The VPascal Primer

Programming Guide for the V++ Precision Digital Imaging System

PO Box 35-715 • Browns Bay • Auckland 1330 • New Zealand
Phone +64 (21) 736 378, +64 (9) 478 5779 • Fax +64 (9) 479 4750
www.digitaloptics.co.nz

The VPascal Primer
Copyright © 1990 – 2000, Digital Optics Limited.
First Edition (online)
Produced in New Zealand.

Table of Contents

Preface

Welcome to *The VPascal Primer* - the complete introduction to the VPascal automation language and its application to practical problems in digital image processing.

VPascal is the powerful built-in automation language in Digital Optics' well known V++ product. Programs written in VPascal (called *automation modules* or just *modules*) can be very simple, perhaps performing just a few common image processing tasks, or extremely complex, controlling an entire experiment consisting of cameras, laboratory equipment and user interaction.

The VPascal Primer is designed to give V++ users an in-depth understanding of how to apply the VPascal automation language to their own imaging applications. It teaches you how to build effective imaging routines, customize the V++ user interface, and automate control of a PVCAM camera. There is also ample material covering the fundamentals or programming for beginners.

Later chapters focus on advanced techniques for developing imaging applications using VPascal. These include structuring your application, using DDE and networking, controlling laboratory equipment and linking custom functions and dialog boxes to your VPascal modules.

Topics are presented in approximate order of increasing complexity and range from fundamentals to advanced topics suitable for OEM programmers. Advanced users may safely skip early chapters that cover material they are already familiar with.

As you learn more about VPascal, please come and visit some of the technical areas of the Digital Optics web site that address VPascal issues:

http://www.digitaloptics.co.nz/

http://www.digitaloptics.co.nz/technical/technical.htm

http://www.digitaloptics.co.nz/technical/articles.htm

http://www.digitaloptics.co.nz/technical/source/sourcecode.htm

On these pages you will find technical articles and sample VPascal source code, much of which has been contributed by other users. Your contributions would also be very welcome.

1.0 Variables and data types

1.1 Scalars (numbers)

A *scalar* is the technical term for number. VPascal supports a wide variety of scalars, as shown in Table 1.1.1 below.

Scalar	**Range**
Integer numbers	-2147483648 to +2147483647
Real numbers	±3.59539E+308
Complex numbers	±3.59539E+308 in each of 2 components
Integer color numbers	0 to 65535 in each of 3 channels
Real color numbers	±3.40282E+38 in each of 3 channels
Boolean numbers	FALSE, TRUE

Table 1.1.1 Scalars in VPascal

The type of a scalar is not fixed at compile-time, but is a fluid quantity that changes as required. For example, the assignment `x := -123` makes `x` an integer-valued scalar because `-123` is compatible with that type. If the next statement is `x := sqrt( x )`, then `x` becomes complex because the square root of a negative number must result in a complex result.

Example 1.1.1: Setting the value of a scalar

This example demonstrates the changing nature of a scalar in a module.

```
var
  a, b, c ;
begin
a := 123 ;                    // a is an integer
b := 567.678 ;                // b is a real nunmber
c := 123 + i*456 ;            // c is a complex number
a := sqrt( -a ) ;             // a is promoted to complex
b := 44 ;                     // b remains a real number
c := MakeRGB( 4, 77, 234 ) ;  // c is promoted to an integer color number
end
```

Listing 1.1.1 Automatic type changing of scalars in a module.

1.2 Arrays (images)

An array is a matrix of numbers. Arrays can be one-, two- or three-dimensional. Furthermore, V++ supports the widest range of array types. (See Section 1.5: Data types.)

Like scalars, arrays can change their types on the fly, but the rules are slightly more restrictive. When assigning to the entire array, the type of the array may be changed to accommodate the type of data being assigned to it. If only a portion (say one pixel, or a range of pixels) of the image is be assigned to, then the type of data being assigned is changed to that of the array.

Example 1.2.1: Setting the value of a pixel or array

This example demonstrates what happens when a value is assigned to a portion of an array or the entire array.

```
var

  A, B, C ;

begin

A := CreateArray( byte, 100, 100 ) ;

B := CreateArray( single, 100, 100 ) ;

C := CreateArray( RGB, 100, 100 ) ;

A[ 3, 45 ] := 123 ;     // A remains a byte array

A[ 3, 45 ] := 2000 ;    // A remains a byte array

A := 2000 ;             // A is converted to small integer

A := C + B ;            // A is promoted to RGB

end
```

Listing 1.2.1 Situations where the type of an array may change automatically.

1.3 Strings (text)

Strings are stored as arrays of characters. Literal strings appear in a module as a single-quoted sequence of characters.

Example 1.3.1: Strings in a module

This example demonstrates some simple string-related operations.

```
var
  a, b, c ;
begin
a := 'Digital' ;
b := 'Optics' ;
c := a + ' ' + b ;     // c is now 'Digial Optics'
c := UpperCase( c ) ; // c is now 'DIGITAL OPTICS'
end
```

Listing 1.3.1 Strings in a module.

1.4 Special variables

Some variables in a module are special in the sense that they do not fall into the conventional category of number, image, string, etc. For example, plot windows are referenced via *plot variables*, editors (text windows) are referenced via *editor variables*, and so on.

1.5 Data types

V++ supports the greatest variety of data types for image processing. The complete range of types available for representing images is shown below in Table 1.5.1.

Type	**Description**
Binary	1-bit unsigned integer
Byte	8-bit unsigned integer
Short integer	16-bit signed integer
Word	16-bit unsigned integer
Long integer	32-bit signed integer
Single	Single-precision floating-point (32 bits)
Double	Double-precision floating-point (64 bits)
Single complex	Single-precision complex (32 bits per component)
Double complex	Double-precision complex (64 bits per component)
RGB-24	24-bit RGB (8 bits per channel)
RGB-48	48-bit RGB (16 bits per channel)
RGB-float	96-bit floating-point RGB

Table 1.5.1 Image data types

1.6 Variables and desktop images

Arrays can be created in a module for various purposes: to hold a table of results, to represent a mathematical equation, or to hold the pixel values of an image.

In order to gain access to the pixel values of a desktop image (displayed in the V++ GUI) the image must be *associated* with a variable. This makes the image available to the module via a variable name, just like any other variable.

There are several different ways that a desktop image can be associated with a variable and each is explored in the examples below.

Example 1.6.1 Using the active image

This example shows how to use the active image on the V++ desktop. The active image is the image that currently has input focus. (A window has focus if its caption bar is not dimmed.) The procedure `GetActiveImage` determines which image is the active image, and then *connects* or *associates* the image with the variable `A`.

```
var
  A ;
begin
GetActiveImage( A ) ;  // Associate active image with variable A
A := not A ;           // Process the image
end
```

Listing 1.6.1 Accessing the active image

Example 1.6.2 Using a named image

This example shows how to use an image whose name is known. The procedure `GetImage` takes the name of the image and return the association in the variable.

```
var
  A ;
begin
GetImage( 'A0001.tif', A ) ;  // Associate image A0001.tif with variable A
A := not A ;                  // Process the image
end
```

Listing 1.6.2 Accessing a named image

Example 1.6.3 Testing if a named image exists

A call to `GetImage` will fail if the named image does not exist. Use the `ImageExists` function to determine if an image of the specified name is present on the V++ desktop.

```
var
  A ;
begin
if ImageExists( 'A0001.tif' ) then
  GetImage( 'A0001.tif', A )
else
  Halt( 'Image does not exist!' ) ;  // Terminate module if cannot find image
A := not A ;
end
```

Listing 1.6.3 Accessing a named image after testing that the image exists.

Example 1.6.4 Iterating over all images on the desktop

This example demonstrates how to iterate over all the desktop images using `GetFirstImage` and `GetNextImage`. The module associates each image, in turn, with the variable A. It then increments a count if the image is a 16-bit unsigned image.

```
var
  Image ;
  Count ;
begin
Count := 0 ;
GetFirstImage( Image ) ;
while IsImage( Image ) do
  begin
  if TypeOf( Image ) = typ_Word then
    Count := Count + 1 ;
  GetNextImage( Image ) ;
  end ;
WriteInfo( Count, ' 16-bit unsigned images were found' ) ;
end
```

Listing 1.6.4 Looping over all desktop images, counting the number of 16-bit unsigned images.

Example 1.6.5 Determining if there are any desktop images

This example shows how to use the `GetImageCount` function to determine if there are any images available on the desktop.

```
begin
if GetImageCount = 0 then
  Halt( 'No desktop images found!' ) ;
{Do something here if there are images}
end
```

Listing 1.6.5 Determining if there are any desktop images available for processing.

Example 1.6.6 Selecting an image from a list

This example demonstrates the `SelectImage` function as a way to allow the user to select an image from a list of all desktop images. `SelectImage` displays a dialog that lists all desktop images. The user may then make a selection.

```
var
  Image ;
begin
SelectImage( 'Choose an image to process', Image ) ;
Image := not Image ;  // Process the image in some way
end
```

Listing 1.6.6 Letting the user select an image from a list.

Example 1.6.7 Selecting an image from a list with error recovery

This example demonstrates the `SelectImage` function as a way to allow the user to select an image from a list of all desktop images. The `SelectImage` is a function that returns a code indicating which button the user pressed on the dialog. This makes it possible to write code that reacts appropriately when the user presses the Cancel button instead of the OK button.

```
var
  Image ;
  Code ;
begin
Code := SelectImage( 'Choose an image to process', Image ) ;
if Code = id_OK then
  Image := not Image ;  // Process the image if OK pressed
end
```

Listing 1.6.7 Letting the user select an image from a list. The module copes with the case where the user presses the Cancel button.

2.0 Expressions

2.1 What is an expression?

An expression is a formula that calculates a result of some kind. An expression may involve strings, numbers, arrays, or even other functions.

The variables and constants involved in an expression are called *arguments*.

The symbols used to combine the arguments are called *operators*.

Some example expressions are shown in Table 2.1.1 below.

Example Expression	Arguments	Operators

`X + 3 – Y`	`X`
	`Y`
	`3`
		`+`
		`-`

`A` **`nand`** `B` **`xor`** `1234`	`A`
	`B`
	`1234`
		`nand`
		`xor`

`( A > 200 )` **`and`** `( A <= 500 )`	`A`
	`200`
	`500`
		`>`
		`<=`

Table 2.1.1 Arguments and operators.

2.2 Image expressions

One of the most powerful features of VPascal is that arrays can be used in expressions just like ordinary numbers. They can be combined with other arrays or with numbers.

Some example image expressions are shown in Table 2.2.1 below.

Example Expression	**Explanation**
`C := A – B`	Image B is subtracted form image A and stored in image C. This forms the basis of easy background subtraction.
`B := 1.25 * ( A – 10 )`	A reference level of 10 is subtracted from image A and the result is scaled by 1.25 before being stored in image B. The result will be left as a floating-point image.
`B := word( 1.25 * ( A – 10 ) )`	Same as the previous example, except the result is converted to a 16-bit unsigned image before storing in image B.
`B := ( A >= 4095 )` **`and`** `( A <= 0 )`	Create a binary image that is 1 (TRUE) wherever the pixels in A exceed the upper or lower thresholds (0 and 4095). Store the result in image B.
`A := ( A < 10 ) * A`	Replace A with a version which is zero wherever the original pixel values are less than the threshold value 10.
`N := SumOf( A = 0 )`	Count the number of pixels equal to zero and store the result in N.

Table 2.2.1 Example expressions involving images and scalars

2.4 Type precedence

Type precedence refers to the rule used to determine the resultant data type of an expression when the arguments have differing data types.

For example, if `A` is a 16-bit *unsigned* image and `B` is a 16-bit *signed* image, should the expression `C := A + B` yield a signed or unsigned result?

To resolve such an ambiguity the type precedence rule determines what happens in any expression.

2.4.1 Type precedence rule

When two arguments are combined in an arithmetic or logical expression, the resultant data type is the more general of the two data types, according to the priorities listed below.

Priority	Type	Description
1	RGB-float	96-bit floating-point RGB
2	RGB-48	48-bit RGB (16 bits per channel)
3	RGB-24	24-bit RGB (8 bits per channel)
4	Double complex	Double-precision complex (64 bits per component)
5	Single complex	Single-precision complex (32 bits per component)
6	Double	Double-precision floating-point (64 bits)
7	Single	Single-precision floating-point (32 bits)
8	Long integer	32-bit signed integer
9	Word	16-bit unsigned integer
10	Short integer	16-bit signed integer
11	Byte	8-bit unsigned integer
12	Binary	1-bit unsigned integer

Table 2.4.1 Priority table showing type precedence

2.4.2 Type coercion

When two differing data types participate in an expression, the least general type must first be converted to the most general type of the two arguments. Any form of automatic type conversion is called *type coercion*.

Type coercion is applied on a strictly mathematical basis, but with a few minor qualifications as listed in the table below.

When these types…	**Are coerced to these types…**	**This special processing takes place…**
Binary, byte, word, short integer, long integer, single, double.	RGB-24, RGB-48, RGB-float.	The original value becomes the red, green and blue components of the color.
Binary, byte, word, short integer, long integer, single, double.	Complex, double complex.	The original value becomes the real part of the complex number. The imaginary part is set to zero.
Complex, double complex.	RGB-24, RGB-48, RGB-float.	The real part of the original value becomes the red, green and blue components of the color. The imaginary part is discarded.
RGB-24, RGB-48, RGB-float.	Complex, double complex.	The red component of the original value becomes the real component of the complex number. The imaginary part is set to zero. The green and blue components are discarded.
Complex, double complex.	Binary, byte, word, short integer, long integer, single, double.	The real part of the complex number becomes the new value. The imaginary part is discarded.
RGB-24, RGB-48, RGB-float.	Binary, byte, word, short integer, long integer, single, double.	The red component of the color becomes the new value. The green and blue components are discarded.

Table 2.4.2 Special coercion conditions when dealing with composite data types.

2.5 String expressions

String expressions are limited to addition (when joining strings), relational operators (when establishing equality or order) and function evaluations (such as converting lower case).

3.0 Accessing pixels in an image

Images are treated as simple arrays of numbers. An image can have one, two or three dimensions. Every pixel, or range of pixels, in an image can be easily accessed using a simple addressing notation.

In all cases pixels, or ranges of pixels, are addressed using square brackets `[ ]` to identify portions of the image.

Within the brackets, the order of the dimensions is always `[ x-addr, y-addr, z-addr ]`.

3.1 Index notation

Index notation is the term used when a single x-, y- or z-index number is used. For example, if A is a one-dimensional array, A[4] refers to element 4 of the array.

3.2 Accessing a single pixel

Example 3.2.1: Reading and writing a single pixel

This example shows how a single pixel is read from an image, and an alternative value written back to the same location. It assumes that the image A is a simple two-dimensional image and not a sequence. If A is a sequence, see Example 3.2.2 below.

```
var

  A, Pixel ;

begin

GetActiveImage( A ) ;

Pixel := A[ 3, 5 ] ;

WriteInfo( 'Pixel at [3,5] is ',Pixel ) ;

A[ 3, 5 ] := 0 ;

end
```

Listing 3.2.1 Accessing a single pixel using index notation.

Example 3.2.2: Reading and writing a single pixel in a sequence

This example shows how to extract a single pixel from a specific frame in a sequence, and write the value to another position in the sequence.

```
var
  A, Pixel ;
begin
GetActiveImage( A ) ;
Pixel := A[ 3, 5, 9 ] ;
WriteInfo( 'Pixel at [3,5] in frame 9 is ',Pixel ) ;
A[ 3, 5, 8 ] := Pixel ;
end
```

Listing 3.2.2 Accessing a single pixel in a sequence using index notation.

3.2 Index-range notation

Index-range notation is the term used when a range of pixels is addressed in the x-, y- or z-directions. Index-range notation is a simple extension of index notation where individual x-, y- or z-indexes are replaced with x-, y- or z-ranges.

A range of addresses is specified using the notation `[ start-location .. stop-location ]`

Index-range notation appears on the right-hand side of an expression when extracting a range of pixels, and appears on the left-hand side of an expression when assigning new values to the range of pixels.

3.3 Accessing a region

A region is a rectangular portion of an image.

Example 3.3.1: Extracting a rectangular region from an image

This example extracts a rectangular region from an image and displays the result as a new image.

```
var

  A, B ;

begin

GetActiveImage( A ) ;

B := A[ 100..200, 150..300 ] ;

Show( B ) ;

end
```

Listing 3.3.1 Extracting a rectangular region from an image.

Example 3.3.2: Moving a rectangular region in an image

This example moves a rectangular region from one part of an image to another location in the same image.

```
var

  A, B ;

begin

GetActiveImage( A ) ;

B := A[ 100..200, 150..300 ] ;

A[ 500..600, 450..600 ] := B ;

Update( A ) ;

end
```

Listing 3.3.2 Moving a rectangular region in an image.

Example 3.3.3: Avoiding temporary storage

Often there is no need for a temporary variable (B in Example 3.3.2) as is shown in this example.

```
var

  A ;

begin

GetActiveImage( A ) ;

A[ 500..600, 450..600 ] := A[ 100..200, 150..300 ] ;

Update( A ) ;

end
```

Listing 3.3.3 Moving a rectangular region in an image without temporary storage.

Example 3.3.4: Setting a rectangular region to the same value

When assigning to a region in an image, the right-hand-side can be a number. This example shows how to set a region of an image to the constant value 128. (This is can be very useful when clearing image boundaries. See Example 3.3.5 below.)

```
var

  A ;

begin

GetActiveImage( A ) ;

A[ 500..600, 450..600 ] := 128 ;

Update( A ) ;

end
```

Listing 3.3.4 Setting all pixels in a rectangular region to the same value

Example 3.3.5: Clearing the boundaries of an image

Often it is necessary to clear (set to zero) the boundaries of an image prior to an operation. This example shows how to clear a 4-pixel wide boundary around the image using index-range notation.

```
var
  A, xSize, ySize ;
begin
GetActiveImage( A ) ;
GetXYSize( A, xSize, ySize ) ;
A[ 0..3, 0..ySize-1 ] := 0 ;               // Left border
A[ ySize-4..ySize-1, 0..ySize-1 ] := 0 ;  // Right border
A[ 0..xSize-1, 0..3 ] := 0 ;               // Top border
A[ 0..xSize-1, ySize-4..ySize-1 ] := 0 ;  // Bottom border
Update( A ) ;
end
```

Listing 3.3.5 Clearing an image boundary.

3.4 Accessing a row

When accessing a single row, rather than a rectangular region, it is possible to mix index and index-range notation.

Furthermore, when referring to an entire row you may omit the starting and stopping indexes, leaving just the range symbol.

Example 3.4.1: Copying part of a row of data

This example copies data from part of one row to another. It mixes the index and index-range notation.

```
var
  A ;
begin
GetActiveImage( A ) ;
A[ 100..300, 5 ] := A[ 100..300, 7 ] ;  // Copy part of row 7 into row 5
Update( A ) ;
end
```

Listing 3.4.1 Copying part of a row of data.

Example 3.4.2: Copying a complete row of data

This example copies one complete row of pixels to another row. It mixes the index and index-range notation and also omits the range start and stop indexes.

```
var
  A ;
begin
GetActiveImage( A ) ;
A[ .., 5 ] := A[ .., 7 ] ;  // Copy row 7 into row 5
Update( A ) ;
end
```

Listing 3.4.2 Copying a complete row of data to another row.

Example 3.4.3: Zeroing hot pixels

This example replaces defective pixels in a row with the constant value 0. (A smarter solution is shown in Example 3.4.4 below.)

```
var
  A ;
begin
GetActiveImage( A ) ;
A[ .., 5 ] := 0 ;  // Zero pixels in defective row 5
Update( A ) ;
end
```

Listing 3.4.3 Zeroing hot pixels in a row.

Example 3.4.4: Repairing hot pixels

This example replaces defective pixels with the average value of rows above and below the defective row.

```
var
  A ;
begin
GetActiveImage( A ) ;
A[ .., 5 ] := ( A[ .., 4 ] + A[ .., 6 ] ) / 2 ;  // Row 5 is defective
Update( A ) ;
end
```

Listing 3.4.4 Repairing a defective row of pixels.

3.5 Accessing a column

When accessing a single column, rather than a rectangular region, it is possible to mix index and index-range notation.

Furthermore, when referring to an entire column you may omit the starting and stopping indexes, leaving just the range symbol.

Example 3.5.1: Copying part of a column of data

This example copies data from part of one column to another. It mixes the index and index-range notation.

```
var
  A ;
begin
GetActiveImage( A ) ;
A[ 5, 100..300 ] := A[ 7, 100..300 ] ;  // Copy part of column 7 into column 5
Update( A ) ;
end
```

Listing 3.5.1 Copying part of a column of data.

Example 3.5.2: Copying a complete column of data

This example copies one complete column of pixels to another column. It mixes the index and index-range notation and also omits the range start and stop indexes.

```
var
  A ;
begin
GetActiveImage( A ) ;
A[ 5, .. ] := A[ 7, .. ] ;  // Copy column 7 into column 5
Update( A ) ;
end
```

Listing 3.5.2 Copying a complete column of data to another column.

Example 3.5.3: Zeroing hot pixels

This example replaces defective pixels in a column with the constant value 0. (A smarter solution is shown in Example 3.5.4 below.)

```
var
  A ;
begin
GetActiveImage( A ) ;
A[ 15, .. ] := 0 ;  // Zero pixels in defective column 15
Update( A ) ;
end
```

Listing 3.5.3 Zeroing hot pixels in a column.

Example 3.5.4: Repairing hot pixels

This example replaces defective pixels with the average value of columns left and right of the defective column.

```
var
  A ;
begin
GetActiveImage( A ) ;
A[ 15, .. ] := ( A[ 14, .. ] + A[ 16, .. ] ) / 2 ;  // Column 15 is defective
Update( A ) ;
end
```

Listing 3.5.4 Repairing a defective row of pixels.

3.6 Accessing frames in a sequence

Accessing one or more frames in a sequence is easy with index-range notation: simply specify the frame index or indexes as part of the addressing notation.

Example 3.6.1: Extracting a complete frame from a sequence

This example extracts a frame from a sequence and displays the result. Note: the x and y index-range symbols must be present since the addressing must always appear in the order x, y, z.

```
var
  Seq, Frame ;
begin
GetActiveImage( Seq ) ;
Frame := Seq[ .., .., 15 ] ;  // Extract frame 15
Show( Frame ) ;
end
```

Listing 3.6.1 Extracting a frame from a sequence.

Example 3.6.2: Extracting part of one frame from a sequence

This example extracts a part of one frame in a sequence and displays the result.

```
var
  Seq, Frame ;
begin
GetActiveImage( Seq ) ;
Frame := Seq[ 100..200, 30..230, 15 ] ;  // Extract part of frame 15
Show( Frame ) ;
end
```

Listing 3.6.2 Extracting part of one frame from a sequence.

Example 3.6.3: Extracting a sub-sequence from a sequence

This example extracts several contiguous frames (a sub-sequence) from a sequence and displays the result.

```
var
  Seq, SubSeq ;
begin
GetActiveImage( Seq ) ;
SubSeq := Seq[ .., .., 15..21 ] ;  // Extract frames 15 thru 21
Show( SubSeq ) ;
end
```

Listing 3.6.3 Extracting a sub-sequence from a sequence.

Example 3.6.4: Setting a range of frames to a constant value

This example sets the specified frames to the value 128.

```
var
  Seq ;
begin
GetActiveImage( Seq ) ;
Seq[ .., .., 15..21 ] := 128 ;  // Frames 15 thru 21 now equal 128
Update( Seq ) ;
end
```

Listing 3.6.4 Setting frames in a sequence to a constant value.

Example 3.6.5: Copying one frame to another location

This example copies one frame to another location in the sequence.

```
var
  Seq ;
begin
GetActiveImage( Seq ) ;
Seq[ .., .., 21 ] := Seq[ .., .., 15 ] ;  // Copy frame 15 to frame 21
Update( Seq ) ;
end
```

Listing 3.6.5 Copying a frame to a new location.

Example 3.6.6: Copying one frame to a range of locations

This example copies one frame to a contiguous range of other locations.

```
var
  Seq ;
begin
GetActiveImage( Seq ) ;
Seq[ .., .., 15..21 ] := Seq[ .., .., 3 ] ;  // Copy frame 3 to frames 15 thru 21
Update( Seq ) ;
end
```

Listing 3.6.6 Copying one frame to a range of locations in a sequence.

Example 3.6.7: Finding the difference between successive frames in a sequence

This example creates a new sequence equal to the difference between adjacent frames in the original sequence. This would be useful when examining changes over time in a series of images.

```
var
  Seq, DiffSeq ;
  z, xSize, ySize, zSize ;
begin
GetActiveImage( Seq ) ;
GetXYZSize( Seq, xSize, ySize, zSize ) ;
DiffSeq := CreateImage( integer, xSize, ySize, zSize-1 ) ;
for z := 0 to zSize-2 do
  DiffSeq[ .., .., z ] := Seq[ .., .., z+1 ] - Seq[ .., .., z ] ;
Show( DiffSeq ) ;
end
```

Listing 3.6.7 Finding the difference between successive frames in a sequence.

3.9 Omitting indexes and index-range symbols

When using index or index-range notation the referenced dimensions always appear in x-y-z order, although in some cases not all dimensions need be specified.

Leading dimensions must always be included, even if they are ".." Trailing dimensions may be omitted, and if so, are assumed to be ".." Some examples are shown in the table below.

Shorthand Reference	Interpretation
`A[ 5 ]`	`A[ 5, .., .. ]`
`A[ .., 5 ]`	`A[ .., 5, .. ]`
`A[ 5..10 ]`	`A[ 5..10, .., .. ]`
`A[ 5..10, 7 ]`	`A[ 5..10, 7, .. ]`
`A[ 5..10, 7..13 ]`	`A[ 5..10, 7..13, .. ]`

Table 3.9.1 Correct interpretation of shorthand notation.

3.10 Omitting range terminals

The starting and stopping indexes on either side of the index-range symbol are called *terminals*. The starting index is called the *lower terminal* and the stopping index is called the *upper terminal*.

The lower, upper or both terminals may be omitted when using index-range notation. When a terminal is omitted, a default value is used as shown below.

When this terminal is omitted...	**This value is used...**
Lower terminal	Zero
Upper terminal	Maximum legal index for the dimension

Table 3.10.1 Default values used when range terminals are omitted.

Example 3.10.1: Zeroing the right-hand side of an image or sequence

This example zeroes the right-hand side of an image or sequence without reference to the actual size of the array.

```
var
  A ;
begin
GetActiveImage( A ) ;
A[ 250.. ] := 0 ;  // Columns 0 thru 249 remain intact
Update( A ) ;
end
```

Listing 3.10.1 Zeroing the right-hand side of an image or sequence.

Example 3.10.2: Setting a sequence of frames equal to the first frame

This example makes all frames in a sequence equal to the first frame, but without reference to the length of the sequence.

```
var
  Seq ;
begin
GetActiveImage( Seq ) ;
Seq[ .., .., 1.. ] := Seq[ .., .., 0 ] ;
Update( Seq ) ;
end
```

Listing 3.10.2 Setting a sequence of frames equal to the first frame.

3.11 Requirements for the right-hand side of an assignment

3.11.1 Right-hand side is a number

When the right-hand side of an assignment is a number, the number is copied to every array location specified on the left-hand side. If the data type of the number is different from the data type of the array, the number is converted to the same type as the array before being assigned.

Example 3.11.1: Assigning a number to an image

This example assigns a number to arrange of pixel locations in an image. The number is the integer value 120 whereas the image is 24-bit color. Consequently the number is converted to the RGB value (120,120,120) before being assigned to the array.

```
var
  Image, Number ;
begin
Image := CreateImage( RGB, 100, 100 ) ; // Image is 24-bit color
Number := 120 ;                         // Number is the integer 120
Image[ 20..70, 35..80 ] := Number ;     // Number converted to RGB on the fly
Show( Image ) ;
end
```

Listing 3.11.1 Assigning a number to part of an image with type-conversion on the fly.

3.11.2 Right-hand side is an array

When the left-hand and right-hand sides of an expression are arrays, they are not required to be exactly the same size, although a few simple requirements must be satisfied.

> The left-hand and right-hand sides are assignment-compatible provided:
>
> *The right-hand side contains more elements than is specified by the index-ranges on the left-hand side*
>
> OR
>
> *The right-hand side contains fewer elements than is specified by the index-ranges on the left-hand side BUT the right-hand array and left-hand index-ranges have the same x-size*.

When the right-hand array has a data type different from the left-hand array the data type of the right-hand array is changed on the fly before the assignment is made.

Example 3.11.2: Array assignment when there are more elements on the right-hand side

In this example, a 70 by 300 region of the image A is extracted to image B and assigned to a 10 by 10 region of the original image A. Only the first 100 pixels from B are required to fulfill the assignment requirement.

```
var

  A, B ;

begin

GetActiveImage( A ) ;

B := A[ 0..69, 100..399 ] ;   // Extract 70 by 300 region to B

A[ 10..19, 30..39 ] := B ;    // Assign first 100 elements of B to A as specified

Update( A ) ;

end
```

Listing 3.11.2 Array assignment when there are more elements on the right-hand side. Elements will be taken from the right-hand side array as required to satisfy the left-hand side index-ranges.

Example 3.11.3: Fixing a CCD image with noisy rows

This example fixes a problem where a CCD camera consistently produces noisy data in the first 3 rows of an image. Rows 0 through 2 are replaced with row 3.

```
var
  A, B ;
begin
GetActiveImage( A ) ;
A[ .., 0..2 ] := A[ .., 3 ] ;
Update( A ) ;
end
```

Listing 3.11.3 Fixing a noisy CCD image by replacing the offending noisy rows with a representative noise-free row.

Example 3.11.4: Creating a "ramp image" from just one row of data

This example creates a one-dimensional ramp and then assigns it to every row in an image. The for-loop is required for the initial ramp only.

```
var
  Ramp ;
  Image ;
begin
Ramp := CreateArray( word, 500 ) ;
Image := CreateArray( word, 500, 400 ) ;
for i := 0 to 499 do  // Make the ramp
  Ramp[ i ] := i ;
Image[ .. ] := Ramp ;  // Copy the ramp to every row of the image
Show( Image ) ;
end
```

Listing 3.11.4 Creating a ramp image using a one-dimensional ramp as a "seed."

Example 3.11.5: Creating a "ramp image" using just one for-loop

This example creates a ramp image where each column is filled simultaneously.

```
var
  Image ;
begin
Image := CreateArray( word, 500, 400 ) ;
for i := 0 to 399 do
  Image[ i, .. ] := i ;  // Set the value of the entire column
Show( Image ) ;
end
```

Listing 3.11.5 Creating a ramp image using one for-loop.

3.12 Copying a column into a row, etc

It is sometimes necessary to transpose data from an array. The most common example is where a column of data from one image becomes the row data for another image. This is easily handled by index-range notation because of the unrestrictive assignment rules.

Example 3.12.1: Copying an image column to a row

This example extracts a column from an image and inserts it into a row. The assignment works because the actual dimensions of the right-hand side are immaterial if there are sufficient pixels to fulfill the assignment operation.

Note: The example assumes that the length of a row is the same as the height of a column. If this is not the case, simply modify the index ranges accordingly.

```
var
  A ;
begin
GetActiveImage( A ) ;
A[ .., 5 ] := A[ 7, .. ] ;  // Copy column 7 to row 5. Assumes same size!
Update( A ) ;
end
```

Listing 3.12.1 Copying a column of pixels into a row, assuming a square image.

4.0 Conditional calculations

4.1 Conditional statements

A statement is *conditional* if its execution depends upon the outcome of a logical test.

Conditional statements are of the form

Conditional statement	**Explanation**

`if` `<condition>` **`then`**
  `<statement>`
	`<statement>` is executed if `<condition>` is TRUE

`if` `<condition>` **`then`**
  `<statement-1>`
`else`
  `<statement-2>`
	`<statement-1>` is executed if `<condition>` is TRUE otherwise `<statement-2>` is executed

In all cases `<condition>` is an expression that evaluates to a binary scalar value.

4.2 Relational operators

A relational operator compares two quantities and returns TRUE if the relation is satisfied, else it returns FALSE. The standard relational operators are show in the table below.

Operator	**Description**
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to
=	Equal to

Table 4.2.1 Relational operators

Notes

1. When the arguments to a relation are of different types, one argument is promoted so that both have the same type in accordance with the type precedence rules.
2. When the arguments are complex (either single- or double-precision) the relation is computed using the magnitudes of the complex numbers.
3. When the arguments are color the relation is computed using the intensities of the color numbers.
4. When both arguments are scalars (simple numbers) the result of the comparison is a scalar binary value.
5. When one argument is a scalar and the other is an array, the result of the comparison is a binary array with the same dimensions as the array argument. The comparison is made between the scalar and every element of the array.
6. When both arguments are arrays, the result of the comparison is a binary array with the same dimensions as the array arguments. (If both arguments are arrays they must have the same dimensions.)

4.2.1 Relational operators applied to scalars

Relational operators applied to scalars yield binary scalar results.

Example 4.2.1: Comparison of two scalars

This example compares two scalar values and reports if one is greater than the other.

```
var
  a, b ;
begin
a := 17 ;
b := 11 ;
if a > b then
  WriteInfo( '17 > 11 is TRUE' )
else
  WriteInfo( '17 > 11 is FALSE' ) ;
end
```

Listing 4.2.1 Comparison of two scalars.

Example 4.2.2: Comparison of two scalars—alternative method

This example compares two scalar values and reports if one is greater than the other. The difference between this and the previous example is that the relation can be tested, stored in a variable, and reused when needed.

```
var
  a, b ;
  Test ;
begin
a := 17 ;
b := 11 ;
Test := a > b ;
WriteInfo( '17 > 11 is ',Test ) ;  // Output will say TRUE
end
```

Listing 4.2.2 Comparison of two scalars. The use of a binary variable to save the result of the comparison can often lead to more elegant and simpler code.

4.2.2 Relational operators applied to images

Relational operators applied to images, or to images and scalars, yield binary arrays.

When comparing two images the images must have exactly the same dimensions. Each binary pixel in the output array is the result of applying the relational operator to the corresponding pixels in the arguments.

When comparing an image and a scalar, each binary pixel in the output array is the result of applying the relational operator to the corresponding pixel in the array argument and the scalar.

Example 4.2.3: Comparison of two images

This example compares two images and displays a "map" of where one is greater than the other.

```
var
  A, B ;
  Test ;
begin
SelectImage( 'Choose first image',A ) ;
SelectImage( 'Choose second image',B ) ;
Test := A > B ;
Show( Test,'AGreaterThanB' ) ;
end
```

Listing 4.2.3 Comparison of two images. The output image Test will be 1 wherever A is greater than B and 0 elsewhere.

Example 4.2.4: Comparison of two images

This example compares an image and a scalar, and displays a "map" of where the array is greater than the scalar.

```
var
  A ;
  Test ;
begin
SelectImage( 'Choose an image',A ) ;
Test := A > 200 ;
Show( Test,'AGreaterThan200' ) ;
end
```

Listing 4.2.4 Comparison of an image and a scalar. The relation is tested between the scalar and every pixel in the array. The output image Test will be 1 wherever A is greater than 200 and 0 elsewhere.

4.3 The Any and All functions

The test performed in a conditional statement must resolve to a binary scalar value. However, a relational operator applied to array arguments yields a binary array result. Frequently, action must be taken if *all* pixel comparisons are true, or if *any* pixel comparisons are true. These tasks are handled by the `Any` and `All` functions.

4.3.1 The Any function

The `Any` function takes one binary array as its argument and returns the binary value TRUE if *any* of the array's pixels are TRUE.

Example 4.3.1: Stop a division operation if any divisor pixels are zero

This example attempts to divide two images but halts if the divisor contains any zero-valued pixels. Note that the `Any` function takes the result of the array-comparison `D = 0` and returns TRUE if at least one pixel is TRUE.

```
var
  N, D, Q ;
begin
SelectImage( 'Choose numerator image',N ) ;
SelectImage( 'Choose denominator image',D ) ;
if Any( D = 0 ) then
  Halt( 'Denominator contains zeros!' )
else
  Q := N / D ;
Show( Q ) ;
end
```

Listing 4.3.1 Division of two images. The denominator is tested for zeros before attempting the division.

4.3.2 The All function

The `All` function takes one binary array as its argument and returns the binary value TRUE if *all* of the array's pixels are TRUE.

Example 4.3.2: Check that all pixels are above zero

This example checks that all pixels in an image are above zero before proceeding. Note that the `All` function takes the result of the array-comparison `A > 0` and returns TRUE if all pixels are TRUE.

```
var

  A ;

begin

GetActiveImage( A ) ;

if not All( D > 0 ) then

  Halt( 'Parts of image saturated at ZERO!' ) ;

end
```

Listing 4.3.2 Checking if an image is saturated at zero. Note that "**not** `All( A > 0 )`" is equivalent to "`Any( A <= 0 )`".

4.4 The Find function

The `Find` function takes a binary array argument and returns the coordinates of all pixels that are TRUE. This function is useful when trying to locate the positions of pixels that obey a specific relation. The coordinates are returned in each row of an array.

If the argument is a simple two-dimensional array `Find` will return a 2 by N array of coordinates. Each row will contain the x-y coordinates of the TRUE pixels.

If the argument is a three-dimensional array (a sequence) `Find` will return a 3 by N array of coordinates. Each row will contain the x-y-z coordinates of the TRUE pixels.

Example 4.4.1: Locate saturated pixels in a camera image

This example assumes that a 12-bit camera produces the image. The image data will therefore saturate at a gray level of 4095. The code outputs the location of all saturated pixels.

```
var
  A, Coords ;
  n ;
begin
GetActiveImage( A ) ;
Coords := Find( A = 4095 ) ;
for n := 0 to GetYMax( Coords ) do
  Writeln( '[',Coords[0,n],',',Coords[1,n],']' ) ;
end
```

Listing 4.4.1 Locating saturated pixels in an image.

Example 4.4.2: Locate saturated pixels in a camera image—more elegant approach

As in the previous example, it is assumed that a 12-bit camera produces the image. The image data will therefore saturate at a gray level of 4095. The code outputs the location of all saturated pixels.

```
var
  n, A, SatMap, Coords ;
begin
GetActiveImage( A ) ;
SatMap := ( A = 4095 ) ;  // Show where pixels are maxed out
if Any( SatMap ) then
  begin
  Show( SatMap,'Saturation Map' ) ;
  for n := 0 to GetYMax( Coords ) do
    Writeln( '[',Coords[0,n],',',Coords[1,n],']' ) ;
  end ;
end
```

Listing 4.4.2 Locating saturated pixels in an image by first creating a "saturation map."

4.5 Counting pixels

It is easy to count the number of pixels that satisfy a relation because when the pixels in a binary array are summed together, the result is the number of pixels set to TRUE.

Example 4.5.1: Number of saturated pixels in an image

This example assumes a 12-bit camera produces image data. The image data will therefore saturate at a gray level of 4095. The code displays the number of saturated pixels in the image.

```
var
  A ;
  nSaturated ;
begin
GetActiveImage( A ) ;
nSaturated:= SumOf( A = 4095 ) ;
if nSaturated > 0 then
  WriteInfo( 'There are ',nSaturated,' pixels in the image.' ) ;
end
```

Listing 4.5.1 Counting the number of saturated pixels in an image.

Example 4.5.2: Number of saturated pixels in an image—more complex case

This example assumes a 12-bit camera produces image data. The image data will therefore saturate at a gray level of 4095. If the gain and offset of the analogue to digital converter are not set correctly then pixels may also saturate at 0. The code displays the number of pixels in the image saturated at 0 and 4095.

```
var
  A ;
  nSaturated ;
begin
GetActiveImage( A ) ;
nSaturated:= SumOf( ( A = 4095 ) or ( A = 0 ) ) ;
if nSaturated > 0 then
  WriteInfo( 'There are ',nSaturated,' pixels in the image.' ) ;
end
```

Listing 4.5.2 Counting the number of saturated pixels in an image.

5.0 Strings and filenames

5.1 String format

A string is stored internally as an array of characters. Accessing characters in a string follows the Pascal convention. That is, the characters in a string are indexed from 1 (not 0, as in C).

5.2 String output functions

The easiest way to get simple results from a module is to output data as a string. This can achieved either by writing to a text window or editor, or by displaying a dialog box.

5.2.1 Write and Writeln functions

The Write and Writeln functions behave just like their traditional Pascal counterparts. Both routines can write to a default editor, or a user-created editor.

The main difference between the two routines is that Writeln automatically appends a carriage-return / line-feed pair to the output text. This has the effect of moving the output caret to the next line of the editor.

Write and Writeln take an arbitrary number of parameters, making it easy to build richly formatted output.

Writeln may be called without arguments in which case the output caret simply moves to the next line.

Example 5.2.1: Using Write and Writeln

This example shows Write and Writeln outputting both strings and scalars in the same line. Since no editor has been created, the output goes to the default output window.

```
var
  A ;
  Mean, Min, Max ;
begin
GetActiveImage( A ) ;
Mean := MeanOfAll( A ) ;  // Average value of image
Min := MinOfAll( A ) ;    // Minimum value of image
Max := MaxOfAll( A ) ;    // Maximum value of image
Writeln( 'The mean value is ',Mean ) ;
Write( 'The extreme values are ' ) ;
Writeln( Min,' and ',Max ) ;
end
```

Listing 5.2.1 Outputting text and numbers using Write and Writeln.

Example 5.2.2: Outputting image size and type

When Write or Writeln is called with an array argument, the size and type of the array is displayed. This can be a useful debugging aid when developing modules.

```
var
  A ;
begin
GetActiveImage( A ) ;
Writeln( A ) ;  // Output is, say, "Array[ 1035,1013 ] of word"
end
```

Listing 5.2.2 Getting array information using Write or Writeln.

Example 5.2.3: Displaying a small array as text

Often it is useful to see the pixel values of an image in text format. This example outputs the top-left 10 by 10 portion of any array.

```
var
  A ;
  procedure WriteArray( var Data ) ;
  var
    x, y ;
  begin
  for y := 0 to 9 do
    begin
    for x := 0 to 9 do
      Write( Data[ x,y ]:6 ) ;
    Writeln ;
    end ;
  end ; {WriteArray}
begin
GetActiveImage( A ) ;
WriteArray( A ) ;
end
```

Listing 5.2.3 Displaying image pixels in an editor.

5.2.2 WriteInfo and WriteError functions

`WriteInfo` and `WriteError` are essentially identical routines: both display an informational message in a dialog that the user must dismiss by pressing the OK button. The difference between the two procedures is that they have different icons.

`WriteInfo` displays an "information icon" to suggest that the content is simply informative.

`WriteError` displays an "error icon" to suggest that something drastic or fatal has happened.

`WriteInfo` and `WriteError` can be called with an arbitrary number of parameters in the same way as `Write` and `Writeln`.

Example 5.2.4: Displaying messages with WriteInfo and WriteError

```
var
  A ;
begin
GetActiveImage( A ) ;
GetXYSize( A, xSize, ySize ) ;
if not ( IsPowerOfTwo( xSize ) and IsPowerOfTwo( ySize ) ) then
  WriteError( 'Size is not a power of two!' )  // Problem
else
  begin
  A := fft( A ) ;
  WriteInfo( 'FFT complete' ) ;  // Done
  end ;
end
```

Listing 5.2.4 Using WriteInfo and WriteError to display information.

5.3 String input functions

5.3.1 GetString function

The `GetString` function prompts the user to enter a string using a dialog box. It is possible to determine whether the user pressed the OK or Cancel buttons to close the dialog.

Example 5.3.1 Using GetString with no error checks

This example assumes that a string is always entered in response to the prompt. See example 5.3.2 to see how to respond when the user presses the Cancel button.

```
begin
GetString( 'Enter a string', Msg ) ;
WriteInfo( 'You entered the string "', Msg, '"' ) ;
end
```

Listing 5.3.1 Using GetString to enter information into a module.

Example 5.3.2 Using GetString with an error check

This example does not assume that a string is always entered in response to the prompt.

```
var
  Code ;
begin
Code := GetString( 'Enter a string', Msg ) ;
if Code = id_OK then
  WriteInfo( 'You entered the string "', Msg, '"' )
else
  WriteError( 'You cancelled the dialog!' ) ;
end
```

Listing 5.3.2 Using GetString robustly to enter information into a module.

5.3.2 SelectString function

The `SelectString` function allows the user to select from a list of strings. The list is created by concatenating a number of smaller strings delimited by semicolons.

Example 5.3.3 Using SelectString to choose from a list of strings

This example creates a list of options and prompts the user to select one of them.

```
var
  Index ;
  List ;
  Item ;
begin
List := 'Lowpass;Highpass;Bandpass' ;
Index := SelectString( 'Select one of these filters:', List, Item ) ;
if Index > 0 then
  WriteInfo( 'You selected the ', Item, ' filter.' )
else
  WriteError( 'You cancelled the dialog!' ) ;
end
```

Listing 5.3.3 Using SelectString to choose from a list of alternatives.

5.4 String-number conversions

5.4.1 Converting a number to a string

A number is converted to a string using the `Str` function. The `Str` function takes one scalar argument and returns a string. The argument may be qualified to control the width of the string holding the number. The syntax of `Str` is shown in the table below.

Syntax	Output	Comment
`s := Str( 123 ) ;`	`'123'`	
`s := Str( 123:6 ) ;`	`'   123'`	Allow 6 spaces in total
`s := Str( 123.0 ) ;`	`'123.0000'`	
`s := Str( 123.0:8:2 ) ;`	`'  123.00'`	Allow 8 spaces in total with 2 decimal places

Table 5.4.1 Sample syntax for the `Str` function

5.4.2 Converting a string to a number

A string is converted to a number using the `Val` function. The `Val` function takes one string parameter and returns the best scalar compatible with the string. The syntax of `Val` is shown in the table below.

Syntax	**Output**
`x := Val( '123' ) ;`	Fixed-point number 123
`x := Val( '123.456' ) ;`	Floating point number 123.456
`x := Val( '123 456' ) ;`	Complex number 123+i456
`x := Val( '123 456 789' ) ;`	48-bit RGB number (123,456,789)
`x := Val( '$FF' ) ;`	Fixed-point number 255

Table 5.4.2 Sample syntax for the Val function

5.5 Common string operations

5.5.1 Extracting a sub-string

Use `ExtractStr` to copy a sub-string from a larger string. Note that strings are indexed from 1 (as in Pascal) rather than 0 (as in C). When using `ExtractStr` specify the starting position in the main string and the number of characters to copy.

Example 5.5.1: Extracting a sub-string from another string

```
var
  Main, Sub ;
begin
Main := 'Digital Optics' ;
Sub := ExtractStr( Main, 9, 6 ) ;  // Extract 6 chars starting from char 9
Writeln( Main ) ;  // 'Digital Optics'
Writeln( Sub ) ;   // 'Optics'
end
```

Listing 5.5.1 Extracting a sub-string from another string using ExtractStr.

5.5.2 Deleting a sub-string

Use `DeleteStr` to delete characters from a string. Note that strings are indexed from 1 (as in Pascal) rather than 0 (as in C). When using `DeleteStr` specify the starting position in the string and the number of characters to delete.

Example 5.5.2: Deleting a range of characters from a string

```
var
  S ;
begin
S := 'Digital Optics' ;
Writeln( S ) ;          // 'Digital Optics'
DeleteStr( S, 9, 6 ) ;  // Delete 6 chars starting from char 9
Writeln( S ) ;          // 'Optics'
end
```

Listing 5.5.2 Deleting a range of characters from another string using DeleteStr.

5.5.3 Inserting a sub-string

Use `InsertStr` to insert a string into another string. Note that strings are indexed from 1 (as in Pascal) rather than 0 (as in C). When using `InsertStr` specify the sub-string to insert, the main string in which to insert the sub-string, and the starting position in the main string.

Example 5.5.3: Inserting a sub-string into another string

```
var
  S ;
begin
S := 'Precision Imaging' ;
Writeln( S ) ;                   // 'Precision Imaging'
InsertStr( 'Digital', S, 11 ) ;  // Insert 'Digital' starting at char 11
Writeln( S ) ;                   // 'Precision Digital Imaging'
end
```

Listing 5.5.3 Inserting a sub-string into another string using InsertStr.

5.5.4 Replacing a sub-string

Use `ReplaceStr` to replace one or more sub-strings in a string with an alternative sub-string. `ReplaceStr` can be made sensitive to case.

Example 5.5.4: Replacing a sub-string with another sub-string

```
var
  FileName ;
begin
FileName := 'C:\Documents\Images\19990426.tif' ;
Writeln( FileName ) ;  // 'C:\Documents\Images\19990426.tif'
ReplaceStr( FileName, '2000', '1999', rs_ReplaceAll+rs_IgnoreCase ) ;
Writeln( FileName ) ;  // 'C:\Documents\Images\20000426.tif'
end
```

Listing 5.5.4 Replacing all occurrences of a sub-string with another sub-string using ReplaceStr.

5.5.5 Joining (concatenating) two strings

Use the + operator to join two or more strings.

Example 5.5.5: Joining strings with the + operator

```
var
  Path, Dir, Name, Ext ;
begin
Dir := 'C:\Documents\' ;
Name := 'A0001' ;
Ext := '.tif' ;
Path := Dir + Name + Ext ;  // 'C:\Documents\A0001.tif'
end
```

Listing 5.5.5 Using the + operator to join strings.

5.5.6 Trimming spaces from a string

It is good practice to remove extraneous spaces from text when the user enters strings. This is especially true when the string is subsequently converted to a number. Three functions are available for removing spaces from strings, as shown in the table below.

Function	Description
`TrimLeft`	Remove leading spaces from the string
`TrimRight`	Remove trailing spaces from the string
`Trim`	Remove leading and trailing spaces from the string

Table 5.5.1 Functions to trim spaces from a string

Example 5.5.6: Trimming spaces from a string

```
var
  S ;
begin
S := '  123   ' ;
Writeln( '<',S,'>' ) ;                    // <  123   >
Writeln( '<',TrimLeft( S ),'>' ) ;        // <123   >
Writeln( '<',TrimRight( S ),'>' ) ;       // <  123>
Writeln( '<',Trim( S ),'>' ) ;            // <123>
end
```

Listing 5.5.6 Removing spaces from a string using the trimming functions.

5.5.7 Comparing strings

To perform a case-sensitive comparison of two strings use the `CompareStr` function or the equals sign.

To perform a case-insensitive comparison of two strings use the `CompareText` function

Example 5.5.7: Comparing strings

```
var

  S ;

begin

S := 'Digital Optics' ;

if CompareStr( S, 'digital optics' ) = 0 then

  Writeln( 'You should NOT see this message!' ) ;

if S = 'digital optics' then

  Writeln( 'Nor should you see this message!' ) ;

if CompareText( S,'digital optics' ) = 0 then

  Writeln( 'You WILL see this message' ) ;

end
```

Listing 5.5.7 Comparing strings using case-sensitive and case-insensitive routines.

5.5.8 Parsing a string into sub-strings

There are many applications where a string is the concatenation of several smaller strings. The smaller strings are called *tokens*. Special characters, called *delimiters*, separate the smaller strings. The act of extracting tokens from the main string is called *parsing*.

For example, the user may be prompted to enter a coordinate (as a string) in the form "123, 456" or "123 456". The task of a parser is to extract the tokens 123 and 456 given that the delimiter characters are a space and / or a comma.

The function `StrParse` is equipped to parse tokens from a string with any number of delimiter characters.

Example 5.5.8: Extracting pixel coordinates from a string

This example extracts two numbers from a string by parsing the string into separate tokens. The user may enter the coordinates separated by a space, a comma, or both.

```
var
  CoordString ;
  x, y ;
  Delimits ;
begin
GetString( 'Enter the coordinates', CoordString ) ;  // Prompt for the coords
Delimits := ' ,' ;  // Two delimiters: space and comma
Token := StrParse( CoordString, Delimits ) ;  // Extract first token
x := Val( Token ) ;
Token := StrParse( CoordString, Delimits ) ;  // Extract second token
y := Val( Token ) ;
WriteInfo( 'Coordinates are (', x, ',', y, ')' ) ;
end
```

Listing 5.5.8 Extracting integer-valued coordinates using `StrParse`.

Example 5.5.9: Extracting an unknown number of values from a string

This example extracts an unknown number of values from a string. A space, comma, tab, or any combination may separate the tokens.

```
var
  String, Token ;
  Delimits ;
  Count ;
begin
GetString( 'Enter the coordinates', String ) ;  // Prompt for a string to parse
Delimits := ' ,' + chr(9) ;  // chr(9) = tab character
Count := 0 ;  // Number of tokens parsed
repeat
  Token := StrParse( String, Delimits ) ;  // Extract token
  if length( Token ) > 0 then
    begin
    Writeln( Token ) ;    // Output value
    Count := Count + 1 ;  // Keep track of number of tokens for later (?)
    end ;
until length( String ) = 0 ;
Writeln ;
Writeln( Count, ' values extracted' ) ;
end
```

Listing 5.5.9 Extracting an unknown number of tokens from a string using `StrParse`.

5.6 Filenames

A filename consists of a number of different components as shown in Table 5.6.1 below.

Component	**What it is**	**Example**
Drive	Disk drive that hosts the file	`C:`
Path	List of directories that "point" to the file	`\Documents\Images\`
Name	Name of the file	`Test`
Extension	Extension that identifies the type of file	`.tif`
Filename	Complete file specification	`C:\Documents\Images\Test.tif`

Table 5.6.1 Filename components.

5.6.1 Extracting filename components

A number of routines are provided for dissecting a filename into its components:

Function	**Purpose**
`ChangeFileExt`	Alter the extension of a filename
`ExpandFileName`	Expand a partial file specification into a full specification
`ExtractFileDir`	Extract the directory portion of the filename
`ExtractFileDrive`	Extract the drive portion of the filename
`ExtractFileExt`	Extract the extension portion of the filename
`ExtractFileName`	Extract the name and extension of the filename
`ExtractFilePath`	Extract the path portion of the filename (includes trailing backslash)

Table 5.6.2 Useful filename manipulation functions.

Example 5.6.1 Extracting filename components

The example shows how to use the functions listed in Table 5.6.2 to extract various components of a filename.

```
var
  FileName ;
  Drive, Dir, Ext, Name, Path ;
begin
FileName := 'C:\Documents\Images\A0001.tif' ;
Writeln( ExtractFileDrive( FileName ) ) ;  // 'C:'
Writeln( ExtractFileDir( FileName ) ) ;    // 'C:Documents\Images'
Writeln( ExtractFileExt( FileName ) ) ;    // '.tif'
Writeln( ExtractFileName( FileName ) ) ;   // 'A0001.tif'
Writeln( ExtractFilePath( FileName ) ) ;   // 'C:Documents\Images\'
end
```

Listing 5.6.1 Extracting filename components

6.0 File handling

6.1 Does a file exist?

When reading information from disk it is a common problem to determine whether a given file exists. The `FileExists` function return TRUE if the specified file is found on disk.

Example 6.1.1:

This example determines if a certain image exists, and then reads the image onto the desktop.

```
begin

if FileExists( 'C:\Documents\Images\A0001.tif' ) then

  OpenToDesktop( 'C:\Documents\Images\A0001.tif' ) ;

end
```

Listing 6.1.1 Using `FileExists` to check for existence of a file.

6.2 Searching a sequence of directories for a file

Often images are stored in a hierarchy of folders organized by experiment or otherwise. To search a list of directories for a specific file, use the `FileSearch` function. This function takes a list of directories, separated by semicolons, and returns the path to the required file, if found.

Example 6.2.1: Searching a list of directories

This example scans two directories, `C:\Documents\Images\Day1` and `C:\Documents\Images\Day2`, for the image `A0001.tif`. If the image is found, it is displayed on the desktop.

```
const
  Path1 = 'C:\Documents\Images\Day1' ;
  Path2 = 'C:\Documents\Images\Day2' ;
var
  List ;
  File ;
begin
List := Path1 + ';' + Path2 ;
File := FileSearch( 'A0001.tif', List ) ;
if length( File ) > 0 then
  OpenToDesktop( File ) ;
end
```

Listing 6.2.1 Using `FileSearch` to scan a list of directories.

6.3 Iterating over a sequence of files

It is possible to iterate over an entire collection of files using the `FindFirstFile` and `FindNextFile` functions. For example, all images from an experiment may be stored in a single folder or perhaps may be stored using a specific naming scheme, such as `A0001.tif`, `A0002.tif`, etc. In either case it is an easy matter to over all the images in a systematic way.

Example 6.3.1: Scanning all files in a specific folder

This example opens all TIFF images found in a specific folder.

```
var
  Name ;
begin
Name := FindFirstFile( 'C:\Documents\Images\*.tif', fa_Archive ) ;
while length( Name ) > 0 do
  begin
  OpenToDesktop( Name ) ;
  Name := FindNextFile ;
  end ;
end
```

Listing 6.3.1 Using `FindFirstFile` / `FindNextFile` to open all TIFF images in a folder.

Example 6.3.2: Scanning all files with a specific filename signature

This example opens all TIFF images of the form `A0001.tif`, `A0002.tif`, etc, found in a specific folder.

```
var
  Name ;
begin
Name := FindFirstFile( 'C:\Documents\Images\A*.tif', fa_Archive ) ;
while length( Name ) > 0 do
  begin
  OpenToDesktop( Name ) ;
  Name := FindNextFile ;
  end ;
end
```

Listing 6.3.2 Using `FindFirstFile` / `FindNextFile` to open TIFF images with the same filename signature.

6.4 File-size and disk-size functions

To determine the size of a specific disk, use the `DiskSize` function.

To determine the free space on a specific disk, use the `DiskFree` function.

To determine the size of a specific file on disk, use the `FileSize` function.

6.5 Creating a directory

Create a new folder or directory using the `CreateDir` procedure.

Example 6.5.1: Create a new directory for storing experimental results

```
const

  NewFolder = 'C:\Documents\Experiment28' ;

begin

CreateDir( NewFolder ) ;

end
```

Listing 6.5.1 Creating a new folder for experiment files.

6.6 Changing the default directory

Change to a different folder or directory using the `SetDir` procedure.

Example 6.6.1: Create a new directory for storing experimental results then make it the default directory

```
const

  NewFolder = 'C:\Documents\Experiment28' ;

begin

CreateDir( NewFolder ) ;

SetDir( NewFolder ) ;

end
```

Listing 6.6.1 Creating a new folder and making it the default folder.

7.0 Menus and toolbars

7.1 Running a module from a menu

A module can be activated from a menu by inserting the reserved word **menu** at the beginning of a module.

Example 7.1.1: A menu-activated module

This example activates a module when 'Invert the Image' is selected from the User menu.

```
menu 'Invert the Image' ;

var

  A ;

begin

GetActiveImage( A ) ;

if IsImage( A ) then

  A := not A ;

end
```

Listing 7.1.1 Menu-activation of a module

7.2 Running a procedure from a menu

An individual procedure in a module can be executed by associating the **menu** reserved word with each procedure that requires it.

In order that a procedure is able to launch from a menu click, it cannot be declared with parameters.

Functions cannot be directly activated from a menu.

Example 7.2.1: Activating individual procedures from a menu

This example activates each procedure when the appropriate entry is selected from the main User menu. The menu entries will appear in the User menu in the order they are declared in the module.

```
  procedure Invert ; menu '-Invert the Image' ;
  var
    A ;
  begin
  GetActiveImage( A ) ;
  A := not A ;
  end ; {Invert}

  procedure ConvertToByte ; menu 'Convert to byte' ;
  var
    A ;
  begin
  GetActiveImage( A ) ;
  if TypeOf( A ) <> typ_Byte then
    ConvertType( A, typ_Byte ) ;
  end ; {ConvertToByte}

begin
end
```

Listing 7.2.1 Menu-activation of procedures.

7.3 Separators and accelerators

When several procedures or modules add entries to the User menu it is often a good idea to group the available options functionally, separated by horizontal lines. These lines are called separators.

To add a separator, make sure the first character of the menu text is '-'. For example, '-Invert the Image' will place a separator before the menu text 'Invert the Image'.

Accelerators are the underlined characters that appear in a menu. When the menu is displayed, the accelerator character can be pressed on the keyboard rather than having to move the mouse.

To specify a character as the accelerator character, place an ampersand immediately before the character. For example, 'In&vert the Image' will be displayed as 'In<u>v</u>ert the Image'.

7.4 Running a module from a toolbar button

Running a module from a button click is just as easy as adding a menu entry. Simply add the reserved word **`button`** at the beginning of a module, followed by the button icon identifier. V++ comes with hundreds of predefined button images. Consult the help file for a complete list.

Example 7.4.1: A button-activated module

This example activates a module when the 'Hand' button is pressed on the toolbar.

```
button btn_Hand ;

var

  A ;

begin

GetActiveImage( A ) ;

if IsImage( A ) then

  A := not A ;

end
```

Listing 7.4.1 Button-activation of a module

Example 7.4.2: A button-activated module with help string

This example activates a module when the 'Hand' button is pressed on the toolbar. It also adds a useful help string that appears when the mouse hovers over the button for a moment. This assists the user in determining the function of a particular button.

```
button btn_Hand, 'Invert the image' ;

var

  A ;

begin

GetActiveImage( A ) ;

if IsImage( A ) then

  A := not A ;

end
```

Listing 7.4.2 Button-activation of a module with help string

7.5 Running a procedure from a toolbar button

An individual procedure in a module can be executed by associating the `button` reserved word with each procedure that requires it.

In order that a procedure is able to launch from a button click, it cannot be declared with parameters.

Functions cannot be directly activated from a button.

Example 7.5.1: Activating individual procedures from a button

This example activates each procedure when the appropriate button is pressed. The buttons will appear on a toolbar in the order they are declared in the module.

```
  procedure Invert ; button btn_I, 'Invert the Image' ;

  var

    A ;

  begin

  GetActiveImage( A ) ;

  A := not A ;

  end ; {Invert}


  procedure ConvertToByte ; button btn_B, 'Convert to byte' ;

  var

    A ;

  begin

  GetActiveImage( A ) ;

  if TypeOf( A ) <> typ_Byte then

    ConvertType( A, typ_Byte ) ;

  end ; {ConvertToByte}


begin

end
```

Listing 7.5.1 Button-activation of procedures.

7.6 Naming a toolbar

All the buttons defined in a specific module appear on the same toolbar. A module's toolbar can be given a useful name using the **`toolbar`** reserved word. Only one **`toolbar`** reserved word is permitted module.

Example 7.6.1: Naming a toolbar for a module

This example has two button-activated procedures. Both buttons appear on the toolbar called 'Custom Tools'.

```
toolbar 'Custom Tools' ;


  procedure Invert ; button btn_I, 'Invert the Image' ;

  var

    A ;

  begin

  GetActiveImage( A ) ;

  A := not A ;

  end ; {Invert}


  procedure ConvertToByte ; button btn_B, 'Convert to byte' ;

  var

    A ;

  begin

  GetActiveImage( A ) ;

  if TypeOf( A ) <> typ_Byte then

    ConvertType( A, typ_Byte ) ;

  end ; {ConvertToByte}


begin

end
```

Listing 7.6.1 Specifying the name of a toolbar.

8.0 Running code at start-up and at shutdown

8.1 Making a module ready to run at start-up

Having modules pre-compiled and ready to run can be very productive, especially when there are a number of novice users of the system. To automatically load and compile a module at startup, follow the steps below.

Step 1	Compile the module and verify that it is in a compiled (ready to run) state using the Module List dialog. (To launch the Module List dialog, select Tools | Module List on the main menu.)
Step 2	On the Module List dialog press the Startup button.
Step 3	Select the "Load and compile this module at startup" option and press the OK button.
Step 4	The next time V++ starts check the Module List dialog and you will see the module listed as ready to run.

8.2 Running a module at start-up

Automatically running a module at startup is an efficient way to ensure that an experimental system always starts from a known state. To automatically load and run a module at startup, follow the steps below.

Step 1	Compile the module and verify that it is in a compiled (ready to run) state using the Module List dialog. (To launch the Module List dialog, select Tools | Module List on the main menu.)
Step 2	On the Module List dialog press the Startup button.
Step 3	Select the "Load and run this module at startup" option and press the OK button.
Step 4	The next time V++ starts the module will automatically load and run.

8.3 Running a procedure at shutdown

It is generally considered good practice to "tidy up" at the end of an automated sequence of operations. For example, computer controlled equipment such as a stage may need to be restored to its "home" position. This action can take place automatically when V++ shuts down by nominating a module, or a specific procedure in a module, as the "shutdown code."

A procedure or module is identified as shutdown code by using the **shutdown** reserved word.

Each module is allowed only one procedure with the reserved word **shutdown**.

Shutdown procedures cannot accept parameters.

Example 8.3.1: Executing a shutdown procedure when V++ terminates

This example uses a shutdown procedure to close a camera at shutdown. Note that the module could have many user-defined procedures and functions, but only the procedure tagged as **shutdown** will be automatically executed when V++ terminates.

```
const
  CamName = 'PXL37' ;
  procedure Finished ; shutdown ;
  begin
  pvcCloseCamera( CamName ) ;
  end ; {Finished}
begin
pvcOpenCamera( CamName ) ;
{Main body of code here performs various image
 processing operations.}
end
```

Listing 8.3.1 Running a shutdown procedure.

9.0 Initialization files

9.1 What are initialization files?

Initialization, or INI, files are simple text files used to store information used by a program. Typically, an application writes information to an INI file when it terminates so that when restarted it can restore certain conditions in the program.[*]

An INI file is divided into a number of sections. Each section has a heading or title, and is followed by a series of lines with the general format keyword=value. An INI file will therefore look something like this:

General Format	**Example**
`[Section1]`	`[Camera]`
`Keyword1=Value1`	`ExposureTime=155`
`Keyword2=Value2`	`FrameCount=3`
`...`	
`KeywordN=ValueN`	`[Database]`
	`ImagePath=C:\Document\Images`
`[Section2]`	`ResultsPath=C:\Documents\Data`
`Keyword1=Value1`	
`Keyword2=Value2`	
`...`	
`KeywordN=ValueN`	

Table 9.1.1 INI file layout

A program can arrange its content logically into sections, and use sensible keywords to save important information.

9.2 Using a private initialization file

A private INI file refers to an INI file created by the program. A private INI file can have any name, but it is usual to have an extension .INI.

To write values to a private INI file use the `WritePrivateINIString`.

To read values from a private INI file use the `ReadPrivateINIString` function.

[*] Note that most modern applications, including V++, store private initialization data in the Windows Registry database. However, INI files remain a useful means of storing application or module data in a human readable form. The V++ INI file is retained for convenience and backwards compatibility only.

Example 9.1.1: Writing to a private INI file

This example writes a number of parameters to a private INI file.

```
const
  IniFile = 'C:\Documents\Experiment57\Setup.ini' ;
  Section = 'Last Image' ;
var
  Image ;
  xSize, ySize ;
begin
GetActiveImage( Image ) ;
GetXYSize( Image, xSize, ySize ) ;
WritePrivateINIString( IniFile, Section, 'Name', GetName( Image ) ) ;
WritePrivateINIString( IniFile, Section, 'Width', xSize ) ;
WritePrivateINIString( IniFile, Section, 'Height', ySize ) ;
end
```

Listing 9.1.1 Saving information to a private INI file.

Example 9.1.2: Reading from a private INI file

This example reads a number of parameters from a private INI file.

```
const
  IniFile = 'C:\Documents\Experiment57\Setup.ini' ;
  Section = 'Camera' ;
var
  ExpTime, Name ;
begin
Name := ReadPrivateINIString( IniFile, Section, 'CameraName' ) ;
ExpTime := Val( ReadPrivateINIString( IniFile, Section, 'Exposure' ) ) ;
pvcOpenCamera( Name ) ;
pvcSetExpTime( ExpTime ) ;
end
```

Listing 9.1.2 Reading information from a private INI file.

Example 9.1.3: Reading from a private INI file with defaults

This example reads a number of parameters from a private INI file. It uses the default feature of `ReadPrivateINIString` so that if a keyword is absent from the INI file, a program-supplied alternative is used instead.

```
const
  IniFile = 'C:\Documents\Experiment57\Setup.ini' ;
  Section = 'Camera' ;
var
  ExpTime, Name ;
begin
Name := ReadPrivateINIString( IniFile, Section, 'CameraName' ) ;
ExpTime := Val( ReadPrivateINIString( IniFile, Section, 'Exposure', '100' ) ) ;
pvcOpenCamera( Name ) ;
pvcSetExpTime( ExpTime ) ;
end
```

Listing 9.1.3 Reading information from a private INI file with defaults. Note that if the Exposure keyword is absent, a default value of 100 is used.

9.3 Using the V++ initialization file

The V++ INI file behaves just like a private initialization file, except you do not need to refer to it by name, nor do you need to know where it is located.

The V++ INI file is accessed using the routines `WriteINIString` and `ReadINIString`. These routines are identical to their private counterparts, except they do not require a path parameter.

Should data be stored in the V++ INI file or a private INI file? Strictly speaking it doesn't matter, but good practice suggests that each major experimental setup should have its own private INI file. In order that modules do not need to be hard-coded with the location of the private INI file, it is a good idea to save the path to the private INI file in the V++ INI file. That way, a module first interrogates the V++ INI file to obtain the location of the private INI file, then reads / writes data from / to the private INI file.

10.0 Plotting

10.1 Creating a plot window

Example 10.1.1: Creating a named plot window

```
var

  P ;

begin

P := CreatePlot( 'My Plot' ) ;

end
```

Listing 10.1.1 Creating a named plot window.

Example 10.1.2: Creating a named plot window with specified position and size

```
var

  P ;

begin

P := CreatePlot( 'My Plot', 20, 20, 300, 200 ) ;

end
```

Listing 10.1.2 Initializing the size and location of a plot window.

10.2 Simple plots

The simplest form of plot is where a one-dimensional array of number needs to be displayed in graphical form. This is very easy to do with the Plot procedure.

Example 10.2.1: A simple plot

```
var
  P ;
  Data ;
begin
P := CreatePlot( 'Simple Plot' ) ;
Data := Sin( MakeLinear( -pi, +pi, 500 ) ) ;  // Create a sine wave
Plot( P, Data ) ;
end
```

Listing 10.2.1 A simple plot using the Plot procedure.

Example 10.2.2: Plotting the mean of each frame in sequence

```
var
  P ;
  Image ;
  Means ;
begin
GetActiveImage( Image ) ;
Means := MeanOfFrames( Image ) ;
P := CreatePlot( 'Sequence Means' ) ;
Plot( P, Means ) ;
end
```

Listing 10.2.2 Displaying useful information in a plot.

10.3 Adding titles and captions

A plot can be annotated with a range of labels as shown in Table 10.3.1 below.

Function	Action
`SetTitle`	Display a title at the top of a plot window
`SetXLabel`	Display a label below the x-axis of a plot window
`SetYLabel`	Display a label to the left of the y-axis of a plot window

Table 10.3.1 Annotating a plot with useful information

In addition to setting the text to be displayed, the title and axis labels can be altered by changing the font, font size and font color.

Example 10.3.1: Plotting the mean of each frame in sequence with useful annotation

```
var
  P ;
  Image ;
  Means ;
begin
GetActiveImage( Image ) ;
Means := MeanOfFrames( Image ) ;
P := CreatePlot( 'Sequence Means' ) ;
Plot( P, Means ) ;
SetTitle( P, 'Mean of Sequence Frames' ) ;
SetXLabel( P, 'Frame Number' ) ;
SetYLabel( P, 'Mean Value' ) ;
end
```

Listing 10.3.1 Annotating a plot window.

10.4 Plotting X and Y data

It is often required to plot one set of data versus another set. This is accomplished using the PlotXY procedure.

Example 10.4.1: A simple XY plot

```
var
  P ;
  XData, yData ;
begin
P := CreatePlot( 'Simple Plot' ) ;
xData := MakeLinear( -pi, +pi, 500 ) ;  // Create x data
yData := Sin( 10 * xData ) ;            // Create y data
PlotXY( P, xData, yData ) ;
end
```

Listing 10.4.1 A simple XY plot using the PlotXY procedure.

11.0 Controlling a PVCAM camera

11.1 Opening and closing a camera

Before a PVCAM camera can be used it must be opened. A camera may be opened in the GUI or from a module. If a camera is opened from within the GUI it does not need to be subsequently opened from a module. In other words, once it's open, it's open.

Sometimes it may be useful to close a camera, making it unavailable for operation. However, most times it unnecessary to explicitly close a camera because an open camera is automatically closed when V++ shuts down. (Furthermore, any cameras that are open just prior to shutdown will be opened automatically when V++ restarts.)

Example 11.1.1: Opening a camera

This example opens a camera with a known name.

```
begin

pvcOpenCamera( 'SenSys1' ) ;

end
```

Listing 11.1.1 Opening a camera.

Example 11.1.2: Closing a camera

This example closes a camera with a known name.

```
begin

pvcCloseCamera( 'SenSys1' ) ;

end
```

Listing 11.1.2 Closing a camera.

Example 11.1.3: Opening a camera without knowing its name

This example obtains a list of all cameras installed in the system and then prompts the user to open a specific camera.

```
var
  NameList ;
  AName ;
begin
NameList := pvcGetCameraList ;
SelectString( 'Choose a camera', NameList, AName ) ;
pvcOpenCamera( AName ) ;
end
```

Listing 11.1.3 Opening a camera based on the list of installed cameras.

11.2 Acquiring and displaying an image

Single-frame images are acquired using the `pvcCapture` function while sequences are acquired using the `pvcSequence` function.

Example 11.2.1: Acquiring a single frame

This example acquires a single frame from the camera and displays the result. It assumes that the camera is already open for operation.

```
var
  Image ;
begin
pvcSetExpTime( 200 ) ;  // Set the exposure time
Image := pvcCapture( 0, 0, 255, 255 ) ;  // Capture top-left 256-square image
Show( Image ) ;
end
```

Listing 11.2.1 Capturing a single frame.

Example 11.2.2: Acquiring the full CCD

This example acquires a single full frame from the camera and displays the result. It assumes that the camera is already open for operation.

```
var
  Image ;
  xSize, ySize ;
begin
pvcSetExpTime( 200 ) ;  // Set the exposure time
pvcGetCCDSize( xSize, ySize ) ;
Image := pvcCapture( 0, 0, xSize-1, ySize-1 ) ;  // Capture full CCD
Show( Image,'CCDImage' ) ;
end
```

Listing 11.2.2 Capturing a full CCD image.

Example 11.2.3: Acquiring a sequence of frame

This example acquires a series of frames from the camera and displays the result. It assumes that the camera is already open for operation.

```
var
  Image ;
begin
pvcSetExpTime( 200 ) ;  // Set the exposure time
Image := pvcSequence( 10, 0, 0, 255, 255 ) ;  // Capture 10 frames
Show( Image ) ;
end
```

Listing 11.2.3 Capturing a series of frames.

11.3 Binning the CCD

Binning improves the signal to noise ratio of an image by combining adjacent pixels on the CCD into a smaller number of super-pixels. Appending two optional parameters to the `pvcCapture` and `pvcSequence` functions controls the extent of binning.

Example 11.3.1: Acquiring the full CCD with binning

This example acquires a single full frame from the camera with two pixels binned in each direction. It assumes that the camera is already open for operation.

```
var
  Image ;
  xSize, ySize ;
begin
pvcSetExpTime( 200 ) ;  // Set the exposure time
pvcGetCCDSize( xSize, ySize ) ;
Image := pvcCapture( 0, 0, xSize-1, ySize-1, 2, 2 ) ;  // 2x2 binning
Show( Image,'BinnedImage' ) ;
end
```

Listing 11.3.1 Capturing a full CCD image with two-by-two binning.

Example 11.3.2: Acquiring a sequence of the CCD with binning

This example acquires a sequence of full frames from the camera with two pixels binned in each direction. It assumes that the camera is already open for operation.

```
var
  Image ;
  xSize, ySize ;
begin
pvcGetCCDSize( xSize, ySize ) ;
Image := pvcSequence( 10, 0, 0, xSize-1, ySize-1, 2, 2 ) ;  // 2x2 binning
Show( Image,'BinnedImage' ) ;
end
```

Listing 11.3.2 Capturing a sequence of the full CCD with two-by-two binning.

11.5 Detecting and handling camera errors

11.5.1 Detecting a camera error

Camera errors are most easily detected using the `pvcError` function. This function returns a Boolean value of TRUE if an error has occurred, otherwise FALSE.

Calling `pvcError` does not reset the error reporting machinery inside PVCAM, but another error-related function does. See `pvcErrorCode` below.

Example 11.5.1: Detecting a PVCAM error

This example attempts to open a camera and acquire an image. If any form of error occurs a message is displayed. Note that this example does not report the actual error condition.

```
var
  Image ;
begin
pvcOpenCamera( 'SenSys1' ) ;
if pvcError then
  Halt( 'Error occurred while opening camera!' ) ;
Image := pvcCapture( 0, 0, 499, 499 ) ;
Show( Image ) ;
end
```

Listing 11.5.1 Detecting a PVCAM error while opening a camera.

11.5.2 Determining the error code

All PVCAM errors have a specific error code associated with them. The error code can be determined by calling the `pvcErrorCode` function. This function returns an integer scalar value equal to the error code.

A call to `pvcErrorCode` resets the error report mechanism in PVCAM so that subsequent calls to `pvcErrorCode` do not report false errors. The examples below demonstrate how to correctly handle this behavior.

Example 11.5.2: Displaying a PVCAM error code

This example attempts to open a camera and acquire an image. If an error occurs the error code is displayed. Note that the error condition is first detected by `pvcError`.

```
var
  Image ;
begin
pvcOpenCamera( 'SenSys1' ) ;
if pvcError then
  Halt( 'Error opening camera. Code = ', pvcErrorCode ) ;
Image := pvcCapture( 0, 0, 499, 499 ) ;
Show( Image ) ;
end
```

Listing 11.5.2 Detecting and displaying a PVCAM error code while opening a camera.

Example 11.5.3: Displaying a PVCAM error code without first using pvcError

This example attempts to open a camera and acquire an image. If an error occurs the error code is displayed. Since the error status is reset immediately after a call to `pvcErrorCode`, the prospective error must first be saved in a variable. If the code is non-zero (indicating a problem) the error is displayed.

```
var
  Image ;
  Code ;
begin
pvcOpenCamera( 'SenSys1' ) ;
Code := pvcErrorCode ;
if Code <> 0 then
  Halt( 'Error opening camera. Code = ', Code ) ;
Image := pvcCapture( 0, 0, 499, 499 ) ;
Show( Image ) ;
end
```

Listing 11.5.3 Detecting and displaying a PVCAM error code while opening a camera, but without using pvcError.

11.5.3 Displaying a useful error message

Error codes are useful programmatically, but a camera user prefers to see a text message. The `pvcErrorMsg` function takes an integer parameter (equal to the error code) and returns an informative text message.

Example 11.5.4: Displaying a PVCAM error message

This example attempts to open a camera and acquire an image. If an error occurs an error message is displayed.

```
var

  Image ;

begin

pvcOpenCamera( 'SenSys1' ) ;

if pvcError then

  Halt( 'Error: ',pvcErrorMsg(  pvcErrorCode ) ) ;

Image := pvcCapture( 0, 0, 499, 499 ) ;

Show( Image ) ;

end
```

Listing 11.5.4 Detecting an error and displaying an error message while opening a camera.

12.0 Dynamic Data Exchange

Dynamic Data Exchange, or DDE, allows Windows programs to communicate with each other and share live data. If you need to integrate image processing functions or camera control into an experimental setup that involves other software products then you may need to use DDE.

In addition to transferring images and modules from one program to another, you can call VPascal procedures from a client program. For example, you can create buttons in a Microsoft Excel spreadsheet that trigger procedures in a VPascal module. Likewise, module variables can be shared so that the client can examine them at any time.

Alternatively, your VPascal module can act as a client to Excel and issue commands that cause operations to take place on the spreadsheet.

In a typical DDE exchange, one application requests data from another. The application servicing requests for data is called the *server*. The application issuing requests or controlling the server is called the *client*. V++ can function as both a server and a client. In the language of DDE a *conversation* is established between two applications and a *transaction* occurs between the client and the server. This terminology is summarized in Table 12.0.1.

Term	Explanation
Conversation	The DDE channel between two applications
Transaction	A specific request made during a conversation
Client	An application that requests and receives data
Server	An application that provides data to clients

Table 12.0.1 DDE terminology

Three names are used to uniquely define any specific piece of data that the server can provide, as shown in Table 12.0.2.

Name	Explanation
Service	The name used to refer to a particular server, like V++
Topic	The name of a broad type of data offered by the server
Item	The name of a specific piece of data under a topic

Table 12.0.2 Transaction terminology

For more information about DDE fundamentals, refer to the V++ on-line help.

12.1 Sending data to applications via DDE

When VPascal sends data to another application it is acting as the client and the other application is the server. The steps involved in controlling a DDE server from within VPascal are summarized in Table 12.1.1.

	Step	**Explanation**	**Functions**
1	Initiate the conversation	Make contact with a server to establish a conversation. The conversation refers to a particular service and topic, and is identified by a handle. The client may maintain several conversations simultaneously.	`DdeInitiate`
2	Transact with the server	After a conversation is established, the client may: send data to the server, request data from the server, or execute commands on the server.	`DdePoke` `DdeRequest` `DdeExecute`
3	Finish the conversation	Terminate a conversation using the appropriate conversation handle.	`DdeTerminate`

Table 12.1.1 Steps involved in controlling a DDE server

12.2 Exchanging data with Excel via DDE

Excel is a powerful adjunct to V++. This section shows a number of examples of how to control the Excel DDE server using a VPascal client.

Example 12.2.1: Initiating and terminating a conversation with Excel

This example opens a conversation with Excel on the topic of a specific worksheet. It assumes that Excel is already running.

```
var
  Ch ;
begin
Ch := DdeInitiate( 'Excel', 'Sheet1' ) ;  // Ch = channel number
{Perform useful client-server actions here}
DdeTerminate( Ch ) ;
end
```

Listing 12.2.1 The basics of all client modules: initiating and terminating a conversation.

Example 12.2.2: Auto-running Excel if necessary

This example attempts to open a conversation with Excel. If the client cannot connect to Excel, it attempts to execute the Excel program and reconnect the conversation. Note that you may need to spell out the full directory path to the Excel executable file (this is normally something like "C:\Program Files\Microsoft Office\Office\Excel.exe").

```
var
  Ch ;
begin
Ch := DdeInitiate( 'Excel', 'Sheet1' ) ;  // Ch = channel number
if Ch = 0 then
  begin
  Execute( 'Excel.exe' ) ;  // Include path if necessary
  Ch := DdeInitiate( 'Excel', 'Sheet1' ) ;
  if Ch = 0 then Halt( 'Cannot connect to Excel' ) ;
  end ;
{Perform useful client-server actions here}
DdeTerminate( Ch ) ;
end
```

Listing 12.2.2 Testing for a valid conversation handle and starting Excel if possible.

Example 12.2.3: Poking data into the current Excel cell

This example pokes a number into the current cell in Excel. The current cell is addressed as 'RC'.

```
var
  Ch ;
begin
Ch := DdeInitiate( 'Excel', 'Sheet1' ) ;
DdePoke( Ch, 'RC', 123.456 ) ;  // Could also use a variable instead of 123.456
DdeTerminate( Ch ) ;
end
```

Listing 12.2.3 Sending data to the current Excel cell by poking.

Example 12.2.4: Poking data into specific Excel cells

This example pokes numbers and text into specific cells in Excel using absolute addressing. In the example, 'R2C1' (row 2, column 1) refers to cell A2. Similarly, 'R2C2' refers to cell B2.

```
var
  Ch ;
  s, x ;
begin
Ch := DdeInitiate( 'Excel', 'Sheet1' ) ;
x := 123 ;
s := 'Text' ;
DdePoke( Ch, 'R2C1', x ) ;  // Put 123 into A2
DdePoke( Ch, 'R2C2', s ) ;  // Put 'Text' into B2
DdeTerminate( Ch ) ;
end
```

Listing 12.2.4 Sending data to specific Excel cells by poking to absolute addresses.

Example 12.2.5: Retrieving data from a specific Excel cell

This example requests the values of a cell in an Excel spreadsheet using absolute addressing. In the example, 'R2C1' (row 2, column 1) refers to cell A2. The `DdeRequest` function returns a string unless the optional `Format` parameter is specified.

```
var
  Ch ;
begin
Ch := DdeInitiate( 'Excel', 'Sheet1' ) ;
WriteInfo( 'Cell A2 contains: ',DdeRequest( Ch, 'R2C1' ) ) ;
DdeTerminate( Ch ) ;
end
```

Listing 12.2.5 Retrieving data from a specific Excel cell

Example 12.2.6: Executing commands in Excel

This example executes a macro in Excel to select a range of cells, and then modifies the appearance of those cells.

```
var
  Ch ;
  s, x ;
begin
Ch := DdeInitiate( 'Excel', 'Sheet1' ) ;
x := 123 ;
s := 'Text' ;
DdeExecute( Ch,'[select("R5C2:R5C2")]' ) ;     // Select cell B5
DdeExecute( Ch,'[format.font(,,,true)]' ) ;    // Italicise cell
DdeExecute( Ch,'[select("R5C2:R6C4")]' ) ;     // Select cell range B5:D6
DdeExecute( Ch,'[alignment(3,false,3,0)]' ) ;  // Center justify cells
DdeTerminate( Ch ) ;
end
```

Listing 12.2.6 Executing Excel macros using `DdeExecute`.

12.3 Controlling V++ with DDE

Other applications can control V++ via the V++ DDE server. For example, a Visual Basic for Applications program (as found in the Microsoft Office suite of products) can get access to VPascal scalars, strings and images, as well as execute procedures and modules. The DDE topics relevant to VPascal are shown in Table 12.3.1. Note: the DDE service name is "Vpp".

Topic	**Explanation**
Var	Access to VPascal variables exposed via a share name
Proc	Access to executable modules and procedures via a share name

Table 12.3.1 DDE server topics relevant to VPascal

12.3.1 Var Topic

The Var topic allows DDE clients access to a module's shared variables.

A module variable is shared using the reserved word **dde**.

Only global variables can be shared via DDE.

External applications or other VPascal modules can both link to shared variables (see 12.4).

Example 12.3.1: Sharing variables via DDE

This example demonstrates how to share global variables via DDE. The variable `A` is shared using the DDE share name `ActiveImage` and the variable `N` is shared via the DDE share name `ObjectCount`. Note that DDE share names must be unique across all modules in V++. Other applications and modules can obtain access to variables `A` and `N` using the topic Var, and Item names `ActiveImage` and `ObjectCount`, respectively.

```
var

  A dde 'ActiveImage' ;

  N dde 'ObjectCount' ;

begin

end
```

Listing 12.3.1 Sharing global variables via DDE.

An application may have a warm or hot link to the shared variables. In such cases, whenever the variable is changed (say, during the normal course of a module's execution) the change is reflected immediately in the client application.

Example 12.3.2: Accessing a variable from Visual Basic

This example demonstrates how a Visual Basic program could monitor the value of a shared variable.

```
var
  Image ;
  Mean dde 'Average' ;
begin
GetActiveImage( Image ) ;
Mean := MeanOf( Image ) ;  // Assignment will cause all links to update
end
```

Listing 12.3.2 Sharing the mean value of an image via DDE.

A connection to Visual Basic could proceed as follows. Add a text-box control (referred to as `Text1` in the example code below) to the Visual Basic form. Ensure that the appropriate VPascal module is running and create a hot link to the `Average` variable with the following code:

```
Text1.LinkMode = 0
Text1.LinkTopic = "Vpp|Var"
Text1.LinkItem = "Average"
Text1.LinkMode = 1
```

Visual Basic code for Listing 12.3.2

An up-to-date value for the `Average` variable will now be visible in the text-box control.

Example 12.3.3: Poking a value into a VPascal shared variable

This example shows how to poke a value into a module variable that has been shared via DDE.

```
var
  Image ;
  Name dde 'Title' ;
  Mean dde 'Average' ;
begin
GetImage( Name, Image ) ;  // Use the name poked into Title
Mean := MeanOf( Image ) ;  // Report mean value in Average
end
```

Listing 12.3.3 Poking values into a shared variable.

The Visual Basic code (taken from Excel) required to poke a new value into the share `Title` is shown below.

```
Channel = Application.DDEInitiate("Vpp", "Var")
Application.DDEPoke Channel, "Title", "A0001"
Application.DDETerminate Channel
```

Visual Basic code for Listing 12.3.3

12.3.2 Proc Topic

The Proc topic allows DDE clients access to a module's shared procedures.

A procedure is shared using the reserved word **`dde`**.

Only procedures without parameters can be shared via DDE.

External applications or other VPascal modules can both link to shared procedures (see 12.4).

Example 12.3.4: Sharing procedures via DDE

This example demonstrates how to share VPascal procedures via DDE. The procedure `Hello` is shared using the DDE share name `Hello.` Note that DDE share names must be unique across all modules in V++.

Clients can execute the procedure Hello using the topic Proc and the Item name Hello. A client may be an external application or another module running in V++ (see 12.4).

```
procedure Hello ; dde 'Hello' ;

begin

WriteInfo( 'Hello there!' ) ;

end;

begin

// the main part of the server module

end
```

Listing 12.3.4 Sharing a procedure via DDE.

By performing a DDE execute transaction an external application or any other VPascal module can trigger actions to be performed by the server module. Visual Basic code (taken from Excel) required execute the shared procedure `Hello` is shown below.

```
Channel = Application.DDEInitiate("Vpp", "Proc")

Application.DDEExecute Channel, "Hello"

Application.DDETerminate Channel
```

Visual Basic code for Listing 12.3.4

12.4 Communication between VPascal modules

When a VPascal module initiates a communication with the V++ server itself the conversation is called a *self-connection*. This proceeds exactly like any other DDE conversation except that the client and server both reside in V++.

This means that you can write one module to act as a server and others to act as clients. In fact, DDE is the preferred way for VPascal modules to communicate with each other. It enables you to put all the complicated code, say for controlling some lab equipment, into a single module and then write various simple modules that call the server, in this case to get access to the hardware.

The client and server modules follow exactly the same pattern as if they were intended for communicating with external programs. Using DDE to communicate with another module is no different to communicating with another application.

Further, a client module can access all the topics and commands available from the V++ DDE server. For example, one module can load, compile and run another module using DDE execute commands under the V++ System topic.

Example 12.4.1: DDE self-connection

This example shows how a module can self-connect to V++ and find out the full version number and the security key serial number. Note the use of a semi-colon in the WriteInfo parameters to indicate a line break.

```
var
  Ch ;
  Vn,Sn ;
begin
Ch := DdeInitiate( 'Vpp','System' ) ;
Vn := DdeRequest( Ch,'BuildNumber' ) ;
Sn := DdeRequest( Ch,'SerialNumber' ) ;
WriteInfo( 'V++ Version ',Vn,';Security key: ',Sn ) ;
DdeTerminate( Ch ) ;
end
```

Listing 12.4.1 DDE self-connection example

Example 12.4.2: Simple DDE server module

The following module shares one variable and one procedure. A client module can read and write the value of the shared variable `Count` and call the `ShowCount` shared procedure. An example client module is shown in Listing 12.4.3 below.

```
Program Server ;
var
  Count dde 'Count' ;
procedure ShowCount ; dde 'ShowCount' ;
begin
WriteInfo( 'The current count is: ',Count ) ;
end;
begin
Count := 123 ;  // Set initial value of Count
end
```

Listing 12.4.2 Simple DDE server module

Example 12.4.3: Simple DDE client module

The client module shown here creates a couple of V++ self-connections (one for variables and one for procedures) and illustrates request, poke and execute transactions. The server is assumed to be the example shown in Listing 12.4.2 above.

```
Program Client ;

var

  vChannel ;

  pChannel ;

  Data ;

begin

{ start two conversations }

vChannel := DdeInitiate( 'Vpp','Var' ) ;

pChannel := DdeInitiate( 'Vpp','Proc' ) ;

{ request the variable value }

Data := DdeRequest( vChannel,'Count',fmt_Number ) ;

WriteInfo( 'Received the number: ',Data ) ;

{ run the shared procedure }

DdePoke( vChannel,'Count',Data + 1 ) ;  // increment Count and send it back

DdeExecute( pChannel,'ShowCount' ) ;

{ end the conversations }

DdeTerminate( vChannel ) ;

DdeTerminate( pChannel ) ;

end
```

Listing 12.4.3 Simple DDE client module

12.5 Remote communication with Network DDE

DDE conversations most often take place between two applications on the same computer. Network DDE allows conversations to take place across a network, between applications that are running on different computers. Network DDE is independent of the actual network connections making up a Microsoft Windows network and can even work across wide area networks. To get a Network DDE conversation going follow the steps described below.

Step 1: Define a DDE share

Create a DDE share on the computer that will be running the DDE server. This is usually done with the DDE Share Manager program (DDESHARE.EXE) from the Windows Resource Kit but can also be done programmatically using the Windows NDDEAPI.DLL library.

To define a DDE share simply choose a share name and use the DDE Share manager to enter the service and topic names to which it should provide access. The DDE share name refers to one service / topic pair so to access multiple topics a share must be created for each. You can restrict conversations to particular items or require passwords for access to the server. The share name can be anything at all and does not have to be the same as the service or topic names. By convention, the share name normally ends with "$".

IMPORTANT NOTE: Although the terminology is similar, DDE network shares are not the same thing as VPascal shared variables and procedures.

Step 2: Run NETDDE on both machines

In the Windows directory you will find a program called NETDDE.EXE which loads the Network DDE layer of the operating system. This program must be run on both the server and client machines before trying to make a network connection. This step is required for Windows 95 and 98 only - it is not necessary for Windows NT.

You may find it useful to have NETDDE automatically loaded – this can be done by copying it to the Windows StartUp folder.

Step 3: Make a connection

To contact the server on a remote machine, the DDE client must use special names in place of the usual service and topic names. Those names are as follows:

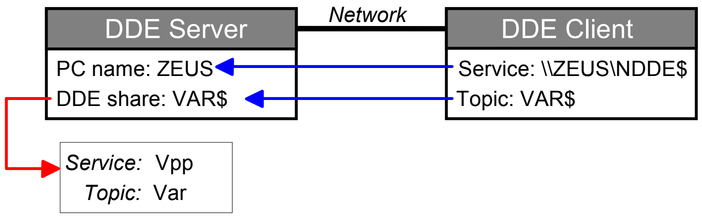
Service:	\\computer\NDDE$
Topic:	DDE share name

where *computer* should be replaced with the actual network name of the machine the DDE server is running on. NDDE$ is like a file share but refers to Network DDE handlers on the machine running the server. The DDE share name allows Windows to redirect the conversation to an actual server and topic on the remote machine.

Example 12.5.1: Using Network DDE

Suppose we have a machine called "ZEUS" running the DDE server. We now create a DDE share called VAR$ on that machine and relate it to the V++ Var topic by specifying the service name "Vpp" and the topic name "Var". When a client on another machine wants to connect, it uses the service name "\\ZEUS\NDDE$" and the topic name "VAR$".

On the machine running the server, the request is translated to the local service name "Vpp" and topic name "Var"



The server and client in this network DDE connection could both be copies of V++ running on different machines. The server doesn't have to do anything special because the Windows DDE sharing mechanism handles all the details. The client simply uses the special service and topic names, as illustrated below:

```
var
  NetCh ;
  Count ;
begin
NetCh := DdeInitiate( '\\ZEUS\NDDE$','VAR$' ) ;
if NetCh <> 0 then
  begin
  Count := DdeRequest( NetCh,'Count' ) ;
  WriteInfo( 'Remote variable = ', Count ) ;
  DdeTerminate( NetCh ) ;
  end;
end
```

Listing 12.5.1 Network DDE client module

13.0 Controlling laboratory equipment

13.1 RS-232 serial communications

Many devices communicate via a serial or RS-232 connection. Mostly, this means connecting an instrument or another computer to the serial port connectors found on the back of a PC. There can be up to 4 serial ports on a standard PC and some hardware allows up to 8 ports. The ports are named COM1:, COM2: and so on, and are therefore sometimes called com ports.

To transmit data to a remote device it is placed in an output buffer and sent through the serial port from there. Data received by the serial port is placed into an input buffer until it is processed.

VPascal modules can communicate with remote devices using the built-in serial communication functions. An outline of performing serial communications follows.

Step	**Action**	**Description**
1	Open a serial port	Before any communication can take place, a serial port must be opened by the module with the `OpenSerial` function. All subsequent serial functions in the module operate on the open port. There are a number of line parameters that must be specified at the time the port is opened, such as port number, baud rate, data bits, etc.
2a	Transmit	Once a serial port has been opened you can transmit VPascal variables or constants to the remote device using the Transmit function. If the remote device needs a special series of characters to tell it when the transmission is finished, this can be automatically appended to every transmission (use `SetTxEnd` to select a termination string).
2b	Receive	Receiving data is slightly more complicated than transmitting it because you need to decide when the transmission is finished and whether to wait for more characters. The `RxWaiting` function tells you how many unprocessed characters are waiting in the input buffer. You can read these one at a time with the `RxChar` function or wait for a given number of characters and then read them all with the `RxString` function. A convenient method for receiving input is to select a termination string which will indicate the end of a particular transmission - set this up with the `SetRxEnd` function. When a termination string has been selected, `RxString` will wait for a correctly terminated string before returning. If there are several terminated strings already in the buffer then these will be returned one at a time on each call to `RxString`. You can limit how long `RxString` should wait for a terminated

		string using the `SetRxTimeout` function.
3	Close the serial port	When your communication is completed you should release the port for use by other modules by calling the `CloseSerial` procedure. If the port remains open then no other module will be able to use it until the current "owner" terminates.

Table 13.1.1 Steps to successful serial communications.

Example 13.1.1: Opening and closing the serial port

This example shows how to correctly open the serial port and close it when complete. The `OpenSerial` procedure attempts to open port 2 at 19200 baud with 8 data bits, no parity and 1 stop bit.

```
begin
OpenSerial( 2, 19200, 8, NoParity, 1 ) ;  // Open serial port
if SerialError <> 0 then
  Halt( 'Failed!' ) ;                     // Check for errors while opening
{Transmit and/or receive data here}
CloseSerial ;                             // Close serial port
end
```

Listing 13.1.1 Opening and closing the serial port.

Example 13.1.2: Sending and receiving data through one serial port

This example shows how to send and receive data through one serial port. For an example of managing more than one serial port at a time see Example 13.1.3 later in this section.

```
begin
{Prepare for communication}
OpenSerial( 2, 19200, 8, NoParity, 1 ) ;
if SerialError <> 0 then Halt( 'Could not open COM2: port' ) ;

{Transmit}
TxFlush ;                 // Flush garbage in transmit buffer
Transmit( 'Hello!' ) ;    // Send information

{Receive}
RxFlush ;                 // Flush garbage in receive buffer
SetRxEnd( chr(13) ) ;     // Wait until a carriage-return is received
SetRxTimeout( 15000 ) ; // Timeout after 15 seconds

{Display results}
WriteInfo( 'Received: ',RxString ) ;

{Close serial port}
CloseSerial ;
end
```

Listing 13.1.2 Communicating over one serial port.

Example 13.1.3: Sending data through two serial ports

This example shows how to send data through two serial ports.

```
begin

{Open 2 serial ports}

OpenSerial( 3, 9600, 8, NoParity, 1 ) ;  // Open port 3 at 9600 baud

OpenSerial( 4, 19200, 8, NoParity, 1 ) ; // Open port 4 at 19200 baud


{Transmit test messages}

SelectPort( 3 ) ;

Transmit( 'Testing port #3' ) ;

SelectPort( 4 ) ;

Transmit( 'Testing port #4' ) ;


{Close ports}

CloseSerial( 4 ) ;

CloseSerial( 3 ) ;

end
```

Listing 13.1.3 Communicating over two serial ports.

13.2 Time lapse imaging

In time lapse imaging, we are required to capture multiple scheduled images, or a sequence with a specific time delay between successive frames. Doing this in VPascal is relatively simple and can be done in the background because modules may multitask with other operations.

Example 13.2.1: Principles of time lapse imaging

This example illustrates the essential features of a time lapse module. It contains an outer loop which performs a task on each iteration (in this case, updating the status bar with a timer display) and synchronization code which ensures that the outer loop only executes once in each time period. This module is designed so that any key press will short cut the current iteration.

IMPORTANT: It may appear that the module is monopolizing the CPU with an inefficient delay loop waiting for a specific time to elapse. However, VPascal is designed to multitask the execution of modules with everything else the computer is doing. Therefore, you will find that you can continue to use V++ and other applications even while the module is running and the timer display is updating.

```
const
  Period  = 2000 ;  // time delay in milliseconds

var
  t1 ;
  i ;

begin

  ClearKeys ;

  i := 0 ;

  repeat

    t1 := Clock ;

    WriteStatus( 'Time lapse: ',i * Period / 1000 :4,' seconds' ) ;

    i := i + 1 ;

    repeat until ( Clock >= t1 + Period ) or KeyPressed ;

  until ( ReadKey = vk_Escape ) ;

  WriteStatus ;

end
```

Listing 13.2.1 Principles of time lapse imaging

Note: This direct technique illustrates the principles of time lapse but for most applications it is much simpler to use the “Timers” timing generator library for VPascal. This is available for free download from the Digital Optics web site: http://www.digitaloptics.co.nz/

13.3 Advanced camera programming

13.3.1 Controlling multiple cameras

VPascal supports multiple PVCAM camera operation. Cameras may be operated consecutively (whereby data is acquired from one camera before switching to another) or concurrently (where more than one camera is acquiring data simultaneously).

Example 13.3.1: Switching between two cameras to acquire data

This example shows how to switch consecutively between two cameras in order to acquire data. In this case the standard routine `pvcCapture` can be used together with `pvcSelectCamera`.

```
var
  ImageA, ImageB ;
begin
{Open cameras, just in case}
pvcOpenCamera( 'SenSys1' ) ;
pvcOpenCamera( 'MicroMax1' ) ;

{Acquire images}
pvcSelectCamera( 'SenSys1' ) ;
pvcSetExpTime( 200 ) ;
ImageA := pvcCapture( 0, 0, 511, 511 ) ;

pvcSelectCamera( 'MicroMax1' ) ;
pvcSetExpTime( 1200 ) ;
ImageB := pvcCapture( 0, 0, 1023, 1023, 2, 2 ) ;

{Display images}
Show( ImageA, 'SenSysImage1' ) ;
Show( ImageB, 'MicroMaxImage1' ) ;
end
```

Listing 13.3.1 Switching between two cameras to acquire data.

Example 13.3.2: Acquiring data from two cameras simultaneously

This example shows how two cameras can acquire data simultaneously. The key is to start acquiring data in each camera, and then monitor the progress of each exposure.

```
const
  Cam1 = 'PXL1' ;
  Cam2 = 'SenSys3' ;
var
  ImageA, ImageB ;
  Cam1Done, Cam2Done ;
begin
{Start exposures}
pvcSelectCamera( Cam1 ) ;
pvcSetExpTime( 1000 ) ;  // One second exposure
pvcStartCapture( 0, 0, 500, 300 ) ;
pvcSelectCamera( Cam2 ) ;
pvcSetExpTime( 2000 ) ;  // Two second exposure
pvcStartCapture( 100, 50, 600, 200 ) ;

{Check status of each and read image when ready}
Cam1Done := false ;
Cam2Done := false ;
repeat
  {Check camera 1}
  if not Cam1Done then
    begin
    Cam1Done := pvcCheckStatus( Cam1 ) = pvc_ReadoutComplete ;
    if Cam1Done then
      begin
      pvcSelectCamera( Cam1 ) ;
      ImageA := pvcEndCapture ;
      end ;
    end ;
  {Check camera 2}
  if not Cam2Done then
    begin
    Cam2Done := pvcCheckStatus( Cam2 ) = pvc_ReadoutComplete ;
    if Cam2Done then
      begin
      pvcSelectCamera( Cam2 ) ;
      ImageB := pvcEndCapture ;
      end ;
    end ;
until Cam1Done and Cam2Done ;

{Show results}
Show( ImageA,Cam1+'Image' ) ;
Show( ImageB,Cam2+'Image' ) ;
end
```

Listing 13.3.2 Acquiring data from two cameras simultaneously.

The example shown in Listing 13.3.2 is just one possibility for capturing data simultaneously. In particular, note that the images are note displayed until both cameras have finished acquiring data. This may be undesirable when there is a large disparity in exposure times, or the exposures are triggered rather than timed.

In order to display the images as soon as they are ready, change the innermost conditional statements for each camera to that shown in Listing 13.3.3 below:

```
    ...
    if Cam1Done then
      begin
      pvcSelectCamera( Cam1 ) ;
      ImageA := pvcEndCapture ;
      Show( ImageA,Cam1+'Image' ) ;  // Move "Show" to here
      end ;
    end ;
    ...
```

Listing 13.3.3 Displaying data as soon as it is ready.

There may also be occasions where many cameras are exposing, in which case it may be desirable to move the `pvcEndCapture` functions outside the main repeat-loop. The modification is shown in Listing 13.3.4.

```
...
{Show results}
pvcSelectCamera( Cam1 ) ;
ImageA := pvcEndCapture ;
Show( ImageA,Cam1+'Image' ) ;


pvcSelectCamera( Cam2 ) ;
ImageB := pvcEndCapture ;
Show( ImageB,Cam2+'Image' ) ;
end
```

Listing 13.3.4 Delaying collection of data until all cameras have finished.

13.3.2 Aborting an exposure

The functions `pvcCapture` and `pvcSequence` cannot be aborted from within VPascal. To allow the user to prematurely terminate a camera exposure you must use the `pvcStartCapture` / `pvcEndCapture` pair.

Example 13.3.5: Manually aborting an exposure operation

This example starts an exposure and then continually checks the status of the exposure to see when it is complete. At the same time, the code is looking for a key press. If a key press is detected the exposure operation is terminated.

```
var
  QuitNow ;
  Image ;
begin
{Set exposure time and start exposure}
pvcSetExpTime( 10000 ) ;
pvcStartCapture( 100,100,800,700 ) ;

{Loop on status}
ClearKeys ;
repeat
  QuitNow := KeyPressed ;
until ( pvcCheckStatus = pvc_ReadoutComplete ) or QuitNow ;

{Show result if not aborted}
if QuitNow then
  pvcAbort
else
  begin
  Image := pvcEndCapture ;
  Show( Image ) ;
  end ;
end
```

Listing 13.3.5 Using a key press to manually terminate an exposure in progress.

Example 13.3.6: Aborting a triggered exposure after a timeout period

This example starts a triggered exposure and then continually checks the status of the exposure to see when it is complete. If the exposure is not complete within a specified time, the exposure operation is terminated. The exposure is also aborted if a key is pressed.

```
const
  Timeout = 10000 ;  // Time out interval is 10 seconds
var
  QuitNow ;
  Image ;
  StopTime ;
begin
{Determine the time after which the operation should abort}
StartClock ;
StopTime := Clock + Timeout ;

{Wait for trigger, then expose for 10ms}
pvcSetExpTime( 100 ) ;
pvcSetExpMode( pvc_ExpTriggerFirst ) ;
pvcStartCapture( 100,100,800,700 ) ;

{Loop on status}
ClearKeys ;
repeat
  QuitNow := KeyPressed or ( Clock > StopTime ) ;
until ( pvcCheckStatus = pvc_ReadoutComplete ) or QuitNow ;

{Show result if not aborted}
if QuitNow then
  pvcAbort
else
  begin
  Image := pvcEndCapture ;

  Show( Image,'myimage' ) ;

  end ;

end
```

Listing 13.3.6 Terminating an exposure after a timeout period, or if a key is pressed.

13.3.3 ICL scripts

PVCAM-compatible cameras support the low-level Imager Control Language (ICL) interface. ICL scripts can be run from VPascal using `pvcRunICL` and `pvcStartICL` / `pvcEndICL`.

Example 13.3.7: Running an ICL script using pvcRunICL

This example uses `pvcRunICL` to execute a simple image acquisition script in ICL. The ICL code is shown after Listing 13.3.6.

```
const

  ICLScript = 'C:\Documents\ICL\SnapCCD.icl' ;

var

  Image ;

begin

Image := pvcRunICL( ICLScript ) ;

Show( Image,'ICLImage' ) ;

end
```

Listing 13.3.7 Executing a simple ICL script.

```
Assume script is saved in file C:\Documents\ICL\SnapCCD.icl

script_begin();

shutter_open();  /* Take exposure */

expose( 200 );

shutter_close();

pixel_readout( 0, 512, 2, 512, 2 );  /* Collect data */

pixel_display( 256, 256 );

script_end( 1 );
```

ICL script for Listings 13.3.6 and 13.3.7

Example 13.3.8: Running an ICL using pvcStartICL and pvcEndICL

This example uses `pvcStartICL` / `pvcEndICL` to execute a simple image acquisition script in ICL. This form of execution makes it possible to abort the script while still executing in the camera. The ICL code is the same as the previous example. (Note that the syntax is identical to that for `pvcStartCapture` / `pvcEndCapture`.)

```
const
  ICLScript = 'C:\Documents\ICL\SnapCCD.icl' ;
var
  Image, QuitNow ;
begin
{Compile and start ICL script}
pvcStartICL( ICLScript ) ;
if pvcError then Halt( pvcErrorMsg( pvcErrorCode ) ) ;

{Loop on status}
ClearKeys ;
repeat
  QuitNow := KeyPressed ;
until ( pvcCheckStatus = pvc_ReadoutComplete ) or QuitNow ;

{Show result if not aborted}
if QuitNow then
  pvcAbort
else
  begin
  Image := pvcEndICL ;
  Show( Image,'ICLImage' ) ;
  end ;
end
```

Listing 13.3.8 Executing an ICL script using pvcStartICL / pvcEndICL.

13.4 Controlling a video frame grabber

If you have a video frame grabber or some other video device installed, you can control it from a VPascal module. The structure of the VPascal video control routines is broadly similar to that used for controlling PVCAM cameras – for example, routine names are usually the same. However, video frame grabbers and PVCAM cameras are controlled by separate sets of routines. The video routines all start with the prefix `vid`.

13.4.1 Video device architecture

V++ can control multiple video devices simultaneously, even if the devices are controlled by different drivers or are from different manufacturers. In VPascal, each device has a name and an index number that you can use to refer to it. Generally, you will use the index number to select a specific device but you can convert a device name to an index using the `vidIndexOf` function.

If there is only one video device installed, it is always selected by default but if there are several you must use `vidSelectDevice` to indicate which one to use.

Example 13.4.1: Looking for video devices

The following simple example displays a list of installed video devices

```
var
  i ;

begin

for i := 0 to vidGetDeviceCount - 1 do
  writeln( i:2,': ',vidGetDeviceName( i ) ) ;

end
```

Listing 13.4.1 List installed video devices

13.4.2 Acquiring and displaying a video image

Single-frame images are acquired using the `vidCapture` function while sequences are acquired using the `vidSequence` function.

Example 13.4.2: Acquiring a single frame

The following example acquires a single frame from the video frame grabber and displays the result. It assumes that the frame grabber is already selected.

```
var
  Image ;
begin
Image := vidCapture ;
Show( Image ) ;
end
```

Listing 13.4.2 Capturing a single video frame

Example 13.4.3: Acquiring a sequence of video frames

This example acquires a series of frames from the frame grabber and displays the result. It assumes that the frame grabber is already selected.

```
var
  Image ;
begin
Image := vidSequence( 10 ) ;  // Capture 10 frames as fast as possible
Show( Image ) ;
end
```

Listing 13.4.3 Capturing a series of video frames.

13.4.3 Using a region of interest (ROI)

If you need to capture a small part of the video frame you must first set up a region of interest, or ROI, for the device. This indicates that only the pixels in the ROI should be returned by the `vidCapture` or `vidSequence` routines.

Example 13.4.4: Capture a region of interest from a video camera

```
var
  x1,y1 ;
  xs,ys ;
  Image ;
begin
{ determine coordinates of central region }
xs := vidGetXSize / 2 ;
ys := vidGetYSize / 2 ;
x1 := xs / 2 ;
y1 := ys / 2 ;
{ set up ROI }
vidSetROI( x1,y1,x1+xs-1,y1+ys-1 ) ;
{ capture }
vidUseROI( true ) ;
Image := vidCapture ;
Show( Image ) ;
end
```

Listing 13.4.4 Capture a region of interest from a video camera

13.4.4 Asynchronous video capture

The `vidStartCapture` procedure is similar to `vidCapture` except that it returns immediately instead of waiting for the acquisition to finish. This allows you to perform other operations while the frame grabber captures the data. When the captured image is ready (check this using the `vidCheckStatus` function) you use `vidEndCapture` to read it out.

Example 13.4.5: Asynchronous control of multiple frame grabbers

As well as providing for pipeline processing (ie. processing frame n while frame n+1 is being acquired) the asynchronous routines allow you to perform simultaneous acquisitions on multiple frame grabbers, as illustrated in the following example.

```
var
  Image0,Image1 ;
  Ready0,Ready1 ;
  Quit ;

begin
{ Start acquiring on device 0 }
vidSelectDevice( 0 ) ;
vidStartCapture ;

{ Start acquiring on device 1 }
vidSelectDevice( 1 ) ;
vidStartCapture ;

Quit := false ;
repeat
  { Check status of devices }
  Ready0 := ( vidCheckStatus( 0 ) = vid_Ready ) ;
  Ready1 := ( vidCheckStatus( 1 ) = vid_Ready ) ;

  { Do other things }
  Quit := KeyPressed ;

until ( Ready0 and Ready1 ) or Quit ;

{ Readout device 0 }
vidSelectDevice( 0 ) ;
Image0 := vidEndCapture ;

{ Readout device 1 }
vidSelectDevice( 1 ) ;
Image1 := vidEndCapture ;

{ Display images }
Show( Image0 ) ;
Show( Image1 ) ;

end
```

Listing 13.4.5 Asynchronous control of multiple frame grabbers

13.5 Controlling a TWAIN scanner or camera

If any TWAIN devices, such as flat-bed scanners or digital cameras, are installed the `TwainAcquire` function opens the interface to the selected device so that an image can be acquired. If the acquisition is cancelled the function returns a null variable. If there is more than one TWAIN device, you can display the Select Source dialog box using `TwainSelectSource`.

You can access TWAIN devices in parallel with PVCAM cameras and video frame grabbers.

Example 13.5.1: Acquiring an image from a TWAIN source

This example shows how a module can ask the user to select a TWAIN device (eg. a scanner or a digital camera) and then open the native image acquisition interface for that device. If an image is captured, it is moved to the `Image` variable.

```
var
  Image ;

begin

TwainSelectSource ;

Image := TwainAcquire ;

if IsImage( Image ) then

  Show( Image )

else

  WriteInfo( 'No image was acquired' ) ;

end
```

Listing 13.5.1 Acquiring an image from a TWAIN device

14.0 Linking to external libraries

14.1 Calling DLL code

Using DLLs is a powerful way to extend the capabilities of V++. You can directly call functions that you write yourself or functions contained in 3rd party libraries. You can even call functions in the Windows API libraries.

14.1.1 Creating the DLL

Use any language to write an external library providing it will generate a standard 32-bit Windows DLL. The functions to be called from VPascal must be compiled in exportable form with standard calling conventions (refer to your compiler documentation to verify that it supports the Windows standard call model). Functions can exported either by name or by an explicit index.

To access image data passed to your function by VPascal you also need to include the Image Descriptor Block (IDB) type definitions in your project.

14.1.2 Writing the module

Declare the DLL functions to use in the VPascal module. Unlike functions defined inside the module itself, external functions must be declared with explicit parameter types.

For example:

```
procedure MyProc( a,b:integer; Image:pointer ) ; external 'MyCode' ;
```

The `external` keyword is used to indicate that the procedure or function is located in a DLL and must be followed by the name of the DLL concerned. The DLL name does not have to be fully qualified and in this example the extension defaults to `'.DLL'` and the library name is therefore `MYCODE.DLL`.

There are seven data types recognized for declaring parameters to external routines: byte, short integer, word, long integer, single, double, and pointer (note that integer is the same as long integer). Image parameters must always be declared as type pointer.

The VPascal external routine declaration above would correspond to one of the following declarations in a DLL written in C:

```
void MyProc( int a, int b, void *Image ) ;

void MyProc( int a, int b, TIDB *Image ) ;
```

In Pascal, the DLL's procedure declaration would be very similar to the first line of the VPascal declaration shown above.

Note: If you declare a parameter with the **var** keyword then a pointer to the variable is passed to the external routine.

14.1.3 Calling an external routine

External routines are called exactly like functions and procedures defined inside the module itself. You can pass virtually all of your VPascal variables to an external function, including both scalars and images. Parameters are usually passed by value, except images that are always passed by reference. You can pass the address of a variable using the `Address` function.

14.1.4 Using images

Images are stored as contiguous packed arrays of pixel values organized in row-wise order. Information about the dimensions of the image, its type, and the memory location of the pixel data is stored separately in an Image Descriptor Block, or IDB.

The VPascal declaration of an external routine always shows an image parameter as a pointer, however there are two ways that the image parameter may actually be passed to the external routine:

1. As a pointer to the Image Descriptor Block
2. As a pointer to the raw image data

The default is the first case. If you call the external routine as follows then a pointer to the IDB will be passed:

```
MyProc( a, b, Image ) ;
```

To pass a pointer to the raw data *only* then call the external routine as follows:

```
MyProc( a, b, Address( Image ) ) ;
```

14.2 Direct access to image memory

14.2.1 Getting a pointer to image memory

To get a pointer to the memory used to store image data use the `Address` function. `Address` is a function that returns a pointer (as a 32-bit integer) to the data area of a variable.

For example, say you have a function that takes a pointer to a block of memory containing signed integers and adds a constant to every integer. In C the external function may be declared like this:

```
void AddConst( int *Mem[], int nPts, int Num ) ;

// Mem = pointer to integers in memory

// nPts = number of integers in memory

// Num = constant to add to integers
```

In VPascal this function would be declared like this:

```
procedure AddConst( Mem:pointer ; nPts,Num:integer ) ; external 'SomeDLL.DLL' ;
```

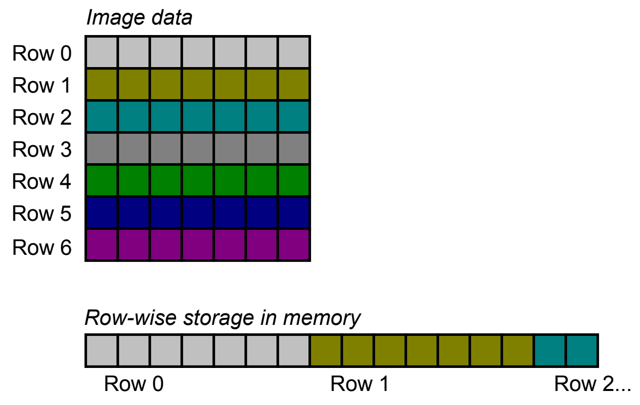
A VPascal code fragment illustrating how this routine might be called is:

```
...

{Get current image and check it is integer}

GetActiveImage( Img ) ;

if TypeOf( Img ) <> integer then

  Halt( 'Image must be integer!' ) ;


{Call external routine}

nPixels := GetXSize( Img ) * GetYSize( Img ) ;

AddConst( Address( Img ), nPixels, 123 ) ;

...
```

14.2.2 Memory Layout

An image is simply a large 2D or 3D array of pixels, each of which represents an intensity value. The data is stored in memory as a single packed contiguous block. Each image is stored row-wise: that is the x-coordinate changes most rapidly, followed by the y-coordinate. Sequences are stored as consecutive images, with the z-coordinate changing least rapidly.

Image memory blocks contain only the image pixels themselves. There are no headers and there is no padding.



14.2.3 Accessing the Image Descriptor Block

Every image in VPascal contains a block of information describing the content and layout of the image. This information is made available when the image variable, say `Img`, is used as a parameter rather than `Address( Img )`. For this to work, the parameter to the external code must be declared as pointer. For example, an external procedure is declared as

```
procedure MyProc( x:pointer ) ; external 'SomeDLL.DLL' ;
```

The external procedure call

```
MyProc( Address( Img ) ) ;
```

passes the address of the image memory to `MyProc` whereas

```
MyProc( Img ) ;
```

passes the address of the image descriptor block to `MyProc`.

14.2.4 Image Descriptor Block (IDB) Layout

The memory layout of the image descriptor block is shown below using a Pascal record and a C structure. When using an image descriptor do not write past the end of the structure.

Pascal Record

```
type
  TType = ( typ_Null, typ_Binary, typ_Byte, typ_Shortint, typ_Word,
            typ_Longint,typ_Single, typ_Double, typ_Complex, typ_DblComplex,
            typ_RGB, typ_RGB48, typ_RGBFloat ) ;


  PIDB = ^TIDB ;
  TIDB = record
           aType : TType ;     // image data type (4 byte enumerated scalar)
           xSize : integer ;   // image width (32-bit signed integer)
           ySize : integer ;   // image height (32-bit signed integer)
           zSize : integer ;   // image depth (32-bit signed integer)
           tSize : integer ;   // reserved for future use (4 bytes)
           Data  : pointer ;   // pointer to start of image memory
         end ;
```

C Structure

```
typedef enum {
  typ_Null,typ_Binary, typ_Byte, typ_Shortint, typ_Word,
  typ_Longint, typ_Single, typ_Double, typ_Complex, typ_DblComplex,
  typ_RGB, typ_RGB48, typ_RGBFloat,
} TType ;


typedef struct {
  TType  aType ;      // image data type (4 byte enumerated scalar)
  int    xSize ;      // image width (32-bit signed integer)
  int    ySize ;      // image height (32-bit signed integer)
  int    zSize ;      // image depth (32-bit signed integer)
  int    tSize ;      // reserved for future use (4 bytes)
  void   *Data ;      // pointer to start of image memory
} TIDB, *PIDB ;
```

14.3 Implementing Custom Dialog Boxes

To add a custom dialog box to V++ you must write a DLL that implements the dialog box and provides an access function that you can call from VPascal to display it and return results. Using the same technique you can make Windows common dialogs available as well (there is an example of this on the Digital Optics web site).

14.4 Calling the Windows API

Using VPascal, you almost never need to worry about Windows. But occasionally, a module may have specialized requirements that require it to call Windows API functions directly.

The Windows API interface is implemented as a set of DLLs (like KERNEL32.DLL, USER32.DLL etc) that export the functions programmers need to communicate with the operating system. If we know the name and parameter list for one of these functions we can link to it in VPascal and call it when the module is running.

For example, consider the Windows `SendMessage` function which is used to send Windows messages to applications or to individual windows. `SendMessage` is exported by the USER32.DLL library and can be declared in VPascal as follows:

```
function SendMessage( Handle,Message,wParam,lParam:integer ) : integer ;
external 'User32' ; name 'SendMessageA' ;
```

`SendMessage` is declared above as a function but it's also possible to declare it as a procedure if you don't need the result. Note also that we used the `name` keyword to specify the actual name used in USER32.DLL to export the function.

Having declared it, you can now call `SendMessage` just as you would call any other VPascal function.

Example 14.4.1: Closing V++ from a module

As a practical example, consider how to shut down V++ from a module (there's no built-in function to do this). You can shut down by using `SendMessage` to send the main V++ window a Windows `wm_Close` message.

In order to send the message, you first need to declare another API function, `FindWindow`, which you use to determine the handle of the main V++ window.

```
function FindWindow( ClassName,WindowName:pointer ) : integer ;
external 'User32' ; name 'FindWindowA' ;

procedure SendMessage( Handle,Message,wParam,lParam:integer ) ;
external 'User32' ; name 'SendMessageA' ;

const
  wm_Close = 16 ;

begin

SendMessage( FindWindow( 'TFrameForm','V++' ),wm_Close,0,0 ) ;

end
```

Listing 14.4.1 Closing V++ from a module

15.0 Advanced Development

15.1 VPascal Architectures

An advanced VPascal programmer can implement a great deal of sophisticated custom functionality within the V++ environment. This might include an extended user interface (toolbars, menus and custom dialog boxes), new imaging routines, custom hardware interfacing, networking, interaction with external applications and more.

There are several broad ways an advanced VPascal programming project can be structured and to describe the options we will borrow some terminology from the networking world.

The key to a successful project is breaking down the problem in an appropriate way and then implementing the component VPascal modules and any external code that you may need.

This chapter assumes that you are already familiar with general VPascal programming issues, DDE communications and linking to external libraries. If not, then please read the appropriate earlier chapters before returning to this chapter.

15.1.1 Front-end shell model

The front-end shell model is the simplest structure from the VPascal point of view. It involves implementing all custom functionality in an external DLL and writing a VPascal module to link it into V++ by providing a toolbar and/or new menu commands.

Although very simple at the VPascal end, this model requires you to be familiar with conventional Windows programming and to be capable of producing a DLL.

The main disadvantage of this model is that you don't make good use of VPascal's powerful image handling capabilities.

15.1.2 Client-server model

Client-server is a very useful architecture for many advanced VPascal projects and is particularly well-suited to hardware integration. In this model, all core functionality is built into a single module which makes a selection of procedures and variables available using DDE sharing. Additional modules (the clients) can then be written as required and can use the functions and variables shared by the server.

For example, to integrate a motorized filter wheel into V++ you would start by writing a server module that includes all of the control functions for the wheel and shares these via DDE. The server might also implement a toolbar and some menu commands to allow the user to manually control the wheel. Client modules can then be written which perform automated filter wheel control sequences by calling the server's shared routines. An example of a VPascal filter wheel controller with a client-server architecture can be downloaded from the Digital Optics web site.

Note that although shared procedures cannot accept parameters, in practice you can pass parameters by setting the values of server shared variables before calling a procedure.

One advantages of the client-server model is the fact that client modules can be small because they do not have to re-implement the functionality that resides in the server. Also, using DDE as the communication mechanism means that the server can be triggered by external applications as well as by other modules. This can even take place across a network.

15.1.3 Peer-to-peer model

This model is really a variation of the client-server model. The difference is that all participating modules may be servers or clients or both. This approach is suited to projects that can be broken down into roughly equal sized units each of which can be implemented as an individual module.

Inter-module communication is still accomplished using DDE and the peer-to-peer model therefore retains all of the advantages of the client-server model.

An additional advantage of peer-to-peer is that you can reconfigure your system simply by selecting which modules will be compiled. This includes the user interface if you have distributed the implementation of toolbars and menus commands among the various modules.

15.1.4 Server-only model

The server-only architecture basically involves implementing only the server side of a client-server system and providing all triggering and user interface code in an external program. This gives you total freedom in the implementation of your project but this model is probably the most difficult to implement well.

Again, the communication mechanism is DDE which provides the external program with access to shared procedures and variables in the VPascal server module. Using DDE, you can also get access to other V++ objects, including desktop images and module editors (regardless of whether they are linked to a VPascal variable).

15.1.5 Hybrid models

The models presented here are examples of how to describe the architectures you use in advanced VPascal projects. However, in reality you can mix the characteristics of several models to suit your own requirements. For example, any of the architectures that use DDE can double as a server-only architecture. Also, you may choose to implement parts of any project in an external DLL library – that approach is not limited to the front-end shell model.

Ultimately, the best architecture is the one that solves your own specific set of problems.