CALL

CALL [**INTERFACE4**] *operand1* [**USING**] [*operand2*] ... 128

This chapter covers the following topics:

- Function
- Syntax Description
- Return Code
- Register Usage
- Storage Alignment
- Adabas Calls
- Direct/Dynamic Loading
- Linkage Conventions
- Calling a PL/I Program
- Calling a C Program
- INTERFACE4

For an explanation of the symbols used in the syntax diagram, see *Syntax Symbols*.

Related Statements: CALL FILE | CALL LOOP | CALLNAT | DEFINE SUBROUTINE | ESCAPE | FETCH | PERFORM

Belongs to Function Group: *Invoking Programs and Routines*

Function

The CALL statement is used to call an external program written in another standard programming language from a Natural program and then return to the next statement after the CALL statement.

The called program may be written in any programming language which supports a standard CALL interface. Multiple CALL statements to one or more external programs may be specified.

A CALL statement may be issued within a program to be executed under control of a TP monitor, provided that the TP monitor supports a CALL interface.

Syntax Description

Operand Definition Table:

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
operand1	C	S				A												yes	no
operand2	C	S	A	G		A	U	N	P	I	F	B	D	T	L	C	G	yes	yes

Syntax Element Description:

INTERFACE4	The optional keyword `INTERFACE4` specifies the type of the interface that is used for the call of the external program. See the section *INTERFACE4* below.
operand1	**Program Name:** The name of the program to be called (*operand1*) can be specified as a constant or - if different programs are to be called dependent on program logic - as an alphanumeric variable of length 1 to 8. A program name must be placed left-justified in the variable.
[USING] ***operand2***	**Parameters:** The `CALL` statement may contain up to 128 parameters (*operand2*), unless the `INTERFACE4` option is used. In that case, the number of parameters is limited by the size of the cataloged object. Depending on all other statements in the Natural object, up to 16370 parameters may be used. Standard linkage register conventions are used. One address is passed in the parameter list for each parameter field specified. If a group name is used, the group is converted to individual fields; that is, if a user wishes to specify the beginning address of a group, the first field of the group must be specified. **Note:** The internal representation of positive signs of packed numbers is changed to the value specified by the `PSIGNF` parameter of the `NTCMPO` macro (Compilation Options) *before* control is passed to the external program.

Return Code

The condition code of any called program (content of Register 15 upon return to Natural) may be obtained by using the Natural system function `RET` (Return Code Function).

Example:

```
...
RESET #RETURN(B4)
CALL 'PROG1'
```

```
IF RET ('PROG1') > #RETURN
  WRITE 'ERROR OCCURRED IN PROGRAM1'
END-IF
...
```

Register Usage

Register	Contents
R1	Address pointer to the parameter address list.
R2	Address pointer to the field (parameter) description list. The field description list contains information on the first 128 fields passed in the parameter list. Each description is a 4-byte entry containing the following information: • the 1st byte contains the type of variable (A,B,...); • if a variable of type A exceeds a size of 32767 bytes, it is passed as type Y; • if a variable of type B exceeds a size of 32767 bytes, it is passed as type X; the types X and Y have been introduced to support long alpha and binary variables with the standard CALL interface. If field type is N or P: • the 2nd byte contains the total number of digits; • the 3rd byte contains the number of digits before the decimal point; • the 4th byte contains the number of digits after the decimal point. If field type is X or Y: • the 2nd byte is unused; • the 3rd-4th byte contain zero; • the length of the field is passed via R4. All other field types: • the 2nd byte is unused; • the 3rd-4th byte contain the length of field.
R3	Address pointer to list of field lengths. Each length field is a 4-byte entry containing the length of each field passed in the parameter list. In the case of an array, the length is the sum of the individual occurrences' lengths.
R4	Only for type X and Y: • a 4-byte long entry for each variable of type A or B that exceeds the size of 32767 bytes.
R13	Address of 18-word save area.

Register	Contents
R14	Return address.
R15	Entry address/return code.

Storage Alignment

See the section *Storage Alignment* in the *Programming Guide*.

Adabas Calls

A called program may contain a call to Adabas. The called program must not issue an Adabas open or close command. Adabas will open all database files referenced.

If Adabas exclusive (EXU) update mode is to be used, the Natural profile parameter `OPRB` (Database Open/Close Processing) must be used in order to open all referenced files. Before you attempt to use EXU update mode, you should consult your Natural administrator.

If a called program issues Adabas commands that begin or end a transaction, Natural will not be able to recognize the change of the transaction status.

Calls to Adabas must comply with the calling conventions for the Adabas application programming interface (API) for the respective TP monitor or operating system. This applies also if Natural is acting as a server, e.g. under z/OS or SMARTS.

Direct/Dynamic Loading

The called program may either be directly linked to the Natural nucleus (that is, the program is specified with the profile parameter `CSTATIC` (Programs Statically Linked to Natural) in the Natural parameter module in the *Operations* documentation, or it may be loaded dynamically the first time it is called.

If it is to be loaded dynamically, the load module library containing the called program must be concatenated to the Natural load library in the Natural execution JCL or in the appropriate TP-monitor program library. Ask your Natural administrator for additional information.

Example:

The example below shows a Natural program which calls the COBOL program TABSUB for the purpose of converting a country code into the corresponding country name. Two parameter fields are passed by the Natural program to TABSUB:

- the first parameter is the country code, as read from the database;
- the second parameter is used to return the corresponding country name.

Calling Natural Program:

```
** Example 'CALEX1': CALL PROGRAM 'TABSUB'
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
```

```
  2 NAME
  2 BIRTH
  2 COUNTRY
*
1 #COUNTRY      (A3)
1 #COUNTRY-NAME (A15)
1 #FIND-FROM    (D)
1 #FIND-TO      (D)
END-DEFINE
*
MOVE EDITED '19550701' TO #FIND-FROM (EM=YYYYMMDD)
MOVE EDITED '19550731' TO #FIND-TO   (EM=YYYYMMDD)
*
FIND EMPLOY-VIEW WITH BIRTH = #FIND-FROM THRU #FIND-TO
  MOVE COUNTRY TO #COUNTRY
  /*
  CALL 'TABSUB' #COUNTRY #COUNTRY-NAME
  /*
  DISPLAY NAME BIRTH (EM=YYYY-MM-DD) #COUNTRY-NAME
END-FIND
END
```

Called COBOL program TABSUB:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TABSUB.
REMARKS. THIS PROGRAM PROVIDES THE COUNTRY NAME
        FOR A GIVEN COUNTRY CODE.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
01 COUNTRY-CODE  PIC X(3).
01 COUNTRY-NAME  PIC X(15).
PROCEDURE DIVISION USING COUNTRY-CODE COUNTRY-NAME.
P-CONVERT.
   MOVE SPACES TO COUNTRY-NAME.
   IF COUNTRY-CODE = 'BLG' MOVE 'BELGIUM' TO COUNTRY-NAME.
   IF COUNTRY-CODE = 'DEN' MOVE 'DENMARK' TO COUNTRY-NAME.
   IF COUNTRY-CODE = 'FRA' MOVE 'FRANCE' TO COUNTRY-NAME.
   IF COUNTRY-CODE = 'GER' MOVE 'GERMANY' TO COUNTRY-NAME.
   IF COUNTRY-CODE = 'HOL' MOVE 'HOLLAND' TO COUNTRY-NAME.
   IF COUNTRY-CODE = 'ITA' MOVE 'ITALY' TO COUNTRY-NAME.
   IF COUNTRY-CODE = 'SPA' MOVE 'SPAIN' TO COUNTRY-NAME.
   IF COUNTRY-CODE = 'UK'  MOVE 'UNITED KINGDOM' TO COUNTRY-NAME.
P-RETURN.
GOBACK.
```

Linkage Conventions

Standard linkage register notation is used in batch mode. Each TP monitor has its own conventions. These conventions must be followed; otherwise, unpredictable results could occur.

The following sections describe conventions that apply for the supported TP monitors.

- CALL Using Com-plete

- CALL Using CICS

CALL Using Com-plete

The called program must reside in the Com-plete online load library. This allows Com-plete to load the program dynamically. The Com-plete utility ULIB may be used to catalog the program.

CALL Using CICS

The called program must reside in either a load module library concatenated to the CICS library or the DFHRPL library. The program must also have a PPT entry in the operating PPT so that CICS can locate the program and load it.

The linkage convention passes the parameter list address followed by the field description list address in the first fullwords of the TWA and the COMMAREA.

The parameter `FLDLEN` in the NCIPARM parameter module controls if the field length list is also passed (by default, it is not passed). The COMMAREA length (0, 8, 12 or 16) reflects the number of list addresses passed (0, 2, 3 or 4). The last list address is indicated by the high-order bit being set. The user must ensure addressability to the TWA or to the COMMAREA respectively. This is only required if the user program has not been defined to Natural as a static or directly linked program, in which case the pointer to the parameter list is passed via Register 1, the pointer to the description list via Register 2, the pointer to the field length list via Register 3, and the pointer to the large field length list is passed in Register 4.

The parameters `FLDLEN` and `COMACAL` in the NCIPARM parameter module control the contents of the COMMAREA.

If you wish the parameter values themselves, rather than the address of their address list, to be passed in the COMMAREA, issue the Natural (call options) terminal command `%P=C` before the call.

Normally, when a Natural programs calls a non-Natural program and the called program issues a conversational terminal I/O, the Natural thread is blocked until the user has entered data. To prevent the Natural thread from being blocked, the terminal command `%P=V` can be used

Normally, when a Natural program calls a non-Natural program under CICS, the call is accomplished by an `EXEC CICS LINK` request. If standard linkage is to be used for the call instead, issue the terminal command `%P=S`. (In this case, the called program must adhere to standard linkage conventions with standard register usage).

In 31-bit-mode environments the following applies: if a program linked with `AMODE=24` is called and the threads are above 16 MB, a "call by value" will be done automatically, that is, the specified parameters which are to be passed to the called program will be copied below the 16 MB memory line.

Return Codes under CICS

CICS itself does not support return codes for a call with CICS conventions (`EXEC CICS LINK`), with the exception of calling C/C++ programs where values passed by the `exit()` function or the `return()` statement are saved in the EIBRESP2 field. However, the Natural CICS Interface supports return codes for the `CALL` statement: When control is returned from the called program, Natural first checks the EIBRESP2 field for a non-zero return code. Then Natural checks whether the first fullword of the COMMAREA has changed. If it has, its new content will be taken as the return code. If it has not

changed, the first fullword of the TWA will be checked and its new content taken as the return code. If neither of the two fullwords has changed, the return code will be `0`.

Note:
When parameter values are passed in the COMMAREA (`%P=C`), only EIBRESP2 field is checked for a return code; that is, for non-C/C++ programs the return code is always `0`.

Example Using CICS:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TABSUB.
REMARKS. THIS PROGRAM PERFORMS A TABLE LOOK-UP AND
         RETURNS A TEXT MESSAGE.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 MSG-TABLE.
   03  FILLER       PIC X(15) VALUE 'MESSAGE1       '.
   03  FILLER       PIC X(15) VALUE 'MESSAGE2       '.
   03  FILLER       PIC X(15) VALUE 'MESSAGE3       '.
   03  FILLER       PIC X(15) VALUE 'MESSAGE4       '.
   03  FILLER       PIC X(15) VALUE 'MESSAGE5       '.
   03  FILLER       PIC X(15) VALUE 'MESSAGE6       '.
   03  FILLER       PIC X(15) VALUE 'MESSAGE7       '.
01 TAB REDEFINES MSG-TABLE.
   03  MESSAGE OCCURS 7 TIMES PIC X(15).
LINKAGE SECTION.
01 TWA-DATA.
   03 PARM-POINTER USAGE IS POINTER.
01 PARM-LIST.
   03 DATA-LOC-IN  USAGE IS POINTER.
   03 DATA-LOC-OUT USAGE IS POINTER.
01 INPUT-DATA.
   03  INPUT-NUMBER      PIC 99.
01 OUTPUT-DATA.
   03  OUTPUT-MESSAGE    PIC X(15).
PROCEDURE DIVISION.
100-INIT.
    EXEC CICS ADDRESS TWA(ADDRESS OF TWA-DATA) END-EXEC.
    SET ADDRESS OF PARM-LIST   TO PARM-POINTER.
    SET ADDRESS OF INPUT-DATA  TO DATA-LOCIN.
    SET ADDRESS OF OUTPUT-DATA TO DATA-LOC-OUT.
200-PROCESS.
    MOVE MESSAGE (INPUT-NUMBER) TO OUTPUT-MESSAGE.
300-RETURN.
    EXEC CICS RETURN END-EXEC.
400-DUMMY.
    GO-BACK.
```

Calling a PL/I Program

A called program written in PL/I requires the following additional procedure:

- Since the parameter list is a standard list and is not an argument list being passed from another PL/I program, the addresses passed do not point at a `LOCATOR DESCRIPTOR`. This problem may be resolved by defining the parameter fields as arithmetic variables. This causes PL/I to treat the parameter list as addresses of data instead of addresses of `LOCATOR DESCRIPTOR` control blocks.

The technique suggested for defining the parameter fields is illustrated in the following example:

```
PLIPROG: PROC(INPUT_PARM_1, INPUT_PARM_2) OPTIONS(MAIN);
      DECLARE (INPUT_PARM_1, INPUT_PARM_2) FIXED;
      PTR_PARM_1 = ADDR(INPUT_PARM_1);
      PTR_PARM_2 = ADDR(INPUT_PARM_2);
      DECLARE FIRST_PARM        PIC '99'   BASED (PTR_PARM_1);
      DECLARE SECOND_PARM       CHAR(12)   BASED (PTR_PARM_2);
```

Each parameter in the input list should be treated as a unique element. The number of input parameters should exactly match the number being passed from the Natural program. The input parameters and their attributes must match the Natural definitions or unpredictable results may occur. For additional information on passing parameters in PL/I, see the relevant IBM PL/I documentation.

The following topics are covered below:

- Example of Calling a PL/I Program
- Example of Calling a PL/I Program which is Operating under CICS

Example of Calling a PL/I Program

```
** Example 'CALEX2': CALL PROGRAM 'NATPLI'
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 AREA-CODE
  2 REDEFINE AREA-CODE
    3 #AC         (N1)
*
1 #INPUT-NUMBER   (N2)
1 #OUTPUT-COMMENT (A15)
END-DEFINE
*
READ EMPLOY-VIEW IN LOGICAL SEQUENCE BY NAME
                 STARTING FROM 'WAGNER'
  MOVE ' ' TO #OUTPUT-COMMENT
  MOVE #AC TO #INPUT-NUMBER
  /*
  CALL 'NATPLI' #INPUT-NUMBER #OUTPUT-COMMENT
  /*
END-READ
*
END
```

Called PL/I program NATPLI:

```
NATPLI:  PROC(PARM_COUNT, PARM_COMMENT) OPTIONS(MAIN);
    /*                                              */
    /* THIS PROGRAM ACCEPTS AN INPUT NUMBER         */
    /* AND TRANSLATES IT TO AN OUTPUT CHARACTER     */
    /* STRING FOR PLACEMENT ON THE FINAL            */
    /* NATURAL REPORT                               */
    /*                                              */
    /*                                              */
   DECLARE PARM_COUNT, PARM_COMMENT  FIXED;
   DECLARE ADDR BUILTIN;
   COUNT_PTR = ADDR(PARM_COUNT);
   COMMENT_PTR = ADDR(PARM_COMMENT);
```

```
   DECLARE  INPUT_NUMBER    PIC '99' BASED (COUNT_PTR);
   DECLARE  OUTPUT_COMMENT  CHAR(15) BASED (COMMENT_PTR);
   DECLARE COMMENT_TABLE(9) CHAR(15) STATIC INITIAL
      ('COMMENT1   ',
       'COMMENT2   ',
       'COMMENT3   ',
       'COMMENT4   ',
       'COMMENT5   ',
       'COMMENT6   ',
       'COMMENT7   ',
       'COMMENT8   ',
       'COMMENT9   ');
    OUTPUT_COMMENT = COMMENT_TABLE(INPUT_NUMBER);
    RETURN;
END NATPLI;
```

Example of Calling a PL/I Program which is Operating under CICS

```
** Example 'CALEX3': CALL PROGRAM 'CICSP'
************************************************************************
DEFINE DATA LOCAL
1 #MESSAGE (A10) INIT <' '>
END-DEFINE
*
CALL 'CICSP' #MESSAGE
DISPLAY #MESSAGE
*
END
```

Called PL/I program CICSP:

```
CICSP: PROCEDURE OPTIONS (MAIN REENTRANT);
       DCL  1        TWA_ADDRESS    BASED(TWA_POINTER);
            2        LIST_ADDRESS   POINTER;
       DCL  1 PTR_TO_LIST           BASED(LIST_ADDRESS);
            2 PARM_01               POINTER;
       DCL MESSAGE CHAR(10) BASED(PARM_01);
       EXEC CICS ADDRESS TWA(TWA_POINTER);
       MESSAGE='SUCCESS'; EXEC CICS RETURN; END CICSP;
```

Calling a C Program

Before using a C program, you need to compile and link it.

- Use for instance IBM's C compiler to build the executable module. Since IBM's C compiler produces LE code, the sample is only executable in an LE environment. To execute LE programs, the Natural front-end needs to be installed LE enabled.
- If you intend to use any other C compiler, such as Dignus or SASC, you need to build a module which is callable from a non-C environment. Refer to the appropriate compiler documentation for further information.
- The include file NATUSER needs to be included in the C program.

C programs written for INTERFACE4 can be used on Mainframe systems as well as on UNIX, OpenVMS or Windows systems. Whereas C programs, written for the standard Call Interface, are platform-dependent.

If it is intended to call the C Program via `CALL INTERFACE4` or if a Natural subprogram is called from the C Program, NATXCAL4 needs to be linked to the executable module. Use one of the INTERFACE4 Call Back Functions to retrieve the parameter description and parameter values. The Call Back Functions are described below.

Use function ncxr_if4_callnat, to execute a Natural subprogram from the C program.

Prototype:

```
int ncxr_if4_callnat ( char *natpgm, int parmnum, struct parameter_description *descr );
```

Parameter description:

natpgm	Name of the Natural subprogram to be invoked.	
parmnum	Number of parameter fields to be passed to the subprogram.	
descr	Address of a `struct parameter_description`. See *Operand Structure for INTERFACE4* for a detailed description of this structure.	
return	Return Value:	Information:
	0	OK If an Natural error occurs while the subprogram is executed, information about this error will be returned in the variable `natpgm` in the form `*NAT` *nnnn*, where *nnnn* is the corresponding Natural error number.
	-1	Illegal parameter number.
	-2	Internal error.

The following topics are covered below:

- Example of Calling a C Program via Standard CALL
- Example of Calling a C Program via CALL INTERFACE4

Example of Calling a C Program via Standard CALL

```
** Example 'CALEX4': CALL PROGRAM 'ADD'
************************************************************************
DEFINE DATA LOCAL
1 #OP1  (I4)
1 #OP2  (I4)
1 #SUM  (I4)
END-DEFINE
*
CALL 'ADD' #OP1 #OP2 #SUM
DISPLAY #SUM
*
END
```

Called C program ADD:

```
/*
**  Example C Program ADD.c
*/
NATFCT ADD (int *op1, int *op2, int *sum)
{
*sum = *op1 + *op2;    /* add opperands */

return 0;              /* return successfully */
} /* ADD */
```

Example of Calling a C Program via CALL INTERFACE4

```
** Example 'CALEX5': CALL PROGRAM 'ADD4'
************************************************************************
DEFINE DATA LOCAL
1 #OP1  (I4)
1 #OP2  (I4)
1 #SUM  (I4)
END-DEFINE
*
CALL INTERFACE4 'ADD4' #OP1 #OP2 #SUM
DISPLAY #SUM
*
END
```

Called C program ADD4:

```
NATFCT ADD4 NATARGDEF(numparm, parmhandle, parmdec)
{
NATTYP_I4 op1, op2, sum;              /* local integers */
int i;                                /* loop counter */
struct parameter_description desc;
int rc;                               /* return code access functions */

/*
** test number of arguments
*/
if (numparm != 3) return 1;

/*
** test types of arguments
*/
for (i = 0; i < (int) numparm; i++)
{
       rc = ncxr_get_parm_info( i, parmhandle, &desc );
      if ( rc ) return rc;

      if ( desc.format != 'I' || desc.length != sizeof(NATTYP_I4) || desc.dimensions != 0 )
      {
              return 2;          /* invalid parameter */
      }
}

/*
** get arguments
*/
rc = ncxr_get_parm( 0, parmhandle, sizeof op1, (void *)&op1 );
if ( rc ) return rc;

rc = ncxr_get_parm( 1, parmhandle, sizeof op2, (void *)&op2 );
if ( rc ) return rc;

/*
```

```
** perform the addition
*/
sum = op1 + op2;

/*
** move the result back to operand 3
*/
rc = ncxr_put_parm( 2, parmhandle, sizeof sum, (void *)&sum );
if ( rc ) return rc;

/*
** all ok, return success to the caller
*/
return 0;
} /* ADD4 */
```

INTERFACE4

The keyword `INTERFACE4` specifies the type of the interface that is used for the call of the external program. This keyword is optional. If this keyword is specified, the interface, which is defined as Interface4, is used for the call of the external program.

The following table lists the differences between the `CALL` statement used with `INTERFACE4` and the one used without `INTERFACE4`:

	CALL statement without keyword INTERFACE4	**CALL statement with keyword INTERFACE4**
Number of parameters possible	128	16370 or less
Maximum data size of one parameter	no restriction	1 GB
Retrieve array information	no	yes
Support of large and dynamic operands	full read access, write without changing size of operand	yes
Parameter access via API	direct	via API

The maximum number of parameters is limited by the maximum size of the generated program (GP) and by the maximum size of a statement. 16370 parameters are possible if the program contains only the `CALL` statement. The maximum number is lower if other statements are used.

The following topics are covered below:

- INTERFACE4 - External 3GL Program Interface
- Operand Structure for INTERFACE4
- INTERFACE4 - Parameter Access
- Exported Functions

INTERFACE4 - External 3GL Program Interface

The interface of the external 3GL program is defined as follows, when `INTERFACE4` is specified with the Natural `CALL` statement:

```
NATFCT functionname (numparm, parmhandle, traditional)
```

USR_WORD	numparm;	16 bit unsigned short value, containing the total number of transferred operands (*operand2*).
void	*parmhandle;	Pointer to the parameter passing structure.
void	*traditional;	Check for interface type (if it is not a NULL pointer it is the traditional CALL interface).

Operand Structure for INTERFACE4

The operand structure of `INTERFACE4` is named "parameter_description" and is defined as follows. The structure is delivered with the header file *natuser.h*.

struct parameter_description		
void *	address	Address of the parameter data, not aligned, `realloc()` and `free()` are not allowed.
int	format	Field data format: NCXR_TYPE_ALPHA, etc. (*natuser.h*).
int	length	Length (before decimal point, if applicable).
int	precision	Length after decimal point (if applicable).
int	byte_length	Length of field in bytes int dimension number of dimensions (0 to IF4_MAX_DIM).
int	dimensions	Number of dimensions (0 to IF4_MAX_DIM).
int	length_all	Total data length of array in bytes.

int	flags	Several flag bits combined by bitwise OR, meaning:	
		IF4_FLG_PROTECTED:	The parameter is write protected.
		IF4_FLG_DYNAMIC:	The parameter is a dynamic variable.
		IF4_FLG_NOT_CONTIGUOUS:	The array elements are not contiguous (have spaces between them).
		IF4_FLG_AIV:	The parameter is an application-independent variable.
		IF4_FLG_DYNVAR:	The parameter is a dynamic variable.
		IF4_FLG_XARRAY:	The parameter is an X-array.
		IF4_FLG_LBVAR_0:	The lower bound of dimension 0 is variable.
		IF4_FLG_UBVAR_0:	The upper bound of dimension 0 is variable.
		IF4_FLG_LBVAR_1:	The lower bound of dimension 1 is variable.
		IF4_FLG_UBVAR_1:	The upper bound of dimension 1 is variable.
		IF4_FLG_LBVAR_2:	The lower bound of dimension 2 is variable.
		IF4_FLG_UBVAR_2:	The upper bound of dimension 2 is variable.
int	occurrences[IF4_MAX_DIM]	Array occurrences in each dimension.	
int	indexfactors[IF4_MAX_DIM]	Array indexfactors for each dimension.	
void *	dynp	Reserved for internal use.	
void *	pops	Reserved for internal use.	

The address element is null for arrays of dynamic variables and for x-arrays. In these cases, the array data cannot be accessed as a whole, but must be accessed through the parameter access functions described below.

For arrays with fixed bounds of variables with fixed length, the array contents can be accessed directly using the address element. In these cases the address of an array element (i,j,k) is computed as follows (especially if the array elements are not contiguous):

```
elementaddress = address + i * indexfactors[0] + j * indexfactors[1] + k * indexfactors[2]
```

If the array has less than 3 dimensions, leave out the last terms.

INTERFACE4 - Parameter Access

A set of functions is available to be used for the access of the parameters. The process flow is as follows:

- The 3GL program is called via the `CALL` statement with the `INTERFACE4` option, and the parameters are passed to the 3GL program as described above.
- The 3GL program can now use the exported functions of Natural, to retrieve either the parameter data itself, or information about the parameter, like format, length, array information, etc.
- The exported functions can also be used to pass back parameter data.

There are also functions to create and initialize a new parameter set in order to call arbitrary subprograms from a 3GL program. With this technique a parameter access is guaranteed to avoid memory overwrites done by the 3GL program. (Natural's data is safe: memory overwrites within the 3GL program's data are still possible).

Exported Functions

The following topics are covered below:

- Get Parameter Information
- Get Parameter Data
- Write Back Operand Data
- Create, Initialize and Delete a Parameter Set
- Create Parameter Set
- Delete Parameter Set
- Initialize a Scalar of a Static Data Type
- Initialize an Array of a Static Data Type
- Initialize a Scalar of a Dynamic Data Type
- Initialize an Array of a Dynamic Data Type
- Resize an X-array Parameter

Get Parameter Information

This function is used by the 3GL program to receive all necessary information from any parameter. This information is returned in the `struct parameter_description`, which is documented above.

Prototype:

```
int ncxr_get_parm_info ( int parmnum, void *parmhandle, struct parameter_description *descr );
```

Parameter Description:

parmnum	Ordinal number of the parameter. This identifies the parameter of the passed parameter list. Range: 0 ... numparm-1.	
parmhandle	Pointer to the internal parameter structure	
descr	Address of a struct parameter_description	
return	Return Value:	Information:
	0	OK
	-1	Illegal parameter number.
	-2	Internal error.
	-7	Interface version conflict.

Get Parameter Data

This function is used by the 3GL program to get the data from any parameter.

Natural identifies the parameter by the given parameter number and writes the parameter data to the given buffer address with the given buffer size.

If the parameter data is longer than the given buffer size, Natural will truncate the data to the given length. The external 3GL program can make use of the function `ncxr_get_parm_info`, to request the length of the parameter data.

There are two functions to get parameter data: `ncxr_get_parm` gets the whole parameter (even if the parameter is an array), whereas `ncxr_get_parm_array` gets the specified array element.

If no memory of the indicated size is allocated for "buffer" by the 3GL program (dynamically or statically), results of the operation are unpredictable. Natural will only check for a null pointer.

If data gets truncated for variables of the type I2/I4/F4/F8 (buffer length not equal to the total parameter length), the results depend on the machine type (little endian/big endian). In some applications, the user exit must be programmed to use no static data to make recursion possible.

Prototypes:

```
int ncxr_get_parm( int parmnum, void *parmhandle, int buffer_length, void *buffer )

int ncxr_get_parm_array( int parmnum, void *parmhandle, int buffer_length, void *buffer, int *indexes )
```

This function is identical to `ncxr_get_parm`, except that the indexes for each dimension can be specified. The indexes for unused dimensions should be specified as 0.

Parameter Description:

parmnum	Ordinal number of the parameter. This identifies the parameter of the passed parameter list. Range: 0 ... numparm-1.	
parmhandle	Pointer to the internal parameter structure	
buffer_length	Length of the buffer, where the requested data has to be written to	
buffer	Address of buffer, where the requested data has to be written to. This buffer should be aligned to allow easy access to I2/I4/F4/F8 variables.	
indexes	Array with index information	
return	Return Value:	Information:
	< 0	Error during retrieval of the information:
	-1	Illegal parameter number.
	-2	Internal error.
	-3	Data has been truncated.
	-4	Data is not an array.
	-7	Interface version conflict.
	-100	Index for dimension 0 is out of range.
	-101	Index for dimension 1 is out of range.
	-102	Index for dimension 2 is out of range.
	0	Successful operation.
	> 0	Successful operation, but the data was only this number of bytes long (buffer was longer than the data).

Write Back Operand Data

These functions are used by the 3GL program to write back the data to any parameter. Natural identifies the parameter by the given parameter number and writes the parameter data from the given buffer address with the given buffer size to the parameter data. If the parameter data is shorter than the given buffer size, the data will be truncated to the parameters data length, i.e., the rest of the buffer will be ignored. If the parameter data is longer than the given buffer size, the data will be copied only to the given buffer length, the rest of the parameter stays untouched. This applies to arrays in the same way. For dynamic variables as parameters, the parameter is resized to the given buffer length.

If data gets truncated for variables of the type I2/I4/F4/F8 (buffer length not equal to the total parameter length), the results depend on the machine type (little endian/big endian). In some applications, the user exit must be programmed to use no static data to make recursion possible.

Prototypes:

```
int ncxr_put_parm        ( int parmnum, void *parmhandle,
                             int buffer_length, void *buffer );
int ncxr_put_parm_array  ( int parmnum, void *parmhandle,
                            int buffer_length, void *buffer,
                           int *indexes );
```

Parameter Description:

parmnum	Ordinal number of the parameter. This identifies the parameter of the passed parameter list. Range: 0 ... numparm-1.	
parmhandle	Pointer to the internal parameter structure.	
buffer_length	Length of the data to be copied back to the address of buffer, where the data comes from.	
indexes	Index information	
return	Return Value:	Information:
	< 0	Error during copying of the information:
	-1	Illegal parameter number.
	-2	Internal error.
	-3	Too much data has been given. The copy back was done with parameter length.
	-4	Parameter is not an array.
	-5	Parameter is protected (constant or AD=O).
	-6	Dynamic variable could not be resized due to an "out of memory" condition.
	-7	Interface version conflict.
	-13	The given buffer includes an incomplete Unicode character.
	-100	Index for dimension 0 is out of range.
	-101	Index for dimension 1 is out of range.
	-102	Index for dimension 2 is out of range.
	0	Successful operation.
	> 0	Successful operation., but the parameter was this number of bytes long (length of parameter > given length).

Create, Initialize and Delete a Parameter Set

If a 3GL program wants to call a Natural subprogram, it needs to build a parameter set that corresponds to the parameters the subprogram expects. The function `ncxr_create_parm` is used to create a set of parameters to be passed with a call to `ncxr_if_callnat`. The set of parameters created is represented by an opaque parameter handle, like the parameter set that is passed to the 3GL program with the `CALL INTERFACE4` statement. Thus, the newly created parameter set can be manipulated with functions `ncxr_put_parm*` and `ncxr_get_parm*` as described above.

The newly created parameter set is not yet initialized after having called the function `ncxr_create_parm`. An individual parameter is initialized to a specific data type by a set of `ncxr_parm_init*` functions described below. The functions `ncxr_put_parm*` and `ncxr_get_parm*` are then used to access the contents of each individual parameter. After the caller has finished with the parameter set, they must delete the parameter handle. Thus, a typical sequence in

creating and using a set of parameters for a subprogram to be called through `ncxr_if4_callnat` will be:

```
ncxr_create_parm
ncxr_init_ parm*
ncxr_init_ parm*
...
ncxr_put_ parm*
ncxr_put_ parm*
...
ncxr_get_parm_info*
ncxr_get_parm_info*
...
ncxr_if4_callnat
...
ncxr_get_parm_info*
ncxr_get_parm_info*
...
ncxr_get_ parm*
ncxr_get_ parm*
...
ncxr_delete_parm
```

Create Parameter Set

The function `ncxr_create_parm` is used to create a set of parameters to be passed with a call to `ncxr_if_callnat`.

Prototype:

```
int ncxr_create_parm( int parmnum, void** pparmhandle )
```

Parameter Description:

parmnum	Number of parameters to be created.	
pparmhandle	Pointer to the created parameter handle.	
return	Return Value:	Information:
	< 0	Error:
	-1	Illegal parameter count.
	-2	Internal error.
	-6	Out of memory condition.
	0	Successful operation.

Delete Parameter Set

The function `ncxr_delete_parm` is used to delete a set of parameters that was created with `ncxr_create_parm`.

Prototype:

```
int ncxr_delete_parm( void* parmhandle )
```

Parameter Description:

parmhandle	Pointer to the parameter handle to be deleted.	
return	Return Value:	Information:
	< 0	Error:
	-2	Internal error.
	0	Successful operation.

Initialize a Scalar of a Static Data Type

Prototype:

```
int ncxr_init_parm_s( int parmnum, void *parmhandle,
    char format, int length, int precision, int flags );
```

Parameter Description:

parmnum	Ordinal number of the parameter. This identifies the parameter in the passed parameter list. Range: 0 ... numparm-1.	
parmhandle	Pointer to the parameter handle.	
format	Format of the parameter.	
length	Length of the parameter.	
precision	Precision of the parameter.	
flags	A combination of the flags IF4_FLG_PROTECTED	
return	Return Value:	Information:
	< 0	Error:
	-1	Invalid parameter number.
	-2	Internal error.
	-6	Out of memory condition.
	-8	Invalid format.
	-9	Invalid length or precision.
	0	Successful operation.

Initialize an Array of a Static Data Type

Prototype:

```
int ncxr_init_parm_sa( int parmnum, void *parmhandle,
    char format, int length, int precision,
    int dim, int *occ, int flags );
```

Parameter Description:

parmnum	Ordinal number of the parameter. This identifies the parameter in the passed parameter list. Range: 0 ... numparm-1.	
parmhandle	Pointer to the parameter handle.	
format	Format of the parameter.	
length	Length of the parameter.	
precision	Precision of the parameter.	
dim	Dimension of the array.	
occ	Number of occurrences per dimension.	
flags	A combination of the flags IF4_FLG_PROTECTED IF4_FLG_LBVAR_0 IF4_FLG_UBVAR_0 IF4_FLG_LBVAR_1 IF4_FLG_UBVAR_1 IF4_FLG_LBVAR_2 IF4_FLG_UBVAR_2	
return	Return Value:	Information:
	< 0	Error:
	-1	Invalid parameter number.
	-2	Internal error.
	-6	Out of memory condition.
	-8	Invalid format.
	-9	Invalid length or precision.
	-10	Invalid dimension count.
	-11	Invalid combination of variable bounds.
	0	Successful operation.

Initialize a Scalar of a Dynamic Data Type

Prototype:

```
int ncxr_init_parm_d( int parmnum, void *parmhandle,
    char format, int flags );
```

Parameter Description:

parmnum	Ordinal number of the parameter. This identifies the parameter in the passed parameter list. Range: 0 ... numparm-1.	
parmhandle	Pointer to the parameter handle.	
format	Format of the parameter.	
flags	A combination of the flags IF4_FLG_PROTECTED	
return	Return Value:	Information:
	< 0	Error:
	-1	Invalid parameter number.
	-2	Internal error.
	-6	Out of memory condition.
	-8	Invalid format.
	0	Successful operation.

Initialize an Array of a Dynamic Data Type

Prototype:

```
int ncxr_init_parm_da( int parmnum, void *parmhandle,
    char format, int dim, int *occ, int flags );
```

Parameter Description:

parmnum	Ordinal number of the parameter. This identifies the parameter in the passed parameter list. Range: 0 ... numparm-1.	
parmhandle	Pointer to the parameter handle.	
format	Format of the parameter.	
dim	Dimension of the array.	
occ	Number of occurrences per dimension.	
flags	A combination of the flags IF4_FLG_PROTECTED IF4_FLG_LBVAR_0 IF4_FLG_UBVAR_0 IF4_FLG_LBVAR_1 IF4_FLG_UBVAR_1 IF4_FLG_LBVAR_2 IF4_FLG_UBVAR_2	
return	Return Value:	Information:
	< 0	Error:
	-1	Invalid parameter number.
	-2	Internal error.
	-6	Out of memory condition.
	-8	Invalid format.
	-10	Invalid dimension count.
	-11	Invalid combination of variable bounds.
	0	Successful operation.

Resize an X-array Parameter

Prototype:

```
int ncxr_resize_parm_array( int parmnum, void *parmhandle, int *occ );
```

Parameter Description:

parmnum	Ordinal number of the parameter. This identifies the parameter in the passed parameter list. Range: 0 ... numparm-1.	
parmhandle	Pointer to the parameter handle.	
occ	New number of occurrences per dimension.	
return	Return Value:	Information:
	< 0	Error:
	-1	Invalid parameter number.
	-2	Internal error.
	-6	Out of memory condition.
	-12	Operand is not resizable (in one of the specified dimensions).
	0	Successful operation.

All function prototypes are declared in the file *natuser.h*.