of a longword bit mask containing the flag. The **flags** argument accepts the following value:

PPL$M_STOP_CHILDREN	Terminates all subordinates created by the caller in addition to terminating the caller itself. (PPL$ makes no effort to delete subordinates at process termination in the absence of a call to this routine specifying this flag value, but note that a VMS subprocess is deleted when the parent terminates.)

DESCRIPTION

The PPL$TERMINATE routine informs the PPL$ facility that the caller is no longer part of the parallel application, and will make no further requests for PPL$ services.

Normally, you need not call this routine. PPL$ automatically performs cleanup operations when the participant completes its execution.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.

Any error returned by $FORCEX or LIB$FREE_VM.

PPL$TRIGGER_EVENT Trigger an Event

The Trigger an Event routine causes the event's state to become *occurred*. You control whether all pending actions for the event are processed (made to occur), or just one is processed. A pending action can be an AST, a signal (condition), or a wakeup.

FORMAT

PPL$TRIGGER_EVENT *event-id [,event-param] [,flags]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

event-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the event. The **event-id** argument is the address of an unsigned longword containing the identifier.

Event-id is returned by PPL$CREATE_EVENT.

event-param

VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

An arbitrary value to be passed to all requests processed for the event as a result of the trigger, or, if there are no queued event notification requests for this event, to the first caller to enable event notification. The **event-param** argument is the address of an unsigned longword containing this value. The value of **event-param** is received by the **output** argument of PPL$AWAIT_EVENT.

If a participant enables delivery of an AST by calling PPL$ENABLE_EVENT_AST, this argument appears in the second longword of the vector specified by the **astprm** argument. If a participant enables delivery of a signal by calling PPL$ENABLE_EVENT_SIGNAL, this argument appears as the third longword in the signal vector when the condition is raised.

flags

VMS usage: **mask_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

PPL$TRIGGER_EVENT

Specifies options for triggering an event. The **flags** argument is the address of a longword bit mask containing the flag. The valid value for **flags** is as follows:

PPL$M_NOTIFY_ONE	Processes exactly one enabled event notification. By default, all pending actions are processed when the event state becomes *occurred.*

DESCRIPTION

PPL$TRIGGER_EVENT sets the event state to *occurred* and processes the queue of requested operations. (The caller controls whether all pending actions for the event are processed, or just one action is processed, by use of the PPL$M_NOTIFY_ONE flag.) A pending action can be an AST, a signal (condition), or a wakeup, as established by corresponding calls to PPL$ENABLE_EVENT_AST, PPL$ENABLE_EVENT_SIGNAL, and/or PPL$AWAIT_EVENT.

PPL$TRIGGER_EVENT initiates the appropriate action, which is finally performed in the context of the participant that enabled the notification. If no participant has enabled notification of the event, the event state remains *occurred.* Triggers are then queued and processed in the order in which they occur, as processes request notification. If one or more participants have enabled notification of the event, the notification resets the state to *not_occurred.* PPL$TRIGGER_EVENT performs these steps as one atomic action; in other words, once this routine begins executing, it completes without interruption from other event operations.

Refer to Section 4.3.7 for more information about triggering an event.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INSVIRMEM	Insufficient virtual memory available.
PPL$_INVARG	Invalid argument.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$UNIQUE_NAME Produce a Unique Name

The Produce a Unique Name routine returns an application-unique name. A system-unique string specific to the calling application is appended to the string specified by the user. The resulting name is identical for all participants in the application, but different from those for all other applications on that system.

FORMAT

PPL$UNIQUE_NAME *name-string ,resultant-string [,resultant-length]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

name-string

VMS usage: **char_string**
type: **character string**
access: **read only**
mechanism: **by descriptor**

The user-supplied string for the unique name. The **name-string** argument is the address of a descriptor pointing to a character string containing this name.

resultant-string

VMS usage: **char_string**
type: **character string**
access: **write only**
mechanism: **by descriptor**

Resulting unique name. The **resultant-string** argument is the address of a descriptor pointing to a character string containing this name. **Resultant-string** consists of the **name-string** string and an appended system-unique string.

resultant-length

VMS usage: **word_unsigned**
type: **word (unsigned)**
access: **write only**
mechanism: **by reference**

Length of the unique name returned as the **resultant-string**. The **resultant-length** argument is the address of an unsigned word containing this length.

PPL$UNIQUE_NAME

DESCRIPTION

PPL$UNIQUE_NAME returns an application-unique name that consists of a system-unique string appended to a string you specify. The resulting unique name is consistent within the application but different from any other name within another application. This means that for a given input string, the resultant name is identical when requested by any participant.

This unique name is useful, for example, when an application creates a scratch file that must not interfere with other users who are also running their own copy of the same application.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INVARG	Invalid argument.
PPL$_INVDESC	Invalid descriptor.
PPL$_WRONUMARG	Wrong number of arguments.

PPL$WAIT_AT_BARRIER Synchronize at a Barrier

The Synchronize at a Barrier routine causes the caller to wait at the specified barrier. The barrier is in effect from the time the first participant calls PPL$WAIT_AT_BARRIER until each member of the quorum has issued the call. At that time, the wait concludes and all are released for further execution.

FORMAT

PPL$WAIT_AT_BARRIER *barrier-id*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

barrier-id

VMS usage: **identifier**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Identifier of the barrier. The **barrier-id** argument is the address of an unsigned longword containing the barrier identifier.

Barrier-id is returned by PPL$CREATE_BARRIER.

DESCRIPTION

PPL$WAIT_AT_BARRIER causes the caller to wait at the specified barrier until the quorum required for conclusion of the barrier wait arrives at the synchronization point. As each participant calls this routine, it is blocked and awaits the arrival of the remaining unblocked participants. When the final unblocked participant calls PPL$WAIT_AT_BARRIER, the wait concludes and all are freed to continue their execution. The caller is blocked by the PPL$ facility's call to the system service $HIBER.

The number of participants required to constitute a **quorum** can be defined by calls to the PPL$CREATE_BARRIER, PPL$SET_QUORUM, and PPL$ADJUST_QUORUM routines.

Note that a call to PPL$ADJUST_QUORUM can result in conclusion of a barrier wait.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type.
PPL$_NOINIT	PPL$CREATE_APPLICATION has not been called.
PPL$_WRONUMARG	Wrong number of arguments.

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

S

T

V

W

Z

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal[1]	————	USASSB Order Processing - WMO/E15 *or* U.S. Area Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

[1]For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

VMS RTL Parallel Processing (PPL$) Manual
AA–LA74B–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

_____

What I like best about this manual is _____

_____

_____

What I like least about this manual is _____

_____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

-- Do Not Tear - Fold Here and Tape ----------

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01-3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

-- Do Not Tear - Fold Here ----------