VMS RTL Parallel Processing (PPL$) Manual

Order Number: AA–LA74A–TE

April 1988

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

Revision/Update Information: This is a new manual.

Software Version: VMS Version 5.0

digital equipment corporation
maynard, massachusetts

April 1988

DEC	DIBOL	UNIBUS
DEC/CMS	EduSystem	VAX
DEC/MMS	IAS	VAXcluster
DECnet	MASSBUS	VMS
DECsystem–10	PDP	VT
DECSYSTEM–20	PDT	
DECUS	RSTS	
DECwriter	RSX	digital™

ZK4375

HOW TO ORDER ADDITIONAL DOCUMENTATION
DIRECT MAIL ORDERS

USA & PUERTO RICO*	**CANADA**	**INTERNATIONAL**
Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061	Digital Equipment of Canada Ltd. 100 Herzberg Road Kanata, Ontario K2K 2A6 Attn: Direct Order Desk	Digital Equipment Corporation PSG Business Manager c/o Digital's local subsidiary or approved distributor

In Continental USA and Puerto Rico call 800-258-1710.

In New Hampshire, Alaska, and Hawaii call 603-884-6660.

In Canada call 800-267-6215.

*Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575).

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Westminster, Massachusetts 01473.

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.

™ PostScript is a trademark of Adobe Systems, Inc.

Contents

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(TM) 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 provides 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.

(TM) VAX is a trademark of Digital Equipment Corporation.
® Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

- 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 optional *Guide to Parallel Programming on VMS* describes the concepts and terminology associated with parallel processing, and introduces the concepts involved in programming an application for parallel execution.

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

Convention	Meaning
$\boxed{\text{RET}}$	In examples, a key name (usually abbreviated) shown within a box indicates that you press a key on the keyboard; in text, a key name is not enclosed in a box. In this example, the key is the RETURN key. (Note that the RETURN key is not usually shown in syntax statements or in all examples; however, assume that you must press the RETURN key after entering a command or responding to a prompt.)
CTRL/C	A key combination, shown in uppercase with a slash separating two key names, indicates that you hold down the first key while you press the second key. For example, the key combination CTRL/C indicates that you hold down the key labeled CTRL while you press the key labeled C. In examples, a key combination is enclosed in a box.
$ SHOW TIME 05-JUN-1988 11:55:22	In examples, system output (what the system displays) is shown in black. User input (what you enter) is shown in red.
$ TYPE MYFILE.DAT . . .	In examples, a vertical series of periods, or ellipsis, means either that not all the data that the system would display in response to a command is shown or that not all the data a user would enter is shown.
input-file, . . .	In examples, a horizontal ellipsis indicates that additional parameters, values, or other information can be entered, that preceding items can be repeated one or more times, or that optional arguments in a statement have been omitted.
[logical-name]	Brackets indicate that the enclosed item is optional. (Brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)
quotation marks apostrophes	The term quotation marks is used to refer to double quotation marks ("). The term apostrophe (') is used to refer to a single quotation mark.

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

New and Changed Features

The PPL$ facility is new for VMS Version 5.0. All of the routines described in this manual are new.

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 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.

Refer to the *Guide to Parallel Programming on VMS* for more information about parallel processing techniques.

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, and increment 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 applications 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. For this version of 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 "threads of execution" that performs an application-defined piece of work. A participant can be either a parent process or a subordinate.

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, or event. When you specify blocking of a participant, the participant is blocked by a PPL$ call to the system service $HIBER, so that ASTs can be delivered.

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 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

To achieve synchronization in a work queue model, you generally use mutual exclusion to access data in shared memory. *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.)

1.5 System Requirements

No privileges are required to use the PPL$ facility. However, 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.

1.5.1 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 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 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.

1.5.3 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 3 per participant.

1.5.4 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 initialize and terminate a process's access to PPL$, create and delete subordinates, retrieve subordinate information, and produce a name consistent throughout the application.

2.1 Accessing the PPL$ Facility

The PPL$ facility provides the following two routines to initialize and terminate your application's access to the facility:

PPL$INITIALIZE	Informs the PPL$ facility that the caller is forming or joining the parallel application
PPL$TERMINATE	Ends the caller's participation in the parallel application

2.1.1 Initializing PPL$

PPL$INITIALIZE informs the PPL$ facility that the caller is forming or joining the parallel application. You are not required to call this routine. The first time you call one of the PPL$ routines listed below, the PPL$ facility is automatically initialized. However, the routine PPL$INITIALIZE is provided in case you want to explicitly initialize the PPL$ facility from your program, or if you want to specify an initial number of pages of memory available to PPL$ that exceeds the default. Note that this is an application-wide setting, and calls by subordinates have no effect on the space allocated.

The routines that perform automatic initialization when first called are as follows:

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

If you do not call PPL$INITIALIZE, PPL$ allocates the default (link time) constant PPL$K_INIT_SIZE pages for its internal data structures. This initial allocation accommodates a minimum of 32 processes, 8 barriers, 8 semaphores, 4 events, 4 spin locks, and 16 global sections. (These numbers represent a rough guideline for combinations of PPL$ components. If you have less than 32 processes, for example, you can have more than 8 barriers, and so forth.) You can specify another value for the **size** argument in PPL$INITIALIZE 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 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$INITIALIZE. You can also specify the PPL$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 subordinate is assigned in order as it joins the application, so that all the subordinates in the application always return an index greater than zero.

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

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

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

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$CREATE_SHARED_MEMORY cannot map the shared memory to the same addresses for all participants, the condition value PPL$_NONPIC is returned. (This might occur when the application executes more than one 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.

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$FLUSH_SHARED_MEMORY routine writes (flushes the contents of) a global section to disk. This global section must have been created by a call to PPL$CREATE_SHARED_MEMORY. 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. If you did not specify a file name in the call PPL$CREATE_SHARED_MEMORY, a default file specification (with a default file type of DAT) is obtained from SYS$SCRATCH. The shared memory name is used as a related file name.

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.

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 from the global section. 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. The following restrictions apply when you are creating a zone:

- You can only call PPL$CREATE_VM_ZONE once per zone in your application. That is, once this routine has been called by any participant, all participants in the application can use the returned zone identifier to call any of the RTL LIB$ routines listed previously.
- 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 freed by another participant.

4 Synchronization Operations

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

- Barriers
- Events
- Semaphores
- Spin locks

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

4.1 Retrieving a Synchronization Element Identifier

Given the name of a barrier, event, semaphore, or spin lock, the PPL$FIND_SYNCH_ELEMENT_ID routine returns the identifier of the associated synchronization element. This routine is useful when you are trying to ensure that a particular synchronization element's identifier is available to all the participants that need access to that element. By naming the element when you actually create it, you can then use PPL$FIND_SYNCH_ELEMENT_ID to let other participants retrieve the identifier of the element of that name. Element names are case sensitive.

You can also retrieve the identifier of an element by naming that element and "re-creating" it. That is, after you have created an element, all participants that need to access that element's identifier can call the appropriate "create" routine, specifying the same name for the element. This returns the identifier of the existing element 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 all 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$CREATE_BARRIER	Creates a barrier synchronization element
PPL$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$FIND_SYNCH_ELEMENT_ID to retrieve the identifier of the barrier with that name.
- Call PPL$CREATE_BARRIER again, specifying the same barrier name, to retrieve the identifier of that barrier.

4.2.2 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.

4.2.3 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)
```

4.2.4 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).

4.2.5 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$WAIT_AT_BARRIER, the quorum will never be reached and the waiting participants will hang. By using PPL$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 six routines that help you implement event synchronization.

PPL$CREATE_EVENT	Creates a user-defined event
PPL$TRIGGER_EVENT	Sets the event state to *occurred*
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$AWAIT_EVENT	Blocks the caller until the event state becomes *occurred*
PPL$READ_EVENT	Returns the current state of the event

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

4.3.2 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 be delivered 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 the queue, and will be processed once the event actually occurs. 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** 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$. The second longword receives the value specified in the call to PPL$TRIGGER_EVENT that results in the delivery of this AST.
- For PPL$-defined events (those not created by the user), the **astprm** parameter must point to a vector of four unsigned longwords that accommodates the following:
  - The user's "context" longword
  - The longword to receive the event's distinguishing condition-value
  - The parameters to the PPL$-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.

4.3.3 Enabling an Event Signal

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

4.3.4 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)
```

4.3.5 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 to the event are processed, so that any enabled ASTs or signals, or both, are delivered, and anyone 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 that 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)
```

4.3.6 Reading an Event

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

4.3.7 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

   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

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

PPL$CREATE_SEMAPHORE	Creates and initializes a semaphore with a waiting queue
PPL$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

The semaphores provided by the PPL$ facility are counting semaphores; that is, they can take any nonnegative integer value. You can use a counting semaphore to control access to multiple physical resources. 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. A binary semaphore acts like a lock bit; it allows only one process at a time to execute a critical section or access a physical resource, such as a line printer or disk.

There are two basic operations that are performed on semaphores to implement mutual exclusion: *wait* and *signal*. The wait operation lets you acquire exclusive access to a critical section by decrementing the value of the semaphore. If the value of the semaphore is zero, the wait operation forces your process to wait until the semaphore has a value greater than zero, and then the wait operation decrements the semaphore, granting you access to the section. The signal operation lets you relinquish your access to a critical section by incrementing the semaphore so that a waiting process can acquire access.

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

4.4.2 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)
```

4.4.3 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)
```

4.4.4 Reading a Semaphore Value

The PPL$READ_SEMAPHORE routine returns the current and/or maximum values 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.

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$CREATE_SPIN_LOCK	Creates and initializes a simple (spin) lock
PPL$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

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

4.5.2 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)
```

4.5.3 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)
```

5 Developing Parallel Processing Applications

The Parallel Processing Facility (PPL$) is a process-oriented library of routines that simplifies development of a parallel application. This chapter discusses some recommended methods for using the Parallel Processing Facility for developing new programs. (Note that in this chapter, a participant can be either a parent process or a subordinate. For this version of PPL$, a subordinate is defined as a VMS subprocess.)

5.1 Comparing the Use of Synchronization Elements

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

5.1.1 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.

5.1.2 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.)

5.1.3 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 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.

5.1.4 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.

5.1.5 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$FIND_SYNCH_ELEMENT_ID, supplying the name of the element.
2. "Re-create" each element by calling the PPL$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 "anonymous" 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).

5.2 Considerations

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

5.2.1 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.

5.2.2 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.

5.2.3 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.

5.2.4 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. (A fine level of granularity means that less work, such as individual machine instructions or a few program statements, is performed by each task. Medium-grained tasks are procedures or subroutines within a program, or code sequences with a single routine of a program. The coarsest level of granularity means that tasks are separate programs.)

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 and VAX FORTRAN. 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. Example 6–1 shows the use of PPL$ routines in BLISS-32 to perform a predefined event test. Further discussion is provided at the end of this program listing.

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 use of synchronization
! mechanisms - in this case, a barrier.
!
!-

library 'sys$library:starlet';

require 'pplmsg';
require 'ppl$def';
require 'ppl$routines';

forward routine
    main,
    handler,
    print;


macro
    fail_badly_ =
       (local i;
        i = .0;          !access violation
        )
        %;
```

Example 6–1 Cont'd. on next page

Examples of Calling PPL$ Routines

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

```


literal
    a_user_defined_condition_value = %x'99880';

literal
    termination_exception = a_user_defined_condition_value + sts$k_info;


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

routine main =
begin
local
    status  : unsigned long;

literal
    num_children = 1;

enable handler;    !set up to get the termination exception


status = PPL$GET_INDEX (index);      !find out who I am
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 %string ('!/!UL:!_created barr= !XL!_stat= !XL'),
                                    .index, .barrier, .status);


case .index from 0 to num_children of
    set
    [0] : !PARENT CODE
            (
            !set up for termination event notification
            status = PPL$ENABLE_EVENT_SIGNAL (%ref(ppl$k_abnormal_exit),
                                              termination_exception);
            if not .status then return signal (.status);

            !create the child
            status = PPL$SPAWN (%ref(num_children), !how many helpers
                                0,                  !run the same image
                                0,                  !don't need the id_list
                                %ref(ppl$m_init_synch   !wait for child init
                                    or ppl$m_nodebug)); !leave out debug
            print (%ascid %string ('!/!UL:!_spawn stat= !XL'),
                                .index, .status);

```

Example 6–1 Cont'd. on next page

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

```

            !The parent feels safe in waiting at this barrier for work
            !completion, since a condition handler has been established.
            !This wait would hang the application if we had not arranged
            !to fix up the barrier in emergencies.

            status = PPL$WAIT_AT_BARRIER (barrier);
            print (%ascid %string ('!/!UL:!_wait stat= !XL'),
                                          .index, .status);
            );


    [1] : !CHILD 1 CODE
            (
            !This child would normally be doing some work for the application,
            !but it breaks, never reaching the barrier as the parent expects.

             fail_badly_;         !terminate

            !The remaining child code never executes...

            status = PPL$WAIT_AT_BARRIER (barrier);
            print (%ascid %string ('!/!UL:!_wait stat= !XL'),
                                    .index, .status);
            );


    [inrange]   : 0;

    [outrange]  : 0;

    tes;

!end; !bind


return ss$_normal;

end; !main
```

Example 6–1 Cont'd. on next page

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

```
;    0906  1    routine handler
;    0907  1        (
;    0908  1              signal_arr  : ref $bblock,      ! Signal vector
;    0909  1              mech_arr    : ref $bblock       ! Mechanism vector
;    0910  1        ) =
;    0911  2     begin
;    0912  2
;    0913  2     !+
;    0914  2     ! This handler is invoked as notification of the predefined event
;    0915  2     ! PPL$K_ABNORMAL_EXIT.  It fixes up the barrier quorum to account
;    0916  2     ! for the lost body, and prevents application-wide hang.
;    0917  2     !-
;    0918  2
;    0919  2       local status;
;    0920  2
;    0921  2
;    0922  2     if (.signal_arr[chf$l_sig_name] eql termination_exception) then
;    0923  3         (
;    0924  3         !avoid application-wide hang by completing the outstanding barrier wait
;    0925  3         status = PPL$ADJUST_QUORUM (barrier, %ref(-1));
;    0926  3         print (%ascid %string ('!/!UL:!_adjust_quorum stat= !XL'),
;    0927  3                                 .index, .status);
;    0928  2         );
;    0929  2
;    0930  2
;    0931  2     ss$_resignal      !this will show us the associated message
;    0932  2
;    0933  2
;    0934  1     end; !handler

        .PSECT  $PLIT$,NOWRT,NOEXE,2

;    0936  1     routine print (ctrstr, p1) =
;    0937  2     begin
;    0938  2
;    0939  2     !+
;    0940  2     ! This formats a string with $fao and writes it.
;    0941  2     !-
;    0942  2
;    0943  2     external routine
;    0944  2         LIB$PUT_OUTPUT;
;    0945  2
;    0946  2     local
;    0947  2         buffer  : $bblock[132],
;    0948  2         desc    : vector[2];
;    0949  2
;    0950  2         desc[0] = %allocation(buffer);
;    0951  2         desc[1] = buffer;
;    0952  2         $faol (ctrstr = .ctrstr, outlen=desc[0], outbuf=desc[0], prmlst=p1);
;    0953  2         LIB$PUT_OUTPUT (desc[0])
;    0954  2
;    0955  1     end;   !print
```

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.

Here, 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 signal as requested by the parent. The parent's condition handler adjusts the barrier quorum to account for this termination, the hang is prevented, and the application completes.

Example 6–2 shows the use of PPL$ routines in VAX FORTRAN to decompose a loop. Further discussion is provided at the end of this program listing.

Example 6–2 Using PPL$ Routines in VAX FORTRAN

```
        PROGRAM EXAMPLE1

C
C PROGRAM DESCRIPTION:
C
C       This example program demonstrates master/slave loop
C       decomposition using PPL$ routines.
C
C***********************************************************************
C DATA DECLARATIONS
C***********************************************************************

C EXTERNAL DEFINITIONS
        EXTERNAL    sts$k_info, sts$k_severe
        EXTERNAL    PPL$K_ABNORMAL_EXIT, PPL$M_NODEBUG
        INTEGER*4   PPL$INITIALIZE, PPL$CREATE_SHARED_MEMORY
        INTEGER*4   PPL$SPAWN, PPL$GET_INDEX
        INTEGER*4   PPL$CREATE_BARRIER, PPL$WAIT_AT_BARRIER
        INTEGER*4   PPL$CREATE_SEMAPHORE
        INTEGER*4   PPL$INCREMENT_SEMAPHORE, PPL$DECREMENT_SEMAPHORE
        INTEGER*4   PPL$ENABLE_EVENT_SIGNAL
        INTEGER*4   LIB$PUT_OUTPUT

C DEFINE ITEMS FOR USE WITH PPL$
        INTEGER*4   spawn_flags
        INTEGER*4   fatal_signal
                        !for event handling
        INTEGER*4   sem_max_val, sem_init_val
            PARAMETER (sem_max_val = 1, sem_init_val = 1)
                        !for a binary semaphore

C DEFINE APPLICATION DATA NEEDS
        INTEGER*4   stride
            PARAMETER (stride = 5)
                    !number of consecutive array indices each party processes
        INTEGER*4   subordinates
            PARAMETER (subordinates = 2)
                    !number of slaves
        INTEGER*4   array_size
            PARAMETER (array_size = 50)
                    !a small array for demonstrative purposes
        INTEGER*4   one_page
            parameter (one_page =  512)

```

Example 6–2 Cont'd. on next page

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

```
C DEFINE DATA TO BE USED LOCALLY BY EACH PARTICIPANT
        INTEGER*4   copies                  !for parent's spawn call
        INTEGER*4   id_list(subordinates)   !"
        INTEGER*4   index                   !each participant's PPL-index
        INTEGER*4   lenadr(2)               !for creating shared memory
        INTEGER*4   status                  !info on each call
        INTEGER*4   mygroup, row, col, offset  !for doing the real work
        CHARACTER*12   my_id, group         !just for execution trace

C DEFINE DATA FOR SHARING
        byte        front_guard(one_page)   !memory is mapped on page boundaries
        INTEGER*4   array1(array_size, array_size)      !input array
        INTEGER*4   array2(array_size, array_size)      !input array
        INTEGER*4   final_array(array_size, array_size) !output array
        INTEGER*4   next_task_number                    !work item info
        INTEGER*4   semaphore_id                        !synchronization
        INTEGER*4   barrier_id                          !"
        byte        rear_guard(one_page)

C PUT ALL THE SHARED DATA IN A COMMON BLOCK, WHICH WILL GET SHARED
        COMMON /pgm_shared_data/ front_guard,
        1                        array1,
        1                        array2,
        1                        final_array,
        1                        next_task_number,
        1                        semaphore_id,
        1                        barrier_id,
        1                        rear_guard


C***********************************************************************
C       CODE FOR ALL PARTICIPANTS STARTS HERE
C***********************************************************************

        type *,'Initializing'
        status = PPL$INITIALIZE ()
        if (.not. status) call LIB$STOP (%val(status))


C MAP SHARED ADDRESS SPACE - enough for the shared variables + the spacers

        lenadr(1) = %loc(rear_guard) + one_page - %loc(front_guard)
        lenadr(2) = %loc(front_guard)
        status = PPL$CREATE_SHARED_MEMORY ('pgm_shared_data', lenadr)
        if (.not. status) call LIB$STOP (%val(status))


C DISPATCH TO ROLE-SPECIFIC CODE - PARENT @100, CHILD @200

        status = PPL$GET_INDEX (index)
        if (.not. status) call LIB$STOP (%val(status))

        if (index .ne. 0) go to 200     !act like a child
        go to 100                       !act like a parent


```

Example 6–2 Cont'd. on next page

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

```
  C*********************************************************************
  C                PARENT CODE HERE
  C*********************************************************************

  C The master performs all set-up functions, initializing both the
  C application's data and the synchronization support.


  100     type *, 'Parent Init'


  C INIT A SEMAPHORE AND BARRIER FOR SYNCHRONIZATION

          status = PPL$CREATE_BARRIER (barrier_id, 'synch_barrier',
          1                           %ref (subordinates + 1))
                          !slaves and master all wait at this barrier
          if (.not. status) call LIB$STOP (%val(status))

          status = PPL$CREATE_SEMAPHORE (semaphore_id, 'mutex',
          1                              sem_max_val, sem_init_val)
          if (.not. status) call LIB$STOP (%val(status))


  C REQUEST A SIGNAL IF SOMETHING UNUSUAL OCCURS IN A SUBORDINATE

          fatal_signal = %loc(sts$k_severe)      !severe means we stop
          status = PPL$ENABLE_EVENT_SIGNAL (%loc(ppl$k_abnormal_exit),
          1                                 %val(fatal_signal))
          if (.not. status) call LIB$STOP (%val(status))


  C CREATE THE SUBORDINATES

          copies = subordinates
spawn_flags = %loc(ppl$m_nodebug) !disable child debug
          status = PPL$SPAWN (copies,            !how many children
          1                       ,              !use current image name
          1                       id_list,       !children IDs
          1                       spawn_flags)   !special D
          if (.not. status) call LIB$STOP (%val(status))


```

Example 6–2 Cont'd. on next page

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

```
C PREPARE FOR TASK (WORK ITEM) ALLOCATION

        next_task_number = 1


C INIT THE DATA TO BE PROCESSED

        do 11 i = 1,array_size          !clear the space for results
            do 12 j = 1,array_size
                final_array(i,j) = 0
12      continue
11      continue

        do 13 i = 1,array_size          !arbitrarily init array 1
            do 14 j = 1,array_size
                array1(i,j) = i
14      continue
13      continue

        do 15 i = 1,array_size          !likewise init array 2
            do 16 j = 1,array_size
                array2(i,j) = 1
16      continue
15      continue



C At this point, all initialization functions have been performed by
C the master, which must now wait for the subordinates to catch up.
C Each of the subordinates waits at this barrier, guaranteeing that
C they all proceed in unison.


C WAIT FOR THE CHILDREN TO INIT

        type *,'Parent waiting for children to init'
        status = PPL$WAIT_AT_BARRIER (barrier_id)
        if (.not. status) call LIB$STOP (%val(status))


C A master might also want to participate in the parallel code sections,
C which would happen right here.
C In this example, the master waits.

```

Example 6–2 Cont'd. on next page

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

```

C WAIT FOR THE CHILDREN TO COMPLETE

        type *,'Parent waiting for work completion'
        status = PPL$WAIT_AT_BARRIER (barrier_id)
        if (.not. status) call LIB$STOP (%val(status))


C VERIFY RESULTS - LEFT AS AN EXERCISE FOR THE READER

C       call verify_results
        write (*,2) ( ( (final_array(i,j), i=1,array_size), j=1,array_size) )
2       format (z12.8)


C WRITE TERMINATION MESSAGE

        type *, 'Parent Terminating'

        go to 999



C************************************************************************
C           CHILD CODE HERE
C************************************************************************


C PREPARE MY INDEX FOR OUTPUT - EXECUTION TRACE

200     write (unit=my_id, fmt='(I12)') index
        status = LIB$PUT_OUTPUT ('child init' // my_id(10:12))


C GET READY, SLAVES

        status = PPL$WAIT_AT_BARRIER (barrier_id)   !parent has to say go
        if (.not. status) call LIB$STOP (%val(status))


```

Example 6–2 Cont'd. on next page

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

```
C PROCESSING LOOP - PERFORM THE INTENDED PROGRAM FUNCTION

        do while (.true.)

            ! FIND OUT WHAT TO DO - IN A CRITICAL REGION

            status = PPL$DECREMENT_SEMAPHORE (semaphore_id)
            if (.not. status) call LIB$STOP (%val(status))

            mygroup = next_task_number
            next_task_number = next_task_number + stride

            status = PPL$INCREMENT_SEMAPHORE (semaphore_id)
            if (.not. status) call LIB$STOP (%val(status))


            ! MAYBE THERE'S NO WORK TO DO

            if (mygroup .gt. array_size) go to 888


            ! EXECUTION TRACE

            write (unit=group, fmt='(I12)') mygroup
            status =
        1   LIB$PUT_OUTPUT ('child/grp:' // my_id(10:12) // group(10:12))


            ! DO THE WORK

            do 333 offset = 0,(stride-1)
                row = mygroup + offset
                do 344 col = 1,array_size
                    final_array(row, col) = 0
                    do 355 i = 1,array_size
                        final_array(row,col) = final_array(row,col) +
        1                                (array1(row,i) * array2(i,col))
355                 continue
344             continue
333         continue

        end do


C CHILD TERMINATION POINT - get here when all work is done

888     status = PPL$WAIT_AT_BARRIER (barrier_id)
        if (.not. status) call LIB$STOP (%val(status))

        status = LIB$PUT_OUTPUT ('termination: child ' // my_id(10:12))

        go to 999
```

Example 6–2 Cont'd. on next page

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

```

C*************************************************************************
C          EXIT
C*************************************************************************

C WRITE STATUS

999     print 1,status
1       format(t8, 'final status= ',z12.8)

        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).

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 the specified barrier, thus allowing a barrier to dynamically alter the number of participants required to conclude a wait on that barrier. The barrier must have been created by PPL$CREATE_BARRIER. (See PPL$CREATE_BARRIER for more information about quorums.)

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$ADJUST_QUORUM allows you to dynamically alter the number of participants expected to wait at a barrier. A barrier has an associated "quorum" of participants. A quorum is the number of participants required to call PPL$WAIT_AT_BARRIER (and thereby be blocked) before all blocked participants are unblocked to pass the barrier.

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$_NORMAL	Routine successfully completed.
PPL$_INVARG	Invalid argument(s).
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*

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.

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.

CONDITION VALUES RETURNED

PPL$_NORMAL	Routine successfully completed.
PPL$_INVELEID	Invalid element identifier.
PPL$_INSVIRMEM	Insufficient virtual memory available.
PPL$_INVARG	Invalid argument(s).
PPL$_WRONUMARG	Wrong number of arguments.

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 anonymous (unnamed) barrier is created. An arbitrary number of anonymous 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 anonymous 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.

Other routines that implement barrier synchronization are as follows:

PPL$WAIT_AT_BARRIER	Wait until the quorum reaches the barrier.
PPL$READ_BARRIER	Return the barrier's quorum and number of waiting participants.
PPL$SET_QUORUM	Establish the initial quorum for the barrier.
PPL$ADJUST_QUORUM	Increment or decrement a barrier's quorum.

CONDITION VALUES RETURNED

PPL$_NORMAL	Routine successfully completed.
PPL$_ELEALREXI	An element of the same name already exists. (Alternate success status.)
PPL$_INCOMPEXI	Incompatible type of element with the same name already exists.
PPL$_INSVIRMEM	Insufficient virtual memory available.
PPL$_INVARG	Invalid argument(s).
PPL$_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 anonymous (unnamed) event is created, which can be referenced only by its identifier. An arbitrary number of anonymous 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.

Routines that implement event operations are as follows:

PPL$TRIGGER_EVENT	Sets the event state to *occurred* and examines the queue of requested operations. If any signals or ASTs have been enabled for the event, or if any participant is waiting for the event, the appropriate action is taken and the event state is reset to *not_occurred*.
PPL$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 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$ENABLE_EVENT_AST	Requests that a specified AST be delivered when the event has occurred. If the state is already *occurred* when this routine is called, the AST is immediately delivered and the state is reset to *not_occurred*. If the state is *not_occurred* when this routine is called, the request is queued to the event, and the AST is delivered as a result of a corresponding trigger for this event.
PPL$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*. 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*.

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 an AST is requested. Refer to the description section of PPL$ENABLE_EVENT_AST for more information.

CONDITION VALUES RETURNED

PPL$_NORMAL	Routine successfully completed.
PPL$_ELEALREXI	An element of the same name already exists. (Alternate success status.)
PPL$_INCOMPEXI	Incompatible type of element with the same name already exists.
PPL$_INSVIRMEM	Insufficient virtual memory available.
PPL$_INVARG	Invalid argument(s).
PPL$_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 anonymous (unnamed) semaphore is created. An arbitrary number of anonymous 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 an unsigned longword 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 longword 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 anonymous semaphores.) Another method is to store the returned **semaphore-id** in shared memory.

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). Routines that implement semaphore synchronization are as follows:

PPL$DECREMENT_SEMAPHORE	Waits for the semaphore to have a value greater than zero, then decrements the semaphore.
PPL$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.

CONDITION VALUES RETURNED

PPL$_NORMAL	Routine successfully completed.
PPL$_ELEALREXI	An element of the same name already exists. (Alternate success status.)
PPL$_INCOMPEXI	Incompatible type of element with the same name already exists.
PPL$_INSVIRMEM	Insufficient virtual memory available.
PPL$_INVARG	Invalid argument(s).
PPL$_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]*

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 value**

Specifies options for creating and mapping shared memory. The **flags** argument is the value 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. Note that the creator of a shared memory section always initializes the memory to zero, because a VMS global section is created. Using the PPL$M_NOZERO flag inhibits that behavior for subsequent callers. Thus, it is not appropriate to initialize data in a section to be shared before the *first* call to PPL$CREATE_SHARED_MEMORY.
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.

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 **file-name** argument is the address of a descriptor pointing to the file name. If you do not specify a file name, a default file specification (with a default file type of DAT) is obtained from SYS$SCRATCH. The shared memory name is used as a related file name.

If you specify a file that does not exist, PPL$CREATE_SHARED_MEMORY creates it. If you specify a file that already exists, the file size defaults to that of the shared memory.

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

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 starting address of the front guard from the last address of the end 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$_NORMAL	Routine successfully completed.
PPL$_CREATED	Shared memory created (success).
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 anonymous (unnamed) lock is created. An arbitrary number of anonymous 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 anonymous locks.) Another method is to store the returned **lock-id** in shared memory.

Routines that implement spin lock synchronization are as follows:

PPL$SEIZE_SPIN_LOCK	Obtain the lock for exclusive access.
PPL$RELEASE_SPIN_LOCK	Release the lock.

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$_NORMAL	Routine successfully completed.
PPL$_ELEALREXI	An element of the same name already exists. (Alternate success status.)
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 following restrictions apply when you are creating a zone.

- Call PPL$CREATE_VM_ZONE once for each zone in your application. That is, once this routine has been called by any participant, all participants in the application can use the returned **zone-id** to call the LIB$ virtual memory zone routines listed previously.
- 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.

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

CONDITION VALUES RETURNED

PPL$_NORMAL	Routine successfully completed.
PPL$_INSVIRMEM	Insufficient virtual memory.
PPL$_INVARG	Invalid argument.
PPL$_INVSTRDES	Invalid string descriptor for **zone-name**.

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. If the value of the semaphore is 0 at the time of the call, the caller is put in the associated queue and is suspended. The semaphore must have been created by PPL$CREATE_SEMAPHORE.

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.)

PPL$DECREMENT_SEMAPHORE

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 caller is blocked by the PPL$ facility's call to the system service $HIBER.

CONDITION VALUES RETURNED

PPL$_NORMAL	Routine successfully completed.
PPL$_INVELEID	Invalid element identifier.
PPL$_NOT_AVAILABLE	Operation cannot be performed immediately; therefore it is not performed.
PPL$_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 value**

Bit mask specifying actions to be performed before deleting the global section. The **flags** argument is 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$DELETE_SHARED_MEMORY deletes or unmaps from a global section that you created using the PPL$CREATE_SHARED_MEMORY routine. A VMS global section is a section of memory potentially available to all processes in the system.

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 another process is using the global section 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$_NORMAL	Routine successfully completed.
PPL$_INVARG	Invalid argument.
PPL$_WRONUMARG	Wrong number of arguments.

Any error returned by the system service $DGBLSC.

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.) 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 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 the 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.

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 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$_NORMAL	Routine successfully completed.
PPL$_INVELEID	Invalid element identifier.
PPL$_INSVIRMEM	Insufficient virtual memory available.
PPL$_INVARG	Invalid argument(s).
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**

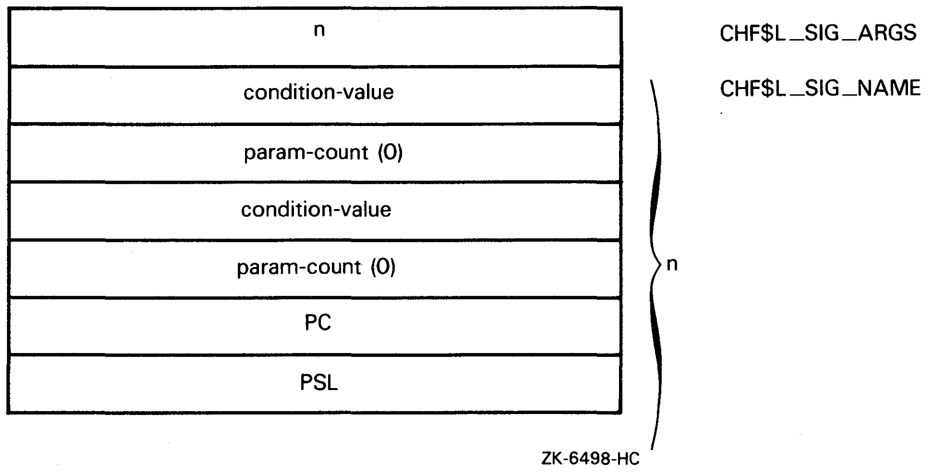
Optional user-defined value to be signaled when the event occurs. The **signal-value** argument is an unsigned longword containing this value.

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.) If the state is already *occurred* when you call this routine, the signal 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 a signal to notify the caller of an event's occurrence is placed in the 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.

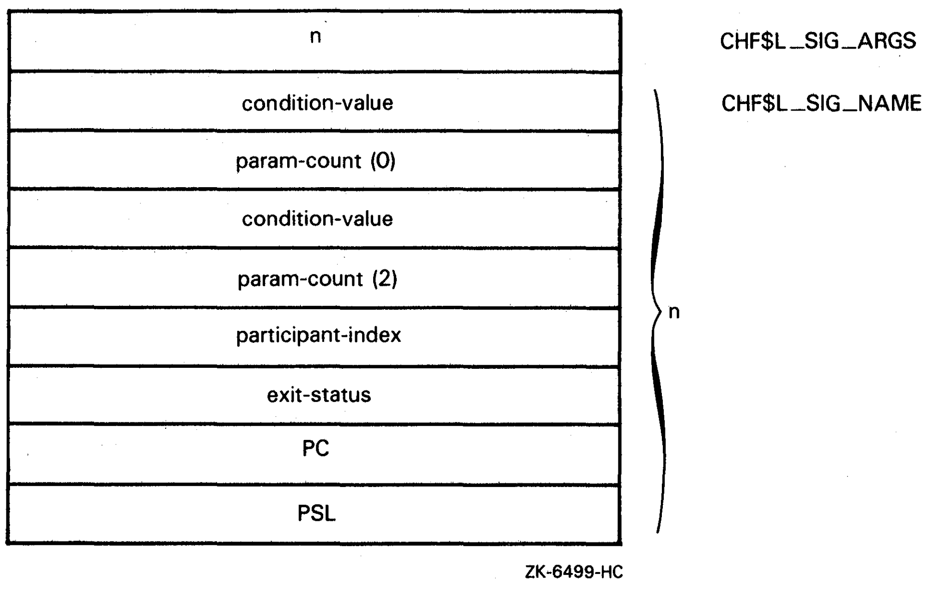
If you specify the **signal-value** argument, that value is the first condition signaled in the signal vector when the event occurs. If you do not specify **signal-value**, PPL$_EVENT_OCCURRED is signaled. If the **event-param** argument is specified in the call to PPL$TRIGGER_EVENT that causes the delivery of this signal, that argument appears as the second condition value in the signal vector. The following figure 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 event automatically. Refer to PPL$CREATE_EVENT for more information about predefined events. The following figure illustrates the structure of a signal vector for a PPL$-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. Usually, you should only call it once for each event for each participant.

CONDITION VALUES RETURNED

PPL$_NORMAL	Routine successfully completed.
PPL$_INVELEID	Invalid element identifier.
PPL$_INSVIRMEM	Insufficient virtual memory available.
PPL$_INVARG	Invalid argument(s).
PPL$_WRONUMARG	Wrong number of arguments.

PPL$FIND_SYNCH_ELEMENT_ID Find Synchronization Element Identification

Given the name of a spin lock, semaphore, barrier, or event, the Find Synchronization Element Identification routine returns the identifier of the associated synchronization element.

FORMAT

PPL$FIND_SYNCH_ELEMENT_ID *element-id ,element-name*

RETURNS

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

ARGUMENTS

element-id

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

Element identifier to be returned. The **element-id** argument is the address of an unsigned longword that receives the associated identifier.

element-name

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

Name of the synchronization element for which to return the associated identifier. The **element-name** argument is the address of a descriptor pointing to a character string containing the (user-defined) name of the synchronization element.

DESCRIPTION

Given the name of a spin lock, semaphore, barrier, or event, PPL$FIND_SYNCH_ELEMENT_ID returns the identifier of an element. An element is any spin lock, semaphore, barrier, or event previously created and named in a call to PPL$CREATE_SPIN_LOCK, PPL$CREATE_SEMAPHORE, PPL$CREATE_BARRIER, or PPL$CREATE_EVENT.

CONDITION VALUES RETURNED

PPL$_NORMAL	Routine successfully completed.
PPL$_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 value**

Bit mask specifying actions to perform before flushing the global section. The **flags** argument is 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. If you did not specify a file name in the call PPL$CREATE_SHARED_MEMORY, a default file specification (with a default file type of .DAT) is obtained from SYS$SCRATCH:. The shared memory name is used as a related file name. Only pages that have been modified are flushed to disk.

CONDITION VALUES RETURNED

PPL$_NORMAL	Routine successfully completed.
PPL$_INVARG	Invalid argument.
PPL$_INVDESC	Invalid descriptor.
PPL$_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 "top" or "main" participant. 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$GET_INDEX returns the unique index of the calling participant within the application. The index of the "top" or "main" execution thread is always zero. The index of each subordinate is assigned in the order in which it joins the application (by a call to PPL$INITIALIZE). For example, the first subordinate 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$_NORMAL	Routine successfully completed.
PPL$_INVELEID	Invalid element identifier.
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 or subprocess 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 or subprocess associated with the specified participant index.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_NO_SUCH_PARTY	The participant specified does not exist in this application.

PPL$INITIALIZE Initialize the PPL$ Facility

The Initialize the PPL$ Facility routine informs the PPL$ facility that the caller is forming or joining the parallel application. Calling this routine is optional, because PPL$ initializes itself at the first call to a PPL$ routine.

FORMAT

PPL$INITIALIZE *[size]*

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 **size** argument is the address of an unsigned longword containing this size value. By default, PPL$ allocates PPL$K_INIT_SIZE pages (available to user programs as a link-time constant) for its internal data structures. This initial allocation provided by PPL$ accommodates a minimum of 32 processes, 8 barriers, 8 semaphores, 4 events, 4 spin locks, 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 barriers, and so forth.) You can increase this allocation by specifying another value, as in the following example:

```
status = PPL$INITIALIZE (3*PPL$K_INIT_SIZE)
```

The **size** argument is ignored in all participants other than the "top" participant, because at the time any subordinate initializes, the size of the PPL$ reserved area has already been established for that application.

DESCRIPTION

PPL$INITIALIZE informs the PPL$ facility that the caller is forming or joining the parallel application. This routine initializes internal data structures to provide the caller with all the PPL$ features. You are not generally required to call this routine because it is performed automatically when you call any one of the PPL$ routines listed in the following table. Note that PPL$ does not automatically initialize when you call routines that require an already created element. This keeps the overhead of these routines — barrier, semaphore, event, and spin lock requests — at a minimum.

The routines that perform automatic initialization when first called are as follows:

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

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

1. Your system quotas are not sufficient for the amount of memory requested by the application.
2. 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.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_INSVIRMEM	Insufficient virtual memory available.
PPL$_WRONUMARG	Wrong number of arguments.

Any condition value returned by SYS$CRMPSC.

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 or subprocess 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.

CONDITION VALUES RETURNED

PPL$_NORMAL	Normal successful completion.
PPL$_NO_SUCH_PARTY	The participant specified does not exist in this application.

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.

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$_NORMAL	Routine successfully completed.
PPL$_INVARG	Invalid argument.
PPL$_INVELETYP	Invalid element type for specified operation.
PPL$_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$_NORMAL	Routine successfully completed.
PPL$_INVARG	Invalid argument.
PPL$_INVELETYP	Invalid element type for specified operation.
PPL$_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**

Value of the semaphore. The **semaphore-value** argument is the address of a signed word containing the current value of the semaphore specified by **semaphore-id**. A nonnegative value indicates the number of available resources associated with this semaphore. One or more participants may be blocked by the semaphore when the value is 0.

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 an unsigned longword containing the maximum value of the semaphore specified by **semaphore-id**.

DESCRIPTION

PPL$READ_SEMAPHORE returns the current or maximum values, or both, of the specified semaphore. If no values are requested, a status code of PPL$_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$_NORMAL	Routine successfully completed.
PPL$_INVARG	Invalid argument.
PPL$_INVELETYP	Invalid element type for specified operation.
PPL$_NOSUCHELE	The element you specified does not exist.
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$_NORMAL	Routine successfully completed.
PPL$_INVELEID	Invalid element identifier for requested operation.
PPL$_LOCNOTEST	The lock was not established.
PPL$_NOSUCHELE	An element with the specified identifier does not exist.
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. This allows you to easily reuse a barrier for different work items with various numbers of participants. The barrier must have been created by PPL$CREATE_BARRIER.

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. This value must be positive.

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.

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(s).
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. On output, this parameter contains the number of subordinates actually created. This value differs from the requested number if a 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**

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 flag. Valid values for **flags** are as follows:

PPL$M_INIT_SYNCH	The caller of this routine and all subordinates it creates are synchronized to continue processing only after each and every subordinate created by this routine has called PPL$INITIALIZE. (See the Description section for more information.) Note that this flag cannot be reliably used if other participants in the application also create subordinates using $CREPRC or LIB$SPAWN. Also, note that a failure of the created subordinate after it successfully starts but before its call to PPL$INITIALIZE can cause difficulties with the use of this flag value.
PPL$M_NODEBUG	Prevents the startup of the VMS Debugger, even if the debugger was linked with the image.

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$M_INIT_SYNCH flag arranges that processing in the parent and the subordinates only continues when each and every child created by this operation has called PPL$INITIALIZE. (Note that this initialization is performed automatically by PPL$ at the first call to a PPL$ routine; see PPL$INITIALIZE for more information.) This synchronization is achieved by blocking the parent in the call to PPL$SPAWN, and blocking each child in its PPL$INITIALIZE 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.

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$_NORMAL	Routine successfully completed.
PPL$_INVNUMPRO	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, $DELPRC, or LIB$FREE_VM_PAGE.

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.

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 value**

Specifies options for triggering an event. The **flags** argument is the value 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$TRIGGER_EVENT initiates the appropriate action, which is finally performed in the context of the participant that enabled the notification. If no participant has enabled notification of the event, the event state remains *occurred*. Otherwise, the notification resets the state to *not_occurred*. PPL$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.

CONDITION VALUES RETURNED

PPL$_NORMAL	Routine successfully completed.
PPLL$_INVELEID	Invalid element identifier.
PPL$_INSVIRMEM	Insufficient virtual memory available.
PPL$_INVARG	Invalid argument(s).
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.

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 tree but different from any other name within another application tree. 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$_NORMAL	Routine successfully completed.
PPL$_ATTUSETWO	Attempted use of two barriers simultaneously. Logic error in user's program that results in deadlock.
PPL$_INVELEID	Invalid element identifier.
PPL$_INVELETYP	Invalid element type for specified operation.
PPL$_WRONUMARG	Wrong number of arguments.

Index

Reader's Comments

VMS RTL Parallel Processing (PPL$) Manual
AA–LA74A–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