DATE: March 29, 1984

TO: R & D Personnel

FROM: Ralph Becker

SUBJECT: NPX User's Guide

REFERENCE: PE-TI-793, PRIMENET Guide

KEYWORDS: NPX

ABSTRACT

This document is intended to be a complete, updated user's guide to the NPX (Network Process Extension) facility. NPX is a remote procedure call mechanism for execution of shared dynamically linkable procedures on remote CPUs, hence allowing access to remote resources. Included is a functional description, internals overview, and calling sequence reference. This document supersedes all previous NPX documents except where noted. The documents archived are:

PE-TI-643 Remote Procedure Call (RPCL) (T. Taylor)
PE-TI-776/1 Subsystems Programming Guide to NPX (T. Taylor)
PE-TI-1031 The Implementation of SLAVID in NPX (H. Chen)
PE-TI-1032 Usage of Asynchronous RPCLs (H. Chen)

Table of Contents

1 Introduction to NPX

Please note: NPX is unreleased for general use. Any engineers thinking about using it should contact the network group for consultation before implementation.

The NPX facility allows a program to execute any direct call or shared library routine on any CPU in the visible (configured) network. For instance, a program running on ENB could call NEXT$ to read the next record of a MIDAS file on END. Each Remote Procedure Call (PCL) made by NPX is actually executed on the remote machine by a special dedicated server process similar to a phantom known as a slave. The process that calls a slave is known as the master.

A slave can serve only one master; that is, once a slave is associated with a master, the relationship between them remains until one side (usually the master) explicitly breaks the relationship and returns the slave to the available pool. A master process may have many slaves, but only one per remote node. If a process attempts to acquire more than one slave on a particular node, NPX will detect this and simply use the slave that already exists on that node. The master-slave relationship, including all status information, is retained until explicitly broken by the master, logout, or unrecoverable node or communications failure.

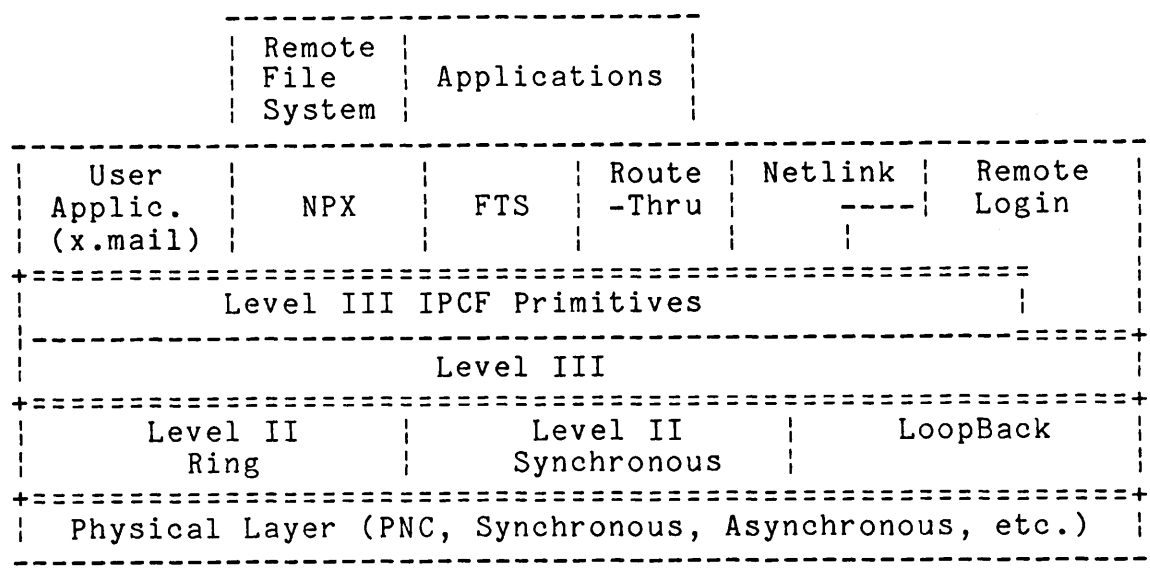
All arguments and the ASCII target subroutine name are copied to the slave process via the network inter process communication facility (IPCF) primitives. The slave then constructs a DYNT (an unsnapped dynamic link) to the target routine and calls it with the user supplied arguments. Return arguments from the target call are sent back to the user just as the input arguments were copied to the slave.

1.1 History

NPX was developed and written in FORTRAN approximately five years ago. Originally, it was planned as the tool to succeed the server-based FAM (File Access Manager) to improve performance and functionality, as well as provide a general purpose Remote Procedure Call mechanism. At PRIMOS (operating system) revision 18, the file system was rewritten to use the newly created FAM II routines.

For the sake of clarity, future references to 'FAM I' will concern the old File Access Manager, obsolete at rev. 19.3, which performed all local and remote file access with a server. The name 'FAM II' refers to the current file system routines that replace FAM I and which use NPX rather than a server to effect remote access. 'NPX' refers to the set of routines that allow access to remote procedures; these are described in this document. NPX, the Network Process Extension, is so named as it is an extension to PRIMENET Level III and uses PRIMENET routines and functionality.

This is how NPX fits into PRIMENET:



Since its inception, NPX has undergone many changes. Most changes have been transparent to most users, while some have been quite radical. Further, many subsystems within Prime have converted to using NPX rather than IPCF routines, as NPX is considered less cumbersome to use and reasonably flexible.

When using NPX, it is important to know the PRIMOS revision on which an application is run. There are currently two distinct, incompatible calling interfaces to NPX; one for all NPX systems up to and including PRIMOS rev. 19.2, and a second for all subsequent revisions (19.3 and greater). Both of these interfaces are described in detail in later chapters of this document.

The interface was changed to accommodate enhancements to the network configurator functionality. The entire network is now allowed to be brought up and shut down independently of the CPU. It is easy to envision a scenario where a user running the old NPX allocates a slave, then the network is taken down, unknown to the user. A new configuration could be brought up with new node numbers; thus, when the user with the allocated slave tries to call a remote procedure, his 'known' node number may not exist or could be another node! Thus, the unfortunate user in this situation will not know anything is wrong unless he tries to make a call while the network is down.

The fix for this problem is the implementation of a system-wide unique slave identifier that is seen in the new NPX interface calls. This makes the NPX calls independent of the system's network configuration. Please see the subsequent chapter on those calls for further details.

The largest and most obvious application of NPX is the file system. The file system itself runs in one of two ways; locally or remote. When a file system operation is requested, the object of the operation, whether it be a file, directory, attach point, or something else, is searched for on the local machine. If it is not there, then it is searched for on the list of remote disks by using NPX.

For example, if the command 'attach blah>frump' is issued, the UFD blah is searched for on the disks of the local system. If it isn't found, then the disks of the remote systems are sequentially searched. This is done by making the same SRCH$$ calls that are made on the local system, only NPX is used to call them on each remote system in turn.

Note that to access a file on a remote system through NPX the disk on which it resides does not necessarily need to be added. However, both nodes must be able to recognize each other on the network for the NPX calls to be successful. Both nodes need only to have each other configured in their respective networks.

2 Performance of NPX vs. FAM I

NPX performance was measured some time ago based on comparison tests to FAM I. In summary, FAM II (NPX) is approximately twice as fast as FAM I in most tests. The tests included real time for File System operations, CPU time in PRWF$$, working set, and overall CPU time. It should be noted that this improvement in speed was made at the expense of the making and breaking of Virtual Circuits. Please refer to PE-TI-793, FAM II Performance (T. Taylor) for complete details.

3 Using NPX

Using NPX is fairly simple for the average user. For most applications, it is a matter of allocating (setting up) a slave process on the remote node, calling the desired remote routine(s) in succession, and releasing the slave.

In order to use NPX, several insert files must be included with programs to obtain values for declarations of constants. Use the following files:

```
SYSCOM>KEYS.INS.language       /* System Keys */
SYSCOM>ERRD.INS.language       /* Error Codes */
INTCOM*>NPXKEY.INS.language    /* NPX Codes   */
```

where language is currently only Fortran (FTN) or PLP. Also note that NPX routines are not in a library. In order to be used, they must be declared with the DYNT subcommand in BIND, or loaded with the binary image of a special miniature PMA program that DYNTs the NPX routines used. All user-accessible NPX routines are PRIMOS gates and can only be accessed in this way.

3.1 NPX Routines

Specifically, there are routines that accomplish the required tasks within NPX. Before any real work is done, a remote node name must be obtained. The node name is then checked; in the old calling interface, R$CVT is used to check the node name and convert it into a node number. In the new calling interface, the optional routine R$CKNT ensures that a given node name is valid. Then, the module R$ALOC performs slave allocation; in the old scheme using the node number, in the new scheme the node name itself is used and a slave identification is returned and is later passed to the other routines. Then, R$CALL (synchronous) or paired calls of R$BGIN and R$END (asynchronous) do the actual calling of the remote routine(s). Finally, R$RLS is used to release the slave.

Both the old and new calling sequences for these routines are outlined in detail in the subsequent chapters. There is also a special chapter that describes a suggested conversion procedure between the old and the new interfaces.

Finally, please note that there is a limit on one slave per user per remote node; that is, a user may have slaves on up to 16 remote nodes, but only one slave may be on each system for that user process.

3.2 User Restrictions

There are several things that NPX cannot or will not do for a user. Many go far beyond the original scope of intended NPX functionality, while others are overly burdensome to implement. Here are some of the things that NPX does not do:

1. The user and/or the subsystem in question may not use common blocks to communicate across the network. The subsystem may use common blocks to communicate between its own internal subroutines, but common blocks can not be used to communicate between distributed subsystems.

2. The NPX mechanism is intended to support up to 15 arguments in each remote call. Arguments are copied from the calling (master) process to the slave process where the target call is made. Return arguments are copied back to the caller in an analogous fashion. The maximum length of all arguments concatenated together is 8k bytes due to ring0 stack size. This is because the slaves copy their arguments into and out of the Ring 0 Stack before and after the procedure call to the target routine.

3. Each argument is either an input argument, an output argument, or both. Pointer or LOC() arguments are also supported, using some special rules that inform the NPX mechanism of the memory size of the referenced argument. INTEGER*2, INTEGER*4, char(*) aligned, char(*) varying aligned, pointer, and both scalar and aggregate data can be passed through the NPX mechanism. Each argument to a local subroutine maps to a triplet of arguments for NPX. The triplet includes the argument itself, its length, and keys describing its data type.

4. The return lengths of aggregate arguments can be specified by other arguments set as a result of the target call. For example, the specified length of the buffer passed as an argument to GPATH$ will affect the total length of the arguments list sent to a slave.

5. NPX does not support 'route-through' by calling R$CALL with a remote procedure name argument of R$CALL. The slave structure does not lend itself to an efficient route-through mechanism, and is not allowed.

3.3 Remote IDs

ACLs are used to restrict the scope of the slaves access on target systems. Typically, the slave inherits its master's user name, project(s) and node name. This gives control of remote resources to the administrator of the target system (the system that owns the resources) not the master's system administrator (the system that wants to consume or use them).

This brings to light the issue of naming spheres. A machine or group of machines is in the same naming sphere so specified in the network configuration. If two nodes are in different naming spheres, then when an attempt is made to use a slave on the remote node, a 'Slave ID Mismatch' or similar error occurs.

This is a security feature that prevents unauthorized access to system administrator specified nodes or groups of nodes. However, if access to the remote node is desired, an authorized person, with a login name and password on that node, can issue a command to permit access. This command is the Add_Remote_ID (ARID) command and is of the form:

```
ARID remote_id [password] -ON node [-PROJECT proj_id] [-PROMPT]
```

Note that the -PROMPT option was added at PRIMOS rev. 19.3.

When an ARID command is issued, all remote access made by the user issuing this command on the specified node uses the given remote id and its corresponding access rights. The ARID command may be issued when not explicitly required if a user needs greater access privileges on a remote node than is provided by default and knows a login name and password there. Please note that NPX (and all subsystems that use it) are the only objects that are affected by the Remote ID principle.

Associated with the ARID command is the List_Remote_ID (LRID) command, which lists the remote IDs that have been specified for all remote nodes. Finally, the Remove_Remote_ID (RRID) command removes a given remote ID and again allows the default remote ID, that being the user name, to be used for subsequent remote accesses. The Remove_Remote_ID command was added to PRIMOS at rev. 19.3. Note that logging out also removes all remote IDs, as well as logging out any slaves that may have been created. Further details on these commands can be found in the PRIMENET Guide.

4 The Calling Interfaces

4.1 The Old Synchronous NPX Interface

In all old-style NPX calls, an error return value is provided for identification of possible errors. In all examples, this code is represented by Rcode. This code generally is a PRIMOS standard return code and may be interpreted by calling ERRPR$ or IOA$ER to display the message associated with a given code. Possible values of Rcode are listed with each routine. The actual numerical value of the codes are found in the insert file ERRD.INS.language. Exceptions to the use of standard return codes are as noted in individual routines.

R$CVT:

Obtains a node number from an ASCII node name.

R$CVT(Nodename, Node_Namlen, Rcode) /* Function returns I*2 */

Nodename (char(*),input) The name of the node (e.g., 'ENB').

Node_Namlen (bin,output) Length of the nodename in bytes.
The octal value 100000 (the largest negative number) is returned if this node is unknown.

R$WHER:

Finds the node on which an object that you wish to access using NPX is physically located.

R$WHER(Key, Obj_Name, Obj_Num, Rcode) /* Function returns I*2 */

Key tells R$WHER what kind of object is being located.

Current KEYS are:

K$NAME returns node number of the pathname specified in Obj_Name

K$UNIT return the node number of the file unit specified in Obj_Num

Obj_Name (char(128) var,input) ASCII name of an object to be located (e.g. file name)

Obj_Num (bin,input) Numeric object data (e.g. file unit)

R$ALOC:

This routine allocates a slave on the node represented by the node number returned from R$CVT.

Since multiple subsystems within a single process may use the same slave, a global per slave allocation count is maintained by all subsystems sharing a slave. To allocate a slave for a specific node, R$ALOC merely increments the per node counter, and no guarantee is made as to whether or not a slave is actually available on a target node. Calls to R$ALOC and R$RLS must be paired in a manner analogous to quit inhibit calls in order to maintain the counter correctly. At least one R$ALOC call must be made for a node before any R$CALL or R$BGIN - R$END can be made.

R$ALOC(Nodenum, Rcode)

Nodenum (bin,input) Node number returned from R$CVT.

R$RLS:

This call releases the slave specified by the given Nodenum. The slave returns to the pool of available slaves.

R$RLS(Nodenum, Rcode)

Nodenum (bin,input) Node number returned from R$CVT.

R$CALL:

The is the actual Remote Procedure Call mechanism. The node number, obtained from R$CVT, is passed along with the ASCII name and arguments of the desired routine. Each passed argument, whether input, output, or I/O, has associated with it a length and one or more keys that describe the data and its direction.

```
R$CALL(Rkey, Nodenumber,  Procname, Procnamlen, Rcode,
          Arg1, Arg1len, Arg1type,
          Arg2, Arg2len, Arg2type,
          ...,
          Argn, Argnlen, Argntype)
```

Rkey (I*2) key to R$CALL

Keys To R$CALL

K$WFRC+n The slave will wait n times 15 seconds for a reconnect dialog if the Virtual Circuit is cleared by network failure during a remote request. n can be 0 to 127. This key gets reset on every request.

K$FUNC This is a function call, please return the L register value set by the target routine to the caller of NPX. Function values not returned in the A or L register are not returnable though NPX.

K$RTRY Retry slave acquisition if none are initially available. Quit will cause an exit and the message "no remote slaves available". This is ignored after the first request.

Nodenumber(bin,input) Node number on which to activate a slave. (See R$CVT to obtain node number from node name.)

Procname (char(*),input) ASCII name of the subroutine being called. Under current search rules this must be dynamically linkable or in a shared library.

Procnamlen(bin,input) Chars in target subroutine name.

Rcode (bin,output) Return code for the "remoteness" of this call, not the code from the target subroutine

Argn (any type,input) Nth argument to the target subroutine

Argnlen (bin,input) Length of the Nth argument in its basic units (ie., bytes, words, double words or quadwords)

Argntype (bin,input) Bits which describe the data type and direction of the nth argument. All keys are additive.

Type

K$FB15,K$I2	Argument is fixed bin(15)
K$FB31,K$I4	Argument is fixed bin(31)
K$FL	Argument is float bin (2 halfwords)
K$DFL	Argument is double floating bin (4 halfwords)
K$CHAR	Argument is a fixed-length char(*) aligned.
K$VCHR	Argument is a PL1 char(*) var aligned.

Direction

K$IN	Input only argument...the target subroutine uses this argument as an input parameter.
K$OUT	Return or output argument. The subroutine sets the value of this argument for use by the caller. K$IN and K$OUT can be summed to specify that an argument is both passed to and returned from the target subroutine.
K$REF + m	Refer to argument number m after the target call is made to obtain the length to return. The length will be an I*2 number representing n units of the data type of the referred to argument. ARGnLEN should be the maximum possible length of the pointed to data structure (this is used for the slave's allocation temporary storage), e.g., if the referred to argument is K$CHAR, then the refer value will be in bytes, if the referred to argument was K$FB15, then the refer length would be understood to be words. K$REF requires that K$OUT be used.

Indirection

KPTR,KLOC	Argument is a pointer or LOC() variable. In this case, argument length is the length of the structure pointed to in 16 bit words. If the argument is also REF, the arguments return length will be in its own data type. Only K$I2 works now.

Possible return codes in RCODE

0	Operation complete.
E$RLDN	Slave's system or link has gone down since the users' last remote request. This request has not been started. If K$WFRC is used, the user will wait n minutes in the abortable reconnect state before returning this code.
E$FONC	Slaves system or link was lost in mid request. The link could not be reestablished in the allowed time. The target call may or may not have been executed.
E$UNOP	Slave was lost due to a cold start or force logout or slave timeout after virtual circuit clearing) since last request. The VC was reestablished but the slave no longer existed.
E$FABT	Slave error.
E$NRIT	Remote subroutine linkage not permitted.
E$NSLA	No slaves available in the time allowed to get one.
E$IREM	No NPX calls allowed to that system from this system.
E$NETE	An uncorrected Low level network error has been detected by R$CALL.
E$PNTF	Target procedure not found.

For example, a PRWF$$ call to read 1436 words on file unit 35 on node BLAH looks like this:

```
Rnode = R$CVT ('BLAH', 3, Code)        /* Get the node number */
CALL R$ALOC (Rnode, Code)              /* Allocate a slave */
   .
   .
CALL R$CALL (0,Rnode,'PRWF$$',6,Rcode,
+             K$READ,1,K$FB15+K$IN,
+             35,1,K$FB15+K$IN,
+             LOC(Buf),1436,K$PTR+K$FB15+K$REF+6,
+             0,1,K$FB15+K$IN,
+             0010240,1,K$FB31+K$IN,
+             Numred,1,K$FB15+K$OUT,
+             Code,1,K$FB15+K$OUT)
   .
   .
CALL R$RLS (Rnode, Code)               /* Release the slave */
```

Note:

Notice that the third argument triplet uses the K$REF key so that only the number of storage units (FB15) actually read will be returned in BUF.

4.2 The New Synchronous NPX Interface

When a user uses NPX to go to a remote system, a virtual circuit is established between the local node and the remote node, and a slave is acquired in the remote system which will serve the master until it is released. If the master process has more than one subsystem (or program) using NPX to the same remote node, this one slave serves all subsystems in the master process. In the case where the virtual circuit is cleared unexpectedly, this slave goes away. If one of the subsystems reestablishes the virtual circuit to the same remote node, a new slave is acquired. Other subsystems in the master process must be notified of this fact when they make subsequent NPX calls.

The implementation of a unique number to a slave (SLAVID) will serve this purpose. It works as follows:

After the initial NPX call (R$ALOC routine), SLAVID is returned to the caller. The caller stores this value and submits it in subsequent NPX calls. NPX checks the validity of this number. If the submitted SLAVID does not match the current slave's ID number, an 'INVALID SLAVE ID' error code is returned.

As the result of the implementation of the new network configurator, the node number has been eliminated from the user interface as of revision 19.3. The SLAVID replaces the node number in most of the NPX modules except R$ALOC. In the R$ALOC routine, Node_Name replaces the node number, and an extra argument Slavid is returned to the caller for use in subsequent calls.

Also, some routines have been removed or replaced in the new interface. R$CVT, which obtained a node number from a given node name, has been rendered obsolete by the slave ID, no longer exists. Also, R$WHER has been removed from the NPX interface, and has been replaced by the OS routine ISREM$. Please refer to the OS/File System group for further information on this routine.

NOTE:

For all NPX routines but one, R$ALOC, the SLAVID is submitted by the user. User must not alter the value of SLAVID after obtaining it from R$ALOC.

R$ALOC:

Allocate a slave on the node specified by the ASCII Node_Name and return a Slavid for use in subsequent NPX calls.

```
R$ALOC(Node_Name, Slavid, Rcode)
```

where:

Node_Name	(char(32) var,input) ASCII name for the target node.
Slavid	(char(8),output) A unique number assigned to the slave. It is a return argument. Caller must store this value, and submit it unaltered in subsequent R$CALL, R$BGIN, R$END and R$ALO1 calls.
Rcode	(bin,output) Returned error code.

The possible values of Rcode are:

0:	Operation complete without an error.
E$MSLV:	Exceeds the maximum number of slaves allowed per user
E$NETE:	Network problem.
E$RLDN:	Remote line is down.
E$NSLA:	No NPX slave available in the target node.
E$BPAR:	User's arguments are bad.
E$RSNU:	Remote system is not up (but on its way up).

NOTE:

The virtual circuit between the local node and the target node is established in R$ALOC when it is called for the first time.

R$ALO1:

This routine is a subroutine similar to R$ALOC. R$ALOC must have previously been called and the Slavid retained. That Slavid is passed here and a slave with that Slavid is created. An invalid Slavid can be detected here, but not in R$ALOC (see note below).

```
R$ALO1(Slavid, Rcode)
```

where:

Slavid	(char(8),input) an unique number assigned to the slave. The caller must submit this input value.
Rcode	(bin,output) Returned error code.

The possible values of Rcode are:

0:	Operation complete without an error.
E$WSLV:	SLAVID is invalid.
E$RLDN:	Remote line is down.
E$VCGC:	The virtual circuit got cleared.

Note: The Distinction Between R$ALOC and R$ALO1

Any process calling NPX modules must call R$ALOC before calling R$CALL, R$BGIN, R$END or R$RLS. In the first R$ALOC call, a virtual circuit to the desired node will be established and a unique slave ID number will be returned. For subsequent allocations, R$ALO1 should be used. R$ALO1 expects the same slave ID that was returned from the original R$ALOC call and using R$ALO1 when re-allocating a slave eliminates a great deal of overhead and can detect if the wrong slave ID was used. Pre-rev-19.3 programs that do not have R$ALO1 available can re-use R$ALOC, at the cost of some efficiency.

The general recommendation concerning R$ALOC and R$ALO1 is as follows: Use R$ALOC when first calling any NPX routines. This returns a Slave ID that is unique system-wide and allocates a slave for that user. Any subsequent calls that would logically require another R$ALOC call should use R$ALO1 for the reasons outlined above. This assumes the user has saved the Slave ID, irrespective of the users' knowledge of the node name. However, there may be occasions when the Slave ID is not known, but the node name is. In this case, the user has no choice but to call R$ALOC.

Please note that R$ALOC and R$ALO1 calls may be nested, provided that an R$ALOC is called first. It is also permitted to call them in sequential blocks, again provided an R$ALOC is called first. Again, each call of either R$ALOC or R$ALO1 must have a corresponding R$END. Both of the following examples are allowed:

```
R$ALOC
   R$ALO1
      R$CALLs
   R$END
R$END
```

```
R$ALOC
   R$CALLs
R$END
R$ALO1
   R$CALLs
R$END
```

R$CALL:

The is the actual Remote Procedure Call mechanism. The Slavid, obtained from R$ALOC, is passed along with the ASCII name and arguments of the desired routine. Each passed argument, whether input, output, or I/O, has associated with it a length and one or more keys that describe the data type and its direction.

```
R$CALL(Rkey, Slavid, Proc_Name, Proc_Namlen, Rcode,
       arg1, arg1len, arg1type,
       arg2, arg2len, arg2type,
       ...,
       argn, argnlen, argntype)
```

where:

Rkey	(bin,input) See next page for NPX keys.

Keys To R$CALL

K$WFRC+n	The slave will wait n times 15 seconds for a reconnect dialog if the Virtual Circuit is cleared by network failure during a remote request. n can be 0 to 127. This key gets reset on every request.
K$FUNC	This is a function call, please return the L register value set by the target routine to the caller of NPX. Function values not returned in the A or L register are not returnable though NPX.
K$RTRY	Retry slave acquisition if none are initially available. Quit will cause an exit and the message "no remote slaves available". This is ignored after the first request.

Slavid	(char(8),input) Slave's unique id.
Proc_Name	(char(*),input) ASCII name of the target subroutine. Under current search rules this must be dynamically linkable; either a gate or in a shared library.
Proc_Namlen	(bin,input) Chars in target subroutine name.
Rcode	(bin,output) Returned error code from the R$CALL, not from target subroutine.

The possible values of Rcode are:

0:	Operation complete without an error.
E$WSLV:	SLAVID is invalid.
E$BPAR:	User's arguments are bad.
E$BCFG:	Network configuration mismatched.
E$VCGC:	The virtual circuit got cleared.

ARGn	(any type,input) nth subroutine argument.
ARGnLEN	(bin,input) The length of the Nth argument in its basic unit (bytes, words, chars, etc.).
ARGnTYPE	(bin,input) Bits which describe the data type and direction of this argument. All keys are additive.

Argument Keys

	Type
K$FB15,K$I2	Argument is fixed bin(15)
K$FB31,K$I4	Argument is fixed bin(31)
K$FL	Argument is float bin (2 halfwords)
K$DFL	Argument is double floating bin (4 halfwords)
K$CHAR	Argument is a fixed-length char(*) aligned.
K$VCHR	Argument is a PL1 char(*) var aligned.

	Direction
K$IN	Input only argument...the target subroutine uses this argument as an input parameter.
K$OUT	Return or output argument. The subroutine sets the value of this argument for use by the caller. K$IN and K$OUT can be summed to specify that an argument is both passed to and returned from the target subroutine.
K$REF + m	Refer to argument number m after the target call is made to obtain the length to return. The length will be an I*2 number representing n units of the data type of the referred to argument. ARGnLEN should be the maximum possible length of the pointed to data structure (this is used for the slave's allocation temporary storage), e.g., if the referred to argument is K$CHAR, then the refer value will be in bytes, if the referred to argument was K$FB15, then the refer length would be understood to be words. K$REF requires that K$OUT be used.

	Indirection
KPTR,KLOC	Argument is a pointer or LOC() variable. In this case, argument length is the length of the structure pointed to in 16 bit words. If the argument is also REF, the arguments return length will be in its own data type. Only K$I2 works now.

R$RLS:

This call releases the slave specified by the given Slavid. All R$RLS calls should have a corresponding R$ALOC call. The slave returns to the pool of available slaves.

```
R$RLS(Slavid, Rcode)
```

where:

Slavid	(char(8),input) the unique number assigned to the slave. Caller must submit this value.
Rcode	(bin,output) Returned error code.

The possible values of Rcode are: ~0: Operation complete without an error. ~E$WSLV: The slavid is invalid. ~E$BVCC: Problem in clearing the virtual circuit. ~E$VCGC: The virtual circuit got cleared.

R$CKNT:

This call checks the validity of a Node_Name and for its existence in the network. This routine is valid only in the new NPX interface.

```
R$CKNT(Node_Name, Rcode)
```

where:

Node_Name	(char(32) var,input) A network node name.
Rcode	(bin,output) the returned error code.

The possible values of Rcode are: ~0: Node_Name is valid and exists in the network. ~E$UNOD: The Node_Name does not exist in the network.

4.3 The Asynchronous NPX Interface

Asynchronous remote procedure call has two operations, R$BGIN and R$END, to complete a remote procedure call. This section describes the use of these two routines. They were implemented in Revision 19.1 with the use of node number. In this documentation, they are replaced for Revision 19.3 in which SLAVID replaces node number.

Asynchronous NPX allows a remote procedure call to be broken into 2 separate operations. They are a begin remote procedure call (R$BGIN) and an end remote procedure call (R$END). In effect, the first call sends the arguments to the slave, starts it running, then returns. R$END is called to check the status of the outstanding call and optionally pick up the return arguments from the slave if complete. This has the effect of being 'asynchronous' in that a procedure call will not 'finish' unless queried by the user.

R$BGIN/R$END pairs functionally replace R$CALLs in NPX operation. When R$END is called, as shown below, an argument is passed that tells NPX how long to wait before testing for return. If this is made infinity by passing V$INFN, then the pair behaves exactly like a single R$CALL.

Currently each user is allowed one outstanding remote procedure call per node. An attempt to queue R$BGIN calls for the same node will result in a return code of E$APND (Already Pending). Also, note that there is currently no method for prematurely terminating an operation begun with an R$BGIN by the user. This functionality cannot be performed, as once a routine call is made, it is impossible to abort it.

R$BGIN:

This routine, when coupled with a corresponding subsequent R$END call, will asynchronously perform a remote procedure call. Please see the description of asynchronous NPX calls for more details.

```
R$BGIN(Rkey, Slavid, Proc_Name, Proc_Namlen,
       Buf, Buflen, Rcode,
       arg1, arg1len, arg1type,
       arg2, arg2len, arg2type,
       .......,
       argn, argnlen, argntype)
```

where:

Rkey	(bin,input) Key to NPX.
Slavid	(char(8),input) slave's unique id number. Caller must submit this value.
Proc_Name	(char(*),input) ASCII name of the target subroutine. Under current search rules this must be dynamically linkable; either a gate or in a shared library.
Proc_Namlen	(bin,input) Chars in the subroutine name.
Buf	(bin(Buflen),input) A scratch buffer for NPX supplied by the caller (see NOTE below).
Buflen	(bin,input) Length of Buf in words. See NOTE.
Rcode	(bin,output) Returned error code from R$BGIN, not from target subroutine.

The possible values of Rcode are:

0:	The request has been transmitted out of the caller's user space and no errors are in the NPX specific arguments, but no assurances about the arguments to the target routine(proc_name) are made.
E$APND:	Asynchronous procedure still pending.
E$BCFG:	Network configuration mismatched.
E$BFTS:	Buffer too small.
E$BPAR:	User's arguments are bad.
E$NBUF:	No buffer space.
E$NENB:	Remote node not enabled.
E$VCGC:	The virtual circuit got cleared.
E$WSLV:	SLAVID is invalid.

ARGn	(any type,input) nth argument to the subroutine.
ARGnLEN	(bin,input) The length of the Nth argument in its basic unit (bytes, words, chars, etc.).
ARGnTYPE	(bin,input) Bits which describe the data type and direction of the nth argument. All keys are additive.

R$END:

This routine checks the status of a single outstanding remote procedure call. If a call is complete, the return arguments specified in the R$BGIN call are filled in and returned.

```
R$END(Rkey, Slavid, Buf, Time, Rcode)
```

where:

Rkey	(bin,input) Any R$CALL key.
Slavid	(Char(8),input) The unique ID of the slave.
Buf	(bin(Buflen),input) A scratch buffer for NPX supplied by the caller (see R$BGIN for Buf).
Time	(bin,input) Time to wait in tenths of seconds. Two special values are: V$INFN and V$NONE, infinite and no time respectively. If an operation completes before the timer runs out, R$END returns with code = 0. If the timer expires without the call completing then E$SPND (Still Pending) is returned.
Rcode	(bin,output) Returned code.

The possible values of Rcode are:

0:	Call complete for the target node. Results, if any, have been placed in the R$BGIN K$OUT arguments.
E$SPND:	Still pending. The remote node is still working on the target call or data is in transit in the network.
E$VCGC:	The virtual circuit got cleared.
E$WSLV:	SLAVID is invalid.

NOTE:

The BUF argument in R$BGIN is a scratch buffer for NPX which must be supplied by the caller. This buffer must exist for the duration of the remote procedure call. Since BUF is allocated from the caller's storage, the caller must not return or release that storage before the matching R$END is called.

The BUFLEN argument is the length of buffer BUF. Its size in words is given by the formula:

```
BUFSIZE = 100 + MAX (length of concatenated_input_args,
          length of concatenated_output_args + 100).
```

For example, if the concatenated length of the input arguments of a procedure call were 600 words, and the length of the output arguments were 400 words, then a buffer that contains at least 700 words must be used. NPX checks the size of the user supplied buffer against the the size of the buffer that it calculates is necessary. An error of E$BFTS is returned in Rcode if the buffer is too small.

There is an upper limit to the size of BUF. It must not exceed 4K words. If the input or the output argument exceeds it, an error code E$NBUF (no buffer space) will be returned. If the tally of all input arguments or the tally of all output arguments (whichever is larger) exceeds this limit, it will result in the same error code.

5 Conversion Between Old and New Interfaces

My thanks go to J. Craig Burley of technical publications for expounding this technique in a December 1983 memo.

This is a discipline for changing a product using NPX so that it will support both old and new NPX versions of PRIMOS. For most of your program, the only change is that the node number argument of NPX calls is a slave id when using new NPX. Whereas the old node number argument is INTEGER*2 or FIXED BIN(15), the new slave id is INTEGER*2 (2) or CHAR(8), i.e. 4 halfwords (one halfword=16 bits).

The most difficult part of the change is where you actually allocate the slave. Here, instead of converting a node name (like ENB) to a node number (like 513) using R$CVT and then calling R$ALOC with the node number, you just call R$ALOC with the node name (ENB) and it returns a slave id. You use this slave id in subsequent calls to NPX, up until the point where you call R$RLS with the slave id. (If you want to do subsequent allocations of the slave after the initial R$ALOC call and before the final R$RLS calls, you match up pairs of R$ALO1/R$RLS calls, where R$ALO1 is a new subroutine that allocates the slave again using the same slave id, and hence does not take a node name).

One problem is that whereas R$CVT took the node name in a CHAR(6) scalar, new R$ALOC uses a CHAR(6) VAR. This is something of a minor headache in FTN programs.

Because R$CVT isn't used in new NPX, the subroutine does not exist. This fact is used, as shown below, to determine whether a program is running under new or old NPX -- by calling CKDYN$ (check whether dynamic link is snapable) with the name R$CVT to see if R$CVT existed or not.

In the following discipline, the node number is called NODNUM, and certain assumptions are made. The most crucial assumption is that the RCVT/RALOC calls in your existing code must be close together. If instead your program calls R$CVT, stores the node number in some area, and later on your program actually uses R$ALOC, then you have to change the node number storage area to store a node name OR node number instead. But the stuff below will still probably help you get an idea of how to do it even if your program works this way.

This example is in FTN, but the translation to PLP is very straightforward, except PLP programmers will have to use a based overlay to do the EQUIVALENCE functionality (used to map the old node number and new slave id into the same place in memory so non-R$ALOC/R$CVT calls don't need to change).

1. Change the declaration for the node number (NODNUM) from INTEGER*2 to INTEGER*4 NODNUM(2). Do this any place the node number is used.

2. In the slave allocation sequence, change the following sequence of pseudo code:

```
          INTEGER*2 NODNUM
          INTEGER*2 R$CVT
              .
              .
          NODNUM=R$CVT(nodename,nodelen)
          IF (NODNUM.EQ.:100000) GO TO error
          CALL R$ALOC(NODNUM,CODE)  /* CODE is optional before 19.
3.
          IF (CODE.NE.0) GO TO error
```

to this new sequence of pseudo code:

```
          INTEGER*4 NODNUM(2) /* Node number or slave id.
          INTEGER*2 CVTNUM    /* For node number from R$CVT.
          INTEGER*2 R$CVT     /* (R$CVT is still an I*2 function).
          INTEGER*2 CVTNAM(4) /* Name of R$CVT subroutine.
          INTEGER*2 NEWNPX    /* -1=???; 0=old NPX; 1=new NPX
          EQUIVALENCE (CVTNUM,NODNUM)  /* "nodnum"="slavid"
C
          DATA NEWNPX/-1/     /* Don't know which yet.
          DATA CVTNAM/5,'R$CVT'/
                  .
                  .
             IF (NEWNPX.NE.-1) GO TO haveit  /* Know which NPX?
             CALL CKDYN$(CVTNAM,CODE)  /* See if R$CVT exists.
             IF (CODE.EQ.0) NEWNPX=0   /* If yes, old NPX.
             IF (CODE.NE.0) NEWNPX=1   /* Else, new NPX.
C
haveit:   IF (NEWNPX.EQ.1) GO TO newnpx  /* New npx?
C
          CVTNUM=R$CVT(nodename,nodelen)  /* Get node number.
          IF (CVTNUM.EQ.:100000) GO TO error
          CODE=0  /* Some old revs don't zero CODE in R$ALOC.
          CALL R$ALOC(CVTNUM,CODE)    /* Old NPX alloc-slave.
          IF (CODE.NE.0) GO TO error
          GO TO havslv                /* Now have the slave!
newnpx:   CODE=0                      /* Just to be safe.
          CALL R$ALOC(nodename,NODNUM,CODE)
          IF (CODE.NE.0) GO TO error
C
havslv:   CONTINUE                        /* Allocation done.
```

Note: Whereas <nodenam>,<nodelen> are the character string/length in the old FTN style (as used in SRCH$$, TSRC$$, and so on), <nodenamlen> is a varying character string, PL/1 style (as used in SRSFX$ for example), where the first 16-bit halfword is the length of the string, and the string itself starts in the second 16-bit halfword of <nodenamlen>.

3. If you might run on a pre-Rev. 19.2 system, you will need to load the CKDYN$ subroutine in with your product, as it only came into existence at Rev. 19.2. Get it from the master disk 19.2, V1 partition, in PRIMOS>R3S>CKDYN$.PLP. Then change the $INSERT line in CKDYN$ to use SYSCOM>ERRD.INS.PL1 rather than *>INSERT>ERRD.INS.PLP.