Santa Teresa
Laboratory
San Jose, CA

TR

CALLS TO APL2

by Michael T. Wheatley

January 1991

TR 03.390

Calls to APL2

Document Number TR 03.390

January 2, 1991.

Michael T. Wheatley

International Business Machines Corporation
Programming Systems
Santa Teresa Laboratory
San Jose, California, USA

Abstract

This document describes facilities which have been added to APL2 to allow applications written in languages other than APL to issue calls to APL2 -- to execute APL expressions, invoke API functions, reference or specify APL variables, or to enter the APL2 interactive environment. Such applications can be invoked independently of APL2, or they can be invoked, using facilities provided, from an active APL2 environment.

Contents

Calls to APL2

Introduction

This document describes facilities which have been added to APL2 to allow applications written in languages other than APL to issue calls to APL2. Such applications can be invoked independently of APL2, or they can be invoked, using facilities provided, from an active APL environment.

An interface routine, called APL2PI (APL2 Program Interface), provides capabilities through which:

- APL2 can be initialized;
- APL2 can be terminated;
- APL functions can be executed;
- APL variables can be referenced or specified;
- APL expressions can be executed;
- control can be passed to the interactive APL session.

A companion APL external function, called APL2PIE, is provided through which:

- non-APL applications can be invoked from an active APL2 environment. Applications so invoked can subsequently make calls to APL2 using the APL2PI interface;
- a request to terminate can be passed to non-APL applications from the active APL2 environment;
- control can be returned to a non-APL application (that previously invoked APL2 or returned control to the active APL environment);
- service requests can be passed from executing APL functions to any of the currently active non-APL applications.

The APL2PI routine provides a relatively high level of interface designed to be imbedded as a callable service in programs written in high level languages such as FORTRAN, PL/I, C, or COBOL. APL2PI is a reentrant routine that can be link edited with application programs or packaged as a separate load module which is dynamically loaded before being called. In the VM/CMS environment, APL2PI may also be loaded as a CMS nucleus extension.

The facilities described in this document are available in current levels of the APL2 and APL2/AE program products (5668-899, 5688-003) Version 1 Release 3.0, with the following PTF's and their prerequisites installed:

APAR	VM/CMS PTF	MVS/TSO PTF
PL54185	UL69734	UL69735
PL57346	UL69736	-
PL57347	UL69737	UL69738,UL69739
PL58236	-	UL69740
PL53174	UL71031	UL71032,UL71033
PL63437	UL74734	UL74735,UL74736
PL68969	UL79882	UL79883
PL68968	UL79924	

These facilities are not described in the documentation provided with the APL2 Release 3.0 or APL2 AE Release 3.0 program products. The information contained in this document has not been submitted to any formal IBM test and is presented on an "as is" basis without any warranty either expressed or implied. The use of this information is a customer responsibility and depends on the customer's ability to evaluate and integrate it into the customer's operational environment.

Overview

The APL2PI interface routine is designed so that it can easily be called from languages such as FORTRAN, COBOL, C, Pascal, PL/I, and Assembler. The form of such calls (using Assembler or FORTRAN syntax) begins with three consistent arguments:

```
CALL APL2PI(REQUEST,TOKEN,RC,....)
```

where:

REQUEST is a 4 character request identifier. The following requests are supported:

INIT	initialize APL2.
TERM	terminate APL2.
APLE	request execution of an APL expression.
APLS	request execution of an APL function.
APLF	request execution of an APL function. This request is more fully functioned than the APLS request, but it is not as simple to use.
APLV	reference or specify an APL variable.
APLX	return control to the APL environment.
APLP	enter or exit the namescope of an APL packaged workspace. Subsequent requests will be made in that namescope unless specifically directed elsewhere.

TOKEN is a token used by the APL2PI interface for correct and efficient operation. It is returned by an INIT call and should be provided on all subsequent calls.

RC is a 2 element return code returned by the APL2PI interface as the result of any call. A return code of 0 0 indicates success.

Most calls to APL2PI require additional arguments specific to the request. These will be described in subsequent sections.

The INIT, TERM, APLE, APLS, APLX, and APLP requests take relatively straightforward arguments that can be easily provided in most high level languages. The APLF and APLV requests, however, are designed to pass arguments to APL and receive results from APL in CDR format. CDR format is a data representation which allows efficient representation of APL arrays including general arrays. While CDR objects can be constructed in many languages that support data structures (e.g.: Assembler, C, PL/I, Pascal), it is a more difficult format to use than that used in the simpler service requests. The CDR format is described in detail in the APL2 Processor Interface Reference manual (SH20-9234).

In the remainder of this section, a simple example will be presented to illustrate the use of this interface. The example will be presented using FORTRAN because of its simple syntax and understandability. The program:

1. defines the necessary data items,
2. causes APL2 to be initialized by means of an INIT service request to APL2PI,
3. prompts the user to enter a set of 3 numbers,
4. computes their average by calling the APL function ***AVG*** in packaged workspace ***STATS***,
5. displays the result returned by the APL function,
6. causes APL2 to be shutdown by means of a TERM service request to APL2PI.

This example provides overly simplistic error handling facilities (at statement labelled 99), that may not be desirable in an operational environment. More complete examples are shown later in the paper.

```
      INTEGER*4 TOKEN,RC,LENGTH
      REAL*8 NUMBERS(3),RESULT
      TOKEN=0
      LENGTH=0
      CALL APL2PI('INIT',TOKEN,RC,'SAMPLE ',0,0,0)
      IF (RC .NE. 0) GOTO 99
      WRITE (6,*) 'Enter 3 numbers'
      READ (5,*) (NUMBERS(I),I=1,3)
      CALL APL2PI('APLS',TOKEN,RC,'STATS ','AVG ',LENGTH,' ',NUMBERS,RESULT)
      IF (RC .NE. 0) GOTO 99
      WRITE (6,*) 'The average is: ',RESULT
      CALL APL2PI('TERM',TOKEN,RC)
      IF (RC .NE. 0) GOTO 99
      RETURN
   99 WRITE (6,*) 'Unexpected error ',RC,' was returned from APL2PI'
      END
```

Figure 1. Sample FORTRAN Program

The ***AVG*** function invoked by this sample program differs from what a APL user might expect:

```
      ∇AVG ARGS;NUMBERS;RESULT
[1]   →(0∨.=3 11 ⎕NA 2 3⍴'PTAATP')/ERROR
[2]   NUMBERS←'E8 1 3' PTA ↑ARGS
[3]   RESULT←(+/NUMBERS)÷⍴NUMBERS
[4]   'E8 1 1' ATP RESULT (1↓ARGS)
[5]   →0
[6] ERROR:'UNEXPECTED ERROR' ⎕ES 9 9
      ∇
```

Figure 2. Sample APL Function

Lines 2 and 4 of this function use the APL external functions ***PTA*** and ***ATP*** to retrieve the argument ***NUMBERS*** passed by the FORTRAN program and to return ***RESULT*** to that program. These functions will be described in the section entitled "External Functions ATP and PTA" on page 16. Their use is required to accommodate the argument passing mechanisms and data types used in non-APL programs.

APL2PI Interface Calls

All calls to the APL2PI interface assume that the caller provides the necessary arguments "by reference" using standard OS linkage conventions. That is to say, it is assumed that the calling program passes control to APL2PI with the following general purpose registers set:

R1 contains the address of a standard OS parameter list, that is, a list of the addresses of the arguments passed on the call. The list is terminated by setting the high order bit in the last address in the list. Elided arguments imbedded in the list are specified as 0.

R13 contains the address of a standard 18-word OS save area which will be used by the APL2PI interface.

R14 contains the return address in the calling program

R15 contains the address of APL2PI.

Assembler (using the CALL macro), FORTRAN, COBOL, and PL/I use these conventions as the default on most calls. C and Pascal, however, often use an extension to these conventions in which a mixture of addresses and values may appear in the parameter list. From C programs, users must ensure that pointers to the arguments, rather than the values of the arguments, are passed. From Pascal programs, users should declare the arguments so that they will be passed by reference rather than by value. Additional information on this subject can normally be found in the Programmer's Guide manual for the language being used. Some additional information will be provided in later language specific sections of this document.

Many of the arguments required by APL2PI must be specified as character strings, sometimes terminated with a blank. Users should note that many languages, such as PL/I and Pascal, allow definition of variable length character strings which are prefixed with a length field. Such arguments are unacceptable to APL2PI because of the length prefix. Such languages typically provide alternate representations, such as fixed length strings, without the length prefix, that are acceptable to APL2PI. C null terminated character strings are acceptable to APL2PI. If an APL2PI argument must be terminated with a blank, a C null terminated string is acceptable if the character preceding the null is a blank. Again, additional information on this subject can typically be found in the language's Programmer's Guide manual.

Certain APL2PI arguments (such as SERVICE on the 'INIT' call and PATTERN on the 'APLV' call) are fullword fields containing addresses. Note that in these situations, the caller's parameter list must contain an address that points to the fullword containing the necessary address. All such arguments and result fields will be identified in the following descriptions with a phrase like "...a fullword field containing the address of...".

Each APL2PI call provides as its third argument a fullword field into which APL2PI will place a return code on completion of the call. All such return codes should be interpreted as a pair of halfwords. 0 0 indicates success; 0 x indicates an error originating in APL2PI, or an alternate successful result; x x indicates an error detected by APL2 (rather than in the APL2PI interface) and can be interpreted as an APL $\square ET$ value. A return code of 1 2 indicates an unexpected ***SYSTEM ERROR*** that may have been detected by either APL2PI or APL2.

Initialization Call

CALL APL2PI ('INIT',TOKEN,RC,NAME,TYPE,ANCHOR,SERVICE,LENGTH,PARMS)

This call provides an explicit mechanism by which APL2 can be invoked. If this call is not issued explicitly by the calling program, and if APL2 is not active, it will be issued implicitly by other calls to APL2PI (except 'TERM'). Since invocation of APL2 is often a lengthy process, the calling program may wish to issue this call explicitly some time before making use of other APL2PI services.

This call also provides the mechanism by which a non-APL application identifies itself to the APL2PI interface and optionally specifies service routine and anchor addresses. Thus it is recommended that this call be issued by all non-APL applications whether or not APL2 was previously activated.

The arguments to this call are:

'INIT' a required argument identifying this request.

TOKEN a fullword integer field into which the interface routine will place a token on successful completion of this call. This token should be retained and used on subsequent calls to provide optimal performance. This field should be zero when the 'INIT' call is issued, or the call will end with an error.

RC a fullword integer field into which the interface routine will place the return code on completion of the call. Return code of 0 0 indicates success. Other return codes are described below.

NAME a name used to identify the calling application program to APL2PI. This name may be subsequently used by the APL2PIE external function to direct requests to this application program. NAME must be 1 to 8 characters in length, and must be terminated with a blank. If this argument is elided or specified as a null or blank, the name ' ' (i.e., a blank name) will be assigned to the calling application program. This poses no problem if only one non-APL application uses the APL2PI interface, but may result in errors or unexpected results if more than one non-APL application is activated. A non-blank name is recommended.

TYPE a fullword integer identifying the type of service routine indicated by the SERVICE argument. A value of 0 means that no service routine is provided: the ANCHOR and SERVICE parameters will be ignored. A value of 1 indicates that the service routine expects its argument and produces its result in non-CDR form. A value of 2 indicates that the service routine expects its argument and produces its result in CDR form. Additional details on service routine arguments and results are presented in section "External Functions APL2PI and APL2PIE" on page 18.

ANCHOR a fullword token passed from the non-APL application. This token will be returned to the non-APL application on every service routine call. Note that updates to this token made during a call to the service routine will not be retained -- the original value of this token will be passed on all service routine calls.

SERVICE a fullword containing the address of a routine in the calling application to which service requests can be directed with an ***APL2PIE*** **3** call from the APL2 environment. If this argument is elided or specified as 0, or if TYPE is specified as 0, ***APL2PIE*** **3** requests will be denied for this application.

LENGTH a fullword integer field specifying the length in bytes of the PARMS argument. If this arguments is elided or specified with a value of 0, the PARMS argument is ignored.

PARMS a character string specifying APL2 invocation parameters. This argument is optional, but it must be provided if the LENGTH argument is specified as non-zero.

When an 'INIT' call is issued, if APL2 is not already active, APL2PI will append any invocation parameters provided on the call to the APL2 invocation command provided in AP2XAPIC CSECT (if AP2XAPIC is link edited with APL2PI) or to the default APL2 invocation command:

```
APL2 QUIET RUN(APL2PI)
```

If the resulting invocation options cause an APL function other than ***APL2PI*** to be invoked, that function is expected to invoke the ***APL2PI*** external function to cause control to be returned to the APL2PI interface routine on completion of APL2 initialization.

If the 'INIT' call is issued when APL2 is already active (i.e.: from a non-APL application invoked via the ***APL2PIE*** external function), a return code 0 1 ('APL already initialized') will be returned.

Termination Call

CALL APL2PI ('TERM',TOKEN,RC)

This call requests termination of APL2. It is effective only when issued by the non-APL application from which APL2 was invoked. If issued from a non-APL application which did not cause APL2 invocation (i.e.: one which was invoked by APL2 using the ***APL2PIE*** external function), it is nilpotent and returns a return code of 0 10 (invalid request).

If APL2 was invoked by a non-APL application, that application must issue the TERM call before its own termination. Failure to do so may cause abnormal termination of APL2, the APL2PI interface and possibly the non-APL application (and possibly even CMS in a VM/CMS environment).

The arguments to this call are:

'TERM' a required argument indicating that APL2 is to be terminated.

TOKEN a fullword integer containing the token returned on the 'INIT' call. If this token is not provided (i.e.: specified as zero), the call will require more CPU time to execute. On completion of the 'TERM' call, this field will be set to zero.

RC a fullword integer field into which the interface routine will place the return code on completion of the call. Return code of 0 0 indicates success. Other return codes are described below.

Execute an APL Function

CALL APL2PI ('APLS',TOKEN,RC,PKGWS,FN,RLENGTH,RESULT,ARG1,ARG2....)

This is one of two calls provided to request execution of an APL function. This call is designed to be easily used in high level language programs (such as FORTRAN or COBOL).

The function specified may reside in a packaged workspace and is called monadically if arguments (ARG1, ARG2, ...) are specified, or niladically if they are not. Arguments, if any, are passed to the function as a vector of fullword integers which represent the addresses of the argument data. The APL function is expected to use ***PTA*** to access the argument data, and ***ATP*** to update it. ***PTA*** and ***ATP*** are APL external functions provided with APL2. They are described in detail in the section entitled "External Functions ATP and PTA" on page 16.

The arguments to this call are:

'APLS' a required argument indicating that an APL function is to be called.

TOKEN a fullword integer containing the token returned on the 'INIT' call. If this token is not provided (i.e.: specified as zero), the call will require more CPU time to execute.

RC a fullword integer field into which APL2PI will place the return code on completion of the call. Return code of 0 0 indicates success. Other return codes are described below.

PKGWS the name of the packaged workspace in which the specified function is to be found and optionally a surrogate name for the function. If this argument is provided,

```
'PKGWS' 11 ⎕NA 'FN'
```

or

```
'PKGWS' 11 ⎕NA 'FN SURROGATE'
```

will be executed before the specified function is called. If this argument is not provided or is coded with an initial blank, no ⎕NA will be issued before calling the function. Thus, if the function exists in a packaged workspace, the first call to it must provide the PKGWS argument, but subsequent calls do not. The PKGWS argument is a character string which is expected to be terminated with a blank, e.g.: `'MYLIB.MYWS '`. If the surrogate name is specified it must be prefixed with a colon, viz:
`'MYLIB.MYWS:SURROGATE '`.

FN the name of the function to be called. This argument is a character string which is expected to be terminated with a blank. It is used as the right argument to ⎕NA if the PKGWS argument is coded and is then used as the name of the function to be called.

RLENGTH a fullword integer field specifying the length of the result field. On completion of this call. this field will be updated to contain the actual length of the result or error message produced (which may be shorter, the same size, or longer than the result field). If no explicit result or error message is produced, this field will be set to -1.

Note that this field is normally updated as a result of this call. It therefore should not be coded as a constant on calls from high level languages for to do so could result in the constant being modified, which in turn could result in subsequent errors in the calling program.

RESULT a field into which the explicit result of the function (if any) will be placed. If the result produced is larger than the length of this field (as specified by RLENGTH), only the first RLENGTH bytes of the result will be placed into the RESULT field and RLENGTH will be updated to reflect the actual total result length. The result is placed in this field as an unmodified byte string in left list order, i.e., as if it had been produced by the expression:

RESULT←(PFA RESULT) ATR RESULT

If an error results from the execution of the specified expression, the error message (*∈⎕EM*) will be placed in the RESULT field and its length in the RLENGTH field. An error message is not produced in all situations. In general, a message will not be produced if the error is detected before execution of the specified function has begun. In such situations, the RESULT field will not be updated.

ARG1,ARG2,... the arguments to the function. The specified function will be passed a vector of integers representing the addresses of these arguments and is expected to use ***PTA*** to access them and ***ATP*** to update them. If no arguments are coded in the call, the specified function will be called niladically. A maximum of 64 arguments are supported.

Note that this call allows an explicit result produced by the APL function to be passed back to the calling routine. The calling routine, however, must anticipate the size of this field in advance, allocate storage for it and pass it to APL2PI as the RESULT argument. The calling routine cannot control the type, structure, or shape of the data returned, nor can it control whether an explicit result or error is returned. In many situations it may be simpler to pass output or input/output arguments to the APL function and structure that function to return its results by updating one or more of the arguments using the APL external function ***ATP***. This approach allows the RESULT field to be used for the return of error information only.

If it is deemed desirable to produce an explicit result in the APL function called, that function can control the data type of the explicit result returned through the use of the external function ***ATR***, viz:

```
LENGTH=8
RESULT=0.0
CALL APL2PI('APLS',TOKEN,RC,'STATS ','AVG ',LENGTH,RESULT,NUMBERS)


     ∇RESULT←AVG ARG;NUMBERS
[1]   →(0∨.=3 11 ⎕NA 2 3⍴'ATRPTA')/ERROR
[2]   NUMBERS←'E8 1 3' PTA ARG
[3]   RESULT←(+/NUMBERS)↑⍴NUMBERS
[4]   RESULT←'E8 1 1' ATR RESULT
[5]   →0
[6] ERROR:'UNEXPECTED ERROR' ⎕ES 9 9
     ∇
```

Figure 3. Sample 'APLS' Call

Execute an APL Expression

CALL APL2PI ('APLE',TOKEN,RC,SLENGTH,STRING,RLENGTH,RESULT)

This call requests execution of an APL expression. The expression to be executed is specified as a character string - effectively the right argument of an ⍎ primitive. The result is returned as a byte string derived from the 'enlist' (∈) of the result of the executed expression. If an error occurred during the execution of the expression, the error message (∈⎕*EM*) will be returned in the result field.

The arguments to this call are:

'APLE' a required argument indicating that an APL expression is to be executed.

TOKEN a fullword integer containing the token returned on the 'INIT' call. If this token is not provided (i.e.: specified as zero), the call will require more CPU time to execute.

RC a fullword integer field into which APL2PI will place the return code on completion of the call. Return code of 0 0 indicates success. Other return codes are described below.

SLENGTH a fullword integer specifying the length of the string to be executed.

STRING the character string to be executed.

RLENGTH a fullword integer field specifying the length of the result field. On completion of this call, this field will be updated to contain the actual length of the result or error message produced (which may be shorter, the same size, or longer than the result field). If no result or error message is produced, this field will be set to -1.

Note that this field is normally updated as a result of this call. It therefore should not be coded as a constant on calls from high level languages for to do so could result in the constant being modified, which in turn could result in subsequent errors in the calling program.

RESULT a field into which the result of the executed expression (if any) will be placed. If the result produced is larger than the length of this field (as specified by RLENGTH), only the first RLENGTH bytes of the result will be placed into the RESULT field. RLENGTH will be updated to reflect the actual total result length. The result is placed in this field as an unmodified byte string in left list order, i.e., as if it had been produced by the expression:

```
RESULT←(PFA RESULT) ATR RESULT
```

If an error results from the execution of the specified expression, the error message (∈⎕*EM*) will be placed in the result field and its length in the RLENGTH field. An error message is not produced in all situations. In general, a message will not be produced if the error is detected before execution of the specified expression has begun. In such situations, the RESULT field will not be updated.

Return Control to APL

CALL APL2PI ('APLX',TOKEN,RC,VALUE,RESULT)

This call is used to return control to APL2, either to an interactive APL session, or to the APL application that invoked or transferred control to the non-APL application.

Control may be subsequently returned to the non-APL application by calling one of the APL external functions ***APL2PI*** or ***APL2PIE***.

The arguments to this call are:

'APLX' a required argument indicating that control is to be returned to APL2.

TOKEN a fullword integer containing the token returned on the 'INIT' call. If this token is not provided (i.e.: specified as zero), the call will require more CPU time to execute.

RC a fullword integer field into which APL2PI will place the return code on completion of the call. Return code of 0 0 indicates success. Other return codes are described below.

VALUE this optional parameter, if specified, must be a fullword field containing 0 or the address of a CDR to be returned as an explicit result to the APL environment. This CDR must contain `0 0` as its first two items of data. If this parameter is specified as 0 or is not specified an explicit result of `0 0` ***MSG*** is returned, where the value of ***MSG*** is defined in the AP2PAPIW packaged workspace which is supplied with APL2.

RESULT this optional parameter, if specified, must be a fullword field which will be updated to contain the address of a result CDR when and if control is returned to the non-APL application. If control is returned with a call to the ***APL2PI*** external function, no result CDR is returned, and this field will be set to 0. If control is returned with a call to the ***APL2PIE*** external function, the left argument to ***APL2PIE*** will be returned as the result. If no left argument is provided, the field will be set to 0.

If the RESULT parameter is not provided, no result will be returned, even if one is provided in the left argument to ***APL2PIE***.

When the 'APLX' called is issued by the non-APL application, the RESULT field may contain a fullword zero, in which case any result CDR will be returned without conversion, or it may contain the address of a pattern CDR (see "Pattern CDR's" on page 16), in which case the pattern CDR will be used to convert any result CDR returned.

Execution of an 'APLX' call is not permitted while a namescope entered with the 'APLP' call is the active namescope, and will be rejected with a 0 10 ('invalid request') return code.

Execute an APL Function

CALL APL2PI ('APLF',TOKEN,RC,PKGWS,FN,RSLT,LARG,RARG)

This is the second of two calls provided to request execution of an APL function. This call is designed for use from languages such as C and Assembler, and it provides greater access to and control over the arguments and result of the specified function. Unlike the 'APLS' call, this call passes arguments and expects results in APL2 CDR format.

The specified function may be in a packaged workspace and may have any valid valence. The arguments to this call are:

'APLF' a required argument indicating that an APL function is to be called.

TOKEN a fullword integer containing the token returned on the 'INIT' call. If this token is not provided (i.e.: specified as zero), the call will require more CPU time to execute.

RC a fullword integer field into which APL2PI will place the return code on completion of the call. Return code of 0 0 indicates success. Other return codes are described below.

PKGWS the name of the packaged workspace in which the specified function is to be found and optionally a surrogate name for the function. If this argument is provided,

```
'PKGWS' 11 ⎕NA 'FN'
```

or

```
'PKGWS' 11 ⎕NA 'FN SURROGATE'
```

will be executed before the specified function is called. If this argument is not provided or is coded with an initial blank, no ⎕NA will be issued before calling the function. Thus, if the function exists in a packaged workspace, the first call to it must provide the PKGWS argument, but subsequent calls do not. The PKGWS argument is a character string which is expected to be terminated with a blank, e.g.: `'MYLIB.MYWS '`. If the surrogate name is specified it must be prefixed with a colon, viz: `'MYLIB.MYWS:SURROGATE '`.

FN the name of the function to be called. This argument is a character string which is expected to be terminated with a blank. It is used as the right argument to ⎕NA if the PKGWS argument is coded, and it is then used as the name of the function to be called.

RSLT when APL2PI is called, this fullword field may be set to 0 or to the address of a 'pattern CDR' (see "Pattern CDR's" on page 16) to be used to convert the result of the APL function. If 0 is specified on the call, the result will be produced as a CDR without conversion. On completion of the APL2PI call, this field will contain 0, if no explicit result or error message was produced, or the address of a CDR representing the result or error message (∈⎕EM) produced by the function. If the function completed with an APL error, the error message will be returned as a default CDR, without any reference to the pattern CDR provided on input. An error message is not produced in all situations. In general, a message will not be produced if the error is detected before execution of the specified function has begun. In such situations, the RSLT field will be set to 0.

LARG a fullword field containing the address of the CDR representing the left argument to the function or containing 0 if no left argument is provided.

RARG a fullword field containing the address of the CDR representing the right argument to the function or containing 0 if no right argument is provided.

Reference or Specify an APL Variable

CALL APL2PI ('APLV',TOKEN,RC,PKGWS,VARIABLE,VALUE,PATTERN)

This call may be used to obtain or specify the value of an APL variable. If a packaged workspace is specified, the specified variable must already exist in the packaged workspace - it cannot be created with this call. To create a new variable in a packaged workspace, use the 'APLP' call to enter the packaged workspace namescope, then this call (with no PKGWS argument) to create the variable and an 'APLP' call to exit the packaged workspace namescope.

The arguments to this call are:

'APLV' a required argument indicating that an APL variable is to be accessed.

TOKEN a fullword integer containing the token returned on the 'INIT' call. If this token is not provided (i.e.: specified as zero), the call will require more CPU time to execute.

RC a fullword integer field into which APL2PI will place the return code on completion of the call. Return code of 0 0 indicates success. Other return codes are described below.

PKGWS the name of the packaged workspace in which the specified variable is to be found and optionally a surrogate name for the variable. If this argument is provided,

```
'PKGWS' 11 ⎕NA 'VARIABLE'
```

or

```
'PKGWS' 11 ⎕NA 'VARIABLE SURROGATE'
```

will be executed before the specified variable is accessed. If this argument is not provided or is coded with an initial blank, no ⎕NA will be issued before accessing the variable. Thus, if the variable exists in a packaged workspace, the first access to it must provide the PKGWS argument, but subsequent accesses do not. The PKGWS argument is a character string which is expected to be terminated with a blank, e.g.: `'MYLIB.MYWS '`. If the surrogate name is specified it must be prefixed with a colon, viz: `'MYLIB.MYWS:SURROGATE '`.

VARIABLE the name of the variable to be accessed. This argument is a character string which is expected to be terminated with a blank. It is used as the right argument to ⎕NA if the PKGWS argument is coded, and it is then used as the name of the variable to be accessed.

VALUE a fullword field containing the address of the CDR representing the value to be assigned to the variable, or containing 0 if the variable is to be referenced. On completion of the call the address of the CDR representing the value of the variable is placed in this field.

PATTERN This is an optional argument and may be specified when a variable is referenced (VALUE = 0 on input). If specified, it is a fullword field which may contain the address of a 'pattern CDR' (see "Pattern CDR's" on page 16) used to convert the value of the variable. If unspecified, or if the field contains zero, default conversion will be used to produce the VALUE CDR.

Enter or Exit a Packaged Workspace Namescope

CALL APL2PI ('APLP',TOKEN,RC,PKGWS)

This call may be executed to enter or exit a specified packaged workspace namescope. Until a packaged workspace namescope is entered, calls to APL2PI will be executed from the namescope established when APL2 was invoked (typically the active workspace namescope). When a packaged workspace namescope is entered via an 'APLP' call, subsequent calls to APL2PI will be executed in that namescope until another 'APLP' call is issued to exit the namescope or enter another.

The arguments to this call are:

'APLP' — a required argument indicating that an APL packaged workspace namescope is to be entered or exited

TOKEN — a fullword integer containing the token returned on the 'INIT' call. If this token is not provided (i.e.: specified as zero), the call will require more CPU time to execute.

RC — a fullword integer field into which APL2PI will place the return code on completion of the call. Return code of 0 0 indicates success. Other return codes are described below.

PKGWS — if specified, this argument identifies the packaged workspace whose namescope is to be entered. If not specified, the request is to exit the current namescope.

If specified, this argument must be terminated with a blank, and must take one of the following forms:

LIBRARY.MEMBER or ***MEMBER***

where ***LIBRARY*** is the DDNAME (TSO) or file name (CMS) of the load library in which the packaged workspace resides, and ***MEMBER*** is the member name of the packaged workspace. The same rules apply to locating the packaged workspace as when such information is provided in the left argument to ***⎕NA***.

Note that execution of the 'APLX' call is not permitted while a namescope entered with an 'APLP' call is the active namescope.

Note also that the uses of 'APLP' are designed to be paired: an 'APLP' call to enter a packaged workspace, followed sometime later by an 'APLP' call (without the packaged workspace specified) to return to the previous environment. Paired 'APLP' calls can be nested - in other words, one packaged workspace can be entered from another, but care must be taken in unwinding the nesting. An attempt to issue an 'APLP' call to exit a packaged workspace that was not paired with a previous 'APLP' call to enter a packaged workspace will result in a 0 10 error return code ("invalid request").

The 'APLP' call operates by accessing ***⎕EA*** in the specified packaged workspace. If, when the packaged workspace was created, a list of accessible objects was specified, and if ***⎕EA*** was not included in that list, requests to enter that packaged workspace namescope will fail.

Return Codes

Each of the calls described above returns a return code in the RC argument field. These return codes are returned as integer fullwords, but are best interpreted as pairs of halfwords. If the first halfword is non-zero, the return code is a ***⎕ET*** value that resulted from APL execution. In addition to the ***⎕ET*** values, the following return codes are defined:

0 0 success

0 1 APL is already initialized. This return code may result from an 'INIT' call and is the expected return code when the 'INIT' call is issued from a non-APL application which was invoked from APL2 using ***APL2PIE***.

0 2 Unexpected shutdown. APL has terminated unexpectedly (perhaps as a result of an ***)OFF*** command or as a result of an unsuccessful 'INIT' call). This return code may result from any call other than 'TERM', in situations where a non-APL application is running independently of APL2. If this return code is received by a non-APL application invoked from APL, or during processing in a routine nominated as a service routine on an 'INIT' call, processing should be terminated in an orderly fashion and control returned to the routine's caller.

0 3 Expected shutdown. This return code can result from any call to APL2PI and indicates a request from APL for the non-APL application to terminate. In response to the request, the non-APL application should terminate and return control to its caller.

0 4 Insufficient space. There is insufficient free memory for the correct operation of the APL2PI routine. A larger region or virtual machine should be used to run the application.

0 5 (TSO only) Not executing under the TSO TMP. The program which issues calls to APL2PI must be invoked under TSO or a TSO TMP, (typically 'IKJEFT01').

0 10 Invalid request, or invalid parameter list. Among other things, this return code may result from an 'APLX' call while a namescope, entered with an 'APLP' call was active, or from an 'APLP' request to exit a packaged workspace when no packaged workspace namescope was active.

0 11 Unexpected internal error in the APL2PI routine.

Normally, return codes in which the first halfword is non-zero originate from APL and should be interpreted as ***⎕ET*** values. The following, however, can originate from the APL2PI routine:

1 2 (System Error) - an unexpected error in the APL2PI routine.

1 5 (No Shares) - the specified packaged workspace cannot be accessed on a 'APLP' call. This error may occur because the specified packaged workspace could not be located or loaded, or because it was already the active namescope.

3 1 (Value Error) - the variable named on an 'APLV' call could not be accessed in the specified packaged workspace.

Using CDR Results

The 'APLX', 'APLF' and 'APLV' calls which return results, return those results in CDR format. These results are always pointer form CDR's and are built as temporary objects in the APL2 workspace. On the next call to APL2PI, these temporary objects are erased before the call is executed. Thus, CDR results returned by APL2PI may not be used as arguments on subsequent calls, and all processing of such results must be performed before any subsequent call to APL2PI.

Pointer form CDR's returned by APL2PI have addresses in the CDR pointer section. That is to say, the CDRPTR fields in that section contain addresses with the high order bit on, and never zeros or tokens.

Pattern CDR's

Pattern CDR's can be specified on 'APLX', 'APLF' or 'APLV' calls to control the creation of the CDR representing the result of an APL function or expression or the value of an APL variable. If specified, APL will attempt to convert the result or value to the data types specified in the pattern CDR. Further, during this conversion, APL will check that the ranks and shapes of the result or value and its items correspond to those specified in the pattern CDR. If the result or value cannot be converted as specified, or if a rank or shape mismatch is detected, an appropriate APL error will be generated.

The format of CDR's is discussed in the APL2 Processor Interface Reference manual (SH20-9234). The format of a pattern CDR matches that of the CDR header and descriptor sections. That is to say that it is just like a CDR without the pointer or data sections. It consists of the CDRFLAGS, CDRDLEN, CDRXRHO, CDRRT, CDRRL, CDRRANK, and CDRRHO fields only. The contents of the CDRFLAGS field must be valid, but they do not influence the type of CDR produced. A pointer form CDR result or value is always produced.

Unlike a CDR, a pattern CDR may have CDRXRHO or one or more elements of CDRRHO specified as X'80000000' or CDRRANK specified as X'8000'. These values indicate that the corresponding fields are unspecified and are not to be used in rank or shape checking. If CDRRANK is so specified for a particular item of the array, CDRRHO fields may not follow it.

The Pattern CDRs used on this interface are similar to the argument patterns used to describe the arguments of an external routine called by APL (see APL Programming - System Service Reference, chapter 23. SH20-9218). Pattern CDRs, however, conform to the true CDR header format (as defined by the AP2CDR macro), while argument patterns are an EBCDIC representation of it.

External Functions ATP and PTA

When an 'APLS' call is issued to execute an APL function, the arguments provided in the call are passed to the APL function as a vector of addresses - one address for each argument. To the APL function this appears as a vector of integer values. The external function ***PTA*** ('Pointers to Array') is provided to allow access to these arguments. ***PTA*** expects a vector of addresses as its right argument and a pattern (similar to the pattern used with the external function RTA) as its left argument, and it will produce an APL array as a result. For example:

```
ARRAY←'(G0 1 3)(I4 0)(E8 1 2)(C1 1 10)' PTA POINTERS
```

will convert a set of three arguments -- a scalar fullword integer, a pair of double precision real numbers, and a 10 byte character string, respectively -- to an APL vector of three items.

The external function ***ATP*** ('Array to Pointer') is provided to allow pointer arguments to be replaced (i.e.: updated) with an APL array. The syntax for use of this function is:

```
PATTERN ATP ARRAY POINTERS
```

where *PATTERN* is a pattern (similar to the pattern used with *ATR*) which describes the data in the desired format, *POINTERS* is the address of the data to be updated, and *ARRAY* is the source array. Note that this function does not produce an explicit result. Further, it makes no check to ensure that the result fields are large enough to hold the source values.

The *PTA* function assumes a one-to-one correspondence between the data descriptors in the left argument, the data items in the array specified in the right argument, and the set of data areas specified by the pointers in the right argument. Thus, to update a set of three data areas, three pointers must be provided, an array containing three items must be provided, and the pattern must specify either a three element simple array or a general array containing three simple arrays, viz:

```
      'C1 1 3' ATP  'ABC' (P1,P2,P3)
```

or:

```
      ARRAY←'ABCD' (4 2⍴⍳8) 1.234
      PATTERN←'(G0 1 3)(C1 1 4)(I4 2 4 2)(E8 0)'
      PATTERN ATP ARRAY (P1,P2,P3)
```

If it is necessary to update a data area with a non-simple APL array (i.e.: put a data structure into a single data area), the non-simple array must be converted to a record using *ATR*, and then *ATP* can be used to move it to the data area, viz:

```
      ARRAY←'ABCD' (4 2⍴⍳8) 1.234
      PATTERN←'(G0 1 3)(C1 1 4)(I4 2 4 2)(E8 0)'
      RECORD←PATTERN ATR ARRAY
      'C1 1 *' ATP RECORD POINTER
```

Using PTA and ATP

Assume that the function *AVERAGE* in packaged workspace *COMPUTE* is called with the following three arguments:

1. a vector of double precision numbers,
2. a fullword integer indicating the number of items in the first argument,
3. a double precision real field in which the function is to place its result.

The APL function might be coded as shown in figure 4.

```
      ∇AVERAGE ARGS;V;N;R
[1]   →0⍴3 11 ⎕NA 2 3⍴'PTAATP'     ⍝ Access PTA and ATP
[2]   N←'I4 0' PTA 2⊃ARGS          ⍝ Get N from 2nd argument
[3]   V←('E8 1 ',⍕N) PTA ↑ARGS     ⍝ Get V from 1st argument
[4]   R←(+/V)÷N                    ⍝ Compute the average
[5]   'E8 0' ATP R (3⊃ARGS)        ⍝ Update 3rd argument (result)
      ∇
```

Figure 4. Using PTA and ATP

For additional information on the patterns used in the left arguments of *PTA* and *ATP*, see the description of *RTA* and *ATR* in the APL2 Programming: Using the Supplied Routines manual (SH20-9233), and the description of argument patterns for Processor 11 in the APL2 Programming: System Services Reference manual (SH20-9218).

External Functions APL2PI and APL2PIE

Two APL external functions, ***APL2PI*** and ***APL2PIE***, are provided to facilitate communication from APL2 to non-APL applications. ***APL2PI*** and ***APL2PIE*** can be accessed by means of ***⎕NA***, viz:

```
0 11 ⎕NA 'APL2PI'
0 11 ⎕NA 'APL2PIE'
```

Note that the first item of the left argument of ***⎕NA*** must be 0 (and not 3) for proper operation of the rest of the APL2PI interface.

APL2PI is a niladic function used to return control to the non-APL application after APL2 initialization or after an 'APLX' call from the non-APL application. It is equivalent to ***APL2PIE*** `0 ''` as described below.

APL2PIE is an ambivalent function which serves a number of different purposes:

- return control from the APL2 environment to the currently active non-APL application,
- invoke a non-APL application from the APL2 environment,
- request termination of the currently active non-APL application,
- issue a service request to a non-APL application.

Calls to ***APL2PIE*** can be imbedded in APL applications which run independently or are invoked through APL2PI from non-APL applications. Use of ***APL2PIE*** takes the following forms:

RESULT APL2PIE `0 ''`

Return control to the currently active non-APL application. This request can be issued immediately after APL2 is invoked from a non-APL application or after a non-APL application has returned control to APL with an 'APLX' call. Attempting to return control in any other situation will result in a 1 0 return code. If issued monadically, no result will be returned to the non-APL application. If issued dyadically, the left argument will be returned to the non-APL application in CDR format if the RESULT parameter was provided on the 'APLX' call.

Note that this ***APL2PIE*** request (or an ***APL2PI*** request which is equivalent to ***APL2PIE*** `0 ''`) causes control to be transferred from the APL environment at the point at which the request is made. Thus, if a request of this type is imbedded in an APL function, the function is suspended at that point and control is transferred to the non-APL application. Subsequent requests from the non-APL application are executed in the context of this suspended function. In particular, a subsequent 'APLX' request will return control to the suspended function. Users should avoid imbedding ***APL2PIE*** `0 ''` calls in APL applications unless their use is clearly understood and planned for.

COMMAND APL2PIE `1` ***NAME***

Invoke a non-APL application using the specified ***COMMAND***. ***COMMAND*** is a character string containing the name of the module to be invoked, optionally followed by one or more arguments to be passed to the module when it is invoked. In the MVS/TSO environment, the specified module must reside in a load library in the user's normal search order. In the VM/CMS environment, the specified module may be the name of an existing CMS nucleus extension, or the name of a relocatable load module residing on an accessible minidisk.

The non-APL application is assigned the specified ***NAME***. That name must match the NAME in any 'INIT' call issued by the non-APL application and in subsequent ***APL2PIE*** `3` calls to the non-APL application.

When the specified module terminates, control will be returned to APL and APL2PIE will return a result of 0 1 RC where RC is the return code resulting from the module. Control may also be returned to APL if the non-APL application issues an 'APLX' call. In this case, the result returned by ***APL2PIE*** will have the form 0 0 MSG and is provided by the VALUE parameter of the 'APLX' call or a default message provided in the ***APL2PAPIW*** packaged workspace.

Note that for successful use of this service, APL2PI must be link edited as a separate module and loaded as a nucleus extension in CMS or placed in the APL2 load library in MVS. Additional information on this subject can be found in the sections entitled "Invoking a non-APL Application through APL2PIE" in the "System Related Considerations" section of this document.

APL2PIE 2 ''

Request termination of the currently active non-APL application. This request simply sends return code 0 3 back to the non-APL application in response to its last call to APL2PI. The non-APL application is expected to honor this request and terminate. Note that ***NAME*** may not be specified in the right argument; only the currently active non-APL application can be terminated.

If the non-APL application terminates as expected, a result of 0 1 RC will be returned from APL2PIE, where RC is the termination return code resulting from the non-APL application.

VALUE APL2PIE 3 NAME

Make a service request to the ***NAME***'d non-APL application. This request is possible only if the non-APL application specified a SERVICE routine address and a TYPE other than 0 on its 'INIT' call. If these requirements are met, this request will cause a subroutine call to that service routine.

If ***APL2PIE*** is called dyadically, the left argument is passed to the service routine in CDR or non-CDR format depending upon the specification of the TYPE parameter on the 'INIT' call from the non-APL application. In non-CDR format, the ***VALUE*** will be passed as a byte string result of the expression:

(PFA VALUE) ATR VALUE

On entry to the service routine of the named non-APL application,

```
R1  => A(VALUE) in CDR or non-CDR format or A(0) if called monadically
       A(Return Code)     => F'0'
       A(Result Pointer)  => A(0)
       A(APL2PI)
       A(ANCHOR)
R13 => 18 word save area
R14 =  return address
R15 =  address of service routine
```

The return code field is a fullword which can be updated by the service routine. It is initialized to zero. If set to a non-zero value, a three element vector (0,1,return code) will be returned as the result of the ***APL2PIE*** function call when control is returned from the service routine. The result pointer field is a fullword, initialized to zero, into which the service routine can place the address of a value which will be used as the explicit result of the ***APL2PIE*** function call when control is returned from the service routine. If the return code is zero, the result pointer field will be examined. If the result pointer is zero, a result of 0 1 0 will be returned. If it is non-zero, the specified result, pointed to by the address in the result pointer field, will be returned.

The result pointer field must contain zero or the address of a result value in CDR or non-CDR form, depending on the specification of the TYPE parameter on the non-APL application's 'INIT' call. If TYPE was specified as 1 (non-CDR), the result will be interpreted as a byte string prefixed with a fullword length field. If desired, it can be converted to a different form using the APL external function ***RTA***.

If TYPE was specified as 2 (CDR), the result will be interpreted as an API array in CDR form. The array may have any value; however, if its first item is zero, it must have the form:

```
N M VALUE
```

where N M may be 0 0, 0 1, or any defined ⎕*ET* value and ***VALUE*** may be any valid APL array, or may be elided

A service request can also result in the following unsuccessful return codes:

0 0 MSG the service routine issued an 'APLX' request to APL2PI resulting in control being returned to APL2. Such a request may result in unexpected behavior and is not recommended.

0 1 RC the non-APL application terminated with return code RC

1 0 an attempt was made to call the APL2PI or APL2PIE external function when the APL2PI routine was not active or when no non-APL application was active on the APL2PI interface.

0 4 insufficient free space for correct execution

0 10 invalid arguments (typically ***NAME*** too long)

1 2 unexpected error

1 5 invalid ***NAME*** or no service routine for the named non-APL application

1 6 this return code results from an attempt to invoke a non-APL application (***COMMAND APL2PIE*** `1 'NAME'`) when the named application is already active on the APL2PI interface.

Note that the service routine entered as a result of this call to ***APL2PIE*** can issue requests to APL2 using the APL2PI interface. Since such requests can theoretically result in a recursive call to the same service routine, provisions for such an event should be incorporated into the design of the non-APL application.

All of the different calls to APL2PI are supported from executing service routines. It is recommended, however, that the 'TERM' and 'APLX' calls be avoided as they may lead to unexpected and undesired results. The 'TERM' may result in termination of APL2 when the call is made or later at some unexpected time. The 'APLX' call will return control to the APL application that invoked the service routine, and it will appear to that application as if the service routine had terminated. If control is subsequently returned to the non-APL application (with an ***APL2PI*** or ***APL2PIE*** `0 ''` call), control will be returned to the executing service routine which will presumably eventually terminate and return control to APL2.

Note that in many circumstances where the non-APL application and/or the service routine is written in a high level language (such as C or PL/I), ***APL2PIE*** `3` calls to service routines will not operate correctly. Typically, two problems prevent correct operation:

1. While high level languages provide mechanisms by which a subroutine address can be passed as an argument on a call, the form in which that subroutine address is passed may not be acceptable to APL2PI. The APL2PI INIT call expects the SERVICE parameter to be provided as a fullword which contains the address of the of the SERVICE subroutine.
2. The linkage conventions expected by a high level language subroutine may not match those provided by APL2PI (as described above) when the service routine is called. Subroutines written in C and PL/I, for example, may require register 12 to be set on entry, and register 13 to point to a save area within a save area stack maintained by the C or PL/I run time environment. These requirements are not fulfilled in the linkage conventions used by APL2PI.

Service routines designed to be used with APL2PI will most typically be written in Assembler language. With some limitations, they can be written in FORTRAN (using the IBM VS FORTRAN program product) if the service routine is structured as a FORTRAN subroutine and the non-APL application is a FORTRAN mainline routine. An example of such a service routine is contained in the section entitled "Using the APL2PI Interface from FORTRAN".

APL2PI and APL2 Calls to Other Languages

Through the use of ⎕NA, APL users can invoke applications written in languages other than APL2. Supported languages include FORTRAN and Assembler language, but users have reported success with COBOL, PL/1, C, and Pascal as well. The APL2PI interface also provides facilities through which APL can be invoked by or can invoke a non-APL application.

When using the APL2PI interface, some care must be taken not to interfere with the operation of the non-APL application by calling other non-APL routines through ⎕NA which are written in the same high level language as the non-APL application interacting with APL2PI.

This situation is of concern because certain high level languages, such as FORTRAN, require access to a "programming environment" for any non-trivial program. Typically, only one instance of the necessary programming environment is supported in a user's address space or virtual machine at any given time. Non-APL applications written in such high level languages that invoke or are invoked by APL2 via the APL2PI interface will typically establish their necessary programming environment as part of their own invocation.

When a non-APL routine is accessed through the use of ⎕NA, the :INIT tag in the NAMES file entry for that non-APL routine specifies whether a programming environment is required for correct execution of that routine. If required, Processor 11 will attempt to initialize the environment when the non-APL routine is first called, or as a result of a specific request from the APL caller. This instance of the environment is not the same as the instance of the environment established in conjunction with the APL2PI interface, and it may not operate correctly or worse, it may cause unexpected or erroneous results.

It is therefore recommended that when an application written in a high level language like FORTRAN invokes or is invoked by APL2 via the APL2PI interface, no other routines written in the same language be invoked via APL2PI or ⎕NA. Note that ESSL and OSL routines also have a dependency on the FORTRAN programming environment and should therefore not be invoked when a non-APL application written in FORTRAN is active on the APL2PI interface.

Additional information on routines accessed through ⎕NA and their requirements in terms of programming environments can be found in the APL2 Programming: System Services Reference manual (SH20-9218) in the chapter entitled "Processor 11 - Calling Compiled Routines".

System Related Considerations

Using APL2PI in a VM/CMS Environment

In the VM/CMS environment, the APL2PI interface is provided with the following components:

AP2VAPI TXT130 - the object module which contains the APL2PI entry point that is called from non-APL applications. This object module can be combined with the non-APL application or generated as a separate module that can be dynamically loaded by the non-APL application, or accessed as a CMS nucleus extension. Each of these alternatives are described below.

This object module can be found in file 5 on the APL2 and APL2 AE basic machine-readable materials tape, or may be provided as a TEXT file supplied with the PTF's listed at the beginning of this document.

AP2XAPIC AP2MSAMP - an Assembler language source file which can be modified by users to alter the command and parameters used to invoke APL2 from a non-APL application. If an invocation command other than the default:

```
APL2 QUIET RUN(APL2PI)
```

is desired, this source file can be modified, reassembled, and combined with the AP2VAPI object module. If AP2XAPIC is combined with AP2VAPI, the invocation command assembled into AP2XAPIC will be used; otherwise, the default invocation command will be used. Note that in either of these cases, the invocation parameters can be supplemented or overridden by means of the PARMS parameter in the 'INIT' call from the non-APL application.

The AP2XAPIC source file can be found in file 3 on the APL2 and APL2/AE basic machine-readable materials tape.

Modifying the APL2 Invocation Command and Options

To change the command or options used to invoke APL2 from a non-APL application:

1. Copy the AP2XAPIC AP2MSAMP file to AP2XAPIC ASSEMBLE
2. Edit AP2XAPIC ASSEMBLE

   Modify the statement labeled APL2CMDN to change the name of the APL2 module invoked. For example, to cause APL2/AE to be invoked, change the statement to:

   ```
   APL2CMDN DC    CL9'APL2AE'                COMMAND
   ```

   Modify the statement labeled APL2CMDO to change the invocation options. It is recommended that the QUIET and RUN(APL2PI) options be left unchanged.

3. After making the necessary changes to AP2XAPIC ASSEMBLE, reassemble it using the following CMS commands:

   ```
   GLOBAL MACLIB AP2MAC
   ASSEMBLE AP2XAPIC
   ```

4. Combine the TEXT file resulting from this assembly with the AP2VAPI object module. This can be done without destroying the original object module with the following CMS commands:

   ```
   COPY AP2VAPI TXT130 A AP2VAPI TEXT A
   COPY AP2XAPIC TEXT  A AP2VAPI TEXT A (APPEND
   ```

   The resulting AP2VAPI TEXT file should then be used in place of AP2VAPI TXT130 in the procedures described below.

Accessing APL2PI from a non-APL Application

AP2VAPI contains the APL2PI entry point that is called from a non-APL application to request services from APL2. It can be made accessible to the non-APL application in a number of ways:

- If the non-APL application is invoked with CMS LOAD and START commands, the AP2VAPI object module can be made available as a TEXT file on an accessible CMS minidisk, and it will be loaded by CMS when the non-APL application is loaded. To make AP2VAPI accessible as a TEXT file, use the following CMS command:

  ```
  COPY AP2VAPI TXT130 A APL2PI TEXT A
  ```

  Note that it is necessary to change its name to APL2PI TEXT since it is referred to by that name in the non-APL application.

- If the non-APL application is generated as a CMS MODULE using the GENMOD command, APL2PI can be simply incorporated in that module when it is built, viz:

  ```
  COPY AP2VAPI TXT130 A APL2PI TEXT A
  LOAD ... non-APL application ...
  GENMOD ... non-APL application ...
  ```

  APL2PI TEXT will be combined with the non-APL application as a result of the LOAD command.

- In a number of situations, it is may be desirable to structure APL2PI as a CMS nucleus extension and cause it to be loaded and accessed dynamically from the non-APL application. This approach has the advantage that APL2PI is placed in CMS protected storage as an entity separate from the non-APL application. To prepare APL2PI to be loaded as a CMS nucleus extension, it can be converted to a module using the commands:

  ```
  COPY AP2VAPI TXT130 A APL2PI TEXT A
  LOAD APL2PI (CLEAR RLDSAVE
  GENMOD APL2PI
  ERASE APL2PI TEXT A
  ```

  The APL2PI module can be subsequently accessed by the non-APL application using the CMS NUCXLOAD and NUCXDROP commands and NUCEXT functions. For additional information on this subject, see the CMS Macros and Functions Reference (SC24-5280) and CMS Command Reference (SC19-6209) manuals.

Invoking a non-APL Application through APL2PIE

The ***APL2PIE*** external function can be used to invoke a non-APL application which can subsequently make use of the APL2PI interface, viz.:

```
0 11 ⎕NA 'APL2PIE'
'COMMAND' APL2PIE 1 'NAME'
```

The user must ensure that when such a request is made, that APL2PI is established as a CMS nucleus extension; otherwise, an error message will be issued by the APL2PIE function and the request will be denied. If APL2 was invoked via APL2PI from a non-APL application, APL2PI will already be established as a CMS nucleus extension, and no other action is necessary. If APL2 was not invoked via APL2PI, then the user must take explicit action to cause it to be established as a nucleus extension.

To explicitly cause APL2PI to be established as a CMS nucleus extension, it must be first created as a CMS module. This can be accomplished with the following CMS commands:

```
COPY AP2VAPI TXT130 A APL2PI TEXT A
LOAD APL2PI (CLEAR RLDSAVE
GENMOD APL2PI
ERASE APL2PI TEXT A
```

The resulting APL2PI MODULE can be established as a CMS nucleus extension with the CMS command:

```
NUCXLOAD APL2PI
```

Parameters may be passed to the non-APL application by specifying them in the ***'COMMAND'*** left argument of ***APL2PIE***, viz:

```
0 11 ⎕NA 'APL2PIE'
'COMMAND PARMS' APL2PIE 1 'NAME'
```

The specified non-APL application (***COMMAND***) is entered with register 0 pointing to a CMS extended parameter list and register 1 pointing to a CMS tokenized parameter list, viz:

```
R0 => A(command verb) => C'COMMAND '
      A(parameters)   => C'PARMS'
      A(end of command)
      A(0)

R1 => CL8'COMMAND '
      CL8'PARMS   '
      X'FFFFFFFF'
```

VM/XA Considerations

In the VM/XA environment, APL2 normally runs in 31-bit mode. In that mode, the APL2 workspace is placed above the 16 megabyte line. In order to obtain results from APL2, the APL2PI routine must therefore run in 31-bit mode. This should pose no problem if the non-APL application runs in 31-bit mode when it calls APL2PI, when its service routine is called by APL2PI, and when it is accessing data in CDR form returned by APL2PI

If the non-APL application must run in 24-bit mode when calling or being called by APL2PI, the APL2 workspace must be forced below the line. This can be done by invoking APL2 with the invocation option XA(24).

Note that if APL2PI is generated as a separate module (so that it can be NUCXLOADed as described above), the correct addressing and residency modes must be specified. If the non-APL application operates in 31-bit mode,

```
GENMOD APL2PI (AMODE 31 RMODE 24
```

is suggested. If the non-APL application operates in 24-bit mode,

```
GENMOD APL2PI (AMODE 24 RMODE 24
```

will be required.

Using APL2PI in an MVS/TSO Environment

In the MVS/TSO environment, the APL2PI interface is provided with the following components:

AP2TAPI the object module which contains the APL2PI entry point that is called from non-APL applications. This object module can be link edited with the non-APL application or as a separate load module accessed by the non-APL application.

This object module can be found in file 3 (JLG1310.F1) on the APL2 and APL2/AE basic machine-readable materials tape and is placed in the APL2.AP2MODS DLIB by the installation process, or may be provided as an object module supplied with the PTF's listed at the beginning of this document.

AP2XAPIC an Assembler language source file which can be modified by users to alter the command and parameters used to invoke APL2 from a non-APL application. If any invocation command other than the default:

```
APL2 QUIET RUN(APL2PI)
```

is desired, this source file can be modified, reassembled and link edited with the AP2TAPI object module. If AP2XAPIC is link edited with AP2TAPIC, the invocation command assembled into AP2XAPIC will be used: otherwise, the default invocation command will be used. Note that in either of these cases, the invocation parameters can be supplemented or overridden by means of the PARMS parameter in the 'INIT' call from the non-APL application.

The AP2XAPIC source file can be found in file 5 (JLG1310.F3) on the APL2 and APL2/AE basic machine-readable materials tape and is placed in the APL2.AP2SOURC DLIB by the installation process.

Modifying the APL2 Invocation Command and Options

To change the command or options used to invoke APL2 from a non-APL application:

1. Make a copy of the AP2XAPIC source file from the APL2 distribution data set.
2. Edit your copy of the AP2XAPIC source file.

   Modify the statement labeled AP2CMDN to change the name of the APL2 module invoked. For example, to cause APL2/AE to be invoked, change the statement to:

   ```
   APL2CMDN DC    CL9 'APL2AE'          COMMAND
   ```

   Modify the statement labeled APL2CMDO to change the invocation options. It is recommended that the QUIET and RUN(APL2PI) options be left unchanged.
3. After making the necessary changes to AP2XAPIC, reassemble it, specifying APL2.AP2MACS as the macro library.
4. When link editing AP2TAPI in the procedures described below, specify an INCLUDE statement for the object module produced by this assembly.

Steps 1-3 in this procedure can typically be performed in a straightforward fashion using ISPF (options 3, 2, and 4). If you are unfamiliar with the use of ISPF, consult your system administrator for assistance. Step 4 is typically accomplished by executing a batch job such as the one shown in the next section.

Accessing APL2PI from a non-APL Application

AP2TAPI contains the APL2PI entry point that is called from a non-API application to request services from APL2. It can be made accessible to the non-API application by link editing it with that application, or by link editing it as a separate module and dynamically loading it from the non-APL application.

To link edit it with the non-APL application, simply INCLUDE AP2TAPI (and optionally AP2XAPIC) in the link edit of the non-APL application.

The following job can be used to link edit AP2TAPI (and optionally AP2XAPIC) as a separate load module:

```
//LINK JOB (ACCOUNT),PROGAMER,CLASS=A,TIME=(1),
//         NOTIFY=USERID,MSGCLASS=A,MSGLEVEL=(1,1)
//LINK EXEC PGM=IEWL,REGION=512K,
//         PARM='NCAL,RENT,REUS,MAP,LIST,LET,SIZE=(512K,64K)'
//SYSPRINT DD SYSOUT=*
//SYSLMOD  DD DISP=SHR,DSN=output data set
//OBJ      DD DISP=SHR,DSN=input data set
//SYSUT1   DD UNIT=SYSDA,SPACE=(13030,(40,20))
//SYSLIN   DD *
 MODE AMODE(31),RMODE(ANY)
 INCLUDE OBJ(AP2TAPI)
 INCLUDE OBJ(AP2XAPIC)
 ENTRY APL2PI
 NAME APL2PI(R)
/*
//
```

The input data set should identify the data set in which AP2TAPI and AP2XAPIC reside. The output data set must be available to the non-APL application when it needs to load APL2PI. The APL2 load library provides a convenient location since that data set must also be available during execution of the APL2PI interface.

Under MVS, APL2 and the APL2PI routine must be invoked under the TSO terminal monitor program, IKJEFT01. This is the normal mode of operation if the application is running in a TSO environment, and no special action is needed. If, however, the application was designed to operate in a batch environment, it must be invoked through the TSO terminal monitor program. For example, a batch program, normally invoked with:

```
//STEP   EXEC PGM=MYPROG
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
```

must instead be invoked with:

```
//STEP   EXEC PGM=IKJEFT01
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
MYPROG
/*
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
```

Finally, in either a TSO or batch environment, the files necessary to run APL2 must be allocated before issuing the APL2PI 'INIT' call from the non-APL application.

Invoking a non-APL Application through APL2PIE

The *APL2PIE* external function can be used to invoke a non-APL application which can subsequently make use of the APL2PI interface, viz.:

```
0 11 ⎕NA 'APL2PIE'
'COMMAND' APL2PIE 1 'NAME'
```

The user must ensure that when such a request is made, that APL2PI is available as a separate load module in the same load library (or in the same concatenated list of libraries) from which APL2 was loaded. Instructions on creating APL2PI as a separate load module are described above.

Parameters may be passed to the non-APL application by specifying them in the *'COMMAND'* left argument of *APL2PIE*, viz:

```
0 11 ⎕NA 'APL2PIE'
'COMMAND PARMS' APL2PIE 1 'NAME'
```

The specified non-APL application (*COMMAND*) is entered with register 1 pointing to a TSO CPPL control block. The first word in this control block points to the TSO command buffer representing this command, viz:

```
R1 => A(command buffer) => command buffer
```

The command buffer begins with a halfword length field indicating the total length in bytes of the command buffer, followed by a halfword offset field indicating the offset from the command name to the beginning of the command parameters, followed by the command. Thus for *'COMMAND PARMS' APL2PIE 1 'NAME'*,

```
R1 => A(command buffer) => X'00170008',C'COMMAND PARMS'
```

MVS/XA Considerations

In the MVS/XA or MVS/ESA environments, APL2 normally runs in 31-bit mode. In that mode, the APL2 workspace is placed above the 16 megabyte line. In order to obtain results from APL2, the APL2 routine must therefore run in 31-bit mode. This should pose no problem if the non-APL application runs in 31-bit mode when it calls APL2PI. when its service routine is called by APL2PI, and when it is accessing data in CDR form returned by APL2PI.

If the non-APL application must run in 24-bit mode when calling or being called by APL2PI, the APL2 workspace must be forced below the line. This can be done by invoking APL2 with the invocation option XA(24).

IMS Considerations

APL2 and APL2/AE are not officially supported in an IMS environment. Some users have successfully used the APL2/AE product, however, under IMS. To do so the application must be invoked by the TSO terminal monitor program, IKJEFT01. Thus, in the IMS environment, the transaction is structured to invoke IKJEFT01 which in turn invokes the application code that uses APL2PI and through it APL2/AE. Note that IKJEFT01 is an authorized program and that this characteristic must be preserved when running under IMS.

```
     ∇AVG ARGS;SIZE;NUMBERS;RESULT
[1]  ⍝ ARGS: Vector of 3 addresses from non-APL application
[2]  ⍝   ARGS[1]→ Number of numbers  (fullword integer)
[3]  ⍝   ARGS[2]→ Vector of numbers  (floating point)
[4]  ⍝   ARGS[3]→ Result field (fullword integer)
[5]
[6]   →(0∨.≠3 11 ⎕NA 2 3⍴'PTAATP')/ERROR
[7]   →(3≠⍴ARGS)/ERROR
[8]
[9]  ⍝ Retrieve size of input vector from ARGS[1]
[10]  SIZE←'I4 0' PTA ARGS[1]
[11]
[12] ⍝ Retrieve vector of numbers from ARG[2]
[13]  NUMBERS←('E8 1 ',⍕SIZE)PTA ARGS[2]
[14]
[15] ⍝ Compute the average
[16]  RESULT←(+/NUMBERS)÷SIZE
[17]
[18] ⍝ Return result to ARGS[3]
[19]  'E8 0' ATP RESULT ARGS[3]
[20]  →0
[21]
[22] ERROR:'Unexpected error' ⎕ES 9 9
     ∇
```

Figure 5. APL2 AVG program used in language related examples

Using the APL2PI Interface from FORTRAN

Most functions available on the APL2PI interface can be used in a simple and straightforward fashion in FORTRAN programs. Since the FORTRAN language does not provide support for data structures or pointers however, the 'APLF' and 'APLV' calls cannot be used effectively, and only limited function is available between APL2 and a FORTRAN service routine.

This section presents three simple examples of the use of the APL2PI interface from FORTRAN programs. Each of the examples shown has the ability to invoke APL2, or be invoked by APL2. The IBM VS FORTRAN Program Product (5668-805) Version 2, Release 4 was used to construct these examples. Other FORTRAN compilers may or may not have similar capabilities.

The first example shows a FORTRAN program which makes use of the APL2PI 'APLS' call to invoke the APL function ***AVG*** in packaged workspace ***PKGLIB.STATS*** to obtain the average of a vector of numbers passed to it. Lines of the FORTRAN routines shown below are numbered on the left for reference in the notes after the figure.

```
 1        REAL*8 NUMBERS(1000),RESULT
 2        INTEGER*4 TOKEN,RC,SIZE,LENGTH
 3        INTEGER*2 RETCODE(2)
 4        EQUIVALENCE (RC,RETCODE(1))
 5        TOKEN=0
 6        LENGTH=0

    C     ---- CALL APL2PI TO INITIALIZE APL2
 7        CALL APL2PI('INIT',TOKEN,RC,'SAMPLE ',0,0,0,16,'SM(OFF) WS(200K)')
 8        IF (RC .GT. 1) GOTO 98

 9        WRITE (6,*)'Enter number of numbers to average'
10        READ  (5,*) SIZE
11        WRITE (6,1)'Enter ',SIZE,' numbers'
12        READ  (5,*) (NUMBERS(I),I=1,SIZE)

    C     ---- CALL APL2PI TO COMPUTE AVERAGE
13        CALL APL2PI ('APLS',TOKEN,RC,'PKGLIB.STATS ','AVG ',LENGTH,' ',
         1             SIZE,NUMBERS,RESULT)
14        IF (RC .NE. 0) GOTO 99

15        WRITE (6,2) 'The average is: ',RESULT

    C     ---- CALL APL2PI TO TERMINATE APL2
16     10 CALL APL2PI ('TERM',TOKEN,RC)
17        RETURN

    C     ---- UNEXPECTED ERROR FROM APL2 INITIALIZATION
18     98 WRITE (6,*) 'Unexpected error during APL2 initialization'
20        WRITE (6,3) 'Return code: ',(RETCODE(I),I=1,2)
21        RETURN

    C     ---- UNEXPECTED ERROR DURING APL2 FUNCTION EXECUTION
22     99 WRITE (6,*) 'Unexpected error during APL2 function execution'
23        WRITE (6,3) 'Return code: ',(RETCODE(I),I=1,2)
24        GOTO 10

25       1 FORMAT(' ',A,I2,A)
26       2 FORMAT(' ',A,F8.3)
27       3 FORMAT(' ',A,2I2)

28        END
```

Figure 6. FORTRAN program demonstrating use of the 'APLS' request

Notes

- Lines 1-2: define the various data items that will be passed as arguments in subsequent APL2PI calls. It is important to ensure that the data types defined match those expected by APL2PI (e.g.: TOKEN, RC, L) and by the APL2 function being called (SIZE, NUMBERS, RESULT).
- Lines 3-4: the return code returned by APL2PI is returned in the fullword (INTEGER*4) field RC. This code is, however, best interpreted as a pair of halfwords (INTEGER*2). The definition of RETCODE as INTEGER*2 and equivalent to RC provides simple and meaningful subsequent access to the return code.

- Lines 7-8: APL2 is initialized with an 'INIT' call to APL2PI. The FORTRAN program is identified to APL2PI as 'SAMPLE'. Since no service routine is required, the TYPE, ANCHOR, and SERVICE parameters are coded as 0 on the call.

  If APL2 is not active when these statements are executed, it will be initialized as a result of this call and a return code of 0 will be returned. If APL2 was already active, this call is used to identify the FORTRAN program to APL2PI and a return code of 1 will be returned. Thus the error routine is only invoked if an unexpected error is encountered.

- Line 13: The function ***AVG*** in packaged workspace ***PKGLIB.STATS*** is invoked with arguments SIZE, NUMBERS and RESULT. Since this APL function is expected not to return any explicit result, a value of 0 is passed as the RLENGTH parameter on the call. Because this field is updated to reflect the actual length of the result, the parameter cannot be coded as 0 in the call itself. Instead, the variable L is defined and initialized to 0 prior to the call. If this call was made repetitively, L would have to be reinitialized prior to each call. Failure to do so could cause unexpected results in the FORTRAN program.

- Line 16: Note that the return code from the 'TERM' call is not checked. Two possible return codes might be expected: 0 0 if the non-APL application originally caused APL to be invoked, or 0 10 if the non-APL application was invoked from an active APL environment.

- Lines 6, 10, 13 and 19 in the APL function ***AVG***: FORTRAN passes arguments to subroutines 'by reference'. That is to say, FORTRAN passes the addresses of argument data rather that the values of the argument data. The external function ***PTA*** allows an APL application to retrieve data passed by reference, and the external function ***ATP*** allows an APL application to update arguments which were passed by reference.

The second example demonstrates the use of the 'APLX' and 'APLE' calls. When this program is executed, it causes APL2 to be initialized and then returns control to the APL environment. When this occurs, the following message will appear on the user's screen:

```
0 0
      +-----------------------------------+
      + ENTER 'APL2PI' TO RETURN CONTROL  +
      +-----------------------------------+
```

At this point the APL user can interact with APL freely and, when finished returns control to APL2PI by calling the APL2PI external function, viz:

```
A←B←C←10
APL2PI
```

```
 1        CHARACTER RESULT(1000)
 2        INTEGER*4 TOKEN,RC,LENGTH
 3        INTEGER*2 RETCODE(2)
 4        EQUIVALENCE (RC,RETCODE(1))
 5        TOKEN=0
 6        LENGTH=1000

    C     ---- CALL APL2PI TO INITIALIZE APL2
 7        CALL APL2PI('INIT',TOKEN,RC,'SAMPLE ',0,0,0,16,'SM(OFF) WS(200K)')
 8        IF (RC .GT. 1) GOTO 99

    C     ---- RETURN CONTROL TO APL2
 9        CALL APL2PI('APLX',TOKEN,RC)
10        IF (RC .NE. 0) GOTO 98

    C     ---- EXECUTE AN APL EXPRESSION
11        CALL APL2PI('APLE',TOKEN,RC,13,''' ''',⎕NL 2 3 4',LENGTH,RESULT)
12        IF (RC .NE. 0) GOTO 97

13        WRITE (6,*) 'Names in APL workspace: ',(RESULT(I),I=1,L)

    C     ---- CALL APL2PI TO TERMINATE APL2
14      10 CALL APL2PI ('TERM',TOKEN,RC)
15         RETURN

    C     ---- UNEXPECTED ERROR FROM APL2 INITIALIZATION
16      99 WRITE (6,*) 'Unexpected error during APL2 initialization'
17         WRITE (6,1) 'Return code: ',(RETCODE(I),I=1,2)
18         RETURN

    C     ---- UNEXPECTED ERROR ON ATTEMPT TO RETURN CONTROL TO APL2
19      98 WRITE (6,*) 'Unexpected error returning control to APL2'
20         WRITE (6,1) 'Return code: ',(RETCODE(I),I=1,2)
21         GOTO 10

    C     ---- UNEXPECTED ERROR ON DURING APL2 EXECUTION
22      97 WRITE (6,*) 'Unexpected error during APL2 execution'
23         WRITE (6,1) 'Return code: ',(RETCODE(I),I=1,2)
24         GOTO 10

25       1 FORMAT(' ',A,2I2)

26       END
```

Figure 7. FORTRAN program demonstrating use of 'APLX' and 'APLE' requests

Notes

- Line 9: causes control to be returned to the active APL session. Note that the VALUE and RESULT optional parameters on this call cannot be used in a FORTRAN program. This is because these parameters involve the use of data in CDR format, and FORTRAN is not capable of dealing with CDR format.
- Line 11: causes the APL expression

  ```
  ' ',⎕NL 2 3 4
  ```

to be executed. Catenating a blank on to the output of ⎕*NL* causes the names to be separated by at least one blank in the result returned to the FORTRAN program. Note that APL characters may be imbedded in FORTRAN source programs but may not be printed correctly in the compiled listings.

The third example demonstrates the use of a FORTRAN service routine which can be accessed from APL. In this example, the FORTRAN mainline routine initializes APL2, specifying the SERVICE subroutine as a type 1 service routine to be used by the APL2PI interface. After APL initialization is complete, the FORTRAN mainline then passes control to the APL environment with an 'APLX' call. When that happens, the following message appears on the APL user's screen:

```
0 0
      +-----------------------------------+
      + ENTER 'APL2PI' TO RETURN CONTROL  +
      +-----------------------------------+
```

and the user can interact with APL freely. For the purposes of this example, the user should enter the following to cause the SERVICE subroutine to be invoked:

```
      0 11 ⎕NA 'APL2PIE'
      1234 APL2PIE 3 'SAMPLE'
```

This causes control to be passed to the service routine (the SERVICE subroutine) of the APL2PI application identified as SAMPLE (the mainline FORTRAN program). The value 1234 is the first of 5 arguments passed to the service routine. All of the arguments can be retrieved by the FORTRAN SERVICE routine, but only the return code argument can be updated to return data from the FORTRAN SERVICE routine to the APL environment.

Once execution of the SERVICE routine is complete, and control is passed back to the APL2 user, the user completes his work and returns control to the FORTRAN mainline by calling the APL2PI external function.

Notes

- Line 8: the fourth argument of the 'INIT' call causes the FORTRAN application to be identified to APL2PI with the name SAMPLE. This name will be subsequently used as the second item of the right argument of *APL2PIE* when control is passed to the SERVICE routine, whose address is provided in the seventh argument of the 'INIT' call.

```
 1        EXTERNAL SERVICE
 2        CHARACTER RESULT(1000)
 3        INTEGER*4 TOKEN,RC,LENGTH
 4        INTEGER*2 RETCODE(2)
 5        EQUIVALENCE (RC,RETCODE(1))
 6        TOKEN=0
 7        LENGTH=1000

    C     ---- CALL APL2PI TO INITIALIZE APL2
 8        CALL APL2PI('INIT',TOKEN,RC,'SAMPLE ',1,0,SERVICE,16,'SM(OFF) WS(200K)')
 9        IF (RC .GT. 1) GOTO 99

    C     ---- RETURN CONTROL TO APL2
10        CALL APL2PI('APLX',TOKEN,RC)
11        IF (RC .NE. 0) GOTO 98

    C     ---- EXECUTE AN APL EXPRESSION
12        CALL APL2PI('APLE',TOKEN,RC,13,'''  ''',⎕NL 2 3 4',LENGTH,RESULT
13        IF (RC .NE. 0) GOTO 97

14        WRITE (6,*) 'Names in APL workspace: ',(RESULT(I),I=1,L)

    C     ---- CALL APL2PI TO TERMINATE APL2
15     10 CALL APL2PI ('TERM',TOKEN,RC)
16        RETURN

    C     ---- UNEXPECTED ERROR FROM APL2 INITIALIZATION
17     99 WRITE (6,*) 'Unexpected error during APL2 initialization'
18        WRITE (6,1) 'Return code: ',(RETCODE(I),I=1,2)
19        RETURN

    C     ---- UNEXPECTED ERROR ON ATTEMPT TO RETURN CONTROL TO APL2
20     98 WRITE (6,*) 'Unexpected error returning control to APL2'
21        WRITE (6,1) 'Return code: ',(RETCODE(I),I=1,2)
22        GOTO 10

    C     ---- UNEXPECTED ERROR ON DURING APL2 EXECUTION
23     97 WRITE (6,*) 'Unexpected error during APL2 execution'
24        WRITE (6,1) 'Return code: ',(RETCODE(I),I=1,2)
25        GOTO 10

26      1 FORMAT(' ',A,2I2)

27        END

28        SUBROUTINE SERVICE(VALUE,RC,RESULT,ADDRESS,ANCHOR)
29        INTEGER*4 VALUE,RC,RESULT,ADDRESS,ANCHOR
30        WRITE (6,*) 'FORTRAN SERVICE routine called by APL2'
31        WRITE (6,*) 'Input values: ',VALUE,RC,RESULT,ADDRESS,ANCHOR
32        RC=9999
33        RETURN
34        END
```

Figure 8. FORTRAN program demonstrating use of a service routine

Using the APL2PI Interface from C

Most functions available on the APL2PI interface can be used in C programs. The 'APLF' and 'APLV' calls and some variants of the 'APLX' call involve the use of data in CDR format and are more difficult, but not impossible, to handle in the C environment. C service routines are not supported.

This section presents a number of examples of the use of the APL2PI interface from C programs. Each of these examples has the ability to invoke APL2, or be invoked by APL2. The IBM C/370 Program Product (5688-039, 5688-040) Version 1 Release 2 was used to construct these examples. Other C compilers may or may not have the same capabilities.

To understand any of the calls to APL2PI from the C environment, the reader must understand the linkage conventions used by the APL2PI interface. All calls to or from the APL2PI interface assume the /370 OS linkage convention. That is to say, when the call occurs, it is expected that the caller will have set the following registers:

- R1 contains the address of the caller's parameter list. The parameter list is expected to contain a list of addresses -- one for each argument in the call.
- R13 contains the address of a save area, 18 fullwords in length, which may be used by the called routine to save the caller's registers.
- R14 contains the return address in the calling routine.
- R15 contains the entry point address in the called routine.

To cause the C program to utilize these conventions, "#pragma linkage (...,OS)" must be used in the C program to define the APL2PI routine.

Further, any arguments which are to be updated by the called routine must be passed as pointers rather than values. In the C language, arrays and strings are always passed as pointers, so they require no special handling. Scalars arguments, however, are not normally passed as pointers and must be prefixed with '&' if they are to be updated, viz:

```
#pragma linkage(ROUTINE,OS)
int input,output,array[5]
input=3
ROUTINE(input,&output,array)
```

causes the procedure ROUTINE to be called with arguments 'input', 'output' and 'array'. The arguments 'output' and 'array' can be updated by the called ROUTINE, but any attempt to update 'input' will not be reflected in the calling program.

Finally, APL2PI expects to be called as a subroutine rather than as a function and thus produces no explicit result.

The first example shown below illustrates a C program which makes use of the APL2PI 'APLS' call to invoke the APL function ***AVG*** (see Figure 5 on page 28) in packaged workspace ***PKGLIB.STATS*** to obtain the average of a vector of numbers passed to it. Lines of the C program are numbered on the left for reference in the notes after the figure.

```
 1  #pragma linkage(APL2PI,OS)
 2  #include <stdio.h>
 3  main()
 4  {
 5      int token=0,size;
 6      union {
 7         int code;                    /* Return code as 1 fullword  */
 8         short rc[2];                 /* Return code as 2 halfwords */
 9      } rc;
10      double numbers[1000],result,value;
11      char parms[]="SM(OFF) WS(200K)";  /* APL initialization parms */
12      int len=sizeof(parms)-1;          /* Length of parms          */

    /*  ---- Call APL2PI to initialize APL2 ----  */
13      APL2PI("INIT",&token,&rc.code,"SAMPLE ",0,0,0,len,parms);
14      if(rc.code > 1) goto error1;

15      printf("Enter numbers to be averaged\n");
16      printf("Terminate input with non-numeric\n");
17      for (size=0 ; 0<scanf("%lf",&value) ;size++)
18          numbers[size]=value;

    /*  ---- Call APL2PI to compute average ---- */
19      APL2PI("APLS",&token,&rc.code,"PKGLIB.STATS ","AVG ",0,' ',
20             size,numbers,&result);
21      if(rc.code != 0) goto error2;

22      printf("\nThe average is: %lf\n",result);

    /*  ---- Call APL2PI to terminate APL2 ----  */
23     shutdown:
24      APL2PI("TERM",&token,&rc.code);
25      return 0;

26     error1:
27      printf("Unexpected error during APL2 initialization\n");
28      printf("Return code: %hd %hd\n",rc.rc[0],rc.rc[1]);
29      return;

30     error2:
31      printf("Unexpected error during APL2 function execution\n");
32      printf("Return code: %hd %hd\n",rc.rc[0],rc.rc[1]);
33      goto shutdown;

34  }
```

Figure 9. C program demonstrating use of the 'APLS' request

Notes

- Line 1: causes APL2PI to be called with OS linkage conventions.
- Line 6-8: the return code produced by APL2PI is returned as a vector of 2 halfwords. Redefinition of the return code field allows simpler comparison to expected values such as 0 0 or 0 1.
- Lines 13, 19, 24: the scalar arguments 'token' 'rc' and 'result' are updated by APL2PI and so must be prefixed with '&'.

The following example demonstrates the use of the 'APLP' and 'APLF' calls in a C program and the use of data in CDR format. This example also shows that the 'APLF' call can be used to request execution of APL primitives or system functions.

```
#pragma linkage(APL2PT,OS)
#include <stdio.h>
#define CDRID 0x80000000
main() {

    struct cdrdesc {              /* CDR Descriptor section */
        union {
            unsigned int  cdrdlen;
            unsigned char cdrflags;
        } cdrhdr;
        int   cdrxrho;
        char  cdrrt;
        char  cdrrl;
        short cdrrank;
    };

    struct cdrptr {               /* CDR Pointer section    */
        int cdrpslen;
        int cdrdslen;
        char *cdrptr;
        int cdrplen;
    };

    struct  {                     /* CDR for vector  2 3 4  */
        struct cdrdesc desc;
        int  rho;
        int  data[3];
    } v234 = {CDRID+16,3,'I',4,1,3,2,3,4},
      *ptr_v234;                  /* Pointer to v234        */

    struct {                      /* CDR for  matrix result */
        struct cdrdesc desc;
        int  rows;
        int  cols;
        struct cdrptr pointers;
    } *result;

    char *result_data;           /* used to point to result data */

    int token=0,i,j;
    union {
        int code;                /* Return code as 1 fullword  */
        short rc[2];             /* Return code as 2 halfwords */
    } rc;
    char parms[]="SM(OFF) WS(200K)"; /* APL initialization parms */
    int len=sizeof(parms)-1;     /* Length of parms          */
```

Figure 10 (Part 1 of 2). C program demonstrating use of the 'APLP' and 'APLF' requests

```
   /*  ---- Call APL2PI to initialize APL2 ----  */
42     APL2PI("INIT",&token,&rc.code,"SAMPLE ",0,0,0,len,parms);
43     if(rc.code > 1) goto error1;

   /*  ---- Call APL2PI to enter STATS namescope ---- */
44     APL2PI("APLP",&token,&rc.code,"PKGLIB.STATS ");
45     if(rc.code != 0) goto error2;

   /*  ---- Call APL2PI to execute □NL 2 3 4 ---- */
46     result = 0;
47     ptr_v234 = &v234;
48     APL2PI("APLF",&token,&rc.code," ","□NL ",&result,0,&ptr_v234);
49     if(rc.code != 0) goto error2;

   /*  ---- Display result returned by APL2 ---- */
50     printf("\nResult returned from execution of □NL 2 3 4\n\n");
51     printf("%s%x\n%s%d\n%s%c\n%s%x\n%s%hd\n%s%i %i\n%s\n%s\n",
52        "CDRDLEN = ",result->desc.cdrhdr.cdrdlen,
53        "CDRXRHO = ",result->desc.cdrxrho,
54        "CDRRT   = ",result->desc.cdrrt,
55        "CDRRL   = ",result->desc.cdrrl,
56        "CDRRANK = ",result->desc.cdrrank,
57        "CDRRHO  = ",result->rows,result->cols,
58        "CDRDATA:",
59        "--------");

60     result_data=result->pointers.cdrptr;

61     for (i=0;i<result->rows;i++){
62         for (j=0;j<result->cols;j++)
63             putchar(*result_data++);
64         putchar('\n');
65     }

66     printf("--------\n\n");

   /*  ---- Call APL2PI to exit STATS namescope ---- */
67     APL2PI("APLP",&token,&rc.code);

   /*  ---- Call APL2PI to terminate APL2 ----  */
68    shutdown:
69     APL2PI("TERM",&token,&rc.code);
70     return 0;

71    error1:
72     printf("Unexpected error during APL2 initialization\n");
73     printf("Return code: %hd %hd\n",rc.rc[0],rc.rc[1]);
74     return;

75    error2:
76     printf("Unexpected error during APL2 function execution\n");
77     printf("Return code: %hd %hd\n",rc.rc[0],rc.rc[1]);
78     goto shutdown;

79 }
```

Figure 10 (Part 2 of 2). C program demonstrating use of the 'APLP' and 'APLF' requests

Notes

- Lines 3, 6-33: this routine makes use of the 'APLF' call which requires that arguments and results passed to and from APL2 be provided in CDR format. CDR format is described in detail in the APL2 Programming: Processor Interface Reference manual (SH20-9234). CDR's passed from C programs to APL2PI may be dense or pointer form CDR's; CDR's returned from APL2PI are always pointer form CDR's.
- Lines 6-15: the descriptor section of a CDR is defined as a C structure. Note that CDRRHO is not included since this CDR field may be a null vector.
- Lines 16-21: the CDR pointer section is defined as a C structure.
- Lines 22-27: a dense form CDR representing the integer vector **2 3 4** is defined and initialized. The address of this CDR is assigned to ptr_v234 on line 47 and that address is passed as an argument on the 'APLF' call on line 48.
- Lines 28-33: the 'APLF' call on line 48 should produce a character matrix result. This result will be returned as the address of a pointer form CDR which is mapped by this structure.
- Line 44: an 'APLP' call is issued to cause the ***PKGLIB.STATS*** namescope to be entered; subsequent APL2PI calls will be executed in that namescope. This technique is necessary if subsequent 'APLF' calls request execution of primitive functions since the packaged workspace argument cannot be provided on such calls.
- Line 48: an 'APLF' call is issued to request execution of the system function **⎕*NL*** with right argument **2 3 4**. Note that a system function name or a primitive function symbol can be specified as the function to be executed on an 'APLF' call. If APL symbols are imbedded in character literals in a C program they may not be displayed correctly in the listing.

  The RESULT, LARG, and RARG arguments of the 'APLF' call are expected to be fullword fields which contain the addresses of CDR's. Therefore, pointers must be specified, using the C '&' operator, when these arguments are passed on the call. Since a left argument is not provided for this call, the LARG field is coded as 0 in the argument list.

Using the APL2PI Interface from COBOL

Many of the functions available on the APL2PI interface can be used in a simple and straightforward fashion in COBOL programs. Since COBOL only provides very rudimentary support for pointers, however, the 'APLF' and 'APLV' calls cannot be used effectively. COBOL service routines are not supported.

The following example shows a COBOL program which makes use of the APL2PI interface to generate a set of random numbers and to compute their average. The program illustrates simple use of the 'INIT', 'TERM', 'APLE' and 'APLS' calls. Lines of the program are numbered on the left for reference in the notes after the figure. The IBM VS COBOL II Program Product (5668-958) Version 1, Release 3 was used to construct this example. Other COBOL compilers may or may not have similar capabilities.

```
 1     Identification division.
 2       Program-id. callapl2.
 3     Environment division.
 4     Configuration section.
 5       Source-computer.  IBM-370.
 6       Object-computer.  IBM-370.
 7     Input-output section.

 8     Data division.
 9     Working-storage section.

10       1 TOKEN picture s9(9) binary value zero.
11       1.
12        2 RCODE.
13          3 RCODE1 picture s9999 binary.
14          3 RCODE2 picture s9999 binary.
15        2 RETCODE redefines RCODE picture s9(9) binary.
16       1 ZEROV picture s9(9) binary value zero.

17       1 OPTIONS picture x(16) value "SM(OFF) WS(200K)".

18       1 QNA-ATR picture x(14) value "0 11 □NA 'ATR'".
19       1 GET-RANDOM picture x(20) value "'E8 1 *' ATR 5 ? 100".

20       1 RESULT-LENGTH picture s9(9) binary.
21       1 RESULT-BUFFER.
22         2 RESULTS computational-2 occurs 5 times.
23       1 NUMBERS-ARRAY.
24         2 NUMBERS picture -ZZ9 display occurs 5 times.
25       1 ITEM picture s9(9) binary.
26       1 ITEMS picture s9(9) binary value 5.
27       1 AVERAGE computational-2.
28       1 DISPLAY-AVERAGE picture -ZZ9.999 display.
```

Figure 11 (Part 1 of 2). COBOL program demonstrating use of the 'APLE' and 'APLF' requests

```
29      Procedure division.

30          Call "APL2PI" using
31              by content "INIT" by reference TOKEN RCODE
32              by content "SAMPLE " ZEROV ZEROV ZEROV
33              length of OPTIONS OPTIONS
34          If RETCODE is > 1 go to ERROR1
35          End-if

36          Call "APL2PI" using
37              by content "APLE" by reference TOKEN RCODE
38              by content length of QNA-ATR QNA-ATR
39              by content ZEROV " "
40          If RETCODE is not = 0 go to ERROR2
41          End-if

42          Move length of RESULT-BUFFER to RESULT-LENGTH
43          Call "APL2PI" using
44              by content "APLE" by reference TOKEN RCODE
45              by content length of GET-RANDOM GET-RANDOM
46              by reference RESULT-LENGTH RESULT-BUFFER
47          If RETCODE is not = 0 go to ERROR2
48          End-if

49          Perform with test after
50              varying ITEM from 1 by 1
51              until ITEM = ITEMS
52            move RESULTS(ITEM) to NUMBERS(ITEM)
53          End-perform
54          Display "Random numbers returned by APL2: "
55                  NUMBERS-ARRAY upon console

56          Call "APL2PI" using
57              by content "APLS" by reference TOKEN RCODE
58              by content "PKGLIB.STATS " "AVG " ZEROV " "
59              by reference ITEMS RESULT-BUFFER AVERAGE
60          If RETCODE is not = 0 go to ERROR2
61          End-if

62          Move AVERAGE to DISPLAY-AVERAGE
63          Display "The average is: " DISPLAY-AVERAGE upon console.

64      Shutdown.
65          Call "APL2PI" using
66              by content "TERM" by reference TOKEN RCODE
67          Stop run.

68      Error1.
69          Display "Error during APL2 initialization: "
70              RCODE1 " " RCODE2 upon console
71          Stop run.
72      Error2.
73          Display "Error during APL2 execution: "
74              RCODE1 " " RCODE2 upon console
75          Go to shutdown.
```

Figure 11 (Part 2 of 2). COBOL program demonstrating use of the 'APLE' and 'APLF' requests

Notes

- Lines 12-15: the return code returned by APL2PI is formally a pair of halfwords in a fullword field. In some situations it is useful to treat it as a single fullword; in others, as a pair of halfwords.
- Line 16: numeric literals cannot be specified as arguments in a COBOL CALL statement, and therefore must be given names in the data division.
- Lines 18-19: specify APL expressions that will later be executed by means of 'APLE' calls to APL2PI. Note that APL characters can be specified in such expressions but may not print correctly in the COBOL program listing.
- Lines 30-33: APL2 is initialized by means of an 'INIT' call to APL2PI. Note that arguments that are to be updated on the call must be passed by reference, while constants and arguments which are not expected to be updated are passed by content. If a 'by content' argument is updated as a result of the call to APL2PI, the updated value will not be available in the COBOL program.
- Line 34: this program is set up to allow it to invoke APL2 or to be invoked by APL2. This is done by accepting a return code of either 0 or 1 from the 'INIT' call.
- Line 39: in this particular example, the calling COBOL program will not bother to check the results of the ***⎕NA*** executed in this call, since the subsequent call will fail with a predictable error if the ***⎕NA*** fails. Therefore, the RLENGTH and RESULT fields are specified as ZERO and " " respectively. APL2PI will update the RLENGTH field with the length of the actual result, but that updated value will not be returned to the COBOL program because the ZERO argument was passed by content.

APL2 and COBOL Data Representations

APL2 typically represents numeric data in a number of different formats in the workspace. Real numbers are represented as double precision floating point values, integers are typically represented using fullword integer representation, and boolean values are often represented as bits. The representation of the value of a variable or the result of an expression is dependent upon the operations performed upon it and cannot be simply predicted. Most of this is transparent to the APL2 user who sees numbers as numbers an lets the computer manage their representation in its internal memory.

COBOL, on the other hand, is a language in which data representation is visible to and carefully managed by the programmer. When data is passed between a COBOL application and APL2 using the APL2PI interface, that data must be transformed to a representation acceptable to APL2 and/or the COBOL application. This same situation exists when other high level languages are used with the APL2PI interface. The APL2 external functions ***PTA***, ***ATP***, ***ATR*** and ***RTA*** are available to assist with such transformation. ***PTA*** and ***ATP*** are described in this document; ***ATR*** and ***RTA*** are described in the APL2 Programming: Using the Supplied Routines manual (SH20-9233).

The following table shows the correspondence between types specified in the COBOL USAGE clause and those specified in the patterns used with the ***PTA***, ***ATP***, ***ATR*** and ***RTA*** external functions:

COBOL Numeric type	Picture and USAGE	RT/RL in pattern
Binary	PIC S9999 BINARY	I2
	PIC S9(9) BINARY	I4
Internal Floating	COMPUTATIONAL-1	E4
	COMPUTATIONAL-2	E8
External Floating	PIC +9(3).99E+99 DISPLAY	none
External Decimal	PIC S9999 DISPLAY	Z5
Internal Decimal	PIC S9999 PACKED-DECIMAL	P3

In addition to allowing numeric data to be represented in these forms, COBOL also separately maintains a scale factor or decimal point position for binary and decimal representations (the scale factor is an inherent part of the floating point representation and consequently does not have to be separately maintained). When COBOL computations are performed on binary and decimal data, COBOL aligns the data around the decimal point to achieve the desired results. When such data is passed to APL, the position of the decimal point is not passed. For example, if the variable

```
01 CASH PICTURE S9999.99 BINARY VALUE -1234.56.
```

was passed to APL, it would be received as the value `¯123456`. If the COBOL program treated the value as dollars and cents, and the APL program treated it as cents, no problem would exist. If the position of the decimal point was variable or significant, it would be lost unless passed as a separate explicit argument to APL.

Similarly, if the value `¯1234567` was placed by an APL application in the COBOL CASH field as defined above, it would be interpreted by COBOL as the value -1234.56. This is because only the data, and not the decimal position is passed between APL and COBOL.

This behavior becomes a little more complex when decimal (packed or external) data is passed from or to a COBOL program. On the System/370, packed and zoned decimal representations allow a very wide range of numbers to be represented (31 digits for packed and 15 for zoned). When a packed or zoned decial number, passed from COBOL, is accessed with the ***PTA*** function, using the 'P' or 'Z' representation types, that number is converted into double precision floating point representation so that it can be subsequently processed by APL. This conversion may lose precision and may change an integral value to a non-integral one (i.e.: a real number). Worse, the ***ATP*** and ***ATR*** external functions will not accept floating point right arguments when a representation type of 'P' or 'Z' is specified in the pattern specified in the left argument. This problem can be circumvented by converting the data to fullword integers using the APL floor (⌊) primitive, viz:

```
      DATA←1.23×100
      DATA
123
      'P2 0' RTA 'P2 0' ATR DATA
DOMAIN ERROR
      'P2 0' RTA 'P2 0' ATR DATA
                        ∧
      'P2 0' RTA 'P2 0' ATR ⌊DATA
123
```

An alternate and often preferable way to avoid such problems is to use the BINARY, rather than DISPLAY or PACKED-DECIMAL, usage clause in COBOL programs for integer data that is passed to or from APL.

Using the APL2PI Interface from PL/I

Most functions available on the APL2PI interface can be used in PL/I programs. The 'APLF' and 'APLV' calls and some variants of the 'APLX' call involve the use of data in CDR format, and are more difficult, but not impossible to handle in the PL/I environment. PL/I service routines are not supported.

The following example shows a simple PL/I program which makes use of the APL2PI interface to call the function ***AVG*** in packaged workspace ***PKGLIB.STATS*** to obtain the average of a vector of numbers passed to it. Lines of the PL/I program are numbered on the left for reference in the following notes. The IBM PL/I Optimizing Compiler Version 2, Release 2 (5668-909) was used to construct this example. Other PL/I compilers may or may not have similar capabilities.

Notes

- Line 3: the APL2PI entry point must be declared in PL/I programs as shown on line 3. This declaration ensures that APL2PI will be called with the correct linkage conventions.
- Line 6: the return code returned by APL2PI is formally a pair of halfwords in a fullword field. In some situations it is useful to treat it as a single fullword; in others, as a pair of halfwords.
- Line 9: numeric literals cannot be specified as arguments in a PL/I CALL statement and therefore must be given names by means of declarative statements.
- Line 14: this program is set up to allow it to invoke APL2 or to be invoked by APL2. This is done by accepting a return code of either 0 or 1 from the 'INIT' call.
- Line 20: note that APL2PI always updates the LEN field as the result of an 'APLS' call, thereby destroying the initial value of this field. If a subsequent 'APLF' call was made by this program, the LEN field would have to be reset before the call.

```
 1  *PROCESS OPT(2);
 2   PLI2APL: proc options(main reentrant) reorder;

 3   dcl APL2PI entry options(asm inter);
 4   dcl (NUMBERS(100), RESULT) float bin(53);
 5   dcl (TOKEN init(0), RC, SIZE, LEN init(0)) fixed bin(31);
 6   dcl RETCODE(2) fixed bin(15) based(addr(rc));
 7   dcl PICSIZE pic 'Z9';
 8   dcl (PICRC1, PICRC2) pic 'ZZZ9';
 9   dcl ZERO init(0) fixed bin(31) static;
10   dcl OPTIONS char(16) init('SM(OFF) WS(200K)') static;
11   dcl OPTLEN init(16) fixed bin(31) static;
12   dcl BUFFER char(72);

      /* ---- call APL2PI to initialize APL2 ---- */
13   call APL2PI('INIT',TOKEN,RC,'SAMPLE ',ZERO,ZERO,ZERO,OPTLEN,OPTIONS);
14   if RC > 1 then goto ERROR1;

15   display('Enter number of numbers to average ') reply(BUFFER);
16   SIZE = BUFFER;
17   PICSIZE = SIZE;
18   display('Enter ' || PICSIZE || ' numbers ') reply(BUFFER);
19   get string(BUFFER) list((NUMBERS(I) do I=1 to SIZE));

     /* ---- call APL2PI to compute average ---- */
20   call APL2PI('APLS',TOKEN,RC,'PKGLIB.STATS ','AVG ',LEN,' ',
                 SIZE,NUMBERS,RESULT);
21   if RC ¬= 0 then goto ERROR2;

22   display('The average is: '||RESULT);

     /* ---- call APL2PI to terminate APL2 ---- */
23   SHUTDOWN:
24   call APL2PI('TERM',TOKEN,RC);
25   return;

     /* ---- unexpected error during APL2 initialization ---- */
26   ERROR1:
27   display('Unexpected error during APL2 initialization');
28   PICRC1 = RETCODE(1);
29   PICRC2 = RETCODE(2);
30   display('Return Code:' || PICRC1 || PICRC2);
31   return;

     /* ---- unexpected error during APL2 function execution ---- */
32   ERROR2:
33   display('Unexpected error during APL2 function execution');
34   PICRC1 = RETCODE(1);
35   PICRC2 = RETCODE(2);
36   display('Return Code:' || PICRC1 || PICRC2);
37   goto SHUTDOWN;

38   end; /* PLI2APL */
```

Figure 12. PL/I program demonstrating use of the 'APLF' request

Concluding Remarks

The APL2PI interface allows applications written in compiled languages to be extended and enhanced with routines written in APL. A wide variety of uses and benefits can be envisaged for such hybrid applications:

- applications written in languages which do not provide sophisticated numerical computational facilities (e.g.: COBOL, C) can be enhanced by exploiting APL's power in the area of numerical computation and vector processing;
- those portions of application which involve complex or changing algorithms might be better or more productively implemented in APL;
- applications can be prototyped by initially implementing large portions of them in APL, capitalizing on the inherent productivity of APL during the application design and implementation phases;
- APL's powerful interactive capabilities can be exploited by applications in which human interaction is an important component. More than just an interactive interface, APL offers an interactive computational facility which can be used to substantially enhance compiled applications;
- APL offers distinct benefits for applications which require substantial and frequent changes. Typically, APL applications, or those sections of applications written in APL can be modified or enhanced much more quickly and at lower cost than applications or routines written in other languages. By implementing those sections of an application that are most subject to change in APL, the developer can benefit from these characteristics of APL, while retaining the advantages of high level languages for other sections of the application.

Appendix A. Implementation Details

The APL2PI interface consists of a complex set of routines which can be used in a variety of ways to allow APL and non-APL applications to interact and benefit from each other's strengths.

The purpose of this paper has been to provide documentation on the interfaces to and from APL2PI and examples of their use. For many applications, documentation at this level will be entirely sufficient. In more sophisticated applications, however, it may be useful to understand some of the inner workings of the APL2PI facilities. It is the objective of this appendix to provide a first glimpse of these inner workings.

The APL2PI interface is comprised of two major sets of routines:

1. A set of routines written in Assembler language which provide the interfaces used by the non-APL application. The main routine in this set is APL2PI (contained in the AP2VAPI object module in VM/CMS, and in the AP2TAPI object module in MVS/TSO). APL2PI is the entry point to which control is passed from the non-APL application when any request is made to the interface.
2. A set of APL external functions delivered in an APL packaged workspace which provide the interfaces used by APL routines or the APL user when communicating with the non-APL application. The two important functions in this set are ***APL2PI*** and ***APL2PIE***. Note that this ***APL2PI*** external function is not the same as the APL2PI routine used by non-APL application programs. The ***APL2PI*** external function is simply an niladic cover function for the ***APL2PIE*** external function. Its use will be described in more detail below.

The non-APL application which uses the APL2PI interface can be invoked independently or from an active APL session. If invoked independently, the non-APL application causes APL invocation to occur on the first call to the APL2PI interface (typically an 'INIT' call). To invoke a non-APL application from an active APL session, the APL user makes use of the ***APL2PIE*** external function. This external function activates the APL2PI interface and through it causes the non-APL application to be invoked. Once the non-APL application is so initialized, it can call the already active APL2PI interface to make requests to the pendant APL session.

It is possible for both modes of operation to be used together. For example, a non-APL application could be invoked using appropriate VM/CMS or MVS/TSO commands, and that application could use APL2PI to cause APL to be invoked and to submit requests to it. One or more of those requests could cause one or more non-APL applications to be activated from the APL environment. All of these non-APL applications could interact using the facilities provided with the APL2PI module and the ***APL2PIE*** external function. Note, however, that at any given time only one application (APL or non-APL) is running -- all of the other applications are in a pendent state.

Non-APL applications invoked independently or from an active APL environment are often mainline programs written in a high level language. Invocation of such high level language mainline programs typically cause a programming environment to be established. Thus when a FORTRAN program is invoked, the VS FORTRAN programming environment is established to support its execution; when a COBOL program is invoked the VS COBOL II programming environment is established to support its execution. Care must be taken when more than one non-APL application programs is activated. Certain languages or versions of languages do not support more than one instance of the programming environment at any given time. Thus, if one non-APL application written in COBOL is activated, it may not be possible to activate a second one written in the same language, because the second instance of its programming environment would interact destructively with the first instance.

Invoking APL from a non-APL Application

When a non-APL application, invoked and running independently of APL, wishes to access APL facilities, it does so through the APL2PI interface. The APL2PI routine (in the AP2VAPI object module for VM CMS or the AP2TAPI object module for MVS/TSO) can be link edited with the non-APL application, or it can be dynamically loaded (e.g.: via a LOAD macro, or SVC 8), or dynamically accessed (e.g.: as a VM CMS nucleus extension) by the non-APL application. Once the APL2PI routine is available to the non-APL application, the non-APL application makes requests by transferring control to APL2PI using the standard OS CALL protocol described earlier in this paper.

The first call typically issued by the non-APL application is an 'INIT' call to request initialization of APL2. If 'INIT' is not the first call made, the APL2PI interface recognizes that APL2 has not yet been invoked and automatically issues the equivalent of an 'INIT' call with default APL2 initialization parameters. Whether this 'INIT' call is issued implicitly or explicitly, APL2PI then causes APL2 to be invoked. In the VM/CMS environment, this is done by issuing a CMS SVC 202; in MVS/TSO, the APL2 load module is loaded by the APL2PI routine, a CPPL control block is created and control is passed to the APL2 module with R1 pointing to the CPPL which describes the arguments to the command. Thus, it looks to the APL2 module as if it had been invoked from a VM/CMS or MVS/TSO command line.

Before invoking APL2, the APL2PI routine will identify its own entry point making it visible in the address space or virtual machine. This is done by issuing an IDENTIFY macro in MVS/TSO or by establishing APL2PI as a CMS nucleus extension in the VM/CMS environment. This is an important step that will allow APL2 to find its way back to the APL2PI routine after APL2 invocation is complete.

The typical command used to invoke APL2 will contain the arguments QUIET and RUN(APL2PI). The QUIET argument suppresses display of the APL2 greeting message and the interaction associated with the RUN(APL2PI) argument. Its use is not essential to the proper operation of the APL2PI interface, but it is recommended since it leads to less confusing interaction in most cases. Once the APL2PI interface is initialized, QUIET will be turned off, so that APL2 output will be displayed normally.

The RUN(APL2PI) argument in the APL2 invocation command is not optional and is required for proper operation of the interface. It causes the ***APL2PI*** external function (in the AP2PAPIW packaged workspace supplied with APL2) to be executed. As noted above, ***APL2PI*** is simply a cover function for ***APL2PIE*** `0 ''`. Execution of the ***APL2PI*** external function (i.e.: ***APL2PIE*** `0 ''`) causes some housekeeping to be performed in the AP2PAPIW package workspace, a ***⎕NA*** to be issued to the APL2PI external routine, and control to be passed to that routine.

Here things become a little confusing, because there is an ***APL2PI*** external function in the AP2PAPIW packaged workspace and an external routine named APL2PI which is, in fact, the same APL2PI entry point that is accessed by the non-APL application. Here is what happens: the ***APL2PI*** external function in the AP2PAPIW packaged workspace executes ***APL2PIE*** `0 ''`. ***APL2PIE*** issues a `0 11 ⎕NA 'APL2PIX'`. In the NAMES file ***APL2PIX*** is defined to be the APL2PI entry point. Since the APL2PI entry point was IDENTIFY'd or made a CMS nucleus extension earlier, this ***⎕NA*** will set up a link between the ***APL2PIE*** external function and the APL2PI routine. Once this link has been established, ***APL2PIE*** calls the APL2PI routine. This call causes the APL2 session to be suspended awaiting the completion of the APL2PI external routine, and control to be passed to APL2PI.

At this point, APL2PI recognizes that the invocation of APL2 is complete and passes control back to the non-APL application if an explicit 'INIT' call was issued, or proceeds with the non-APL applications request if the 'INIT' call was implicitly issued.

This initialization process may be clearer if considered as an ordered set of events:

1. The non-APL application is activated by the user.

2. The non-APL application calls APL2PI with an 'INIT' request.
3. APL2PI invokes APL2 with options that include RUN(APL2PI).
4. The RUN(APL2PI) option causes the ***APL2PI*** external function in packaged workspace AP2PAPIW to be run when APL2 invocation is complete.
5. ***APL2PI*** calls the external function ***APL2PIE*** in the same packaged workspace.
6. ***APL2PIE*** calls the external routine APL2PI, which is, in fact, the same routine that was called in step 2 by the non-APL application.
7. When the APL2PI routine receives control from ***APL2PIE***, it realizes that APL2 initialization is complete, and it returns control to the non-APL application that called it in step 2.

When and if the non-APL application issues a request other than 'INIT' or 'TERM' to the APL2PI interface, APL2 will be in a state where it is awaiting the completion of execution of the APL2PI external routine. 'APLE', 'APLS', 'APLF' and 'APLV' requests are completed by APL2PI using "callback" requests to APL2, i.e., a combination of 'XE' and 'XF' service calls as documented in the APL2 Programming: Processor Interface Reference manual (SH20-9234). The 'APLP' request is executed by returning control from APL2PI to the ***APL2PIE*** external function which executes the request using local logic and then returns control to the APL2PI routine.

The 'APLX' request causes the APL2PI routine to return control to ***APL2PIE*** which issues a message and returns control to the APL user or application which called it. When the user or APL application subsequently issues an ***APL2PI*** or ***APL2PIE*** `0 ''` request, control returns to ***APL2PIE*** in the AP2PAPIW packaged workspace and from there to the APL2PI routine. Finally, APL2PI sets return codes and return values appropriately and returns control to the non-APL application which called it.

When the non-APL application issues a 'TERM' request to APL2PI, the APL2PI routine returns control to ***APL2PIE*** which stacks an ***)OFF*** command and returns to its caller. When the ***)OFF*** command is executed, it causes APL to be terminated and control to be returned to the routine that originally invoked APL i.e., APL2PI. The APL2PI routine cleans up and deallocates its own resources and returns control to the non-APL application.

Invoking a non-APL Application from APL

A non-APL application can be invoked from the APL environment using the ***APL2PIE*** external function, viz.:

```
0 11 ⎕NA 'APL2PIE'
'ROUTINE ARGUMENTS' APL2PIE 1 'NAME'
```

When ***APL2PIE*** receives this request, it calls the external routine APL2PI with a request to activate the specified non-APL application routine. APL2PI initializes the interface and then loads and calls the non-APL application routine.

In the VM/CMS environment, APL2PI first looks for an existing CMS nucleus extension whose name matches that of the specified routine. If one is found, its address is used as the entry point address for the non-APL application. If no matching CMS nucleus extension is found, APL2PI issues a CMS NUCXLOAD for a relocatable CMS module with the specified name. If that NUCXLOAD command is successful, the address of the loaded routine is used as the entry point address of the non-APL application. If the NUCXLOAD fails, control will be returned to ***APL2PIE*** and then to the APL caller with a return code indicating that the routine could not be found.

If the specified routine is found as an existing CMS nucleus extension or is successfully loaded as a result of the NUCXLOAD command, APL2PI builds a parameter list matching CMS SVC 202 conventions and enters the non-APL application.

In the MVS TSO environment, APL2PI issues a LOAD (SVC 8) for the specified routine, constructs a parameter list in CPPL format, and enters the non-APL application.

When the non-APL application receives control from APL2PI, it can issue calls to APL2PI to make service requests. The first request issued should be an 'INIT' request. Although APL and APL2PI have already been initialized, this 'INIT' request allows the non-APL application to identify itself by name to the APL2PI interface, and it allows a service routine to be specified if desired.

Other APL2PI requests can be subsequently issued by the non-APL application. APL2PI processes these requests appropriately, invoking APL services as necessary, and when the request is completed, it returns control to the non-APL application. The 'APLE', 'APLS', 'APLF' and 'APLV' requests are completed by APL2PI using "callback" requests to APL2, i.e.: a combination of 'XE' and 'XI' service calls. The 'APLP' request is executed by returning control from APL2PI to the pendant ***APL2PIE*** external function. ***APL2PIE*** processes the request and returns control to APL2PI.

The 'APLX' request causes the APL2PI routine to return control to the pendant ***APL2PIE*** external function and from it to the APL application or user that originally called it. When the APL user or application subsequently issues an ***APL2PI*** or ***APL2PIE*** `0 ''` request, control is returned to the ***APL2PIE*** external function and from there to the APL2PI external routine. Finally, APL2PI sets return codes and return values appropriately and returns control to the non-APL application which calls it.

Before terminating, the non-APL application should issue a 'TERM' request to APL2PI. When this request is received, the non-APL application is deleted from APL2PI internal tables, and control is returned to that application. The application is then free to terminate. When it does so, control is returned to APL2PI since APL2PI originally invoked the non-APL application. APL2PI in turn returns control to the ***APL2PIE*** external function that called it. Finally, ***APL2PIE*** returns control to the APL application or APL user that last called it.

Environment Isolation

APL2, when it is invoked, establishes STAE and STAX exits so that asynchronous or unexpected events (attention signals, program checks, ABENDs, etc.) are captured and properly handled. Many non-APL applications, particularly those written in high level languages, need to do much the same thing. When cooperating APL and non-APL applications are run via the APL2PI interface, APL2PI must take care to keep the APL and non-APL environments separate so that things like SPIE, STAE and STAX exits do not interact or cancel each other out, and so that each application is properly notified of events appropriate to it.

In the MVS/TSO environment, this is done by using MVS task isolation. APL and each non-APL application is established as a separate MVS task, and APL2PI activates and deactivates the appropriate tasks as control flows between APL and a non-APL application. Since each MVS task may have its own set of SPIE, STAE, and STAX exits, no conflicts exist between APL and any of the non-APL applications, and events are directed to the task that is currently active.

In the VM/CMS environment, multitasking facilities, and therefore task isolation, are not available. In order to provide the necessary isolation, therefore, APL2PI manages the boundary crossing between APL and non-APL applications. As control flows through APL2PI between the non-APL and APL applications, APL2PI saves the SPIE, STAE and STAX information for the application giving up control and reestablishes the SPIE, STAE, and STAX information for the application to which control is being passed.

There is another aspect of the implementation in the VM/CMS environment that deserves mention. In CMS when a command is issued (by the user or by an application program using SVC 202), an SVC level is added to the CMS SVC save area chain. Typically this is entirely transparent to the user or application program and is simply a detail in the internal operation of CMS. The important part for APL2PI users is that APL2 is dependent on running all of its operations at the same CMS SVC level. Thus APL2 would ABEND if it passed control to an external routine which changed the SVC level and returned to APL2. For this reason, when a non-APL application is invoked from an active APL environment using ***APL2PIE***, APL2PI simulates the operation an linkage of SVC 202 rather than simply issuing an SVC 202 to invoke the non-APL application. In this way, the non-APL application and APL both operate at the same CMS SVC level as they pass control back and forth between each other.

There are many cases in CMS where applications can issue commands and thereby cause additional levels to be added to the SVC chain. For example, a REXX application could call an XEDIT session which in turn could call an XEDIT macro which could execute CMS commands. Each of these calls would introduce another SVC level to the SVC chain. If such an application were to make calls to APL2PI, all calls would have to occur at the same SVC level. Further, if that application was invoked from APL via ***APL2PIE***, all calls to APL2PI would have to be made from the SVC level at which the application was invoked. If these rules are not followed, APL2 will ABEND when control is returned to it at an invalid SVC level.

Termination Processing

A well behaved non-APL application issues an 'INIT' request to APL2PI as its first request and a 'TERM' request before it terminates execution. Further, well behaved APL users or APL applications which invoke non-APL applications via ***APL2PIE*** ensure that those non-APL applications are terminated before the APL session is terminated with an ***)OFF*** command.

Unfortunately, it is often difficult to ensure that APL users or non-APL applications are always well behaved. APL2PI, therefore, takes certain precautions to ensure orderly shutdown in unusual situations.

When APL2PI is first invoked, as part of its initialization, it tells APL that it needs to be notified of APL termination. Thus, when an ***)OFF*** command is issued from the APL environment, APL2PI receives control. If any non-APL applications invoked from the APL environment, or any service routines for non-APL applications are pendant at that time, they are sent a 'shutdown' request (in the form of a 0 2 return code) from APL2PI. Those applications are expected to recognize this return code, complete their processing and return control to there callers immediately. In each of these cases, APL2PI is the caller, and thus APL2PI is able to determine when all of these non-APL applications and service routines have shut down. When that occurs, APL2PI frees its own resources and returns control to APL which then terminates gracefully.

Note that if APL was invoked from a non-APL application, that non-APL application is not notified by APL2PI (via a 0 2 return code) during APL ***)OFF*** processing. Instead, APL termination proceeds normally, and when complete, control is returned to APL2PI because APL2PI originally invoked APL. When that occurs, APL2PI frees its own resources and returns control to the non-APL application which originally invoked it. The non-APL application will be given a successful return code of 0 0 if the termination occurred as a result of 'TERM' request that it made; otherwise, it will be given a 0 2 (unexpected termination).