2 The Apply Language

The basic form of an Apply procedure is as follows:

```
PROCEDURE function-name (function-args)
IS
      variable-declarations
BEGIN
      statements
END function-name;
```

NOTE

> Apply is not case sensitive. In discussions of the Apply language, keywords are shown in uppercase for emphasis. Examples of Apply program segments are shown in all lowercase.

The components of the Apply procedure are defined as follows:

PROCEDURE	This keyword indicates the beginning of an Apply procedure.
function-name	The name of the procedure. This name follows the keywords PROCEDURE and END. The name can be any combination of letters and digits, but the first character must be a letter. Underscores are also allowed within the name, but must not be consecutive.
function-args	A set of arguments to the procedure that define the names and types of data items and declare whether they are input, output, or constant data.
IS	This keyword follows the procedure arguments and precedes local variable declarations.
variable-declarations	The local variable declarations that follow the keyword IS.
BEGIN	This keyword separates the variable declarations from the procedure statements. All statements that follow the keyword BEGIN are executed over the images being processed.
statements	The Apply statements that make up the body of the procedure.
END	This keyword defines the end of the procedure and must be followed by *function-name*.

Appendix A provides a complete definition of the grammar of the Apply language.

Arguments

Each Apply procedure has an argument list containing input, output, or constant arguments to the procedure. The general form of an argument declaration is as follows:

```
function-argument : data-distributor type
```

The following list defines the components of the argument declarations:

function-argument — The name of the variable.

data-distributor — Defines the type of data distribution for *function-argument*. *data-distributor* can be IN, OUT, or CONST. The data distribution types are discussed following this list.

type — The data type of the argument. *type* can be BYTE, INTEGER, REAL, or DOUBLE. BYTE and INTEGER can be preceded by the keyword SIGNED or UNSIGNED. INTEGER can also be preceded by the keyword SHORT. The default for INTEGER and SHORT INTEGER is SIGNED. The default for BYTE is UNSIGNED. The following example declares an input argument of type unsigned integer

inimage : in unsigned integer

You can also define a rectangular computation window around the current pixel point by using the keywords ARRAY and OF with *type* as in the following example:

```
inarray : in array( -2..3, -2..3 ) of byte
```

This declaration specifies that the values in the input image ranging from -2 to +3 pixel locations from the current location in both the x and y directions can be accessed. Figure 2-1 illustrates the computation window defined by the declaration.

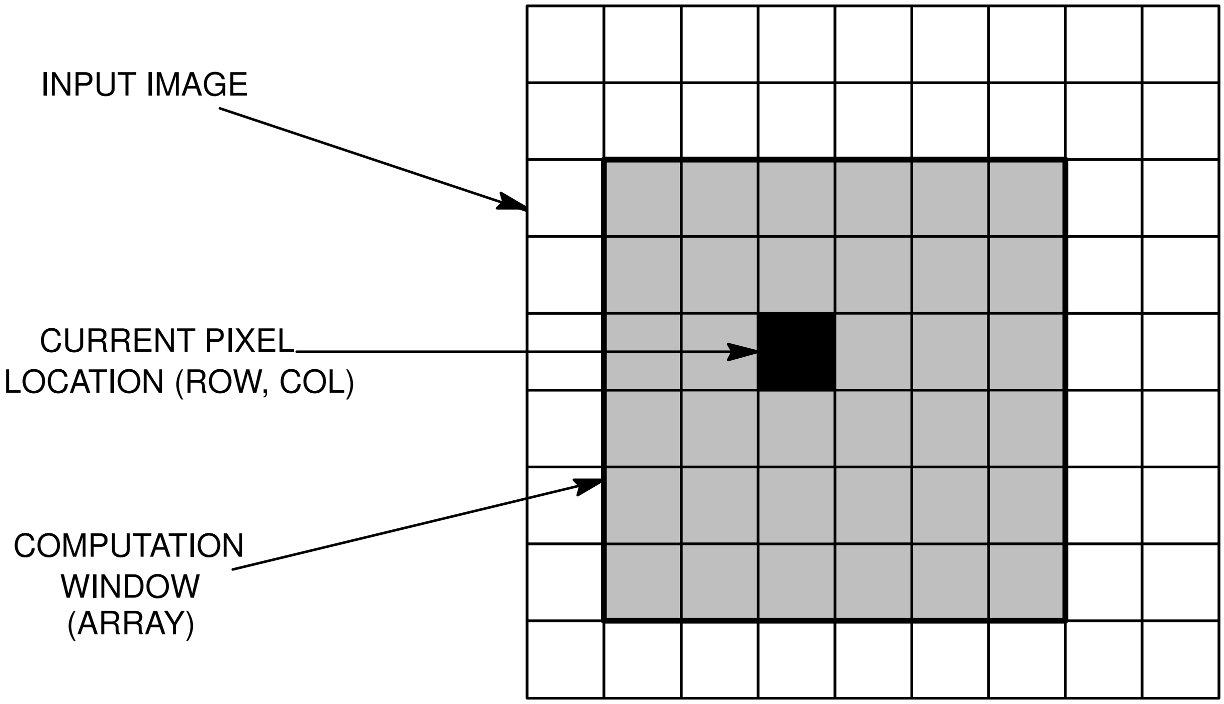

Figure 2-1: ARRAY declaration window

Likewise, you can also define a rectangular window for placing values in the output array, as in the following example:

```
outarray:out array( -2..3, -2..3 ) of byte
```

This notation is designed to maintain consistency with input declarations. In the example, outarray(–2,–2) would be mapped to outarray(0,0), outarray(0,0) would be mapped to outarray(2,2), and so on.

Output image array declarations increase the output image size by the size of the range in each direction. For the example shown, the declaration increases the output image size by a factor of six in both the x and y directions.

Input arguments: IN

Input arguments are defined by the keyword IN. Input arguments have the following properties:

- The argument is an image (a two dimensional array).
- The argument is an input variable and can not appear on the left hand side of an assignment statement.
- The data from the array is distributed across the processors used in the computations.

When declaring a computation window with the keyword ARRAY, IN can be further modified by the keywords BORDER, REPLICATED, MIRRORED, and SAMPLE.

BORDER

In a window operation, the window edges cannot extend beyond the edge of the image. Therefore, any image for which an array declaration has been made will have an output image that is smaller than the input image. The output image will still use the amount of storage space defined by the array declaration, but the border elements of the output image array will not be filled with values. The keyword BORDER increases the size of the input image so values can be computed for all positions of the output image, and the output image will be the same size as the original input image.

Assume we have a four by four input array containing the following values, and the following declaration using BORDER:

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

```
inarray : in array( -1..1, -1..1 ) of byte border 0
```

This declaration extends the input array by one row on the top and bottom and one column on each side. The border is filled with the constant specified in the declaration, in this case 0, resulting in the following input array:

0	0	0	0	0	0
0	1	5	9	13	0
0	2	6	10	14	0
0	3	7	11	15	0
0	4	8	12	16	0
0	0	0	0	0	0

REPLICA TED

You can use the keyword REPLICATED in place of a value in the BORDER declaration. REPLICATED causes the edge values of the image to be copied into the border .

Assume we have a four by four input array containing the following values, and the following declaration using REPLICATED:

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

```
inarray : in array( -2..2, -2..2 ) of byte border replicated
```

The REPLICATED declaration extends the border by two pixels on all sides and copies the values from the edges of the input array into the border pixels, resulting in the following input array:

1	1	1	5	9	13	13	13
1	1	1	5	9	13	13	13
1	1	1	5	9	13	13	13
2	2	2	6	10	14	14	14
3	3	3	7	11	15	15	15
4	4	4	8	12	16	16	16
4	4	4	8	12	16	16	16
4	4	4	8	12	16	16	16

MIRRORED

You can use the keyword MIRRORED in place of a value in the BORDER declaration. MIRRORED causes an image to be mirrored about its edges, assuring continuity of the image at the boundaries out to the edge of the border. Row 0 is copied into row -1, row 1 is copied into row -2, and so on. Likewise, column 0 is copied into column -1, column 1 is copied into column -2, and so on. The corner values in the border are copied from the corresponding mirrored values within the image, so pixel (-1,-1) would get its value from pixel (0,0), pixel (-2,-2) gets its value from pixel (1,1), and so on.

Assume we have a four by four input array containing the following values, and the following declaration using MIRRORED:

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

```
inarray : in array( -2..2, -2..2 ) of byte border mirrored
```

The MIRRORED declaration extends the border by two pixels on all sides and mirrors the values from the input array into the border pixels, resulting in the following input array:

6	2	2	6	10	14	14	10
5	1	1	5	9	13	13	9
5	1	1	5	9	13	13	9
6	2	2	6	10	14	14	10
7	3	3	7	11	15	15	11
8	4	4	8	12	16	16	12
8	4	4	8	12	16	16	12
7	3	3	7	11	15	15	11

NOTE

When using MIRRORED, the input image must be at least as large as the magnitude of the largest array dimension.

SAMPLE

The keyword SAMPLE allows creation of an output image which is smaller than the input image. In the following example, the SAMPLE declaration specifies that every second pixel in the x and y direction of the input image is used for the output image, thus creating an output image that is one fourth the size of the input image.

```
procedure samplexample(inimage : in byte sample(2,2),
                       outimage : out byte)
```

```
is
begin
   outimage := inimage(0,0);
end samplexample;
```

Assume we have a four by four input array containing the following values:

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

If we process the array with the preceding Apply procedure that uses the SAMPLE declaration to specify that only every second pixel in the x and y direction of the input image is used for the output image, the following output array results:

1	9
3	11

NOTE

An image that is processed with SAMPLE has an actual size that is the sample rate multiplied by the height and width specified in the -h and -w options of the Apply command (refer to chapter 3). For example, if -h is 240 and -w is 128, and SAMPLE(2,4) is declared, then the actual height and width of the image are 480 and 512. Another image in the same program without SAMPLE declared would be 240 x 128.

When you use SAMPLE with an ARRAY declaration that defines an input window, the sampling defines the movement of the window. The following example declaration illustrates the use of SAMPLE with ARRAY.

```
inarray : in array( −1..1, −1..1 ) of byte sample(2,2)
```

This declaration would cause the window to move two pixel positions after each set of computations on the current window.

Output arguments: OUT

Output arguments are defined by the keyword OUT. Output arguments have the following properties:

- the argument is an image (a two dimensional array)
- the argument should be assigned a value in the Apply program
- the data for the array is gathered from across the processors used in the computations

Constant arguments: CONST

Constant arguments are defined by the keyword CONST. Constant arguments have the following properties:

- the argument is a scalar unless specifically declared as an array
- the argument is an input value and can not appear on the left hand side of an assignment statement

Variables

Variable names in Apply have the following properties:

- they can contain letters, numbers, and underscores only
- underscores can not appear at the beginning or end of a name, and they can not be consecutive
- they must begin with a letter
- they can be any length
- case is not significant

Variables can be of type BYTE, INTEGER, SHORT INTEGER, REAL, or DOUBLE. BYTE, INTEGER, and SHORT INTEGER variables can be SIGNED or UNSIGNED.

Statements

Apply has the following types of statements:

- assignment statements
- conditional statements
- looping statements

Assignment Statements

An Apply assignment statement consists of a left hand side, followed by the assignment operator :=, followed by a right hand side. The right hand side of the assignment statement consists of an expression followed by a semicolon.

Expressions in Apply are combinations of variables and literals joined by operators. Table 2-1 lists the Apply arithmetic and bitwise operators, and Table 2-2 lists the Apply boolean operators.

Table 2-1: Apply Arithmetic and Bitwise Operators

Symbol	Operation
+	addition
–	subtraction
*	multiplication
/	division
%	modulo
^	bitwise exclusive or
|	bitwise inclusive or
&	bitwise and
!	bitwise complement
abs(x)	absolute value (integer argument)
sqrt(x)	square root (real argument)
max(x)	maximum (any type argument)
min(x)	minimum (any type argument)
real(x)	convert to real (any type argument)
double(x)	convert to double (any type argument)
fabs(x)	absolute value (real argument)
log(x)	natural logarithm (real argument)
log10(x)	common logarithm (real argument)
exp(x)	exponential (real argument)
sin(x)	sin (real argument)
cos(x)	cosine (real argument)
tan(x)	tangent (real argument)
asin(x)	inverse sine (real argument)
acos(x)	inverse cosine (real argument)
atan(x)	inverse tangent (real argument)
atan2(x)	inverse angle (real argument)
integer(x)	convert to integer (any type argument)

Table 2-2: Apply Boolean Operators

Symbol	Operation
AND	logical and
OR	logical or
NOT	logical not
<	less than
<=	less than or equal
=	equal
>=	greater than or equal
>	greater than
/=	not equal

The precedence and associativity relationships for Apply operators are the same as precedence and associativity relationships in the C programming language with one exception. In Apply, bitwise and logical NOT have the same precedence.

Mixing of integer, byte, and real variables within expressions, and assignment of integer-to-real and real-to-integer variables follow the rules of the C programming language. You can convert variable types to ensure agreement by using the pseudo-functions REAL, INTEGER, and DOUBLE. When converting from type real to type byte, real data should be converted to type integer before assignment to a byte variable.

Conditional Statements: IF

IF statements serve as conditional statements in Apply procedures. IF statements have the following two general forms:

```
IF boolean-expr THEN
   statements
END IF;
```

or:

```
IF boolean-expr THEN
   statements
ELSE
   statements
END IF;
```

The following is an example of an Apply IF statement:

```
if x = 1 then
   image1 := 0;
else
   image1 := 1;
end if;
```

Looping Statements: FOR and WHILE

The FOR statement and the WHILE statement are used for looping in Apply procedures. The FOR statement has the following general form:

```
FOR integer-var IN range LOOP
   statements
END LOOP;
```

integer-var is any variable of type integer.

range is two integer expressions separated by two dots.

The following is an example of an Apply FOR statement:

```
for i in 1 .. 10  loop
   x(i) := 1;
end loop;
```

The WHILE statement has the following general form:

```
WHILE boolean-expr LOOP
   statements
END LOOP;
```

The following is an example of an Apply WHILE statement:

```
while i < 10  loop
   x(i) := 1;
   i := i + 1;
end loop;
```

ROW and COL

The reserved words ROW and COL contain the row and column indices of the current processing location as defined by the underlying unsampled image. The first pixel processed is always (0,0), the second is (0,1), and so on. You can use ROW and COL in loop blocks to control operations over selected parts of an image. The following example illustrates an IF statement that uses ROW and COL to set the pixel located at column 4, row 6 to zero.

```
if (COL = 4 & ROW = 6) then
   imageout := 0;
end if;
```

You can not assign values to ROW or COL, since the order of image processing is under control of the Apply compiler.