2 The Adapt Language

The basic form of an Adapt procedure is as follows:

```
PROCEDURE function-name (function-args)
IS
      global-declarations

FIRST
      local-declarations
BEGIN
      section-statements
END FIRST

NEXT
      local-declarations
BEGIN
      section-statements
END NEXT

COMBINE
      local-declarations
BEGIN
      section-statements
END COMBINE

LAST
      local-declarations
BEGIN
      section-statements
END LAST

END function-name;
```

NOTE

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

The components of the Adapt procedure are defined as follows:

PROCEDURE	This keyword indicates the beginning of an Adapt 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 or output images, or global input or output parameters.
IS	This keyword follows the procedure arguments and precedes global variable declarations.
global-declarations	The global variable declarations that follow the keyword IS. These variables are common to all sections of the Adapt program and can be accessed from any section.
BEGIN	This keyword separates the variable declarations from the section statements. The keyword BEGIN follows each section declaration.
FIRST	This keyword declares the FIRST section of the program. This section is used to initialize variables, and it is executed before any computation is done.
NEXT	This keyword declares the NEXT section of the program. This section is executed once per pixel, in raster order, across an image.
COMBINE	This keyword declares the COMBINE section of the program. This section combines results from two independent regions of the image.
LAST	This keyword declares the LAST section of the program. This section does final computation and discards intermediate results. This section is executed once after the entire image has been processed.
local-declarations	The variables declared locally within a section of the Adapt program. These variables are local to the section in which they are declared.
section-statements	The statements that make up a given section of the Adapt program.
END	When followed by a section name, this keyword defines the end of that section. When followed by *function-name*, this keyword defines the end of the Adapt procedure.

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

Arguments

Each Adapt procedure has an argument list that can contain any of the following:

- input images
- output images
- global input parameters
- global output parameters

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 or OUT. If the argument is an image, it can be declared as IMAGE or TRANSPOSED IMAGE.

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

inimage : in image 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 image 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.

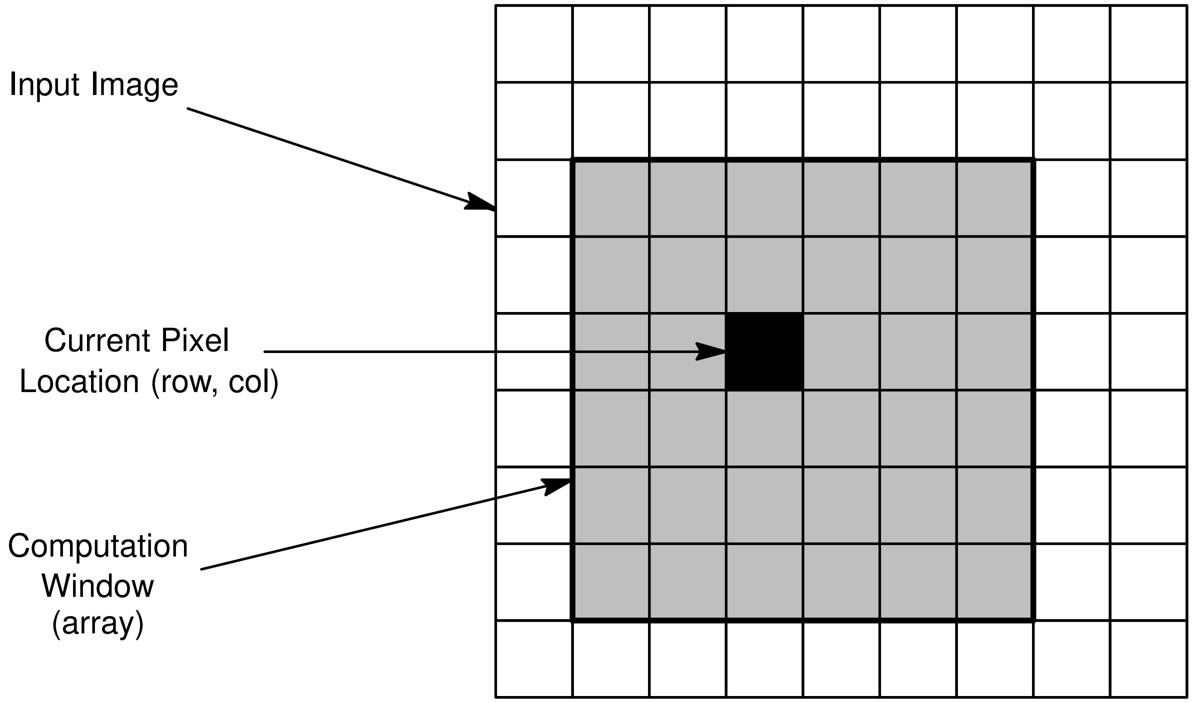

Figure 2-1: ARRAY declaration window

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

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

This notation is designed to maintain consistency with input declarations.

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 can be image or global.
- The argument is an input variable and can not appear on the left hand side of an assignment statement.
- If the input data is an image, the image array is distributed across the processors used in the computations. If it is global, it is broadcast to all processors.

When declaring a computation window with the keyword ARRAY, IN can be further modified by the keywords BORDER, REPLICATED, MIRRORED, and SAMPLE. These keywords are described in the sections that follow.

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 image 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

REPLICATED

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 image 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 image 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 that 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 image byte sample(2,2),
                       outimage : out image byte)
is
next begin
   outimage := inimage(0,0);
end next
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 Adapt 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 parameters passed in the Adapt procedure call. For example, if an adapt procedure is called with a height of 256 and a width of 128, then the actual size of an image passed to a parameter with SAMPLE(2,4) must be 512 x 512. Another image parameter in the same procedure call without SAMPLE must be passed an image that is 256 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 image 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 or global parameter
- the argument should be assigned a value in the Adapt program
- the data for an image array is gathered from across the processors used in the computations

Variables

Variable names in Adapt 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. The one exception to this rule is a global variable preceded by an underscore, by which the count from another cell is obtained for combining results. This syntax is described in the Adapt histogram example in Chapter 1.
- They must begin with a letter.
- They can be any length.
- Case is not significant.

Variables can be of type BYTE, INTEGER, FLOAT or LONG_FLOAT. BYTE and INTEGER variables can be SIGNED or UNSIGNED. Table 2-1 lists the correspondence between variable types for Adapt and C.

Table 2-1: Correspondence between Adapt and C types

Adapt Type	C Type
signed_byte	char
unsigned_byte	unsigned char
byte	unsigned char
signed_integer	int
unsigned_integer	unsigned int
integer	int
float	float
long_float	double

Statements

Adapt has the following types of statements:

- assignment statements
- conditional statements
- looping statements

Assignment Statements

An Adapt 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 Adapt are combinations of variables and literals joined by operators. Table 2-2 lists the Adapt arithmetic and bitwise operators, and Table 2-3 lists the Adapt boolean operators.

NOTE

To have access to the math functions in the following table, you must use a #include statement to include the file `math.ad` with your program. This file is in the standard Adapt include area.

Table 2-2: Adapt Arithmetic and Bitwise Operators

Symbol	Operation
+	addition
–	subtraction
*	multiplication
/	division
**	exponentiation
mod	modulo
rem	remainder
lxor	bitwise exclusive or
lor	bitwise inclusive or
land	bitwise and
lnot	bitwise complement
lsh	left shift
rsh	right shift
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 2-3: Adapt 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 relationships for Adapt operators are shown in Table 2-4.

Table 2-4: Adapt Operator Precedence

Level	Operators
1	or xor and
2	lor lxor land
3	= /= < <= > >=
4	lsh rsh
5	+ – not lnot
6	* / mod rem
7	**
8	– (unary minus)

Conditional Statements: IF

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

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

or:

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

The following is an example of an Adapt IF statement:

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

Looping Statements: FOR, WHILE, and EXIT

The FOR, WHILE, and EXIT statements are used for looping in Adapt 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 Adapt 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 Adapt WHILE statement:

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

The EXIT statement has the following general form:

```
EXIT loop-identifier WHEN boolean-expr
```

The following is an example of an Adapt EXIT statement:

```
loop1: for i in -1..1 loop
   loop2: for j in -1..1 loop
      total := total + img(i,j);
      exit loop1 when (total > threshold);
   end loop;
end loop;
```

CASE Statement

The CASE statement has the following general form:

```
CASE expression IS
WHEN choice | choice => statements
WHEN OTHERS => statements
END CASE;
```

OTHERS is an optional keyword that defines what takes place if *expression* does not evaluate to any defined *choice*.

The following is an example of an Adapt CASE statement:

```
case (img1) is
   when 1 | 2 | 3 => img2 := 7;
   when 4 | 5 => img2 := 13;
   when others => img2 := 0;
end case;
```

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 land ROW = 6) then
   imageout := 0;
end if;
```

You cannot assign values to ROW or COL, since the order of image processing is under control of the Adapt compiler.