



PPL-VM API Reference

**Revision 1.3
2/2/2005 6:06 PM**

Copyright © IP Fabrics, Inc. 2005

IP Fabrics Corporate Headquarters
14964 NW Greenbrier Parkway
Beaverton, OR 97006
Telephone (main line): 503 444-2400
Telephone (FAX line): 503 444-2401
Website: <http://www.ipfabrics.com/>

Information in this document is furnished in connection with IP Fabrics products. No license, express or implied, to any intellectual property rights is granted by this document. This document and the software described in it are furnished under license and may only be used or copied in accordance with the terms of the license.

Copyright © 2005, IP Fabrics, Inc. All rights reserved.

Packet Processing Language™, PPL™ and PPL-VM™ are owned and copyrighted by IP Fabrics, Inc. Microsoft®, Windows® and Windows® XP are registered trademarks of Microsoft Corporation.

Linux® is a registered trademark of Linus Torvalds.

Red Hat® is a registered trademark of Red Hat, Inc.

MontaVista® is a registered trademark of MontaVista Software Inc.

Intel® and Pentium® are registered trademarks of Intel Corporation.

*Other brands, trademarks and names are property of their respective owners.

Table of Contents

1	INTRODUCTION	5
1.1	Revision History	5
1.2	Related Documents	5
1.3	Definition of Terms	6
1.4	Naming Conventions.....	7
1.4.1	Bit and byte order conventions	7
1.5	Copyrights	7
1.6	Document Scope	8
1.7	Software Layering Diagram.....	8
1.8	SysApp Composition.....	8
1.9	Functional Partitioning.....	8
2	IPF PUBLIC API.....	10
2.1	Overview	10
2.2	IPF Public Data Types.....	10
2.3	IPF Public API	11
2.4	Send a Packet or Frame to the VM	11
2.5	Signal an Event to the VM	12
2.6	Signal an Event to the VM with Data.....	12
2.7	Complete an Array Definition.....	12
2.8	Get a Pointer to an Array.....	13
2.9	Release a Pointer to an Array	14
2.10	Get the Size of an Array	14
2.11	Get Most Recent System Log Data	15
2.12	Get New System Log Data.....	15
2.13	Get System Log Record Size	16
2.14	Get Most Recent Header Debug Log Data.....	16
2.15	Get New Header Debug Log Data	17
2.16	Get Most Recent Packet Debug Log Data.....	17
2.17	Get New Packet Debug Log Data	18
2.18	Get VM Statistics Data	18
2.19	Start the VM.....	19
2.20	Stop the VM.....	19
2.21	Coldstart the System.....	20
2.22	Restart the System	21
2.23	Shutdown the System.....	22
2.24	Get System State	22
2.25	Get a Packet/Data from the VM	22
3	SYSAPP ERRORS.....	24
3.1	Error Codes	24
3.2	Return Codes.....	25
4	STATISTICS.....	26
4.1	Counter definitions	26

Table of Tables

Table 1: Revision History 5
Table 2: Reference Documents 5
Table 3: Terms and Definitions 6
Table 4: Error Codes 24
Table 5: Return Codes 25

Table of Figures

Figure 1. Byte Ordering 7
Figure 2. Typical Software Layering Diagram 8
Figure 3. SysApp Component Diagram 8
Figure 4. SysApp Control Paths 9

1 Introduction

1.1 Revision History

Rev	Date	Reason for Changes
1.0	10/30/2004	Initial public version
1.1	11/1/2004	Format and misc. corrections.
1.2	1/26/2005	Return code updates
1.3	1/28/2005	Added ipf_release_array_base()

Table 1: Revision History

1.2 Related Documents

	Document Title	Rev	Ref #
[1]	Packet Processing Language	1/24/2005	
[2]	PPL IXP2000-Family Device Map	10/28/2004	
[4]	PPL Compiler Design Document	1/18/2005	
[6]	Intel IXA Portability Framework Developer's Manual	August 2004	278662-007
[7]	Intel IXA Portability Framework Reference Manual	August 2004	278663-007
[8]	Intel IXP2400/IXP2800 Network Processor Programmers Reference Manual	May 2004	278746-016
[9]	Intel IXP2400/IXP2800 Network Processors Development Tools User's Guide	July 2004	278733-015
[10]	Intel IXA Software Building Blocks Developer's Manual	August 2004	278664-008

Table 2: Reference Documents

1.3 Definition of Terms

Term	Definition
Core Component	An executable module running on the XScale processor that initializes, processes data for, and communicates with microblocks running on the microengines.
Core Component Infrastructure (CCI)	An API library that facilitates the initialization of Core Components and communication with microblocks and other Core Components.
DeviceMap	The means by which a PPL programmer can specify the hardware environment he expects to execute his PPL program on.
IXA Portability Framework	An Intel software architecture designed to allow a network processor programmer to quickly create applications using both the microengine and XScale processors. Provides numerous APIs, functional blocks, and software layers to accomplish packet processing and message handling.
PPL	Packet Processing Language. A high-level language developed by IP Fabrics that allows OEMs and network operators to direct management and data plane processing functions on network traffic going through equipment running the PPL virtual machine (VM).
Resource Manager (RM)	A functional block of the Portability Framework that is the central object through which memory allocation, communication, and Core Component, and other functional support must go through.
Scratch Ring	A hardware-accelerated data structure residing in scratch memory that allows fast communication between microengines and the XScale.
System Application (or SysApp)	Linux application used to initialize PPL system.
System Repository	A function of the Resource Manager that contains a tree-oriented data structure of properties used in the system. The Resource Manager provides an API to access and manage the repository.
VM (or PPL VM)	Virtual Machine. A body of microcode & hardware acceleration that execute byte code instructions for expression evaluation and action execution as packets are received by the virtual machine.

Table 3: Terms and Definitions

1.4 Naming Conventions

The following is a list of the typographical conventions used in this document.

- Constant Width Used examples to show the contents of files, output of commands and typed content.
- Italic* Used for file and directory names, program and command names, command-line options and pathnames
- nnnN Used for hex numbers. All numbers in this document are base 10 unless designated otherwise.

1.4.1 Bit and byte order conventions

Bit and byte order will always be represented in big endian convention with the most significant bit and byte always being furthest to the left.

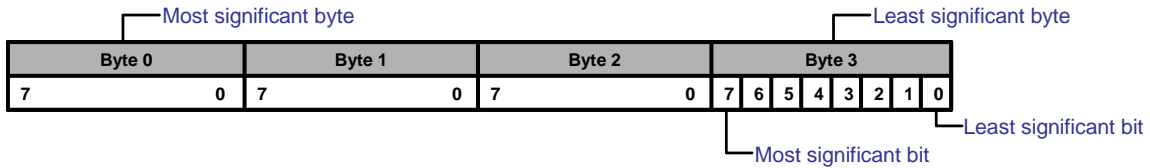


Figure 1. Byte Ordering

1.5 Copyrights

- © IP Fabrics Incorporated 2003-2005. All rights reserved.
- © Packet Processing Language, © PPL are copyrights of IP Fabrics Inc.
- Intel ® is a registered trademark of Intel Corporation.
- Linux ® is a registered trademark of Linus Torvolds.
- Brands, trademarks and names are property of their respective owners.

1.6 Document Scope

This document describes the public API calls that are used with the PPL-VM technology.

1.7 Software Layering Diagram

The layering of the major software components running on the XScale processor for a typical NPU application are shown below:

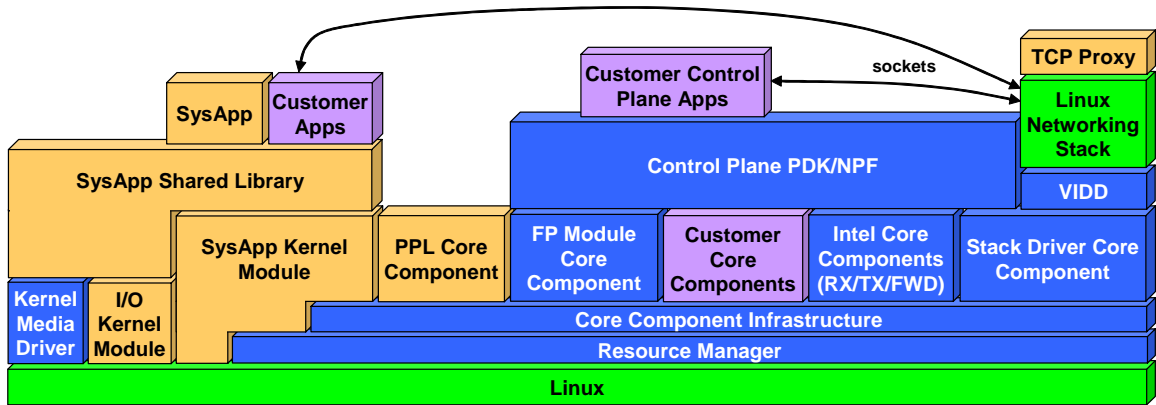


Figure 2. Typical Software Layering Diagram

1.8 SysApp Composition

The SysApp is composed of an executable and a kernel module as shown below. It is statically linked with all Core Components in the system and with the Core Component Infrastructure and Resource Manager libraries.

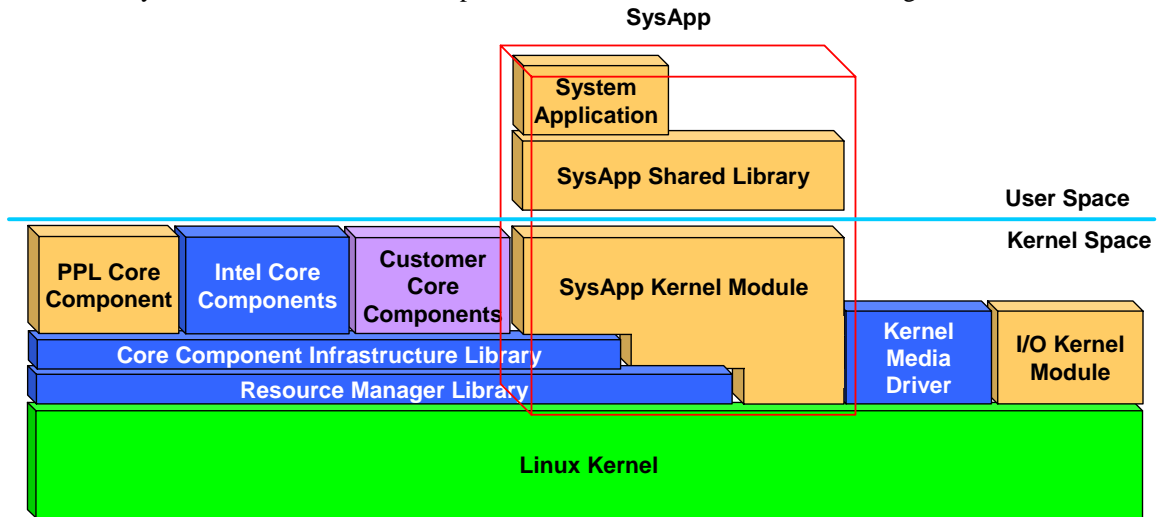


Figure 3. SysApp Component Diagram

1.9 Functional Partitioning

The user space SysApp performs the following tasks:

1. Processes command line arguments
2. Permits command line shutdowns and restarts of the system

The SysApp shared library (running in user space) performs the following tasks:

1. Opens and releases the SysApp, Kernel Media, and special I/O kernel modules

2. Opens and reads the compiler .bin output file and passes sections down to the SysApp kernel module using its ioctl commands

The kernel space SysApp performs the following tasks:

1. Initializes the IXA SDK infrastructure (Resource Manager, CCI, etc.)
2. Accepts .bin file sections (as pointers to buffers containing the sections) and processes them, then copies them to SRAM memory buffers allocated using the Resource Manager
3. Reads .UOF microcode file into kernel memory, patches symbols, and writes result to control stores
4. Initializes and instantiates all Core Components
5. Allocates all system resources (memory, scratch rings, buffer freelists, etc.) using the Resource Manager
6. Starts and stops the VM

The following diagram shows the relationship of the user space SysApp (command line program) and the kernel space SysApp (kernel module).

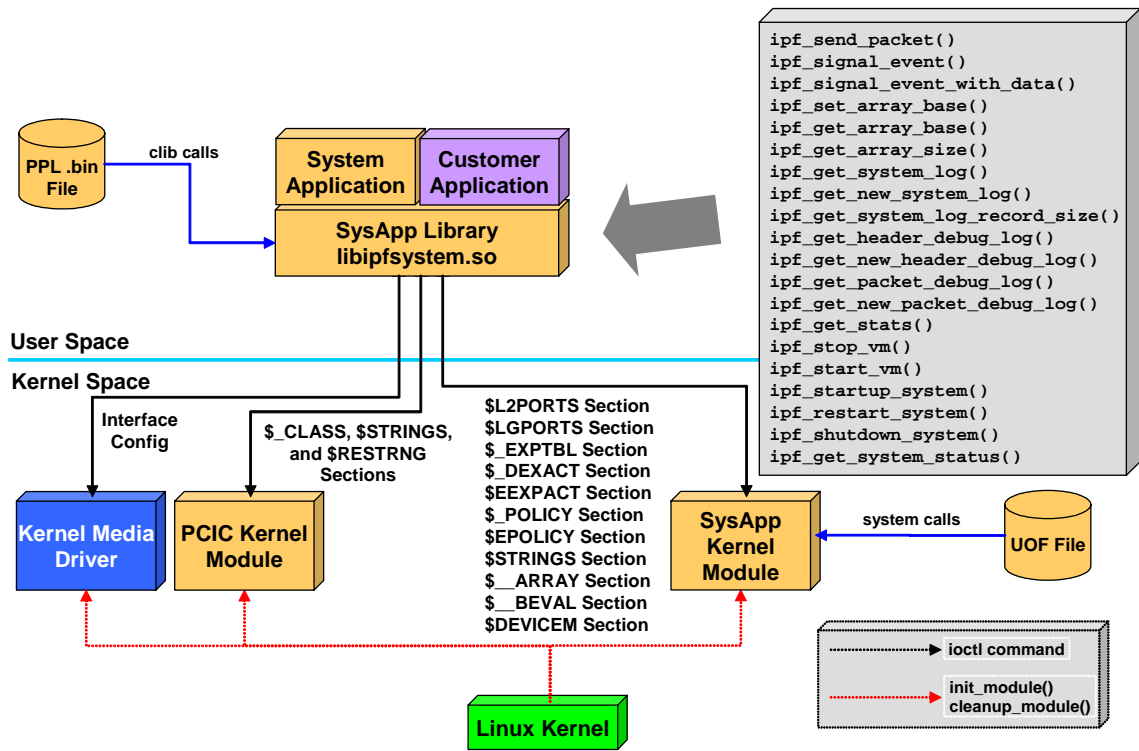


Figure 4. SysApp Control Paths

2 IPF Public API

2.1 Overview

The IPF API is intended to allow user space customer control plane applications running on the XScale to perform some system functions. Access to these functions is through a shared library (libipfsystem.so).

2.2 IPF Public Data Types

```
typedef unsigned char UCHAR;
typedef unsigned short USHORT;
typedef unsigned long ULONG;
typedef unsigned long long ULONGLONG;
typedef enum {
    systemshutdowncomplete = 0,
    systemshuttingdown,
    systemstartupcomplete,
    systeminitialized,
    systemstarted,
} status;
typedef enum {
    type_unknown = 0x00,
    type_ethernet = 0x01,
    type_ppp = 0x02,
    type_ppp_over_eth_discovery = 0x06,
    type_ppp_over_eth_session = 0x07,
    type_mpls_unicast = 0x08,
    type_mpls_multicast = 0x09,
    type_ipv4 = 0x21,
    type_ipv6 = 0x57,
    type_data = 0xfe,
} header_type;
typedef struct taglogheader {
    ULONG counter;
    ULONG timestamp;
    ULONG value;
    ULONG lpn;
} logheader;
typedef struct tagdbgloughdentry {
    ULONG counter1;
    ULONG atimestamp;
    ULONG event_data;
    UCHAR header_data[80];
    ULONG counter2;
} dbgloughdentry;
typedef struct tagdbglogpktentry {
    ULONG counter1;
    ULONG atimestamp;
    ULONG event_data;
    UCHAR packet_data[2048];
    ULONG counter2;
} dbglogpktentry;
```

```

typedef struct tagfilenames {
    char binfn[_POSIX_PATH_MAX]; // PPL binary
    char sucmdfn[_POSIX_PATH_MAX]; // startup command
    char sdcmdfn[_POSIX_PATH_MAX]; // shutdown command
    char mcodefn[_POSIX_PATH_MAX]; // microcode
} filenames;

const ULONG IPF_AUTOSTART = 1;
const ULONG IPF_CFG_IFC = 2;
const ULONG IPF_RESERVED = 4;

```

2.3 IPF Public API

All methods in this section are named with an “ipf_” prefix to denote their availability for use by any application.

All methods return IPF_SUCCESS for successful completion or a negative value when errors occur. In this case, the library sets *errno* to the values specified for each method below.

2.4 Send a Packet or Frame to the VM

```

int ipf_send_packet(UCHAR *data, ULONG size, header_type type,
                   ULONG lpn);

```

Synopsis: Send a packet

Parameters:

- data** Pointer to a buffer containing the frame or packet contents (XScale virtual address)
- size** Size, in bytes, of the frame data in the buffer pointed to by *data*
- type** An enumerated value indicating the type of the outermost header in the packet contained in *data*. Valid types are: *type_ethernet*, *type_ipv4*
- lpn** The logical port number the packet should be sent to

Returns: 0 if the call successfully completes, or one of the following error codes.

- ENOTSTARTED** The system is not in the *systemstarted* state
- EINVAL** *data* is NULL, *size* is greater than 2048, *lpn* is not defined in the PPL device map in use on the VM or is otherwise invalid, *type* is invalid
- EFAULT** Access errors to *data*

Description: Sends a frame or packet to the VM. The calling application must allocate a local buffer and initialize it with the frame or packet data as desired before calling this interface. This raw buffer is passed to the SysApp Kernel Module via an *ioctl* command, whereupon a buffer handle is retrieved from the freelist and the data is copied to the packet buffer starting at an offset into the buffer determined from the *type* parameter and the NPU type.

The raw buffer is then passed to the state lookup engine of the VM, which will treat the packet as if it were of the type specified by *type* for the purposes of setting up offsets for use by the PPL program. The packet will arrive in the PPL program on the event associated with the logical port number specified by *lpn*. *lpn* must be defined in the device map.

2.5 Signal an Event to the VM

```
int ipf_signal_event(ULONG event);
```

Synopsis: Signal a PPL event

Parameters:

event The event number to signal

Returns: 0 if the call successfully completes, or one of the following error codes.

ENOTSTARTED The system is not in the *systemstarted* state

EINVAL *event* is not specified in the PPL program running on the VM or is otherwise invalid

Description: Signals a packet-less event to the VM. The calling application specifies the event to signal using the *event* parameter. The PPL program running on the VM will then receive a packet-less event at that event number. Unless the packet-less event makes a packet current, packet operations performed by that event will result in exceptions being raised.

2.6 Signal an Event to the VM with Data

```
int ipf_signal_event_with_data(ULONG event, const ULONG *data,
                               ULONG size);
```

Synopsis: Signal a PPL event and pass data to it

Parameters:

event The event number to signal

data The pointer to buffer containing the data

size The number of 32-bit data items in the buffer (up to 32)

Returns: 0 if the call successfully completes, or one of the following error codes.

ENOTSTARTED The system is not in the *systemstarted* state

EINVAL *size* is not 0 and the *data* is NULL, the *event* not specified in the PPL program running on the VM or is otherwise invalid, *size* is out of range (0 – 32)

EFAULT Access errors to *data*

Description: Sends a message to the VM to signal an event and passes the event data. . The calling application must allocate a local buffer and initialize it with the data items as desired before calling this interface. The application also specifies the event to signal using the *event* parameter. Up to 32 data items, each of size 32 bits can be specified by *size* parameter. If *size* is 0, then *data* can be NULL. This case would be exactly equivalent to calling `ipf_signal_event()`.

2.7 Complete an Array Definition

```
int ipf_set_array_base(const char *external_name, ULONG base);
```

Synopsis: Complete the initialization of array descriptors whose memory was specified with a symbolic address

Parameters:

external_name A null-terminated string containing the name of the memory block holding the array, as specified in the ARRAY_MAP device map entry

base A pointer to the base address of the buffer to be used to hold this array. This value must be a physical address, not a virtual address (as would be returned by malloc, for example). If user

space access to this array is required, this parameter **must** lie on a physical page boundary (4k on Linux systems). If user space access to this array is not required, the base address must be aligned on a 16 byte boundary

Returns: 0 if the call successfully completes, or one of the following error codes.

ENOTSTARTED	The system is not in the <i>systeminitialized</i> state
EINVAL	<i>external_name</i> is NULL
ENOENT	<i>external_name</i> is not found in the device map

Description: The ARRAY_MAP device map command has a form that allows the user to specify an external name to use for storage of the array's contents rather than a fixed memory address. The format of this command is:

```
ARRAY_MAP(array_name, external_name)
```

If this command exists in a program's device map, 'array_name' will not be allocated at system startup by SysApp, but instead will be linked to 'external_name'. It is then up to the system designer to ensure that memory for the array is allocated using this API call, which links an actual memory location to 'external_name'. This allows systems to exercise increased control in managing their memory maps.

This API also causes initialization of the array memory to take place which had to be deferred until this call was made because the physical address of the array memory was not known at system initialization time.

2.8 Get a Pointer to an Array

```
int ipf_get_array_base(const char *name, void **address);
```

Synopsis: Get a pointer to a PPL array

Parameters:

name	A null terminated string giving the name of the array to which a pointer is desired
address	The address of a pointer to set to the base of the array requested. Upon successful completion of this command, the pointer this pointer points to will be set to the base of the array and may be accessed like any other malloc'ed structure

Returns: 0 if the call successfully completes, or one of the following error codes.

ENOTSTARTED	The system is not in the <i>systeminitialized</i> or <i>systemstarted</i> state
EINVAL	<i>name</i> is NULL or invalid, <i>address</i> is NULL
EFAULT	The array has an incomplete definition. This is usually caused by an array allocated by an ARRAY_MAP to an 'external_name' for which ipf_set_array_base() has not been called
ENOENT	<i>name</i> does not refer to an array defined in the currently running PPL program

Description: Returns a pointer to the base of the array named by *name*. This pointer can then be cast and dereferenced like any other malloc'ed structure.

2.9 Release a Pointer to an Array

```
int ipf_release_array_base(const char *name, void *address);
```

Synopsis: Release a pointer to a PPL array

Parameters:

name	A null terminated string giving the name of the array to which the pointer should be released
address	The pointer to the base of the array requested. Upon successful completion of this command, the virtual address in the pointer can no longer be used.

Returns: 0 if the call successfully completes, or one of the following error codes.

ENOTSTARTED	The system is not in the <i>systeminitialized</i> or <i>systemstarted</i> state
EINVAL	<i>name</i> is NULL or invalid, <i>address</i> is NULL
EFAULT	The array has an incomplete definition. This is usually caused by an array allocated by an ARRAY_MAP to an 'external_name' for which ipf_set_array_base() has not been called
ENOENT	<i>name</i> does not refer to an array defined in the currently running PPL program

Description: Releases system resources used to map a PPL array into the calling process's address space.

2.10 Get the Size of an Array

```
int ipf_get_array_size(const char *name, ULONG *size);
```

Synopsis: Get the size of an array

Parameters:

name	A null terminated string giving the name of the array of which the size is desired
size	A pointer to the size of the array in bytes

Returns: 0 if the call successfully completes, or one of the following error codes.

ENOTSTARTED	The system is not in the <i>systeminitialized</i> or <i>systemstarted</i> state
EINVAL	<i>name</i> is NULL or invalid, <i>size</i> is NULL
EFAULT	The array has an incomplete definition. This is usually caused by an array allocated by an ARRAY_MAP to an 'external_name' for which ipf_set_array_base() has not been called
ENOENT	<i>name</i> does not refer to an array defined in the currently running PPL program

Description: Returns the *size* of an array whose *name* is passed as a parameter. The array must already be fully defined in order to get its size, otherwise an error code is returned.

2.11 Get Most Recent System Log Data

```
int ipf_get_system_log(UCHAR *buffer, ULONG records,
                      ULONG *actual);
```

Synopsis: Get the most recent system logging information

Parameters:

buffer Pointer to an allocated buffer that will be used to hold the log data

records The maximum number of log records to retrieve

actual A pointer to the actual number of log records retrieved

Returns: 0 if the call successfully completes, or one of the following error codes.

ENOTSTARTED The system is not in the *systemstarted* state

EINVAL The value of either *buffer* or *actual* is NULL

EFAULT Accessing either *buffer* or the value pointed to by *actual* results in an access error

EPERM No system log memory was allocated in the device map

ENOSYSLOG No PPL system log is enabled

Description: Retrieve the most recent data from the system log into the buffer supplied. The length of system log entries is variable, but can be retrieved using `ipf_get_system_log_record_size()`. The start of each entry is given by the structure `logheader`, with the last word of the record being a replication of the counter found in that structure. Log entries are returned in reverse chronological order from the last valid entry in the log going back in time until there are no more entries or the number of entries indicated by *records* has been read.

2.12 Get New System Log Data

```
int ipf_get_new_system_log(UCHAR *buffer, ULONG recordnum,
                          ULONG records, ULONG *actual);
```

Synopsis: Get the system logging information starting at a certain log record

Parameters:

buffer Pointer to an allocated buffer that will be used to hold the log data.

recordnum The counter value for the first record to retrieve. If this record exists in the log, it will be the first record returned in the results.

records The maximum number of log records to retrieve.

actual A pointer to the actual number of log records retrieved.

Returns: 0 if the call successfully completes, or one of the following error codes.

ENOTSTARTED The system is not in the *systemstarted* state

EINVAL The value of either *buffer* or *actual* is NULL.

ENOENT The record indicated by *recordnum* cannot be found in the system log.

EFAULT An access error is received while attempting to access either *buffer* or *actual*.

ENOSYSLOG No PPL system log is enabled

Description: Retrieve system log records into the supplied buffer starting at the entry specified by *recordnum* and continuing forward in chronological order. The length of the entries is variable, but can be retrieved using `ipf_get_system_log_record_size()`. The start of each entry is given by the structure `logheader`, with the

last word of the record being a duplicate of the counter field found in that structure. The first entry returned by this call always is the record with counter equal to *recordnum*.

2.13 Get System Log Record Size

```
int ipf_get_system_log_record_size(ULONG *size);
```

Synopsis: Get the system logging record size in bytes, including the log header and trailer.

Parameters:

size A pointer to the location to store the record size, in bytes

Returns: 0 if the call successfully completes, or one of the following error codes.

ENOTSTARTED The system is not in the *systemstarted* state

EINVAL The value of *size* is NULL.

ENOENT System logging has been disabled.

Description: Retrieve the number of bytes in each system log entry. This function is typically used to properly allocate buffers for use by `ipf_get_system_log()` and `ipf_get_new_system_log()`. The format of each log entry is a **logheader** followed by some variable amount of user data, followed by a second counter which will match the counter in **logheader**.

2.14 Get Most Recent Header Debug Log Data

```
int ipf_get_header_debug_log(UCHAR *buffer, ULONG records,
                             ULONG *actual);
```

Synopsis: Get the most recent header debug logging information

Parameters:

buffer Pointer to an allocated buffer that will be used to hold the log data

records The maximum number of log records to retrieve

actual A pointer to the actual number of log records retrieved

Returns: 0 if the call successfully completes, or one of the following error codes.

ENOTSTARTED The system is not in the *systemstarted* state

EINVAL The value of either *buffer* or *actual* is NULL.

EFAULT Accessing either *buffer* or the value pointed to by *actual* results in an access error.

ENODBGLOG No debug log is enabled

Description: Retrieve the most recent data from the header debug log into the buffer supplied. The length and format of the header debug log entries is given by the structure **dbgloghdentry**. Log entries are returned in reverse chronological order from the last valid entry in the log going back in time until there are no more entries or the number of entries indicated by *records* has been read.

2.15 Get New Header Debug Log Data

```
int ipf_get_new_header_debug_log(UCHAR *buffer,
                                ULONG recordnum,
                                ULONG records,
                                ULONG *actual);
```

Synopsis: Get the header debug logging information starting at a certain log record

Parameters:

buffer	Pointer to an allocated buffer that will be used to hold the log data
recordnum	The counter value for the first record to retrieve. If this record exists in the log, it will be the first record returned in the results
records	The maximum number of log records to retrieve
actual	A pointer to the actual number of log records retrieved

Returns: 0 if the call successfully completes, or one of the following error codes.

ENOTSTARTED	The system is not in the <i>systemstarted</i> state
EINVAL	The value of either <i>buffer</i> or <i>actual</i> is NULL.
ENOENT	The record indicated by <i>recordnum</i> can not be found in the header debug log
EFAULT	An access error is received while attempting to access either <i>buffer</i> or <i>actual</i>
ENODBGLOG	No debug log is enabled

Description: Retrieve header debug log records into the supplied buffer starting at the entry specified by *recordnum* and continuing forward in chronological order. The length and format of the header debug log entries is given by the structure **dbgloghdentry**. The first entry returned by this call always is the record with counter equal to *recordnum*.

2.16 Get Most Recent Packet Debug Log Data

```
int ipf_get_packet_debug_log(UCHAR *buffer, ULONG records,
                             ULONG *actual);
```

Synopsis: Get the most recent packet debug logging information

Parameters:

buffer	Pointer to an allocated buffer that will be used to hold the log data.
records	The maximum number of log records to retrieve
actual	A pointer to the actual number of log records retrieved

Returns: 0 if the call successfully completes, or one of the following error codes.

ENOTSTARTED	The system is not in the <i>systemstarted</i> state
EINVAL	The value of either <i>buffer</i> or <i>actual</i> is NULL.
EFAULT	Accessing either <i>buffer</i> or the value pointed to by <i>actual</i> results in an access error.
ENODBGLOG	No debug log is enabled

Description: Retrieve the most recent data from the packet debug log into the buffer supplied. The length and format of the packet debug log entries is given by the structure **dbglogpktentry**. Log entries are returned in reverse chronological order from the last valid entry in the log going back in time until there are no more entries or the number of entries indicated by *records* has been read.

2.17 Get New Packet Debug Log Data

```
int ipf_get_new_packet_debug_log(UCHAR *buffer,
                                ULONG recordnum,
                                ULONG records,
                                ULONG *actual);
```

Synopsis: Get the packet debug logging information starting at a certain log record

Parameters:

buffer	Pointer to an allocated buffer that will be used to hold the log data
recordnum	The counter value for the first record to retrieve. If this record exists in the log, it will be the first record returned in the results
records	The maximum number of log records to retrieve
actual	A pointer to the actual number of log records retrieved

Returns: 0 if the call successfully completes, or one of the following error codes

ENOTSTARTED	The system is not in the <i>systemstarted</i> state
EINVAL	The value of either <i>buffer</i> or <i>actual</i> is NULL
ENOENT	The record indicated by <i>recordnum</i> can not be found in the header debug log
EFAULT	An access error is received while attempting to access either <i>buffer</i> or <i>actual</i>
ENODBGLOG	No debug log is enabled

Description: Retrieve packet log records into the supplied buffer starting at the entry specified by *recordnum* and continuing forward in chronological order. The length and format of the packet debug log entries is given by the structure **dbglogpktentry**. The first entry returned by this call always is the record with counter equal to *recordnum*.

2.18 Get VM Statistics Data

```
int ipf_get_stats(ULONG *buffer, ULONG buffersize);
```

Synopsis: Get statistics data from the VM

Parameters:

buffer	A pointer to a locally allocated buffer in which to store the VM statistics
buffersize	The size of the buffer pointed to by <i>buffer</i> . Currently, this should be 512

Returns: 0 if the call successfully completes, or one of the following error codes.

ENOTSTARTED	The system is not in the <i>systemstarted</i> state
EINVAL	The value of <i>buffer</i> is NULL or <i>buffersize</i> is too small
EFAULT	An access error is received while writing to <i>buffer</i>

Description: Retrieve PPL VM statistics counter values and store them in the buffer provided by the user. For each of the sixteen microengines, there are 8 32bit counters, for a total of 512 bytes of counter data. The meaning of the various counter values is described in section 4.1 of this document.

2.19 Start the VM

```
int ipf_start_vm(void);
```

Synopsis: Start the VM by starting all PPL microengines

Returns: 0 if the call successfully completes, or one of the following error codes.

ENOTSTARTED The system is not in the *systeminitialized* state
EPERM The VM cannot be started, usually because it was already started.

Description: Starts the VM microengines associated with the PPL VM.

This function should only be called when the system has been started (using the `ipf_startup_system()` API) with the `IPF_AUTOSTART` flag *not* specified. If such a startup is successful, all initialization possible has been performed except that the VM microengines have not been started. `ipf_start_vm()` can then be used to start the VM microengines.

This function is useful for instances when user-allocated arrays are used in the PPL program. In this case, SysApp cannot fully initialize the system until the base addresses of all array memory is known (using the `ipf_set_array_base()` API) or a VM exception will occur the first time the array is accessed in the PPL program. For such a case, the system is started without autostarting the VM, the array base addresses are specified, then the VM is started using this API.

This function leaves the system in the *systemstarted* state if successful.

2.20 Stop the VM

```
int ipf_stop_vm(void);
```

Synopsis: Perform an orderly stop of the VM

Returns: 0 if the call successfully completes, or one of the following error codes.

ENOTSTARTED The system is not in the *systemstarted* state
EPERM The VM cannot be stopped, usually because it was already stopped.

Description: The VM is stopped by halting the microengines associated with the PPL VM.

This function leaves the system in the *systeminitialized* state if successful.

2.21 Coldstart the System

```
int ipf_startup_system(ULONG flags, filenames *fn);
```

Synopsis: Startup the system from a shutdown condition

Parameters:

flags Set to any of: `IPF_AUTOSTART`, `IPF_CFG_IFC` (can be OR-ed together)

fn Pointer to the **filenames** structure.

Returns: 0 if the call successfully completes, or one of the following error codes.

ENOTSHUTDOWN The system is not in the *systemshutdowncomplete* state

Description: The system is initialized.

An initialized `filenames` structure must be passed to this API to tell it what filenames to use for various system operations. At a minimum, the *binfn* member must be initialized to a string containing the filename of the binary file produced by the PPL compiler. All unused filename strings must be null (first character is the NUL character – ‘\0’).

Specifying the `IPF_AUTOSTART` flag causes SysApp to start the VM after it has completed loading the system. This is analogous to *not* specifying the “-i” command line parameter to the usermode SysApp and is useful if the PPL program uses arrays whose physical memory is not known at SysApp runtime. In that case, the VM should not be allowed to run as any references to elements of such arrays will cause VM exceptions or memory corruption.

Specifying the `IPF_CFG_IFC` flag causes SysApp to configure the physical interfaces specified in the device map. This is analogous to *not* specifying the “-l” command line parameter to the usermode SysApp and is useful for running on NPUs that do not have the capability to configure the MACs.

Upon successful completion of the startup tasks, the system is left in the *systemstartupcomplete* state if the `IPF_AUTOSTART` flag was specified or the *systeminitialized* state if the flag was not specified. Errors cause the system to be left in the *systemshutdowncomplete* state.

2.22 Restart the System

```
int ipf_restart_system(ULONG flags, filenames *fn);
```

Synopsis: Complete system restart

Parameters:

flags Set to any of: `IPF_AUTOSTART`, `IPF_CFG_IFC` (can be OR-ed together)

fn Pointer to the **filenames** structure.

Returns: 0 if the call successfully completes, or one of the following error codes.

ENOTSTARTED The system is not in the *systemstarted* state during the shutdown phase

ENOTSHUTDOWN The system is not in the *systemshutdowncomplete* state during the startup phase

EPERM The system cannot be restarted, usually because the system was stopped or a restart was already in progress.

Description: : Packet reception is stopped and restarted by calling the Kernel Media Driver. All resources are freed and infrastructure torn down, then the system is reinitialized identically to powerup. Used when a new PPL program needs to be executed.

An initialized `filenames` structure must be passed to this API to tell it what filenames to use for various system operations. At a minimum, the *binfn* member must be initialized to a string containing the filename of the binary file produced by the PPL compiler. All unused filename strings must be null (first character is the NUL character – ‘\0’).

Specifying the `IPF_AUTOSTART` flag causes SysApp to stop the VM when it is shutting down the system and to start the VM after it has completed reloading the system. This is analogous to *not* specifying the “-i” command line parameter to the usermode SysApp and is useful if the PPL program uses arrays whose physical memory is not known at SysApp runtime. In that case, the VM should not be allowed to run as any references to elements of such arrays will cause VM exceptions or memory corruption.

Specifying the `IPF_CFG_IFC` flag causes SysApp to disable the physical interfaces specified in the device map when it is shutting down the system and to configure and enable the physical interfaces when it is loading the system. This is analogous to *not* specifying the “-I” command line parameter to the usermode SysApp and is useful for running on NPUs that do not have the capability to configure the MACs.

Upon successful completion of a restart, the system is left in the *systemstartupcomplete* state if the `IPF_AUTOSTART` flag was specified or the *systeminitialized* state if the flag was not specified. Errors in the shutdown portion cause the system to be left in the *systemstartupcomplete* or *systeminitialized* state, whichever it was in before the restart API was called. Errors in the startup portion cause the system to be left in the *systemshutdowncomplete* state.

2.23 Shutdown the System

```
int ipf_shutdown_system(ULONG flags, filenames *fn);
```

Synopsis: Orderly shutdown of system

Parameters:

flags Set to any of: `IPF_AUTOSTART`, `IPF_CFG_IFC` (can be OR-ed together)

fn Pointer to the **filenames** structure.

Returns: 0 if the call successfully completes, or one of the following error codes.

ENOTSTARTED The system is not in the *systemstarted* state

Description: New packets at the ports are dropped by calling the Kernel Media Driver. The VM microengines are stopped. All resources are freed and infrastructure torn down, files closed, etc.

An initialized filenames structure must be passed to this API to tell it what filenames to use for various system operations. For shutdown, no filenames are required, however the strings must all be null (first character is the NUL character – ‘\0’).

Specifying the `IPF_AUTOSTART` flag causes SysApp to stop the VM when it is shutting down the system. This is analogous to *not* specifying the “-i” command line parameter to the usermode SysApp.

Specifying the `IPF_CFG_IFC` flag causes SysApp to disable the physical interfaces specified in the device map when it is shutting down. This is analogous to *not* specifying the “-I” command line parameter to the usermode SysApp and is useful for running on NPUs that do not have the capability to configure the MACs.

Upon successful completion of a shutdown, the system is left in the *systemshutdowncomplete* state. Errors cause the system to be left in the *systemstartupcomplete* or *systeminitialized* state, whichever it was in before the API was called.

2.24 Get System State

```
int ipf_get_system_state(status *systemstatus);
```

Synopsis: Get the current status of the system

Parameters:

systemstatus A pointer to an enumeration of the system’s status. Values are defined by the **status** enum.

Returns: 0 if the call successfully completes, or one of the following error codes.

EFAULT A NULL pointer was passed for *systemstatus*

Description: Retrieve the current system status.

2.25 Get a Packet/Data from the VM

There is no explicit API provided to access a packet/data that is sent to a XScale application by the VM. However any packet/data that was sent to the XScale through the RM can be accessed using the function calls from the PCAP library.

This is of course in addition to any higher level application on the XScale which has been set up to receive the packets (E.g. Sockets). These applications will receive the data/packets through the Local IP Stack on the XScale. The RM/Stack Driver CC will route the packets coming from the VM to the Local IP Stack.

The packet/data is sent by the VM to the RM using *forward* or *program* instruction in the PPL program. The device map that is associated with PPL program should contain a PROG interface that maps the *lpn* used in the *forward* or *program* instruction to the Stack Driver’s Comm ID (*external name*). The Stack Driver is assigned the Comm ID value of **89** by the RM/SysApp. Any packet/data that is sent to this Comm ID (89) is routed to the Stack Driver CC by the RM.

The packet/data routed to the Stack Driver is seen as arriving on the network interface corresponding to its input port (if it entered the system through hardware Port 0, then it is available on the network interface corresponding to Port 0 and so

on). Any packet that was internally created (either by the VM or XScale) will have its input port set to 0. As such if these packets need to be accessed then a hardware Port 0 should be present.

If the PCAP library is used, the packet/data itself can be accessed by using the using the *pcap_open_live* call to open a session to the interface and then using the *pcap_next* call to obtain the packet/data buffer. A sample application called *ppl_pktio* has been provided in the API samples section. It shows the use of the PCAP library calls to access a packet that was sent by the VM to the XScale App.

3 SysApp Errors

3.1 Error Codes

The following are error codes that the API returns if it encounters any problems during execution:

Error Description	Error Code	Error Value	Type
Success	IPF_SUCCESS	0	
Linux error codes		1 to 124	Varies
Cannot open compiler object file	EBINARYFILEOPEN	200	Fatal (exit after cleanup)
Cannot open SysApp driver	ESYSAPPDRIVEROPEN	203	Fatal (exit after cleanup)
Cannot open kernel media driver	EKMDDRIVEROPEN	204	Fatal (exit after cleanup)
Error reading compiler object file	EBINARYFILEREAD	210	Fatal (exit after cleanup)
Error reading microcode file	EUCODEFILEREAD	211	Fatal (exit after cleanup)
Error initializing Resource Manager	ERMINIT	220	Fatal (exit after cleanup)
Error initializing DeviceMap	EDEVMAPINIT	221	Fatal (exit after cleanup)
Error initializing Core Component Infrastructure	ECCIINIT	222	Fatal (exit after cleanup)
Error initializing System Repository	ESYSREPOSITORYINIT	223	Fatal (exit after cleanup)
Error initializing microengine communication	EMECOMMIT	224	Fatal (exit after cleanup)
Error creating Core Component	ECCCREATE	225	Fatal (exit after cleanup)
System not shutdown	ENOTSHUTDOWN	226	Fatal (exit after cleanup)
System not started	ENOTSTARTED	227	Fatal (exit after cleanup)
Error allocating buffer handle freelist	EBHFREELISTALLOC	230	Fatal (exit after cleanup)
Error allocating connections table	ECXTABLEALLOC	231	Fatal (exit after cleanup)
Error allocating association table	EASSOCTABLEALLOC	232	Fatal (exit after cleanup)
Error allocating SADB	ESADBALLOC	233	Fatal (exit after cleanup)
Error allocating logging buffer	ELOGBUFFERALLOC	234	Fatal (exit after cleanup)
Error allocating journal buffer	EJOURNALBUFFERALLOC	235	Fatal (exit after cleanup)
Error allocating scratch rings	ESCRATCHRGINALLOC	236	Fatal (exit after cleanup)
Error patching microcode	EUCODEPATCH	240	Fatal (exit after cleanup)
Error writing microcode	EUCODEWRITE	241	Fatal (exit after cleanup)
Error starting VM	ESTARTVM	260	Fatal (exit after cleanup)
Error stopping VM	ESTOPVM	261	Fatal (exit after cleanup)
Compiler major version mismatch	EVERNUMMISMATCH	300	Fatal (exit after cleanup)
Compiler output file checksum error	ECHECKSUM	301	Fatal (exit after cleanup)
No PPL syslog configured	ENOSYSLOG	310	Fatal (exit after cleanup)
No PPL debug log configured	ENODBGLOG	311	Fatal (exit after cleanup)
Name not found	ENAMENOTFOUND	312	Fatal (exit after cleanup)

Table 4. Error Codes

3.2 Return Codes

The following are codes returned by the SysApp when it exits. These can be used in shell scripts to act on any errors encountered during SysApp execution.

Return Code	Description
0	Successful execution
< 0	Error codes as described in Table 4

Table 5. Return Codes

4 Statistics

4.1 Counter definitions

The following tables summarize the definitions of counters defined for GA release 1.0. These definitions are subject to redefinition in future releases.

CE (Normal or Demo):

Counter Index	Variable	Function
0	Sls_entries_rcvd	increments: scratch ring entry dequeued
1	S3ls_outputs_sent	increments: output sent through nn regs
2	Sls_unresolved_lpons	increments: LPN is not defined in logical port table
3	Sls_lpons_oor	increments: packets sent in with invalid lpn
4	Sls_rej_malforms	increments: reg/malform criteria met
5	Sls_vm_drops	increments: CE packets dropped
6	Sls_nn_fulls	increments: nn ring put failed due to ring full
7	100ms_tick	increments: every 100ms

BE (Normal or Demo):

Counter Index	Variable	Function
0	Pkt_rx	increments: packet received from CE
1	Pkt_tx_ae	increments: packet sent to AE
2	Be_rule_cnt	totals possible true rules sent to AE (not valid without DEMO switch)
3	Ring_full	increments: SRAM ring put failed due to ring full
4	<unused>	
5	<unused>	
6	<unused>	
7	<unused>	

AE (Normal only):

Counter Index	Variable	Function
0	E_pkt_rx	increments: packet or container received by AE early
1	E_contianer_rx	increments: container received by AE early (AE early dropped the container)
2	L_pkt_rx	increments: packet received by AE late
3	L_pkt_fwd	increments: packet forwarded by AE late
4	L_pkt_drop	increments: packet drop by AE (only late can drop packet)
5	T_excep_raise	increments: total exception raised
6	E_l_scan	increments: early and late total scan instruction executed
7	100ms_tick	increments: every 100ms

AE (Demo only):

Counter Index	Variable	Function
0	E_pkt_rx	increments: packet or container received by AE early
1	E_contianer_rx	increments: container received by AE early (AE early dropped the container)
2	L_pkt_rx	increments: packet received by AE late
3	L_pkt_fwd	increments: packet forwarded by AE late
4	L_pkt_drop	increments: packet drop by AE (only late can drop packet)
5	E_false_rule	increments: early evaluate false rule number
6	L_rule_cnt	increments: total true rule late executed
7	100ms_tick	increments: every 100ms

PE:

Counter Index	Variable	Function
0	Pkt_scheduled	increments: packet scheduled for transmit
1	Schedule_hold	increments: scheduling is required, but TX is slow, so scheduling is dismissed (indicating TX is slow)
2	Enqueue_failed	increments: no packet to schedule while requested (indicating TX is faster)
3	<unused>	
4	<unused>	
5	<unused>	
6	<unused>	
7	100ms_tick	increments: every 100ms

IPv4 (Normal or Demo):

Counter Index	Variable	Function
0	Pkt_rx	increments: packet received from AE
1	Pkt_tx	increments: packet forward out to TX
2	Pkt_drop	increments: packet dropped
3	Pkt_to_Xscale	increments: packets sent to Xscale core
4	<unused>	
5	<unused>	
6	<unused>	
7	<unused>	

Ethernet TX (Normal or Demo):

Counter Index	Variable	Function
0	Pkt_tx_16p_tx_request_rxed	increments: packet sent request received
1	Pkt_tx_16p_pkt_discarded	increments: packet discarded due to internal queue full
2	<unused>	
3	<unused>	
4	<unused>	
5	<unused>	
6	<unused>	
7	<unused>	

Ethernet RX (Normal or Demo):

Counter Index	Variable	Function
0	Pkt_rx_num_proper_mpkts_rxed	increments: non-error mpackets received
1	Pkt_rx_num_er_mpkts_rxed	increments: error mpackets received
2	Pkt_rx_drop_scr_ring_full	increments: packet drop due to ring (towards CE) full
3	Pkt_rx_num_pkts	increments: packet sent to next stage
4	Pkt_rx_drop	increments: every packet drop
5	Pkt_rx_buf_alloc_fail	increments: fail to allocate a buffer handle
6	Pkt_rx_sesp_pkt	increments: single buffer packet received
7	Pkt_rx_number_pause_time	increments: every time pause kicked in