2.0 Language Report

This section defines the Ubercode computer language. It describes the language semantics (meaning), and the Appendix on Syntax Diagrams defines the syntax. The following language elements are defined:

(2.1) Lexical Elements
(2.2) Program Structure
(2.3) Functions
(2.4) The Type Model
(2.5) Scalar Types
(2.6) Structured Types
(2.7) Program Statements
(2.8) Expressions and Operators
(2.9) Error Handling
(2.10) File Input and Output
(2.11) Windows and Menus
(2.12) Printing

2.1 Lexical Elements

These are the words, symbols and numbers that make up a program. The following lexical elements are used, which are explained in more detail below:

- keywords
- symbols
- identifiers
- separators
- numeric literals
- string literals
- comments

The language is not case sensitive, so keywords, symbols, and identifiers can be written in lower or upper case. The end of line is a separator so a single lexical element may not be split across lines. Statements, which consist of several lexical elements, may be split across several lines if needed.

Keywords

These are special reserved words used for parts of a program. They are usually connected with declarations (such as **const**, **type**, **var**) or with control flow (such as **if**, **else**, **end if**). They are printed in bold case in the manual, and there is a list in the Reserved Words topic in the Appendix.

Symbols

These are non alphanumeric characters (such as **>=**, **+**) or short words (**and**, **not**) which are usually operators. There is a complete list in the Reserved Words topic in the Appendix.

Identifiers

These are the names used for all constants, types, variables, classes and functions in a program. The maximum identifier length is limited to 20 characters. The first character must be a letter, and the remaining characters can be letters, numbers or the underscore. For example:

```
i123
COUNT10
InitWindowString
very_long_identifier
ZetaMax
```

Some identifiers such as *Sqr*, *Ord* and *Val* are part of the run time library. These are listed in the Command Reference section. A class name has similar restrictions to an identifier, but is not allowed to contain the underscore character. Classes are always stored in separate files and the class name must be the same as its file name. For example the **system** class is stored in a file called "system.cls". This helps the compiler find the class.

Separators

All the lexical elements have separators between them so the compiler can tell them apart. The separating characters are the space (Chr(32)), tab (Chr(9)), and end of line characters (Chr(13)), (Chr(10)). Separators are also called white space because this is how they appear when printed.

Literals

Literals are constant values which are written as an actual value without using an identifier. Numeric literals are used for integer, fixed point and real type, and string literals are used for string types.

Numeric literals

These are numbers which can be of integer, fixed point or real type. An integer literal is a digit sequence with an optional plus or minus sign in front:

```
0
100
-3000
+600000000
```

A fixed point literal is a literal of fixed type, which is written as a digit sequence with a fractional part and an optional sign:

```
-1.0
+2005.235
0.0
```

A real literal is a digit sequence with an optional fractional part, followed by an exponent. The exponent follows the letter *e*:

```
1e3
6.023e+23
-1.6E-32
+1.6E+32
```

String literals

A string literal is a sequence of characters from the ASCII character set. The characters must be printable characters which include any ASCII character between chr(32) and chr(126), the space to the tilde. Characters from outside this range produce different effects on different computers and are not portable. Strings are delimited by a double quote:

```
"Hello World"
"They said ""Hello"""
"A"
" "
""""
```

A quote in a string is represented by two adjacent quotes. Thus the last three string literals all have a length of one. They contain the letter A (chr(65)), a space (chr(32)) and a double quote character (chr(34)) respectively.

String literals cannot extend across multiple lines of a program because the end of the line is a separator. Strings that span several lines of program text can be produced by joining up string literals with the string concatenation operator (the **+** sign):

```
"The quick brown fox " +
"jumps over the lazy dog."
```

The predefined string NL means new line and represents the end of a line in a string. A string containing NL is a multi line string and is useful for defining sentences of text. For example the following multi line string contains nine lines:

```
"I know you all, and will a while uphold"     + NL +
"The unyoked humour of your idleness."        + NL +
"Yet herein will I imitate the sun,"          + NL +
"Who doth permit the base contagious clouds"  + NL +
"To smother up his beauty from the world,"    + NL +
"That when he please again to be himself,"    + NL +
"Being wanted he may be more wondered at"     + NL +
"By breaking through the foul and ugly mists" + NL +
"Of vapours that did seem to strangle him."   + NL
```

Comments

Comments are parts of the program that are ignored by the compiler and are of interest to human readers. They are characters following the double slash ***//*** character pair and extending to the end of the line. For example:

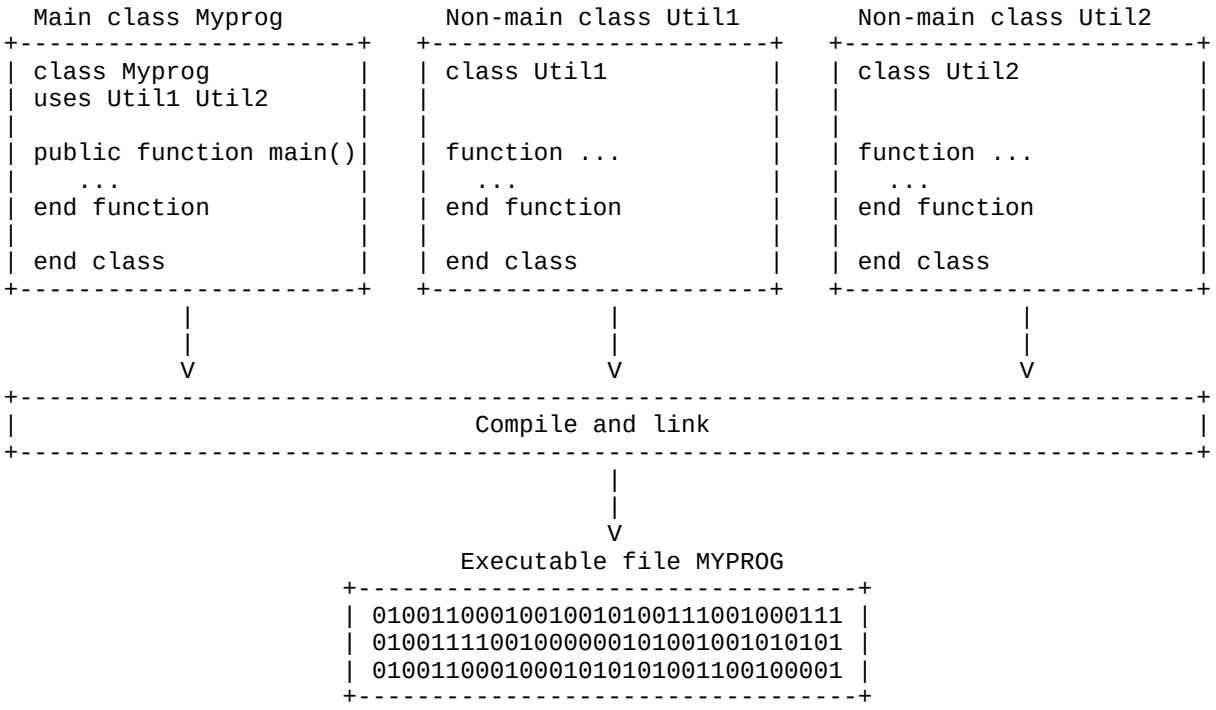
```
//////////////////////////
// this is a comment
// so is this  // and this
//////////////////////////
```

Comment characters do not apply in a string. For example:

```
"// This is not a comment"  // but this is.
```

2.2 Program Structure

A program consists of one or more Ubercode source files. Each file contains a single class and is compiled separately. The program must contain one main class and may optionally contain any number of non-main classes. The following diagram shows a program consisting of three source files:

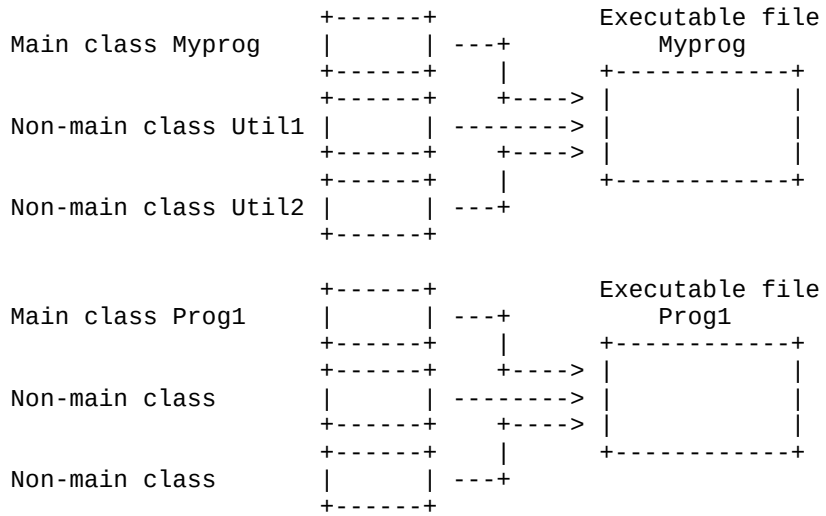


This diagram shown the main class Myprog being combined with non-main classes Util1 and Util2 to make the binary executable file Myprog. When Myprog is run by the operating system, the program starts at function main in the main class.

Multiple Executable files

A large application may consist of many classes, and be too large to be compiled into a single executable file. In such a case the application must be split up into multiple executable files, by splitting the classes up into separate groups.

Executable files can call up other executable files with the Run statement and pass them values as command line arguments. This is a remote procedure call which is similar to a normal function call, except the called class is in a separate executable file. The next diagram shows two separate programs, Myprog and Prog1. The executable file Myprog can call up Prog1.



Class Structure

Ubercode programs are built from one or more classes, and each class is a file which is compiled independently. Classes have a heading, a uses clause, constants, types and functions (which may be public or private), and an ending. The structure of a class is shown next:

```
// Class heading
Ubercode 1 class ClassName

// Uses clause
uses OtherClassName ...

 // Types, constants, functions (public)
public type TypeName ...
public const ConstName:TypeName ...
public function FunctionName(Parameters) ...

 // Types, constants, functions (private)
private type TypeName ...
private const ConstName:TypeName ...
private function FunctionName(Parameters) ...

// End of class
end class
```

Ubercode 1 class. This specifies the class name and the version number. Class names always match the file name, for example the class "myclass" must be stored in the file "myclass.cls".

Uses clause. This is a list of other classes on which this class depends. The effect of using another class is to make its types, constants and functions available to this class. This class is known as a client class of the other classes.

Public type, const, function. A class may declare public types, constants and functions for use by other classes. These are also known as *exported types*, *exported constants* and *exported functions*. When classes do this they define Abstract Data Types (ADTs) with the following parts [Martin 1986]:

- The type. This is represented by the public types.
- Operators. These are represented by the public functions.
- Axioms. These define the ADT in terms of how the functions and types interact. The axioms are the preconditions and postconditions used with the public functions.

Public types, constants and functions can be fully declared using the **public** keyword. Alternately they can be declared using a prototype followed by a full declaration later on in the same class. The prototypes allow public elements to be grouped near the top of the class, which makes it easier to see what the class exports.

Private type, const, function. These are types, constants and functions declared locally by the class. These private elements cannot be used by any other classes. They must be full declarations, therefore types must fully specify the data type and any components, constants must specify their type and value, and functions must contain a complete function body.

If types, constants and functions use neither the **private** or **public** keyword, they default to private scope.

Global Variables are not allowed, because they make software harder to understand (see [Levy 1982, Er 1985] in the Bibliography). Some writers [Feldman 1986] consider languages that allow them to be poorly designed.

End class. This keyword marks the end of the class. There should not be any program code following this point, although comments are allowed. Although it would be possible for the compiler to stop compiling when it reaches the end of the source file instead of explicitly requiring the **end class** keyword, use of **end class** is safer because it guards against the case where part of the class is missing. When copying files, it is possible for the last part to be missing if the copy was unsuccessful.

Main class, Non-main class. A program consists of a main class, and any number of non-main classes. Program execution always starts at function main in the main class. The difference between a non main class and a main class is the latter declares a public function main().

Both types of class may declare any number of private and/or public types, constants and functions. A main class must always declare *main* as a public function. A main class may not be inherited by other classes, to avoid loops in the uses clause. (If a non-main class could inherit a main class, eventually the non-main class would be inherited by the main class and the uses clause would form a loop). Allowing public identifiers in a main class is a textual convenience which makes it easier to convert a non-main class into a main class. In effect only function main() is inherited by the operating system.

Classes cannot be nested because the compiler processes one class at a time. The correct way to use types, constants and functions from another class is to name it in the **uses** clause. A class is also known as a compiland because it is compiled independently. *Compiland* simply means "a file which can be compiled as a single element".

Uses clause

The **uses** clause is an optional part of a class that appears immediately after the class heading. It is written as **Uses** *name1* ... where *name1* ... specifies one or more classes inherited by this class. This class is a *client* of the other classes. All public types, constants and functions from the inherited class are available in this class. This allows inheritance as required by object oriented programming, and allows inheritance of utility functions from code libraries.

The compiler adds the public symbols of each used class to the symbol table of the class being compiled. In line with [Booch 1991 p50], inheritance relationships are defined at the start of the class, so they are available to all declarations within the class being compiled.

Declarations

A declaration is where a class, function, type, constant, variable, input, inout or output parameter is given a name. After being declared, they can be referred to elsewhere by their name. For example, a function is declared with its parameters, calculations and name. It is called up from elsewhere by use of its name.

The *scope* of a declaration is the part of the program where the name can be used. This is also called *visibility* because a name can only be understood if its declaration is visible. If a name is used outside the scope of its declaration, or if its declaration is not visible which is the same thing, the program cannot be compiled. The name is an undeclared identifier which causes compile time error ERR_UNDECLARED_ID to occur.

Scope

All classes have three levels of scope, *public* scope, *private* scope and *function* scope:

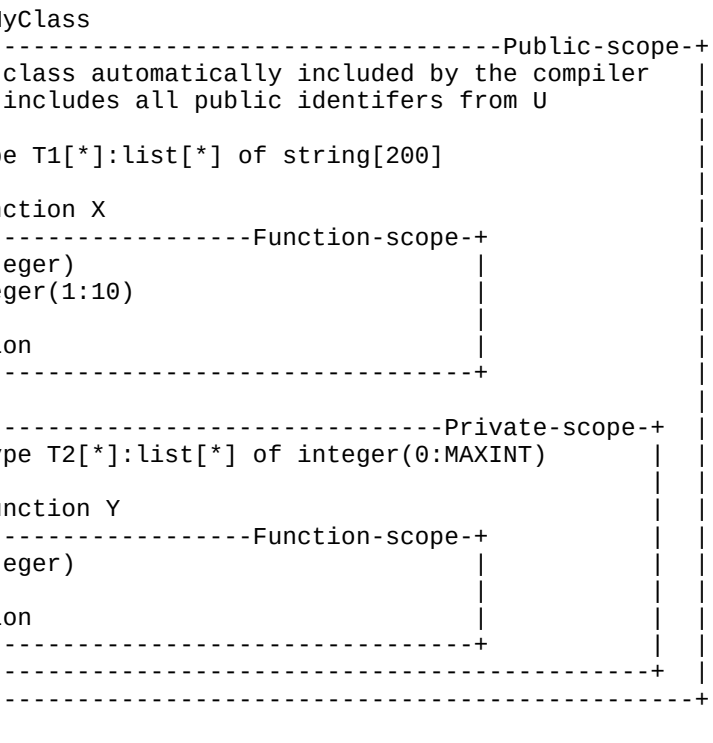
Public scope. All identifiers declared as **public**, and all identifiers imported from used classes have public scope. Identifiers in the run time library also have public scope, because they are in the system class which is automatically inherited. Identifiers at public scope are visible through the entire class.

Private scope. All identifiers declared as **private** have private scope and can only be used in the class they are declared in. Identifiers declared without a scope keyword default to private scope.

Function scope. All identifiers declared in a function, such as constants, types, variables, input, inout and output parameters have function scope. These are also known as local variables and local constants. The actual function name is declared at public or private scope, depending on whether the **public** or **private** keyword was used at the start of the function declaration.

The following diagram shows the different scopes in a class *MyClass*. Public identifiers from the System class and inherited class U are at public scope. Type T1 and function X are declared at public

scope. Type T2 and function Y are declared at private scope:



Identifiers must be declared before being used

All identifiers must be declared before being used. For example:

```
const MAX : integer <- 100
type  T   : record
                data   : string[MAX]
                length : integer(0:MAX)
            end record
var   V   : T
```

The constant MAX, type T and variable V have to be declared in the order shown. The type uses the constant so the constant comes first. The variable uses the type, so it comes after the type. Identifiers can also be used from wider scope levels, which allows constants, functions and types to be inherited from a used class. Compile time error ERR_UNDECLARED_ID occurs if identifiers are used that are not declared.

Functions must be declared before being called. Functions are declared when the compiler finds either a function prototype or the complete function declaration. Only public functions may use prototypes, and all prototypes must have a full declaration later on in the same class. A function prototype is known as a forward declaration in Pascal.

To make sure functions are declared before they are called, they can be ordered so they appear in the source code before any call is made to them. If you prefer to order functions by category or alphabetically, declare the functions using prototypes, then fully declare the functions in any order later on in the class.

When using Recursive Functions the function header is the declaration, therefore the function is correctly declared before the recursive call. In the case of Mutual Recursion it is not possible to fully declare both functions before they are called. Therefore one or both function headers must be declared using a function prototype.

Duplicate identifiers not allowed

An identifier can only be declared once at a given level of scope, or the compiler will report a Duplicate Identifier error. For example, consider a class containing a constant and a function, both at private scope. They cannot have the same name as they are both at the same scope level.

This rule is applied differently for record fields. A record field name can be used by other types at the same scope level because it was not used as the name of the record type itself. However the field names within a single record must be different. The next example shows record declarations that are allowed:

```
+-------------------------------------+
|  type X    : integer                |
|  type REC1 : record                 |
|                X : string[10]       |
|              end record             |
|  type REC2 : record                 |
|                X : string[20]       |
|              end record             |
+-------------------------------------+
```

The identifier *X* is used as a type name, then as a field in type *REC1*, then as a field in type *REC2*.

Also the Duplicate Identifier rule does not apply to constructor functions. These are functions that return a value of a specified type, and that have the same name as the type. The following example shows a type *T* and its constructor function:

```
+-------------------------------------+
|  type T[*]:list[*] of string[100]   |
|  function T(... out result:T[*])    |
+-------------------------------------+
```

The type *T* and its constructor function *T()* have the same name, and the constructor has an **out** parameter of type *T*.

Identifiers can be re-used at different scope levels

When an identifier is used at different scope levels, it always applies to the innermost declaration. This is shown in the next example. Use of C in function X refers to the local declaration. Use of C in function Y refers to the declaration at private scope, which is the only visible declaration:

```
+-------------------------------------------Private-scope-+
|     private const C:integer <- 1                        |
|     private type T[*]:string[*]                         |
|                                                         |
|     private function X()                                |
|  +----------------------------Function-scope-+         |
|  |  const C : real(0:1e9)                    |         |
|  |  code                                     |         |
|  |    // C here refers to local declaration  |         |
|  |  end function                             |         |
|  +-------------------------------------------+         |
|                                                         |
|     private function Y()                                |
|  +----------------------------Function-scope-+         |
|  |  code                                     |         |
|  |    // C, T refer to private declaration   |         |
|  |  end function                             |         |
|  +-------------------------------------------+         |
+---------------------------------------------------------+
```

This is called scope overriding and it makes possible the re-declaration of identifiers from an imported class. Imported identifiers have public scope and are visible to the entire class that uses them. If these

identifiers are re-declared at private scope or function scope, they override the ones imported from the used class.

Type identifiers are an exception to the rule, since they cannot be re-declared at different levels of scope. If a type *T* is declared with public scope, or is imported from another class, *T* cannot be re-declared at any other scope level. The reason is the language uses the name equivalence type system. If a function was declared using type *T* and *T* was re-declared, calls made to the function using the new *T* would be passing data of the incorrect type. To avoid this, types cannot be re-declared.

2.3 Functions

A function is a block of code with a name and **in**, **inout** and **out** parameters. Functions are allowed to modify the inout and out parameters. There can be any number of in and inout parameters, but only a single out parameter. Functions can access global constants and types, although nested functions are not allowed. The following diagram shows the structure of a function, then the different parts are explained:

```
+-------------------------------------------------------------------+
|  [public|private] [callback] function name(in parameters          |
|                                          inout parameters)        |
|  precond BooleanExpression                                        |
|  postcond BooleanExpression                                       |
|  type type-declaration                                            |
|  const const-declaration                                          |
|  var var-declaration                                              |
|  code                                                             |
|     statement...                                                  |
|  end function                                                     |
+-------------------------------------------------------------------+
```

Function. This is the start of the function signature, also known as the function prototype or function heading. The signature is the function name followed by the parameter types. The parameters are enclosed in round brackets, and the brackets are left empty if there are no parameters.

The optional keywords **public**, **private** or **callback** may be used as part of the function heading. When these keywords are used, they should be on the same line as the function heading. This helps the Developer Environment when it automatically adds and removes window functions from classes.

Public/**Private**. If the function is to be made available to other classes, the **public** keyword must be used in the function heading. If the function is not being made public, either the **private** keyword is used, or neither keyword is used.

Public functions can also be declared in two stages. To do this, the function prototype is declared first, then the full function declaration is made later on in the same class. When the function is fully declared the heading must be identical. This means the same number of parameters, the same direction (in, inout or out), the same types and the same parameter names.

Callback. This is an optional keyword used in the function heading. It specifies a window function which handles events occurring in a window. Callback functions use specific parameters as described in the How Windows work topic.

In, Inout. These optional elements are part of the function signature and indicate whether the parameters following them may be modified. In parameters may not be modified within the function and are constant. Inout parameters carry an initial value into the function, which may be modified by the function, and which is returned when the function returns.

Out. The Out parameter is also optional. An Out parameter does not carry an initial value into the function, but it may be modified within the function. The out parameter is the return value when the function is called from an expression. Functions with an out parameter are not allowed inout parameters, because function calls made from expressions are not allowed side effects. This is

discussed under Pure Functions later on.

Precond, postcond. These optional elements are boolean conditions immediately following the function heading.

A function's precondition is a boolean condition that should be true when the function is called. If precondition checking is enabled, a test is made at run time to make sure. The preconditions act as additional documentation about the function, make clear any assumptions made by the function about its inputs, and make possible the design of abstract data types.

A function's postcondition is another boolean condition that should be true immediately the function ends. If postcondition checking is enabled, a test is made at run time to make sure. The postcondition documents what operations the function is capable of doing, assuming its preconditions are met. Preconditions, postconditions and Abstract Data Types are covered in more detail in the section on Object Oriented Programming.

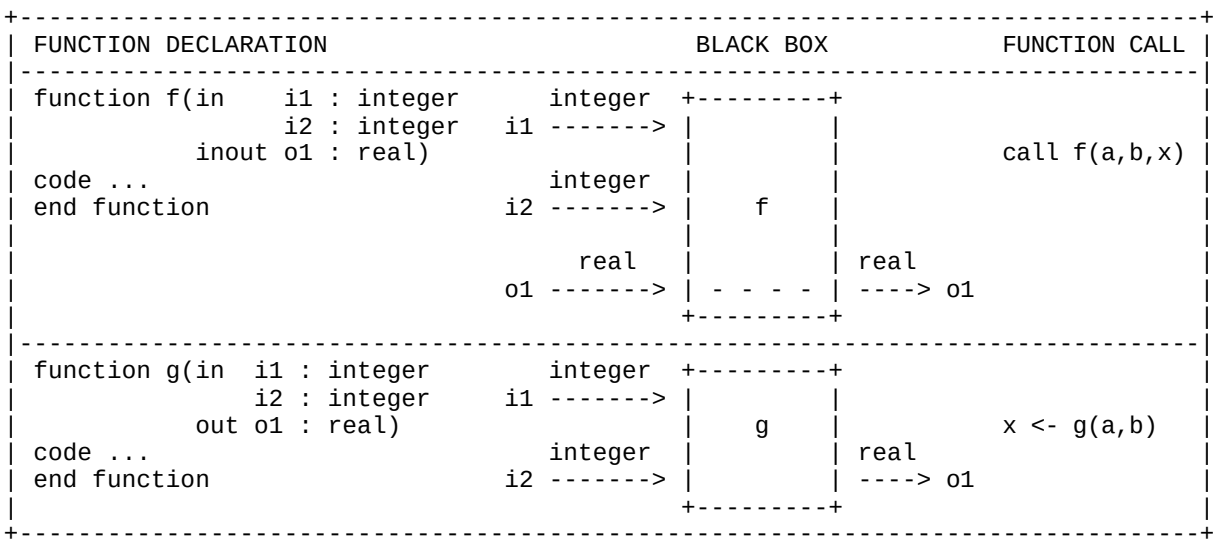
Type, const, var. These are local declarations which are in scope for the entire function. The **var** declaration sets up local variables which retain their value while the function is active. After the function returns the local variables lose their values and are reset if the function is called again.

Code. This keyword marks the start of the instructions or statements of the function. Statements are covered in more detail in the Program Statements section of the Language Report.

End function. This keyword marks the end of the function's statements. When program flow reaches this point, the function returns and any **inout** or **out** parameters are made available to the caller. Note the **end** keyword is also used at the end of statements and classes. This saves having to remember special ending keywords.

Parameter Passing

When using a high level language you should not have to worry about whether to pass parameters by reference, by value or by name. In Ubercode the only concern is whether a parameter is an input (**in**), combined input and output (**inout**), or an output (**out**) of a function. These are shown next:



The In parameters (*i1* and *i2* above) carry information into the function. They cannot have their values altered within the function, so they cannot be assigned to nor passed to other functions for modification. The Inout parameters (such as *o1* above) can have their values altered and they carry information in and out of the function. Any initial value they had at the start is available.

Instead of the inout parameters there can be a single Out parameter which carries information out of the function. This cannot be used to pass data in, since it represents the function's return value in an

expression. Here is the calling rule:

```
+-----------------------------------------------------------+
|  functions with inout       use a call statement          |
|                             e.g. call f(a,b,x)            |
|-----------------------------------------------------------|
|  functions with out         use an expression             |
|                             e.g. x <- g(a,b)              |
+-----------------------------------------------------------+
```

Function parameters and arguments

The words *function arguments* and *function parameters* are often used when discussing functions. The function parameters follow the **in**, **inout**, or **out** keyword in the function heading and represent a value used in the function. The arguments are the actual variables or values used in the code where the function is called. Some languages refer to parameters as *formal parameters* and arguments as *actual parameters*. The following table shows parameters and arguments:

```
+---------------------------------+------------------------+
| Parameters: i1 i2 o1 o2         | Arguments: a b x y     |
|---------------------------------+------------------------|
| function f(in    i1 : integer   | call f(a,b,x,y)        |
|                  i2 : integer   |                        |
|            inout o1 : real      |                        |
|                  o2 : real      |                        |
|   code ...                      |                        |
| end function                    |                        |
+---------------------------------+------------------------+
```

A function has one set of parameters declared in its heading, and can be called from different places with different arguments. The parameters in a function heading have the same format as variable declarations. Any type name, including the standard scalar types, can be used for parameters. The sizing notation in square brackets must be included if it is part of the type name. This allows strings, arrays, sets, lists and tables of different sizes to be used as arguments.

The next example shows a two dimensional array type *Tmat[*:*,*:*]* used as a parameter type. The function parameter does not use numeric bounds, which makes it possible to pass arrays of different sizes to a function. In Pascal this is called a parametric array type, and in Ubercode the same technique can be used with string, set, array, list and table types:

```
+-----------------------------------------------------------+
|  type Tmat[*:*,*:*]:array[*:*,*:*] of real (-1e12:1e12)   |
|                                                           |
|  function Inverse(in m : Tmat[*:*,*:*]                    |
|                      i : integer)                         |
|  code                                                     |
|    ...                                                    |
|  end function                                             |
|                                                           |
|  function Main()                                          |
|  var m1 : Tmat [1:10, 1:10]                               |
|      m2 : Tmat [1:100, 1:100]                             |
|  code                                                     |
|      call Inverse(m1, 10)                                 |
|      call Inverse(m2, 100)                                |
|  end function                                             |
+-----------------------------------------------------------+
```

A function's **out** parameter has a default out value which is returned by the function unless modified by code inside the function. The default value of an **out** parameter is the same as the Default Value of a variable of the same type if declared as a dynamically sized variable.

Since sets and arrays used as **out** parameters are initialized as empty resizable structures, you may want to use Redim to change their size before returning from the function. When a function having a

set or array **out** parameter returns, the function return value is a resizable set or resizable array with the bounds most recently set by the called function. If you want to copy the returned value to a fixed size variable it must have the same bounds. Run time error ERR_ARRAY_COPY occurs if you copy a resizable array to a fixed size array with mismatching bounds.

Polymorphic functions

A polymorphic function is one that accepts different parameters of different types (see [Tennent 1981 p194] in the Bibliography). This is done by declaring multiple functions of the same name with different parameters. This is also called *overloading* because a single function name is overloaded with several different functions.

This does not cause a problem with duplicate identifiers being used at the same scope level, because the compiler considers the parameters of a function to be a part of its name. In mathematical terms the function' s parameters are part of its signature so functions of the same name with different parameters are different. For example function *Str* shown next is overloaded three times, allowing it to be called with an argument of integer, fixed point or real type:

```
+-----------------------------------------------------------+
|  function Str(in i:integer out s:string[*])               |
|  code                                                     |
|    ...                                                    |
|  end function                                             |
|                                                           |
|  function Str(in i:real out s:string[*])                  |
|  code                                                     |
|    ...                                                    |
|  end function                                             |
|                                                           |
|  function Str(in i:fixed out s:string[*])                 |
|  code                                                     |
|    ...                                                    |
|  end function                                             |
+-----------------------------------------------------------+
```

When polymorphic functions are called, the actual function used is the one matching the types of the arguments. For example, the following statement calls all three functions, first calling the integer version of *Str*, then the real number version, then the fixed point version:

```
 s <- Str(100) + " " + Str(1e9) + " " + Str(10.25)
```

This example shows a problem when polymorphism is mixed with automatic type conversions because it is not always clear which version of the polymorphic function should be used. This is known as an ambiguous function call - for example does *Str(100)* mean "use the integer version of the function on the integer 100", or "use the real version of the function on the integer 100 converted to a real number"?

When resolving a call to a polymorphic function, the compiler searches first for a function signature that matches exactly the supplied parameters. If this search was unsuccessful structured types are converted into Generic Types and a new search is made using the modified type list. This gives the following search order:

(1) Search for a version of the polymorphic function using the original arguments.

(2) If no matching function was found, convert any structured types to generic types.

(3) Search for a version of the polymorphic function using the generic types.

Polymorphism cannot work if the different functions have exactly the same parameters. This cannot happen within a class, because it would have duplicate identifiers and would not compile. But this could happen if two separate classes declared identical public functions and a client tried to use both classes. The client class could not be compiled because both functions are at public scope and

therefore clash.

Polymorphism also makes possible functions with optional parameters - this is done by overloading several functions with the same name having different numbers of parameters. This is possible because polymorphism allows the number of parameters to be changed as well as their types. An example of this is with the Msgbox function. One version of the function allows the buttons in the message box to be specified, but by leaving out the button parameter the message box defaults to a single OK button.

Constructor functions

These are functions used for initializing variables of any type, except pre-defined types. When declaring a type *T* it is possible to declare a constructor function for *T* also. The function has the same name as the type, it may have any number of **in** parameters, and it must return type *T* as an **out** parameter. The next example shows a type *Tdata* and its constructor function:

```
+-------------------------------------------------------+
| type Tdata:record                                     |
|   name:string[50]                                     |
|   address:string[300]                                 |
|   species:string[50]                                  |
| end record                                            |
|                                                       |
| function Tdata(in name:string[*] addr:string[*]       |
|                out result:Tdata)                      |
| code                                                  |
|   result.name <- name                                 |
|   result.address <- addr                              |
|   result.species <- "earthling"                       |
| end function                                          |
+-------------------------------------------------------+
```

In the example shown, *Tdata* is the type and *Tdata* is also the constructor function. If a variable *v* of type *Tdata* is declared, *v* can be initialized using the constructor function. For example the following statement:

```
 v <- Tdata("John Smith", "123 High St")
```

will correctly initialize the variable *v*. Constructors are useful when classes declare public types, since they make it easy to initialize variables using the type in other classes. Although it is not necessary to use the constructor in the class declaring it (it is easy to use structured expressions instead), this does not apply in other classes. The other classes cannot use structured expressions to initialize the type, and must use the constructor function instead.

Iterator functions

These return a list or array as an **out** parameter and are normally called from a **for each** loop. The loop steps a variable through each item in the returned list/array. The following example declares an iterator function *Weekdays* that returns the days of the week, then uses the For each loop to print out the days:

```
// Iterfunc.cls
Ubercode 1 class Iterfunc

type TstringList[*]:list[*] of string[20]

function Weekdays(out days:TstringList[*])
code
  days <- {"Monday","Tuesday","Wednesday","Thursday","Friday"}
end function

public function main()
var day:string[*]
code
  for each day in Weekdays()
    call Msgbox("Days", day + " is a weekday")
  end for
end function

end class
```

The example declares the *TstringList[*]* type returned by the iterator function. *Weekdays* is the actual iterator function, which initializes the list and returns it as an **out** parameter. The *For each* loop in function *main* prints each day in a message box. *Iterfunc* is a main class because it declares *main* as a public function.

Iterator functions often return visual objects that represent user interface elements. The following iterator functions are available in the run time library:

Applications returns all the running programs.

Controls returns all the controls used in a window. Both visible and hidden controls are included in the list.

Printers returns all the connected printers. If the computer has no printers attached, the returned list is empty.

Windows returns all the windows that are currently loaded. Both visible and hidden windows are included in the list.

When iterator functions return visual objects, they should be used as soon as they are returned and should not be stored for later use. The state of a visual object changes when windows are loaded or unloaded, or when printer jobs are started or ended. A copy of a visual object is not valid after its state changes. Functions that change a visual object' s state are Load / Unload for windows, and Startprint / Endprint for printers.

Recursive functions

A recursive function is a function that calls itself. The next example shows a factorial function that calls itself to calculate the next lowest number:

```
+-----------------------------------------------------------+
|  function factorial(in n:integer out result:integer)     |
|  code                                                     |
|    if n = 0 then                                          |
|      result <- 1                                          |
|    else                                                   |
|      result <- n * factorial(n-1)                         |
|    end if                                                 |
|  end function                                             |
+-----------------------------------------------------------+
```

Recursion is a useful technique when a problem can be defined in terms of itself. If recursion occurs unintentionally, or the recursive function has no exit condition, the program enters an infinite loop and

a stack overflow occurs. Infinite recursion can also occur when using graphical events if the code handling an event triggers the same event again.

Mutual recursion

Mutual recursion occurs when two Recursive Functions call each other. The next class shows functions *a* and *b* which are mutually recursive:

```
Ubercode 1 class Mutrec
public function b()

public function a()
code
  call b()
  // other code
end function

public function b()
code
  call a()
  // other code
end function

end class
```

Function *b* is declared first as a public prototype, to make it available to function *a*. After function *a* is declared, function *b* is then fully declared. This order of declaration makes sure both functions are in scope for each other. Mutual recursion is a complex technique occasionally needed by some algorithms. As with normal recursion, there must be some condition under which the recursive functions return, otherwise the program will enter an infinite loop.

Pure functions

A pure function is a mathematical function - a mapping of any number of inputs of any type into any number of outputs. A particular set of inputs must always produce the same output whenever the function is called. This is also called a deterministic function.

Pure functions make software more reliable and easier to maintain. A function can be fully understood without reference to its calling environment, because global data and the order of calls cannot affect how a function works.

Unfortunately functions are not pure in most computer languages. A random number generator is impure - otherwise it would always give the same output for the same inputs and be a very predictable random number generator! Other impure functions are those that find the time or date, amount of free disk space, or read input from the program operator.

Functions in Ubercode are as close to pure functions as possible. To achieve this, Global Variables and Pointer Type are not included in the language, and aliasing is restricted.

Aliasing

Aliasing occurs when different variables refer to the same storage location. It can happen if the same variable is passed to a function more than once. For example:

```
call f(a,a,b,b)
```

Aliasing is only allowed between inputs. It is not allowed between an **in** and **inout** parameter, nor between two **inout** parameters. Assume we have a function *fn*, and variables *a* through to *f*, declared as follows:

```
+-----------------------------------------------------------+
|  function fn(in    i1 : integer                           |
|                    i2 : integer                           |
|              inout o1 : integer                           |
|                    o2 : integer)                          |
|  code ...                                                 |
|  end function                                             |
|                                                           |
|  type vect[*:*] : array [*:*] of integer(1:1000)          |
|                                                           |
|  var  a         : integer(1:1000)                         |
|       b         : integer(1:1000)                         |
|       c         : integer(1:1000)                         |
|       d         : vect [1:10]                             |
|       e         : vect [1:10]                             |
|       f         : vect [1:10]                             |
+-----------------------------------------------------------+
```

The following function calls are legal. The first and second calls are allowed because the same variable or array may be used for different inputs. The third and fourth calls are allowed because two different variables *c* and *d* are used as the **inout** parameters:

```
+-----------------------------------------------------------+
|   call fn(a, a, b, c)                                     |
|   call fn(d[1], d[1], e[1], f[1])                         |
|   call fn(a, b, c, d[c])                                  |
|   call fn(a, b, c, d[a])                                  |
+-----------------------------------------------------------+
```

The following examples are not allowed, because a variable or variables are aliased, either by being used as an **in** and an **inout** parameter, or by being used twice as an **inout** parameter:

```
+-----------------------------------------------------------+
|  call fn(a, 1, a, b)         // a is aliased              |
|  call fn(a, b, b, a)         // a and b are aliased       |
|  call fn(a, b, c, c)         // c is aliased              |
|  call fn(d[1], a, d[2], b)   // array d is aliased        |
|  call fn(a, b, e[1], e[2])   // array e is aliased        |
+-----------------------------------------------------------+
```

Aliasing is disallowed in this way to prevent **in** and **inout** values from being altered by other **inout** values. If it was not disallowed, then an alteration to an inout variable could cause another input value or inout variable to change as well. See [Tai 1982 p28] in the Bibliography for a detailed discussion.

2.4 The Type Model

This section describes the type model. The actual data types themselves are described in more detail in later sections, under Scalar Types and Structured Types.

What is a Data Type?

In computing, it is important to classify values so we know what they represent. For example computer languages distinguish between real numbers, integers and arrays of integers. The classifications are called *types* and all values such as constants, variables, functions and expressions have a type. The type denotes the set of allowed values, along with the permitted operations. For the concept of type in Ubercode, see [Dahl 1972 p92-3] in the Bibliography:

(1) A type denotes the set of values which may be assumed by a constant, variable, function or expression.

(2) Every value belongs to one and only one type.

(3) The type of every value may be deduced from its form or declaration at compile time.

(4) All operators expect operands of a particular type and return values of a particular type. If a symbol is applied to several different types, such as **+** which is used for addition of integers as well as real numbers, the symbol denotes several different actual operators.

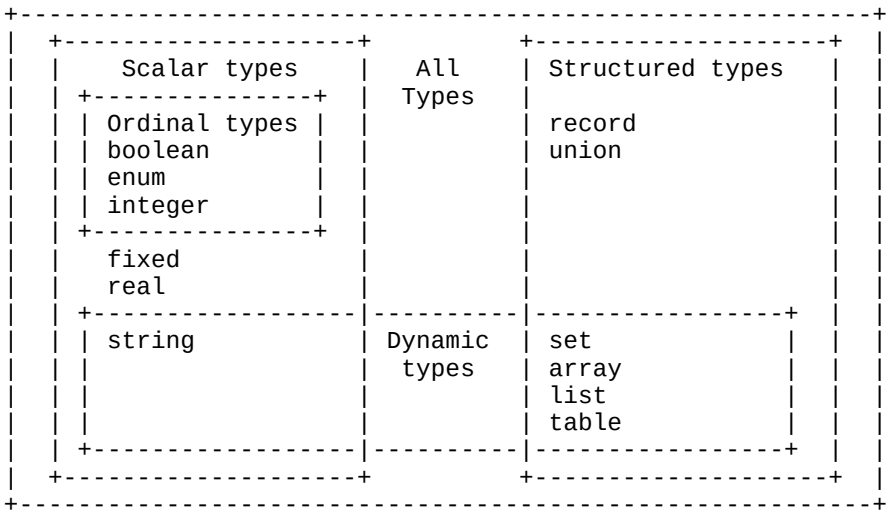
Data Types

The *Data Types* consist of six scalar types and six structured types. The scalar types are implemented as primitive types on most computers, and the structured types are built out of other types. They are:

boolean type True and False values.
enum type Enumerations of defined values.
integer type Integer values.
fixed type Fixed point notation for financial calculations.
real type Floating point numbers.
string type Variable length strings.
record type Multiple components of different type.
union type Components selected by a tag value.
set type Sets of integers.
array type Multiple components of one type.
list type Multiple components selected by integer index.
table type Record components selected by a key value.

Type diagram

The following diagram shows how the data types are related:



Scalar Types. These have a defined ordering (see [Wirth 1976 p5] in the Bibliography). They can be compared with the **=**, **/=**, **>**, **<**, **>=**, **<=** operators.

Ordinal Types. These can be used as selectors in **select** statements and can be used with the Ord function.

Structured Types. These are built from components of structured or scalar type. All structured types are ultimately built from scalar types or type identifiers.

Dynamic Types. These change size as the program runs, and can be stored in memory or on disk. If they are used as components of structured types, a maximum size must be given. Therefore dynamic types can be dynamically sized, or variable to a fixed maximum size.

Ubercode uses *strong typing* which means all identifiers in a program have a single type which is known at compile time. When the program runs, variables may not adopt values of other types [Sebesta 1989 p122]. By using strong typing, many errors are found by the compiler, which reduces debugging time [Rowe 1984 p56].

High Level Data Types

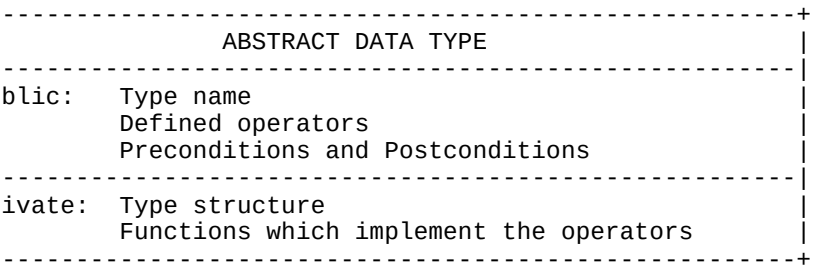
Ubercode provides high level data types such as lists and tables. These types can be of dynamic size, so they can contain any number of components. List components are selected by a number and table components are selected by one or more keys of any scalar type. These high level data types mean programming is not needed for linked lists, balanced binary trees or hash tables.

Abstract Data Types

An Abstract Data Type is a new data type defined by an application for its own use. Abstract data types are defined by their type name, operators, and the axioms describing the relationships between the operators and the types (see [Martin 1986] in the Bibliography). An abstract data type is specified as follows:

- The type name. This is represented by a public type identifier.
- The operators. These are represented by the public functions.
- The axioms. These are represented by the precondition and postcondition of each public function. They define the abstract data type in terms of how the functions and types interact.

Although the type name, operators, preconditions and postconditions are declared public, the internal structure of the type and the operators are private. A client class may use the abstract data type by declaring variables of its type. An abstract data type is represented as follows:



To construct an abstract data type, write a class with a public type, public functions that operate on the type, and use preconditions and postconditions with the functions. Classes can inherit functions and types from other classes, and the structure of inherited types is always hidden. Such a class meets the requirements of an abstract data type used for object oriented programming (see [Ellis 1991 p65-66, p187-188] in the Bibliography).

Type Declarations

Type declarations in Ubercode specify an unlimited range of values. For example when declaring an integer, string or array type you don' t need to specify the range of integer values, the maximum length of the string or the bounds of the array.

In other languages such as Pascal and Ada, types can be restricted by defining a subtype. Values of the subtype also belong to the parent type. This violates a key principle of type theory that all values belong to one type only (see [Harland 1982 p65-9] in the Bibliography). A good type model is at the core of a computer language, therefore Ubercode rejects the notion of subtype.

This means strings are type compatible with other strings of different lengths, and array variables of the same type can be declared with different bounds. This allows functions to be passed different sized arrays and strings as inputs. This approach has been tried in other languages - see [Hennessy 1982] and [Ghezzi 1982 p76] in the Bibliography for more details.

Each type represents a set of values which is potentially infinite. A variable declaration limits the set to a predefined range, so it can be represented in the computer' s memory. Some type declarations are shown next:

```
+-----------------------------------------------------------+
| // Scalar types...                                        |
| type                                                      |
|   bool           : boolean                                |
|   days           : enum (mon,tue,wed,thu,fri,sat,sun)     |
|   temperature    : integer                                |
|   money          : fixed                                  |
|   epsilon        : real                                   |
|   person[*]      : string[*]                              |
+-----------------------------------------------------------+

+-----------------------------------------------------------+
| // Structured types...                                    |
| type                                                      |
|   rec            : record                                 |
|                      ComponentType                        |
|                    end record                             |
|   un             : union                                  |
|                      ...                                  |
|                    end union                              |
|   numset[*:*]    : set[*:*]                               |
|   vector[*:*]    : array[*:*] of ComponentType //1d       |
|   matrix[*:*,*:*] : array[*:*,*:*] of ComponentType //2d  |
|   lst[*]         : list[*] of ComponentType               |
|   tab[*]         : table[*] of ComponentType              |
|                    ...                                    |
|                  end table                                |
+-----------------------------------------------------------+
```

All these types can be used as function parameters and variable declarations. Variables of these types may be copied to one another, loaded and saved from files, and converted to and from strings and XML format.

All structured types (and strings) may be declared public and made available for other classes. Structured types are allowed constructor functions for initializing variables declared of the type. Constructor functions have the same name as the type, and return an instance of the type as an **out** parameter.

The dynamic types (strings, sets, arrays, lists and tables) have a sizing notation - the square brackets and asterisks. This notation is part of the type name, and must be included wherever the name is used. For example if the one-dimensional array type *vector[*:*]* shown above is used as a parameter type, it must include the *[*:*]* element. The compiler needs this so it knows the number of dimensions.

As with functions, public types can be declared in two stages. To do this, the type prototype is declared first, then the full type declaration is made later on in the same class. This is useful for grouping the public elements of a class at the top of the source file.

When public types are declared in two stages, the type' s full declaration is required before the type can be used in other declarations. This is so the compiler knows what the type is. For example this is correct:

```
public type T[*]            // Type prototype
public type T[*]:string[*]  // Full type declaration
const X:T[*] <- "xyz"       // Now we can use the type
```

and this is not:

```
public type T[*]            // Type prototype
const X:T[*] <- "xyz"       // Error, T is not fully declared
public type T[*]:string[*]  // T is fully declared here
```

The error occurs when *T[*]* is used to declare the constant X before *T[*]* has been fully implemented. The compiler does not know what type *T[*]* is yet, so it cannot check the constant initializations and error ERR_TYPE_UNIMPLEMENTED occurs. Types must be fully implemented, not only for constant declarations, but for variables, components of other types, and function parameters as well.

To summarize, the key elements of the type system are:

```
+-----------------------------------------------------------+
|  There are no subtypes.                                   |
|                                                           |
|  The size of a structure is not part of its type.         |
|                                                           |
|  The sizing notation is part of a type name.              |
+-----------------------------------------------------------+
```

Var Declarations

When variables are declared they are given constraints which define the allowed range of values. This also applies when variables are used as components of structured types. For example, integers, real numbers and fixed point numbers are given a minimum and maximum value, and strings are given a maximum length. These constraints ensure portable software because the compiler makes sure they are satisfied on different computers (see [Tremblay 1985 p96] in the Bibliography).

As discussed in the section on Type Declarations constraints are not part of the data type. For example, two **string** variables of different lengths are the same type. The type declares the theoretically infinite set of values allowed, and the variable declaration lists the subset used. The range is therefore a constraint on the values allowed for the variable.

Variable declarations and their constraints are shown in the next diagram. These are based on the types shown previously:

```
+-----------------------------------------------------------+
| // Variables using scalar types...                        |
| var                                                       |
|   b      : boolean          // implicit constraint       |
|   day    : days(mon:sun)    // enum constraints          |
|   temp   : integer(1:100)   // integer constraints       |
|   amount : fixed(0:10000.0) // fixed point constraints   |
|   eta    : real(-1e9:1e9)   // real number constraints   |
|   name1  : string[100]      // string of max length 100  |
|   name2  : string[*]        // dynamically sized string  |
+-----------------------------------------------------------+

+-----------------------------------------------------------+
| // Variables using structured types...                    |
| var                                                       |
|   r      : rec                // implicit constraint     |
|   u      : un                 // implicit constraint     |
|   hand   : numset[1:52]                                   |
|   v1     : vector[1:10]       // v1 and v2               |
|   v2     : vector[1:100]      //   are the same type     |
|   mat1   : matrix[1:10, 1:10]  // mat1 and mat2          |
|   mat2   : matrix[1:100,1:100] //   are the same type    |
|   l1     : lst[10]            // fixed size list         |
|   l2     : lst[*]             // dynamic sized list      |
|   t1     : tab[10]            // fixed size table        |
|   t2     : tab[*]             // dynamic sized table     |
+-----------------------------------------------------------+
```

The constraints for variables of enum, integer, fixed point and real types are in round brackets, and

constraints for variables of dynamic types (string, set, array, list, table) are in square brackets. There is a reason for the different brackets. Round brackets contain the lower and upper limits on values that can't be subdivided, such as integers and real numbers. Square brackets define the maximum number of elements in variables that store repeated items. The constraints in square brackets are also known as sizing notation since they define the number of elements in dynamic types.

When a variable of dynamic type is declared it may be constrained to a fixed maximum size by replacing the asterisks with integer values, for example *name1*, list *l1* and table *t1* above. Or it may be dynamically sized by leaving the asterisks in the declaration, for example *name2*, list *l2* and table *t2* above. A variable of fixed maximum size may vary in size up to the maximum declared bound, and a dynamically sized variable can vary to any size, limited only by available memory.

For example the string variable *name1* shown above has a fixed maximum size and may contain any number of characters between 0 and 100 inclusive. As the number of characters varies, its length (as returned by the length function) varies between 0 and 100. The string variable *name2* is dynamically sized and may contain any number of characters between 0 and the maximum memory block size. Similarly its length will vary between zero and the maximum memory block size.

If the variable is a component of a structured type, it must be constrained to a fixed maximum size. This is done by replacing the asterisks with integer values, and is required because all components have a fixed maximum size in case the structured type is stored on disk.

To summarize, the key elements of variable declarations are:

```
+-------------------------------------------------------------------+
|  (1)  Declarations of enum, integer, fixed and real types have    |
|       constraints in round brackets ( ).                          |
|                                                                   |
|  (2)  Declarations of string, set, array, list and table types    |
|       have constraints in square brackets [ ]. These declarations |
|       can replace each asterisk from their type declaration with  |
|       a number, to declare a variable of fixed maximum size.      |
|                                                                   |
|  (3)  Declarations of boolean, record and union types have        |
|       implicit constraints.                                       |
|                                                                   |
|  These declarations (1) to (3) can be used as component types.    |
|                                                                   |
|  (4)  Declarations of string, set, array, list and table types    |
|       can leave the asterisk in the type name. These declarations |
|       are dynamically sized and cannot be used as component       |
|       types.                                                      |
+-------------------------------------------------------------------+
```

Const Declarations

Constant declarations are used for declaring non modifiable values. The identifiers may be used locally to a function, or within a class, or may be public for use by other classes. Constants do not need constraints since they have a fixed value and cannot be modified.

```
+-----------------------------------------------------+
| const                                               |
|   YES      : boolean <- True                        |
|   MAX_PATH : integer <- 255                         |
|   FIXNUM   : fixed <- 1000.0                        |
|   REALNUM  : real <- 6.023e+23                      |
|   STRTEXT  : string[*] <- "Big" + "Cat"             |
|   LETTERS  : set[*:*] <- [Ord("A"):Ord("Z")]        |
+-----------------------------------------------------+
```

The constant's value is defined to the right of the arrow, as shown above. Constants can use a single value, or may consist of other constant values combined by constant expressions. When constant expressions are used, all parts of the value must be available when the program is compiled. By

convention, constants use upper case, and integer constants that define the maximum sizes of strings or other structures start with *MAX_* as shown. These conventions are optional.

Default value

When variables are declared they automatically have an initial value or default value, which applies until a new value is copied into the variable. This is for consistency when an uninitialized variable is used in an expression. The default value of each type is as follows:

Type	Default value
Boolean	False
Enum	Ordinal value of 0
Integer	0
Fixed point	0.0
Real number	0e0
String	""
Record,union	All fields set to default
Set	[ ]
Resizable array	Empty array with no elements
Fixed array	All elements set to default
List,table	Length of zero

Type Compatibility

Most computer languages define their Type Equivalence system, which is how they decide whether two values are compatible types. Type compatibility is needed if one value is to be copied or compared to another. Values passed to a function must be compatible with the corresponding function parameters declared in the function header.

In Ubercode Name equivalence is used, so two variables are the same type if they were declared with the same type name. The constraints are not taken into account because they are not part of the data type. For example:

```
type num         : integer
     vector[*:*] : array [*:*] of integer (1:MAXINT)
     meters      : real
     yards       : real
var  n1          : num     (1:10)         // n1 and n2 are same type
     n2          : num     (1:100)
     i1          : integer (1:1000)       // i1 and i2 are same type
     i2          : integer (1:10)
     a1          : vector  [1:10]         // a1 and a2 are same type
     a2          : vector  [1:20]
     len1        : meters  (-1e12:1e12)   // len1 and len2 are different
     len2        : yards   (-1e12:1e12)   // types because of
                                          // different type names
```

In the declarations above, *n1* and *n2* are the same type because they are both *num* type. Also *i1* and *i2* are the same type because they are both integer. Arrays *a1* and *a2* are the same type because they are both *vector* type - the different bounds do not affect type compatibility. But *len1* and *len2* are different types, because the type names *meters* and *yards* are different. This type difference is important - if *len1* contained 10 meters, copying *len1* to *len2* must be forbidden because 10 yards is not the same as 10 meters. When declaring *len1* and *len2* it makes no difference that both are based on real number type - the difference in names is the important factor.

The type compatibility rule is extended to make boolean, integer, fixed point, real, string and set types compatible with their descendents. Thus if we have two types, one of which is **boolean**, **integer**, **fixed**, **real**, **string** or **set**, and the other of which is a descendent type of the first, the types are

compatible. This addition to the rule allows literals of boolean, integer, fixed, real, string and set type to be copied to descendent types. Using the example code shown previously, the type *num* is compatible with integer literals and the types *meters* and *yards* are compatible with real number literals. Therefore the following are allowed:

```
n1 <- 10
len1 <- 1e3
len2 <- 2e3
```

Without the extension to the compatibility rule these assignments would not be possible, as *n1* is of type *num* and 100 is of type **integer**, which are incompatible according to the strictest interpretation of name equivalence. Thus to summarize the type compatibility rule used in Ubercode:

```
+-------------------------------------------------------------------------+
|  (1)  Types are compatible if they have the same type name.             |
|  (2)  Types descended from boolean, integer, fixed, real, string or set |
|       type are compatible with the same top level type. This allows:    |
|          type logical:boolean                                           |
|               text[*]:string[*]                                         |
|          var  x:logical                                                 |
|               y:text[*]                                                 |
|          code x <- True                                                 |
|               y <- "Hello"                                              |
|  (2b) Type identifiers based on enum type are compatible with the top   |
|       level type "enum".                                                |
|  (3)  Fixed type, or a type descended from fixed, is compatible with    |
|       the top level type "integer".                                     |
|  (4)  Real type, or a type descended from real, is compatible with      |
|       the top level types "integer" and "fixed".                        |
+-------------------------------------------------------------------------+
```

2.5 Scalar Types

The scalar types include **boolean** for true or false values, **enum** for an enumeration of values, **integer** for whole numbers, **fixed** for fixed point decimal numbers, **real** for an approximation to the real numbers and **string** for printable characters.

Scalar types have a defined ordering (see [Wirth 1976 p5] in the Bibliography) which means they can be compared using the boolean relational operators. They are also known as primitive type because they are implemented directly in the instruction set of most computers [Dahl 1972].

The ordinal types are a subset of scalar type, which includes **boolean**, **enum** and **integer** type. These types have a unique predecessor and successor [Ghezzi 1982 p80]. They can be used as the selector in **select** statements and with the Ord function. Variables of scalar type have these operators available:

```
+-----------------------------------------------------------+
| = /= > < >= <=    Relational.                             |
| <-                Assignment.                             |
| Ord(v)            Converts ordinal types to integer.      |
| Str(v)            Converts scalar type to a string.       |
| Val(v)            Converts string to a scalar type.       |
+-----------------------------------------------------------+
```

When declaring types, constants and variables a useful naming convention is to use an initial capital T for types (Tsometype), all capitals for constants (PI), and all lower case for variables. This convention is followed for the examples in this chapter. However it is only a suggestion and programmers are free to use any naming convention.

Boolean Type

The two values of boolean type are **true** and **false**. The boolean operators are **and**, **or**, and **not** which

have their usual meanings as follows:

```
+-------------------------------------+
|   a  b   |  a and b   a or b   not a |
|----------+---------------------------|
|   F  F   |     F        F        T   |
|   F  T   |     F        T        T   |
|   T  F   |     F        T        F   |
|   T  T   |     T        T        F   |
+-------------------------------------+
```

A boolean type T, variable v and boolean constant C are declared:

```
+-------------------------------------+
|  type  T : boolean                  |
|  var   v : boolean                  |
|  const C : boolean <- false         |
+-------------------------------------+
```

Enum Type

The **enum** type is declared by listing or enumerating all the values up to a maximum enum value of 255. This allows 256 different values, each with an ordinal value from 0 to 255. These values are available as constants of the type, and the scope of an enum constant is the same as the scope of its type. To declare an enum type Tdays, enum variable v and enum constant C:

```
+-----------------------------------------------------------+
|  type  Tdays : enum (mon,tue,wed,thu,fri,sat,sun)         |
|  var   v     : Tdays (mon:sun)                            |
|  const C     : Tdays <- mon                               |
+-----------------------------------------------------------+
```

Integer Type

Integer type denotes the set of whole numbers. When a variable is declared of integer type, the set of allowable values is included in the declaration. This does not create a subtype or a subrange, instead the allowable values act as constraints on the variable. To declare an integer type T, integer variable v and integer constant C:

```
+-------------------------------------+
|  type  T : integer                  |
|  var   v : integer (1:10000)        |
|  const C : integer <- 365           |
+-------------------------------------+
```

The standard arithmetic operators are available for integers, which are addition (**+**), subtraction (**-**), multiplication (*****), integer division (**div**), integer remainder (**mod**), and unary negation (Neg).

Fixed Type

Fixed type denotes a subset of numbers and fractions that can be represented exactly. Addition, subtraction, and integer multiplication work exactly, making this type useful for financial calculations. When a variable is declared of fixed type, it is constrained, which means the range of allowable values is included in the declaration. To declare a fixed type T, variable v and fixed point constant C:

```
+-------------------------------------+
|  type  T : fixed                    |
|  var   v : fixed (0.0:1000.0)       |
|  const C : fixed <- 100.00          |
+-------------------------------------+
```

The standard arithmetic operators are available for addition (**+**), subtraction (**-**), multiplication (*****),

division (*/*) and unary negation (Neg).

Conversion operators are available to find the fractional part of a number (Frac), and to convert a number to an integer (Int). This last operator works by rounding the fixed point number down to the next lowest integer. If the number is already an exact integer value, it is not altered.

Real Type

The type **real** denotes a subset of real numbers, also known as floating point numbers. These numbers are represented with a finite set of digits, so real number arithmetic is of limited accuracy. The IEEE standard (see [IEEE 1985] in the Bibliography) is used which stores values from 5.0e-324 to 1.7e308, with 15 digits of accuracy. To declare a real type T, variable v and real constant C:

```
+-------------------------------------+
|  type  T : real                     |
|  var   v : real (-1e9:1e9)          |
|  const C : real <- 3.1415926        |
+-------------------------------------+
```

The standard arithmetic operators available are addition (**+**), subtraction (**-**), multiplication (*****), division (*/*) and unary negation (Neg). Other mathematical operators are squaring (Sqr), square root (Sqrt), raising to a power (Pwr), the natural logarithm (Ln), the exponential function (Exp), and trigonometric functions (Sin, Cos, Tan, Atan).

Conversion operators are used to convert a real number to its fractional part (Frac), to a fixed point value (Fix), or to an integer (Int). The value returned by Int is the largest number less than or equal to the real number. For example:

```
Int(2.1) = 2
Int(2.0) = 2
Int(1.9) = 1
Int(1.0) = 1
Int(0.1) = 0
Int(0.0) = 0
Int(-1e-6) = -1
Int(-0.1) = -1
Int(-1.0) = -1
Int(-2.0) = -2
Int(-2.1) = -3
```

If the real number is already an exact integer value, it unaltered. Otherwise it is rounded down to the next lowest integer. In some languages [Klerer 1991 p109] the **int** function is called the entier function. Note that for negative real values, the **int** operator behaves differently from the trunc function of Borland Pascal and the automatic float to int conversion of C.

String Type

The type **string** is used for characters from ISO 646 - the ASCII Character Set as shown in the Appendix. When variables of string type are declared they can remain dynamically sized, or they can be constrained to a fixed maximum size. Strings must be constrained when used as component type. Declarations of a string type T, variables v1 and v2, and a string constant C are as follows:

```
+-------------------------------------+
|  type  T [*] : string[*]            |
|  var   v1    : string[10]           |
|        v2    : string[*]            |
|  const C     : string[*] <- "Hello" |
+-------------------------------------+
```

A string may contain lines of text separated by the new line character NL. This is called a multi line string and is used for text. The ASCII null character (chr(0)) has no special meaning in a string. The null character is important only if passing strings to programs written in C or C++ which expect zero

terminated strings. String comparison uses the ASCII Character Set and stops at the first pair of characters that do not match. For example:

```
" " < "A"
"A" < "B"
"AA" < "AB"
"100" < "200"
"FRED" < "fred"
"Fred" < "fred"
"FRED" < "FREd"
```

For strings of unequal length, pairs of characters are compared until the end of the shorter string is reached. If the strings are still equal up to this point, then the shorter string is the lesser of the two. Thus:

```
" " < " A"
" " < "  "
"" < Chr(0)
"AA" < "AAAA"
"AA" < "ABAA"
"AB" < "ABAA"
"1000" < "200"
"TIM" < "TIMMY"
"TIMMY" < "TIMS"
```

Note the empty string "" is less than Chr(0). The empty string has a length of zero. Chr(0) contains the single character with an ASCII value of zero. Therefore the string Chr(0) has a length of one, and is greater than a string of length zero.

The string operators are concatenation (**+**), copying part of the string (Strcopy), counting the number of lines in a string (Strcount), deleting part of the string (Strdel), finding a part of the string (Strfind), converting strings to lower and upper case (Strlower and Strupper), getting a given line from a string (Strline), making a string from a given character (Strmake), using a new string to replace part of the old one (Strset), and finding the current length (Length).

Also there is a highly optimized function Strch which gets or updates a single character in a string. This function makes it possible to write new string functions that are as efficient as the existing library functions.

The bounds operators (Lbound and Ubound) can be used to find the size constraints of a string. The lower bound of any string is 1. The ubound of a fixed maximum size string is the numeric bound used in its declaration. The ubound of a dynamically sized string is not a meaningful value as the upper limit depends on the available memory, therefore Ubound returns a negative value when used with a dynamically sized string. The Isdynamic function checks whether a string is dynamically sized.

The Chr operator converts an integer into the corresponding single character string from the ASCII character set. Some examples are shown below, where *i* is an integer with constraints (0:255) and *s* is a string of length 1:

```
Ord(Chr(i)) = i
Chr(Ord(s)) = s
Chr(65) = "A"
Str(65) = "65"
```

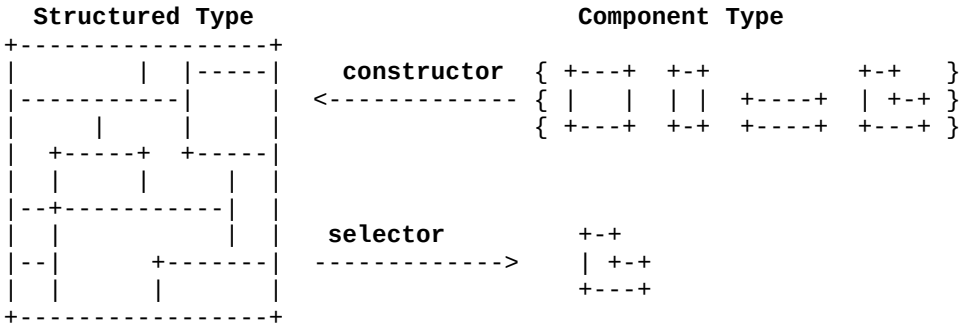
The last two examples show how Chr creates a single character from a numeric ASCII value, whereas Str converts any scalar type into a direct string representation.

2.6 Structured Types

The structured types are the **record**, **union**, **set**, **array**, **list** and **table**. These are built from other types that have already been declared, which are the components of the structure and are therefore

called component type. Any types may be used as component type. Structured types can be built to any degree of complexity.

Operators available for all structured types are the test for equality (**=**) and assignment (**<**). Structured types also have constructors to assemble them from their component types, selectors to access individual components in the structure, and in some cases iterators to loop through their components.



Record Type

The **record** type groups information allowing multiple fields to be processed at once. It stores multiple components of different types. Each component is called a *field*, and can be of any type. Fields declared of dynamic type must have fixed numeric bounds, so that records have a fixed size and can map onto fixed size components of disk files.

A record type *T* is declared by listing all the component fields *f1* to *fn* which are the component types. Types *Tc1* to *Tcn* can be any type. A record type and a record variable are shown next:

```
+--------------------------------------+
|  type  T : record                    |
|              f1 : Tc1                |
|              f2 : Tc2                |
|              .....                   |
|              fn : Tcn                |
|            end record                |
|  var   v : T                         |
+--------------------------------------+
```

Values of record type are initialized using a record constructor. This can use constructor functions or structured expressions. Assume *C1* to *Cn* are constants of type *Tc1* to *Tcn* (shown above). A record variable *v* can be initialized using a structured expression as follows:

```
+--------------------------------------+
|    v <- {C1, C2, ... Cn}             |
+--------------------------------------+
```

The record selector uses the dot notation to access and update individual fields. To select a component, the record name is followed by a dot (**.**) and the field name:

```
+--------------------------------------+
|                 v.f1                 |
+--------------------------------------+
```

Union Type

The **union** type groups different types so that an entire object can be dealt with as a single unit. Although the union has multiple components, only one can be used at a time. This is the reason for the name *union* - because it unites different types. The current active type is indicated by a tag value, also called a discriminant. The tag value is an integer type declared as the first field.

A union type *T* is declared with the tag field, followed by the component fields *f1* to *fn*. The components are of constrained component type *Tc1* to *Tcn*. A union type and union variable are shown next:

```
+-----------------------------------------------------+
|  type  T : union                                    |
|                tf : integer(1:n)                    |
|                with T                               |
|                    case 1 => f1 : Tc1               |
|                    case 2 => f2 : Tc2               |
|                    ...                              |
|                    case n => fn : Tcn               |
|                end with                             |
|            end union                                |
|  var   v : T                                        |
+-----------------------------------------------------+
```

In the type declaration each **case** label declares the component fields belonging to one tag value. Although the example above shows each case label as having one component field, a case label may have zero or more fields. The label is always required even if there are no fields. The tag value must be:

- An integer value greater or than equal to zero.
- In a continuous increasing sequence, without gaps between values. If there are no component fields belonging to a tag value, the **case n =>** element must still be used but the declaration can be left out.
- A value that is available at compile time, which means it must be an integer literal or integer constant declared in the class being compiled. Constants declared in other classes are not allowed as their value is not available at compile time (the value is private to the other class).

Values of union type are initialized using the union constructor. This can use constructor functions or structured expressions. A structured expression uses a tag value constant followed by the components belonging to the tag value. Assume a constant *C1* is declared of type *Tc1* shown above. The union variable *v* is initialized using a structured expression as follows:

```
+-------------------------------------+
|            v <- {1, C1}             |
+-------------------------------------+
```

As another example the value {2,C1} is invalid and would cause a compiler error, because the component *C1* of type *Tc1* does not match the tag value 2. The only way of changing a union' s tag value is with a constructor, or by assigning another union variable of the same type.

The union selector uses a With statement to access the component fields *f1..fn* that match the current tag value.

```
+-------------------------------------+
|  with v                             |
|     case 1 => .....                 |
|     case 2 => .....                 |
|     ...                             |
|     case n => .....                 |
|  end with                           |
+-------------------------------------+
```

The compiler ensures that only field f1 is accessed in case 1, f2 in case 2 and so on. This approach to unions is both efficient and secure. All type checking is done at compile time, and no run time checks are needed to ensure the accessed components match the tag value. It is completely safe because the tag value and component fields always match. This makes it impossible to convert a variable of one type to another by declaring them as fields of a union, and relying on both fields being stored at the same memory location.

This approach can be compared to other languages. A union is called a variant record in Pascal, and the insecurities in the variant are described in [Ghezzi 1982 p84-5] in the Bibliography. The union type

in C has no tag field at all.

Set Type

The **set** type stores integer elements. The ordering of the elements is not significant so for example the set [1,3] is the same as [3,1]. Elements are not stored multiple times, so adding 3 to the set [1,3] would still produce [1,3]. Declarations of a set type *T*, set variables *v1* and *v2* and a set constant *C* are shown:

```
+-------------------------------------+
| type  T [*:*] : set [*:*]           |
| var   v1      : T [1:10]            |
|       v2      : T [*:*]             |
| const C       : T [*:*] <- [1,2,3:5] |
+-------------------------------------+
```

Individual values of a set are separated by commas, and a range of values is given by separating the two values with a colon.

When a set type T[*:*] is declared, it is not constrained to hold any particular range of integers, as represented by the asterisks in the type name. A resizable set variable can be declared by leaving the asterisks in the declaration (v2 above), or the variable can be constrained to a fixed range by replacing the asterisks with numeric bounds (v1 above). When used as components of other structures, sets must always be constrained. At run time the set bounds determine the lowest and highest integers that can be stored. The bounds are obtained with the Lbound and Ubound functions, and a resizable set has bounds as defined with Redim, whereas a fixed size set has the bounds defined in its declaration. The lower bound is always less than or equal to the upper bound, and both bounds are between MINBOUND and MAXBOUND.

Sets are initialized using the set constructor. This can use constructor functions or a list of elements in brackets. For example the set variable *v1* declared above is initialized by specifying its elements as follows:

```
+-------------------------------------+
|        v1 <- [1, 3, 5, 7, 9]        |
+-------------------------------------+
```

The set operators are set intersection (*****), set union (**+**), set difference (**-**), set membership (**in**) and set complement (Complement). The set difference operator removes all elements in the second operand from the first and returns the result. For example [1:4] - [3,5] = [1,2,4].

The complement operator makes a set contain all the elements it did not contain previously. Thus if *v1* shown previously = [1,3,5,7,9] then *Complement(v1)* = [2,4,6,8,10]. The complement is always calculated without changing the set bounds, so if *v1* were dimensioned [1:20] and had the value [1,3,5,7,9] then *Complement(v1)* = [2,4,6,8,10:20]. A set expression has bounds that change at run time, so the complement of a set expression produces an indeterminate result. This is because the bounds within which the set bits are complemented are changing. Thus Complement([]) is indeterminate. To avoid problems with the complement of set expressions, the complement function should be used with variables or constants, such as *v1*, *v2* and *C* shown previously.

The Lbound and Ubound functions are used to find the lowest and highest possible elements of a set. For a dynamically sized set these are the bounds most recently defined with Redim, and for a fixed maximum size set these are the bounds defined in the declaration. The Redim function changes the bounds of a resizable set, and Isdynamic tests whether a set is resizable.

When using the set intersection, union or difference operator in an expression, the resulting set will always be large enough to hold both of the operand sets. This process is known as set widening because the result set is widened to allow for the bounds of both operands.

Array Type

The **array ... of ...** type declares an array type which stores one or more components. All components of a given array are the same type. Arrays have a fixed number of dimensions, and when selecting components an integer index is needed for each dimension. An array type and array variable are declared as shown:

```
+-----------------------------------------------------------+
| type Tvector[*:*]    : array [*:*] of component-type      |
|      Tmatrix[*:*,*:*]: array [*:*,*:*] of component-type  |
| var  v1              : Tvector [1:5]                      |
|      v2              : Tvector [*:*]                      |
|      mat             : Tmatrix [1:3,1:4]                  |
+-----------------------------------------------------------+
```

The asterisks in the type names Tvector[*:*] and Tmatrix[*:*,*:*] are part of the names, and must be included whenever the types are used for function parameters or for declaring other types, variables or constants. Each ***:*** is a dimension of the array and represents a pair of integers for the lower and upper bounds.

A resizable array variable can be declared by leaving the asterisk in the declaration (such as *v2* above), or the array variable can be constrained to a fixed size by replacing the asterisks with numeric bounds (such as *v1* above). When used as components of other structure, arrays must always be constrained.

Arrays are initialized using the array constructor. This can use constructor functions or structured expressions. Assume constants C_{ij} ($1 \le i \le 3$ and $1 \le j \le 4$) are declared using the component type of the array. The array variable *mat* shown previously can be initialized with a structured expression as follows:

```
+-------------------------------------+
|  mat <- {{ c11 , c12 , c13 , c14 },  |
|          { c21 , c22 , c23 , c24 },  |
|          { c31 , c32 , c33 , c34 }}  |
+-------------------------------------+
```

Individual components are accessed and updated with the array selector. This is written in the usual way using square brackets:

```
+-------------------------------------+
|              mat[i,j]               |
+-------------------------------------+
```

At run time the array bounds determine the range of array index values that are allowed. If an array index outside the bounds is used, this is known as an array subscript error, array index error, or out of bounds reference, and run time error ERR_ARRAY_REF occurs. The bounds of an array are obtained with the Lbound and Ubound functions. A dynamically sized array has the bounds most recently set with Redim. An array of fixed maximum size has the bounds it was declared with. Each lower bound of an array is always less than or equal to its upper bound, and all array bounds are between MINBOUND and MAXBOUND. The Redim function changes the bounds of a resizable array, and Isdynamic finds whether a given array is resizable.

More complex arrays are normally updated with **for** loops and a selector, to avoid writing large constructors. Some more complex array types are declared below. Type *Tarray1[*:*]* is a one dimensional array, each element being a 10 character string. Type *Tarray2[*:*]* is a one dimensional array, each element being a list of integers:

```
+-----------------------------------------------------------+
|  type Tlist[*] : list[*] of integer (0:1000)              |
|  type Tarray1[*:*] : array[*:*] of string[10]             |
|  type Tarray2[*:*] : array[*:*] of Tlist[10]              |
+-----------------------------------------------------------+
```

List Type

The **list ... of ...** type declares a list which stores zero or more components of the same type. Components are selected using an integer value between one and the number of components. When variables of list type are declared they can be dynamically sized or they can be constrained to a fixed maximum size. Declarations are shown next:

```
+--------------------------------------+
|  type  T[*]  : list[*] of Tc         |
|  var   list1 : T[*]                  |
|        list2 : T[10]                 |
+--------------------------------------+
```

Tc is the component type of the list. To declare dynamically sized list variables the asterisk is left in the type name, as shown by *list1* above. For lists of a fixed maximum size the asterisk is replaced with a numeric upper bound, as shown by *list2* above. The fixed size list *list2* can vary in size up to ten components and is allowed as component type, whereas *list1* is not. However both variables shown are type compatible, because a constraint is not part of a variable' s type.

Lists are initialized using the list constructor. This can use constructor functions or structured expressions. Assume *C1* to *C3* are constants of type *Tc* (shown above). List variables *list1* and *list2* can be initialized using structured expressions as follows:

```
+--------------------------------------+
|   list1 <- {C1, C2, C3}              |
|   list2 <- {C1, C2, C3}              |
+--------------------------------------+
```

After these assignments both lists contain three components, and have a length of three. However the fixed maximum size list *list2* could only fit another seven components, whereas *list1* is dynamically sized and could fit many more.

A list can only have components added or removed at its end. This makes sure external lists are efficient when their components are stored on disk. Listadd appends a new component to the end of a list and Listdel deletes the component at the end of the list. These are the only operators that change the number of components in the list. Other list operators are used for finding the current length of the list (Length), getting a component of the list (Listread), and updating a component of the list (Listwrite).

The Lbound and Ubound operators find the size constraints of a list. The lower bound of any list is 1. The ubound of a fixed maximum size list is the bound it was declared with. The ubound of a dynamically sized list is not a meaningful value as the maximum number of elements depends on the available memory, therefore Ubound returns a negative value when used with a dynamically sized list. The Isdynamic function checks whether a list is dynamically sized.

List Iterator

This is a loop which reads through all the components in a list. The next example loops through *list1* and loads each component into the *buffer* variable. The buffer must be type compatible with the component type of the list:

```
+--------------------------------------+
|  for i from 1 to length(list1)       |
|    call Listread(list1, i, buffer    |
|    buffer <- ...                     |
|    call Listwrite(buffer, i, list1)  |
|  end for                             |
+--------------------------------------+
```

The effect of the **for** loop and the call to *Listread* is to read through all the list components starting from the first. If the **for** loop was intended to update the list, the code in italics shows how to modify the buffer and write back the modified component. When iterating through the list, the index variable (*i*

above) should be between 1 and the length of the list, otherwise error ERR_LIST_REF occurs.

Table Type

The **table ... of ...** type stores zero or more components, similar to an indexed database file. The components are of record type and the table has indexes to order the records.

A table type is declared by specifying its component type which is a record. Then the indexes are declared using the **index** keyword, and are numbered from 1 upwards. The indexes contain fields from the component record. An index with multiple fields is called a segmented index or a composite key. Declarations are shown next:

```
+-----------------------------------------------------+
|  type  Tabrec    : record                           |
|                      f1 : TC1                       |
|                      f2 : TC2                       |
|                      f3 : TC3                       |
|                      f4 : TC4                       |
|                    end record                       |
|  type  Ttable[*] : table[*] of Tabrec               |
|                      index 1 (f1,f2)                |
|                      index 2 (f1,f3)                |
|                    end table                        |
|  var   table1    : Ttable[*]                        |
|        table2    : Ttable[100]                      |
|        buffer    : Tabrec                           |
+-----------------------------------------------------+
```

In this example the table's component type is the record *Tabrec*, and the actual table type is *Ttable[*]*. The table has two segmented indexes, index 1 of fields *f1* and *f2*, and index 2 of fields *f1* and *f3*. Fields *f1*, *f2* and *f3* must exist in *Tabrec* and must be scalar type. Two table variables are declared, *table1* and *table2*. The variable *table1* is dynamically sized because of the asterisk in its declaration. It can store any number of records. The variable *table2* is fixed maximum size and can store up to 100 records.

In database terminology the fields *f1* to *f4* are the table columns and the records of data are the table rows. The number of columns is fixed and the number of rows can vary. Assume we have values *Cij* where i is the row number ($1 \le i \le n$) and j is the column number ($1 \le j \le 4$). A table is represented as follows:

```
        +-----------+-----------+-----------------------------------------------+
        |  Index 1  |  Index 2  |                     Data                      |
        |-----------+-----------+-----------------------------------------------|
        | f1  | f2  | f1  | f3  | Type TC1  | Type TC2  | Type TC3  | Type TC4  |
        |=====+=====*=====+=====*===========+===========+===========+===========|
row 1   | c11 | c12 | c11 | c13 |   c11     |   c12     |   c13     |   c14     |
        |-----+-----+-----+-----+-----------+-----------+-----------+-----------|
row 2   | c21 | c22 | c21 | c23 |   c21     |   c22     |   c23     |   c24     |
        |-----+-----+-----+-----+-----------+-----------+-----------+-----------|
        | ... | ... | ... | ... |   ...     |   ...     |   ...     |   ...     |
        |-----+-----+-----+-----+-----------+-----------+-----------+-----------|
row n   | cn1 | cn2 | cn1 | cn3 |   cn1     |   cn2     |   cn3     |   cn4     |
        +-----------+-----------+-----------------------------------------------+
```

Tables are initialized using the table constructor. This can use constructor functions or structured expressions. Assume *Cij* are values as shown in the previous diagram. The table variable *table1* shown previously is initialized using a structured expression as follows:

```
+-------------------------------------+
| table1 <- {{c11, c12, c13, c14},    |
|            {c21, c22, c23, c24},    |
|             .................       |
|            {cn1, cn2, cn3, cn4}}    |
+-------------------------------------+
```

Individual records in a table can be read and written with a table selector. If we have index key values *key1* and *key2* having types *Tc1* and *Tc2*, we can read and write from the table variable *table1* as follows:

```
+-------------------------------------------------------+
|  call Tabread(table1, 1, {key1,key2}, buffer)         |
|  call Tabwrite(buffer, 1, {key1,key2}, table1)        |
+-------------------------------------------------------+
```

The call to Tabread reads from the table variable *table1*, using index 1, finds the record matching *{key1,key2}* and reads it into the *buffer* variable. The buffer variable must be type compatible with the component type of the table. The call to Tabwrite writes back the *buffer* variable, using index 1 and *{key1,key2}* to find the record in *table1* that is being updated. The key values must identify a record that is in the table, otherwise error ERR_TABLE_REF occurs. Tabread and Tabwrite have an optional *found* parameter that returns FALSE if the record does not exist. This is useful when it is not known whether the record exists.

Note that Tabwrite finds an existing record and updates it, it does not add a new record to the table. The indexed fields in buffer and the values *{key1,key2}* must all locate the same existing record, therefore *buffer.f1* should equal *key1* and *buffer.f2* should equal *key2*. If this condition is not met error ERR_TABLE_UPDATE occurs.

Other table operators add a new record (Tabadd), delete a record (Tabdel), find the number of records in a table (Length), count the records between two index values (Tabcount), search for a specific record (Tabfind) and recreate the table indexes (Reindex).

The Lbound and Ubound functions get the size constraints of a table. The lower bound is always 1. The ubound of a fixed maximum size table is the numeric bound used in its declaration. The ubound of a dynamically sized table is not a meaningful value as the maximum number of elements depends on the available memory, therefore Ubound returns a negative value when used with a dynamically sized table. The Isdynamic function checks whether a table is dynamically sized.

Tables can be updated as the program runs by adding and deleting records, and the indexes are updated automatically. The records in the table can only be accessed by the indexed fields. This is the difference between lists and tables - data in a list is accessed by its position through the list which is an integer, and data in a table is accessed by one or more keys of any scalar type. The order of insertions and deletions is not important, because the table is always ordered according to its indexes.

Table Iterator

The table iterator is used for processing all the records in a table. In the following example the **iterate** loop reads the entire table, loading each record into the *buffer* variable. The record can then be processed and copied back to the table:

```
iterate buffer through table1
  buffer <- process(buffer)
  call Tabwrite(buffer, 1, {buffer.f1,buffer.f2}, table1)
end iterate
```

The *buffer* variable is a record with the same component type as *table1* and the effect of the iterate loop is to set *buffer* equal to each record in *table1* in turn, and to carry out the commands in the loop. The *process* function represents code using the *buffer* variable. Fields in *buffer* may be modified, except that if Tabwrite is called to write the *buffer* variable back to the table, fields that are part of an index must not be modified. If indexed fields are modified, error ERR_TABLE_UPDATE occurs.

There is another form of table iterator which uses Tabfind instead of the iterate loop. Tabfind is called using the FIRST argument to get the first record from the table, then following calls use NEXT to get the remaining records. This process is repeated until there are no more records. For example:

```
call Tabfind("FIRST", table1, 1, buffer, status)
while status /= 0
  // use buffer here but don't change the indexed fields
  call Tabfind("NEXT", table1, 1, buffer, status)
end while
```

The call to Tabfind(FIRST) reads the first record using index 1. If *table1* is not empty *buffer* stores the first record and *status* returns a non-zero value. Fields in *buffer* can be updated, although indexed fields must not be modified if the *buffer* variable is to be written back to the table. The call to Tabfind(NEXT) reads the next record from the table and updates the *status* variable. The **while** loop continues until *status* returns zero meaning there are no more records.

Usually it is simpler to use the iterate loop instead of the Tabfind loop. The performance of the two loops is similar.

2.7 Program Statements

The statements of a language are the actual computations of a program. Statements are used for calculations and for controlling the flow of control through a program. In Ubercode they go between the **code** keyword and the end of a function:

```
+-------------------------------------+
|    function f(...)                  |
|    // declarations ...              |
|    code                             |
|        // statements ...            |
|    end function                     |
+-------------------------------------+
```

The following list shows all the statements in Ubercode:

if statement
select statement
loop statement
while statement
for loop
for each loop
iterate loop
with statement
assignment statement
call statement
run statement

Ubercode uses self bracketing statements which means all statements have keywords that mark where they start and finish. Compound statements, enclosed with "begin" and "end" in Pascal or in curly brackets in C, are not needed.

The semicolon is not required as a terminator symbol at the end of a statement. Semicolons were originally used as statement terminators to make it easier to write the compiler (see [Perelgut 1988] in the Bibliography). However the art of compiler writing has improved, and it is no longer necessary to complicate a language with unnecessary syntax. Multiple statements may be placed on a line, and the end of a line of text has no significance, other than being treated as white space.

If statement

The **if** statement is used to make decisions. It evaluates the expression following it, then does the statements following the **then** if the expression is true, otherwise it does the statements between the **else** and **end if**. Here is a simple if statement:

```
if a then
    statement
end if
```

Simple if/else statement:

```
if a then
    statement
else
    statement
end if
```

Multiple if statement:

```
if a then
    statement  // a is true
elseif b then
    statement  // b is true
end if
```

Multiple if/elseif/else statement:

```
if a then
    statement  // a is true
elseif b then
    statement  // b is true
else
    statement  // a is false and b is false
end if
```

The compound if statement is also known as a *multi way if statement*.

Select statement

The **select** statement chooses one of several options. The program tests the condition following the **select** keyword, jumps to one of the **case** labels or the **else** label, then continues from **end select**. The condition following **select** is called the selector expression and must be enum type, integer type, string type or control type. The syntax is as follows:

```
select expression
    case c1[:c2] [,...] => statement
    else                => statement
end select
```

If the selector expression does not correspond to a **case** option, the statement following **else** is executed. If the selector expression does not correspond to a **case** option and there is no **else** part, the program continues running from after **end select**. The constants *c1* to *c2* are the *case labels* and they must be the same type as *expression*. Case labels are constants, or a range of constants separated by a colon, or several constants (or pairs) separated by commas.

When case labels are separated by commas, the *expression* is tested against each case label. When case labels are separated by colons, the *expression* is tested to see if it falls between the case label values using ' >=' and ' <=' for enum, integer and string expressions. When control objects are separated by colons, the expression is tested by checking if it is in a control array formed by the two control objects.

The **select** statement is equivalent to the case statement or switch statement found in other languages. Note that a break statement is not needed at the end of the code belonging to a case label.

Loop statement

The **loop** statement is used to repeatedly do a statement until a condition is satisfied. The ones below will loop until *expression* is true. Here is a loop that exits from the top:

```
loop exit when expression
    statement
end loop
```

Loop to exit from the bottom:

```
loop
    statement
    exit when expression
end loop
```

Loop to exit from the middle:

```
loop
    statement
    exit when expression
    statement
end loop
```

The first loop is equivalent to **while...do...** of Pascal and **while {...}** of C. There is also a While statement which is described later on. The second loop is equivalent to **repeat...until** and **do...while**. The third loop has no equivalent in Pascal or C, but is nonetheless very useful. For example, consider a list of items to be processed by a loop. The items are processed by:

(1) Initial calculations to prepare the item.

(2) See whether the prepared item is acceptable.

(3) If so, do detailed calculations based on the item.

The loop would be of the form:

```
loop
    prepare_next_item
    exit when item_not_acceptable
    calculations
end loop
```

While statement

The **while** statement is used to repeat a group of statements any number of times.

```
while expression
    statement
end while
```

The *expression* is of boolean type. When the flow of control reaches **while** the boolean expression is tested and if true the *statement* part is executed. When **end while** is reached the loop returns to the **while** part and tests *expression* again. The loop continues running until the expression tests false.

An **exit when** command can be used to force early exit of the while statement. However this is usually unnecessary since the expression following **while** provides control over the loop.

For loop

The **for** loop executes some statements a given number of times, and is used when the number of iterations is known beforehand. Each time around the loop, the loop counter is increased or decreased.

```
for i from start  {   to   } finish
                  { downto }
    statement
end for
```

At the start of the loop, the integer expressions *start* and *finish* are evaluated and the loop counter is given the value of *start*. If *i* is less than or equal to *finish* then *statement* is done, *i* is incremented to the next value and then tested again. The loop continues until *i* is greater than *finish*. A downward counting **for** loop is possible by replacing **to** with **downto**.

The loop counter *i* must be an integer variable and its value must not be changed by statements in the loop because it is incremented and decremented automatically. The *start* and *finish* expressions are evaluated once only at the start of the loop, so if they are changed in the loop this does not affect the number of iterations. The value of the *finish* expression is saved; after the loop *i* will have this value regardless of how many times the loop was executed. Thus the final value of loop counters is always the value of the *finish* expression. For example:

```
var
  s:string[*]
  i:integer(1:1000)
  start:integer(1:1000)
code
  s <- "Hello"
  start <- 1
  for i from start to Length(s)
    s <- s + "."
    start <- start * 2
  end for
end function
```

The loop will be executed five times. After the loop, *s* = "Hello.....", *start* = 32, *length(s)* = 10 and *i* = 5. The reason why the counter has the value 5 at the end of the loop because it always takes the value of the *finish* expression which was calculated as 5 when the loop started.

An **exit when** command can be used to force an early exit. The next example searches the string *s* for an asterisk:

```
s <- "*hello"
ptr <- 0
for i from 1 to Length(s)
  ptr <- i
  exit when Strcopy(s,ptr,1) = "*"
end for
if (ptr > 0) and (Strcopy(s,ptr,1) = "*") then
  // found it!
end if
```

The separate variable *ptr* is needed because *i* will have the value 6 when the loop finishes. Remember that at the start of the loop, the *start* and *finish* expressions are evaluated. Regardless of how the loop comes to an end, *i* always has the value of the *finish* expression after the loop ends.

For each loop

The **for each** loop carries out one or more statements for each element returned by an iterator function. Iterator functions return a value of list or array type. The iteration is safe against code inside the loop that changes the number of returned items or the order of iteration.

```
for each element in iterator([params])
    statement
end for
```

Iterator is the iterator function and *Params* represents its input parameters, if any. *Element* must be

type compatible with the components of the iterated list or array.

At the start of the **for each** loop, the iterator function is called and the returned list or array is stored for the duration of the loop. Then the *element* variable is set equal to the first item in the returned list and the *statement* part of the loop is executed. After *statement* program flow returns to **for each** and *element* is set equal to the second item in the list. This process continues until all items in the list have been iterated. If the iterator returned an empty list *statement* will not be executed.

Because the loop gets the list of iterated items once only at the start of iteration, it is useful when there is a danger of the code in the loop interfering with the next element to be obtained from the list.

Some languages have a collection type which is an unordered group of objects of the same type. A collection is equivalent to a list type used with a **for each** loop. The **for each** loop guarantees that when processing the objects in the collection, it does not interfere with the order of the objects.

Iterate loop

The **iterate** loop processes the records in a table. The loop makes it possible to choose which records are processed, and the order of the processing.

```
iterate recvar through tab
    [where expression]
    [order by (field,...) [asc|desc]]
    statement
end iterate
```

The loop sets *recvar* equal to each record in the table, then runs the *statement* part of the loop. *Recvar* is the table record which must be type compatible with the components of the table variable *tab*. The optional **where** command modifies the loop so it only processes records for which the **where** condition is true. The optional **order by** command determines the order of iteration and must be followed by a list of fields (*field*) that correspond to one of the table' s indexes. The **order by** command may include the **asc** or **desc** commands to process the record in ascending or descending order.

In the *statement* part of the loop the indexed fields of *recvar* should not be changed. This makes sure the records are processed in the correct order, because the indexed fields are used for finding the next record. The *statement* part will not be iterated at all if the **where** condition returns false for all the table records, or if *tab* contains no records.

The following example declares a table, adds some records to it, then uses the **iterate** loop to select some of the records:

```
// Testiter.cls
Ubercode 1 class Testiter

public function main()
type
  Tdata:record
    name:string[20]
    year:integer(1:9999)
  end record
  Ttab[*]:table[*] of Tdata
    index 1 (name,year)
  end table
var
  tab:Ttab[*]
  buffer:Tdata
code
  call Tabadd({"Richard", 1972},tab)
  call Tabadd({"James",   1976},tab)
  call Tabadd({"Ronald",  1980},tab)
  call Tabadd({"Ronald",  1984},tab)
  call Tabadd({"George",  1988},tab)
  call Tabadd({"William", 1992},tab)
  call Tabadd({"William", 1996},tab)
  call Tabadd({"George",  2000},tab)
  call Tabadd({"George",  2004},tab)
  iterate buffer through tab where buffer.name >= "R"
    call Msgbox("Iterate", Str(buffer))
  end iterate
end function

end class
```

The type *Tdata* is the record structure used for the table. Type *Ttab[*]* is the table type declared with a segmented index consisting of the *name* and the *year* fields. The index has to be segmented because the same name may be added to the table twice over (for example there are two Ronalds and two Williams) so we need the extra field to avoid a duplicate record which is not allowed. The records are added to the table variable *tab* by calling the Tabadd function. Finally the **iterate** statement loops through all records in the table, selecting those with a name alphabetically following "R". The values printed out are:

```
+-------------------------------------+
|            Richard  1972            |
|            Ronald   1980            |
|            Ronald   1984            |
|            William  1992            |
|            William  1996            |
+-------------------------------------+
```

If records are added or deleted while iterating a table, this may change the number of records iterated through. Unlike the For each loop, **iterate** does not make a temporary copy of the table before starting because this would be inefficient for a large table. Instead **iterate** does a search using the index and loads up the next record each time around the loop. As with the **for** loop, an **exit when** command can be used to make the loop come to an early end.

With statement

The **with** statement is used with variables of union type and ensures the only fields that are modified are the ones belonging to the current tag value. Also it is a shorthand way of referring to the fields of a union.

```
with union-var
    case 1 => statement
    case 2 => statement
    case 3 => statement
end with
```

The **with** statement has a case label for each tag value of a union. The tag values are defined when the union type is declared. For example the **with** statement shown above is for a union declared with tag values of 1 to 3. When the **with** statement is compiled the compiler checks there is a **case** option for each tag value. Also it checks each **case** option only modifies union fields having the same tag value.

Also the **with** statement allows the fields of *union-var* to be used without being qualified. In the **with** statement, all occurrences of identifiers that match fields of *union-var* are interpreted as union fields even if there is an identifier of matching name.

At run time the **with** statement examines the tag value of *union-var*, then jumps to the **case** option matching the current tag value. This ensures the only union fields modified are those corresponding to the current tag value. The **with** statement does not change the tag value itself, to do this assign other union values or union structured expressions to the entire union variable.

The **with** statement is not allowed to have ranges of case labels (**case** 1:3 etc.). This is because in the union' s declaration each tag value has different fields,so a with statement with a range of labels could potentially modify fields not associated with the current tag value. Also the **with** statement does not allow a **case else** option for the same reason - **case else** would be a catch-all for multiple tag values meaning the program could modify fields not associated with the current tag value. Case ranges and case else both defeat the purpose of a union, which is that the fields and tag value are always modified consistently.

Assignment statement

This statement copies one variable to another of the same type. Two variables are the same type if they were declared with the same type name - see the section on Type Compatibility for more details. The assignment symbol is the left pointing arrow. This symbol was chosen because in languages such as Basic beginners sometimes write:

```
5 = a
```

intending to copy the value 5 to the variable *a*. In Ubercode this is written:

```
a <- 5
```

to make it clearer. However you cannot write 5 -> a. The value being assigned to must be modifiable, so it must be declared as a **var**, **inout** or **out** parameter or a component of one of these. Values declared as **const** or **in** parameters cannot be modified. Assignment is allowed with variables of any type, no matter how complex their structure. For example:

```
type Ttab[*]       : table[*] of ...
                         ...
                     end table
     Tmat[*:*,*:*] : array[*:*,*:*] of real(-1e9:1e9)
var  s1 : string [*]
     s2 : string [5]
     t1 : Ttab [*]
     t2 : Ttab [100]
     m1 : Tmat [1:10,1:10]
     m2 : Tmat [1:10,1:10]
     m3 : Tmat [1:100,1:100]
code
    t1 <- t2     // OK
    m1 <- m2     // OK
```

After an assignment *v1* <- *v2*, *v1* always equals *v2* by definition. When assigning values of dynamic types it is possible for the left hand variable to overflow, for example if it has fixed bounds and the value being assigned is larger. Assignments to dynamic types are dealt with as follows.

Strings, lists and tables of fixed maximum size. The assignment will change the length of the left

hand variable, but it will never change the maximum size (the upper bound), thus the left hand variable must be large enough to fit all the elements from the right hand side. If there are too many elements, one of the run time errors ERR_STRING_FULL, ERR_LIST_FULL or ERR_TABLE_FULL will occur. For example if the fixed maximum size string *s2* shown above is assigned s2 <- "123" the Length is 3, the Ubound is 5 and the assignment is successful. If the string is assigned s2 <- "123456" the Ubound is still 5 and error ERR_STRING_FULL occurs.

Strings, lists and tables that are dynamically sized. The left hand variable will be resized to match the number of elements in the right hand value. After the assignment the lengths of the left hand and right hand values are equal.

Assigning to sets. If the left hand variable is resizable it is resized to match the bounds of the right hand side value. The set bitmap is then copied, after which the left hand variable has the same bounds as the right hand value and the two are equal. If the left hand variable is of fixed size it does not need the same bounds as the right hand value, as long as all the elements of the right hand set fit in the left hand set. If right hand elements do not fit error ERR_SET_COPY occurs. Assignment never changes the bounds of a fixed size set.

Assigning to arrays. If the left hand variable is resizable it is resized to match the bounds of the right hand value. Copying then takes place, after which the left hand variable has the same bounds as the right hand value and the two are equal. If the left hand variable is of fixed size its bounds must equal the bounds of the right hand, otherwise error ERR_ARRAY_COPY occurs. For example if the fixed size arrays *m1* and *m3* shown previously were assigned *m1* <- *m3*, or *m3* <- *m1*, error ERR_ARRAY_COPY occurs because the left hand array in both cases has different bounds and cannot be resized. Assignment never changes the bounds of a fixed size array.

Assignment and type compatibility. The Type Compatibility rules cannot guarantee that two variables can be assigned. For example when assigning variables of the same dynamic type it is possible for there to be an overflow if the left hand variable has a fixed maximum size. In the array example shown previously, the assignment m1 <- m3 caused the run time error ERR_ARRAY_COPY because the bounds of m1 and m3 are different. This is true even though the arrays are the same type and could both be passed to a function as a parameter of type Tmat[*:*,*:*].

The reason is the size of a dynamic variable is not part of its type, so assignment errors may occur if one dynamic variable is too large to be copied to another. This situation occurs in other languages, for example in C, Basic and Pascal if you declare a string large enough to store 5 characters and copy a larger string into it, either an error occurs or the two values are not equal after the assignment. Some versions of Basic avoid an error by only copying the characters that fit, but this means the strings are unequal after assignment.

Shallow copy and Deep copy. In most computer languages structured types are stored using multiple blocks of memory. This may be a conscious decision of the programmer, who has for example decided to store an array of structures in the form of an array of pointers to separately allocated blocks of memory, instead of storing the structures in a single large block. Or this may occur because of the internal workings of the language being used.

When structured values are stored in multiple memory blocks, there are two ways the assignment operator could work. Consider the case of an array of structures as described previously, if we have two variables x and y being assigned x <- y. Variable x is currently empty and y is an array of pointers to separate blocks of memory which contain the structures.

The first assignment method is shallow copy which copies the top level block of memory only. It works by allocating a block of memory for x and copying y's pointers into it. This assignment is valid because the variables are equal after assignment. However if y is modified later on in the program, x will be indirectly modified as well since x's structure pointers point to y's structures. There is also a problem that when y is finished with, we can't immediately free the memory as it may still be in use by x. This problem is normally dealt with by garbage collection which means memory is not reused until all references to it are dead.

The second assignment method is deep copy which copies all the blocks of memory used by the

variable. In the case of x and y we allocate blocks of memory for x and for all the structures, then copy each of y' s structures into the new blocks allocated for x, then set up x' s pointers to point to its own structures. This method of assignment is also valid because the variables are equal after assignment. There is no danger of aliasing or floating pointers because x and y do not share memory, so they can be copied and deallocated independently.

Both these assignment methods have approximately the same performance, as although the deep copy is slower when doing an assignment, it does not need to pause at regular intervals for garbage collection.

Ubercode uses deep copy for all its assignments, to avoid aliasing problems and the need for garbage collection. To compare with other languages, C++ can use either method depending on how the assignment operator is written and Eiffel uses shallow copy. Visual Basic uses deep copy, for example in the case where you declare an array of structures (user defined types).

Call statement

A call statement transfers control to a function. Functions in Ubercode perform the same tasks as functions in C, procedures and functions in Ada and Pascal, and subroutines in Fortran. The called function must be in the same class as its caller, or it must be declared public in an inherited class.

When the function call is reached, the **in** parameters are evaluated. Then if precondition checking is active, the precondition is tested. Program flow then reaches the function, and the function performs its calculations. When the function returns, the postcondition is tested if postcondition checking is active. Finally the program continues with the next statement after the function call. Any **inout** or **out** parameters are available at this point.

Refer to precond and postcond expressions for more information about the effects of precondition and postcondition testing.

All functions must be declared before being called. Functions are declared by **(1)** importing the function from an inherited class or **(2)** using a public function prototype or **(3)** fully declaring a private or public function. If an undeclared function is called, compiler error ERR_UNDECLARED_ID will occur. The reason for declaring functions before calling them is so the compiler can check the function arguments are correct.

Some algorithms use Recursive Functions that call themselves. A single recursive function satisfies the rule that functions are declared before use, because the function heading declares the function and the call occurs after this declaration. In more exotic cases an algorithm may use Mutual Recursion, which occurs when several recursive functions call each other. In this case, one or both functions must be declared using a public prototype.

Function calls can be made by a **call** statement or from an expression. However if the function heading has an **out** parameter, it must be called from an expression because the **out** parameter is the function result. For example:

```
+-----------------------------------------------------------+
| call f(a,b,x)    function f(in i1:integer i2:integer      |
|                            inout o1:real)                 |
|                  code ...                                 |
|                  end function                             |
|-----------------------------------------------------------|
| x <- g(a,b)      function g(in i1:integer i2:integer      |
|                            out o1:real)                   |
|                  code ...                                 |
|                  end function                             |
+-----------------------------------------------------------+
```

Functions normally have Inout parameters or an Out parameter so that data may be returned from the function. There can only be a single **out** parameter, but any number of **inout** parameters are allowed. A function cannot have both. This table summarizes the calling rule:

```
+-----------------------------------------------------------+
|  functions with inout       use a call statement          |
|                             e.g. call f(a,b,x)            |
|-----------------------------------------------------------|
|  functions with out         use an expression             |
|                             e.g. x <- g(a,b)              |
+-----------------------------------------------------------+
```

Run statement

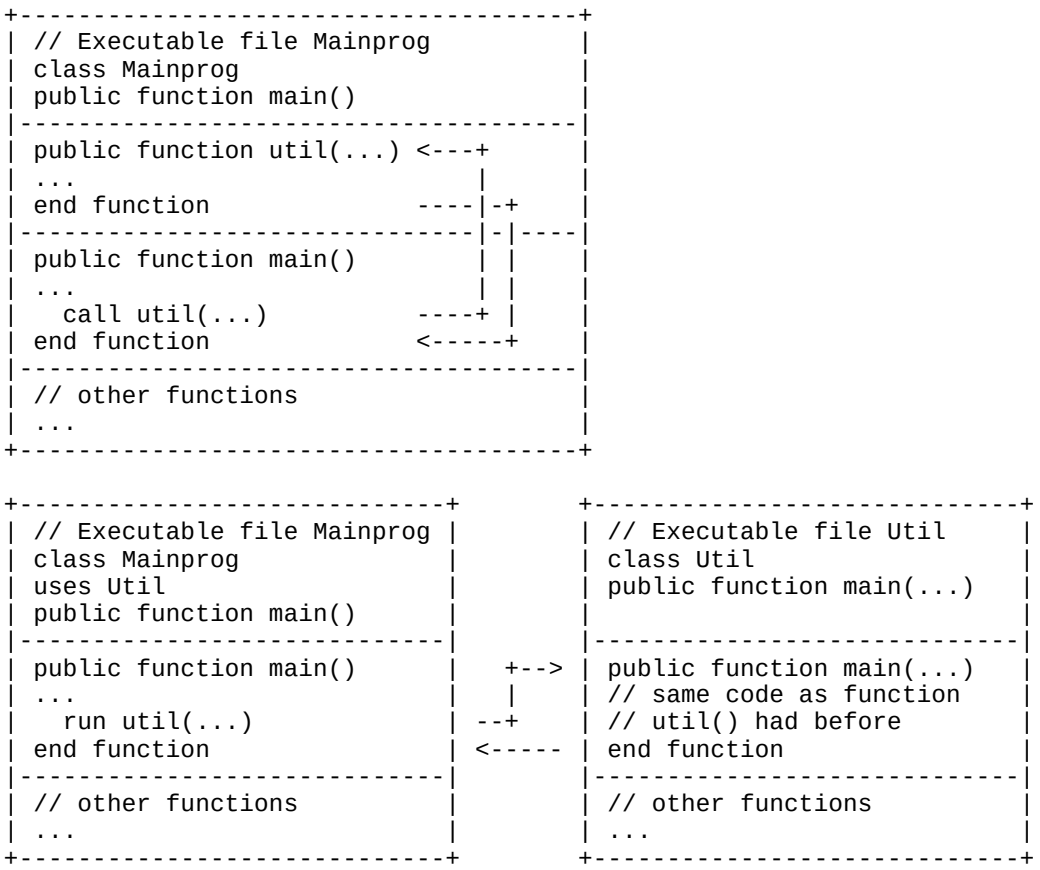
The **run** statement calls up external program files. Control is transferred to the called program, and when this finishes control resumes from after the run statement. The caller remains in memory and the called program only stays in memory while being run. The syntax of **run** is:

```
run name(param)
```

The *name* element is the executable file to run in the form of an identifier without quotes and without ".exe" on the end. The *param* element is an optional string parameter and if present is passed as command line arguments to *name*.

The **run** statement automatically appends ".exe" to the name, then searches for the called executable program. Run searches for the program first in the directory containing the calling application (the directory returned by the Dirstart function), then in the calling application' s current logged directory (as returned by the Dirpath function), then in the Windows system directory (\Windows\System, \Winnt\System32 or \Windows\System32), then in the Windows home directory (\Windows or \Winnt), then in the directories listed in the PATH environment string. This is the same order as used by Exec when called without a fully qualified name.

The run statement is similar to the call statement. A **call** transfers control to a function in the same executable file, whereas **run** transfers control to a separate executable file. These similarities are shown in the next diagram:



In the top part of the diagram the main class Mainprog has a function *main()* calling function *util()* from the same class. At run time both functions are in the same executable file. The second part of the diagram shows how *util()* can be split off into a separately compiled program. A new class Util is created with all the code that was previously in function util() in Mainprog. Also Mainprog is altered to use the statement *run util()* instead of *call util()*. The call and run statements have the same effect, assuming class Util has the same code as util() had previously.

This is similar to code overlays used in earlier languages. It is possible to call up several executable files all occupying the same memory space, as long as each is unloaded before the next one starts.

When running an application the flow of control starts at function Main which is always the first function run. Main is allowed a single string input parameter. When running other executable files the effect is to suspend the calling application, and load and run the executable file. The calling application continues running after the called executable file has completed. A single string may be passed as the command line arguments of the executable file being run.

Labels and goto Statements

Labels and goto statements are not supported, because Ubercode has enough constructs to make them unnecessary. The **for** loop, **for each** loop, **iterate** loop and **while** loop can be exited prematurely using the **exit when** condition, making it unnecessary to jump out of a loop. Multiple function return statements or jumps to the end of a function are not needed to deal with errors occurring within the function, because the precondition mechanism can be used to ensure the function has valid inputs. Jumps to error handlers are not needed because the error handler is automatically called when needed.

2.8 Expressions and Operators

An expression combines values with operators to form new values. The values being combined are

operands and can be variables, constants, functions or other expressions. The operators always expect a certain number of operands of particular types. If an expression consists of constants only, it is called a constant expression and wherever possible is evaluated at compile time. Structured expressions consist of curly brackets enclosing zero or more normal expressions separated by commas, and are used for initializing values of structured types. Structured expressions are discussed in more detail later on.

Operator precedence

When an expression has more than one operator, the order of calculation is determined by the rules of operator precedence (also known as binding). Operators with the highest precedence are calculated first. Operators with equal precedence are calculated from the left to the right. The following table shows the precedence of the operators, with the highest priority at the top:

Description	Operators
Unary operators	not
Multiplying operators	* / mod div
Adding operators	+ -
Relational operators	= /= > < >= <= in
Logical operators	and or

Round brackets (parentheses) take priority over all the operators. This is useful for controlling the order of calculation. For example *a/(b+c)* is calculated as *b+c* first, then *a* is divided by the result. Without the brackets, *a/b+c* is calculated as *a/b* and the result is added to *c*.

Unary operators

The unary operators include just the **not** operator:

Operator	Operation	Operand types	Return type
not	logical negation	boolean	boolean

The **not** operator reverses a boolean value. Numeric unary negation is carried out with the Neg function.

Multiplying operators

The multiplying operators include multiplication, division, set intersection, and the **div** and **mod** operators:

Operator	Operation	Operand types	Return type
*	multiplication	integer, integer	integer
		fixed, fixed	fixed
		real, real	real
*	set intersection	set[*:*],set[*:*]	set[*:*]
/	division	integer, integer	real
		fixed, fixed	real
		real, real	real
mod	modulus	integer, integer	integer
div	integer division	integer, integer	integer

If only one of the operands of ***** or **/** is of real type the other will be converted to real type. Similarly, if only one operand is of fixed type, the other will be converted to fixed type. The division operator **/**

always returns real type, and run time error ERR_NUM_DIVZERO occurs if an attempt is made to divide by zero.

The set intersection operator ***** returns a set containing only those elements that are in both of its operand sets. The resulting set has bounds corresponding to the overlapping portion of the operands.

The integer division operator **div** divides the first operand by the second. If the result of the division is not a whole number, it is rounded down to the next lowest integer. The value *n div d* is the same as *Int(n/d)*, the only advantage to using div is the division takes place as an integer operation, whereas *Int(n/d)* requires a function call and a real number division which is slower. The results of div are shown next:

```
+4 div +3 = +1  // round +4/3 down to +1
+4 div -3 = -2  // round -4/3 down to -2
-4 div +3 = -2  // round -4/3 down to -2
-4 div -3 = +1  // round +4/3 down to +1
```

The integer remainder operator or modulus operator gives the integer remainder after an integer division takes place. If *n* below is the numerator (the number being divided) and *d* the non-zero integer divisor, **mod** is defined as:

```
n mod d = n - (n div d) * d
```

Also (n div d) equals int(n/d) so we also have:

```
n mod d = n - int(n/d) * d
```

The next tables show the result of div and mod with positive and negative numerators and divisors (n and d). The result of mod has the same sign as the divisor, and the result of div has the same sign as the mathematical quotient (*n/d* expressed as a real number).

n	d	n div d	int(n/d)	n mod d
6	3	2	2	0
5	3	1	1	2
4	3	1	1	1
3	3	1	1	0
2	3	0	0	2
1	3	0	0	1
0	3	0	0	0
-1	3	-1	-1	2
-2	3	-1	-1	1
-3	3	-1	-1	0
-4	3	-2	-2	2
-5	3	-2	-2	1
-6	3	-2	-2	0

n	d	n div d	int(n/d)	n mod d
6	-3	-2	-2	0
5	-3	-2	-2	-1
4	-3	-2	-2	-2
3	-3	-1	-1	0
2	-3	-1	-1	-1
1	-3	-1	-1	-2
0	-3	0	0	0
-1	-3	0	0	-1
-2	-3	0	0	-2
-3	-3	1	1	0
-4	-3	1	1	-1
-5	-3	1	1	-2
-6	-3	2	2	0

Adding operators

The adding operators are:

Operator	Operation	Operand types	Return type
+	addition	integer, integer	integer
		fixed, fixed	fixed
		real, real	real
+	string concatenation	string, string	string
+	set union	set[*:*], set[*:*]	set[*:*]
-	subtraction	integer, integer	integer
		fixed, fixed	fixed
		real, real	real
-	set difference	set[*:*], set[*:*]	set[*:*]

If one of the operands of addition or subtraction is of real type, the other operand is converted to real type. Similarly if one of the operands is fixed type, the other is converted to fixed type.

The string concatenation operator joins two strings together.

The set union operator returns a set containing all elements from both operand sets. The set difference operator returns its first operand after removing all elements that were in the second operand. Set widening is used with the set union, difference and intersection operators, to make sure the resulting set contains all applicable elements from the operand sets. With these operators the result set is widened to hold a range of elements from the lowest bound of the operands to the highest

bound.

The following function shows some set operations:

```
function SetOperations()
type
  Tnumset[*:*]:set[*:*]
var
  small    : Tnumset[1:10]
  big      : Tnumset[10:1000]
  UnionSet : Tnumset[1:1000]
  DiffSet  : Tnumset[1:1000]
code
  small <- [1, 10]
  big <- [10, 100, 1000]
  UnionSet <- small + big
  // now UnionSet = [1,10,100,1000]
  DiffSet <- small - big
  // now DiffSet = [1]
  small <- small + big
  // causes run time error because resulting set is 1:1000
end function
```

Relational operators

The relational operators include the comparison operators and the **in** operator for set membership. All relational operators return boolean values:

Operator	Operation	Operand types	Return type
=	equal	any, any	boolean
/=	not equal	any, any	boolean
>	greater than	any scalar, any scalar	boolean
<	less than	any scalar, any scalar	boolean
>=	greater or equal	any scalar, any scalar	boolean
<=	less or equal	any scalar, any scalar	boolean
in	set membership	integer, set[*:*]	boolean

The equals operator **=** and not equals operator **/=** can be used with any two values of the same type. Integers are automatically converted to fixed or real type, and fixed type is converted to real type when needed, as discussed later under automatic type conversions. Be careful when comparing two real numbers for equality **=**, because rounding errors may cause the numbers to be extremely close but not the same.

The other four relational operators **<**, **>**, **<=**, **>=** can be used with any two values of the same scalar type. Automatic conversions of integer and fixed types are done where necessary.

The ordering of enum types is defined by their declarations starting at 0, each constant having a value one less than the following constant. Boolean types are ordered by False having the value 0 and True the value 1.

Strings are ordered according to the ASCII character set. The maximum length of the string (the ubound) does not matter because any two variables of string type are type compatible with each other. The string comparison is done by comparing pairs of characters from each string, and the string with the lower ASCII value is the lesser. If the strings are equal when the end of one of them has been reached, the shorter string is the lesser.

The **in** operator tests whether the integer operand is in the set operand, and returns **true** if it is.

Logical operators

The logical operators **and** and **or** have two boolean operands and return a boolean value:

Operator	Operation	Operand types	Return type
and	logical and	boolean, boolean	boolean
or	logical or	boolean, boolean	boolean

The logical operators are evaluated from left to right, and evaluation stops as soon as the result of the expression is known. This is called short circuit evaluation or jumping code and the next example shows why this is useful. The **while** loop searches an array *a* containing *max* elements for an element equal to *target*:

```
i <- 1
while (i <= max) and (a[i] /= target)
  i <- i + 1
end while
```

If *target* is not in the array and the search is unsuccessful, *i=max+1* when the **while** condition is tested for the last time. Short circuit evaluation ensures when the test *i<=max* fails we leave the loop without testing the second part of the expression, *a[i]/=target*. Without this strict left to right evaluation, *a[i]* would be tested using an index of *max+1* and the program would fail with an array index error at run time.

Automatic type conversions

These occur when values of one type are automatically converted to another under the Type Compatibility rules. They are a useful addition to the rules because they extend in a safe way the situations under which two types are compatible. Automatic type conversions are also known as *type promotions* or *type coercions*. They occur as follows:

(1) When you declare a type identifier which renames or inherits from the predefined boolean, integer, fixed, real, string or set type, values of the predefined type are automatically converted into the inheriting type where required. This conversion allows literals of the predefined types to be used with the inheriting type, and enables code such as the following:

```
type Tlogical:boolean
     Ttext[*]:string[*]
var  x:Tlogical
     y:Ttext[*]
code x <- True
     y <- "Hello"
```

(2) Integer type is automatically converted to fixed point type where required. For example if a fixed point variable *f* and an integer variable *i* are declared:

```
f <- 100     // conversion of integer to fixed
f <- i       // conversion of integer to fixed
```

(3) Integer and fixed point types are automatically converted to real number type where required. For example if we have variables *r*, *f* and *i* of type real, fixed and integer:

```
r <- 100     // conversion of integer to real
r <- i       // conversion of integer to real
r <- 10.25   // conversion of fixed to real
r <- f       // conversion of fixed to real
```

A more interesting example of automatic type conversion is:

```
r <- i + f + r
```

In this example *i+f* is the first addition so the integer is converted to fixed point type and a fixed point addition occurs which results in a fixed point temporary value. The next addition is to add the temporary value to *r*, so the temporary value is converted to real type then a real number addition occurs. Finally the real number result is copied to the variable *r*. Such expressions are also called *mixed type expressions* because the type changes through the expression.

Type demotions

Type demotions occur when real is converted down to fixed or fixed to integer type. Demotions cannot be done automatically because they lose numeric accuracy, so the Type Conversion Functions described later must be used instead. The following example shows type demotions - none of these are allowed:

```
i <- 1.25   // error - can't convert fixed point to integer
i <- f      // error - can't convert fixed point to integer
f <- 1e32   // error - can't convert real to fixed point
f <- r      // error - can't convert real to fixed point
i <- 1e32   // error - can't convert real to integer
i <- r      // error - can't convert real to integer
```

Numeric conversions

Numeric type conversions are automatic type conversions of integer to fixed point to real type that occur when needed. Numeric type conversions take place in expressions, but not in structured expressions (values between curly brackets). For example if *i*, *f* and *r* are integer, fixed point and real type variables, the following are allowed:

```
f <- i
r <- i
r <- f
```

As described under Type demotions it is not possible to convert back the other way, because accuracy is lost. Instead Int and Fix can be used as Type conversion functions as follows:

```
i <- Int(f)
i <- Int(r)
f <- Fix(r)
```

The term rounding is used when Int and Fix are used for converting back from real to fixed to integer type, because accuracy is lost during conversion.

Type Conversion functions

Type conversions are for converting values of one type to another. The Type Compatibility rules describe the automatic type conversions that occur. In other cases you can use conversion functions to explicitly convert one type to another. In some cases data is lost when using the conversion functions. For example when converting a real number to an integer, the fraction is lost. For this reason, type conversion functions are not automatic and must be called explicitly. The following conversion functions are available:

Function	Operation	Input type	Result type
Chr	Convert to string of length 1	integer	string
CvtB	Convert from xbase type	Tlogical	boolean
CvtI	Convert from xbase type	Tnumeric	integer
CvtF	Convert to fixed point	integer	fixed
CvtR	Convert to real	integer	real
		fixed	real
Enumval	Convert to enum type	integer	enum
Fix	Convert to fixed point	real	fixed
Frac	Find fraction	fixed	fixed
		real	real
Hexstr	Convert to hex string	integer	string
Hexval	Convert hex string to integer	string	integer
Int	Round down to integer	fixed	integer
		real	integer
Ord	Find ordinal value	boolean	integer
		enum	integer
		integer	integer
		string	integer
Str	Convert to string	any	string
Val	Convert string to other type	string	any

Chr function.
Chr converts an integer into the corresponding character from the ASCII character set. For example, Chr(65) = "A". The string returned by Chr always has a length of 1.

CvtB, CvtI functions.
These functions convert a data type stored in an xbase file back to normal data types. CvtB converts the logical type Tlogical to boolean, and CvtI converts the numeric type Tnumeric back to integer.

CvtF, CvtR functions.
CvtF converts integer to fixed point type, and CvtR converts integer and fixed type to real type. These functions are not often used since automatic type conversions do the same thing. They are used in situations where you want to force a value to be treated as a fixed point or a real number, for example when calling a polymorphic function that requires one of these types.

A good example of the use of CvtF is when calling the Delay function, which pauses for a number of seconds as given by an argument of fixed type. The call Delay(1) would cause a compile time error as Delay is not overloaded to accept an integer parameter. The call Delay(CvtF(1)) would compile successfully as the integer is converted to fixed point which is accepted as a parameter of the Delay function.

Enumval function.
This function converts an integer into an enumerated type. When using the function you need to make sure the integer has an equivalent enumerated value. Enumerated values start at zero and increase by one for each enumerated constant, so an enumerated type with 7 different constants has values from zero to 6. For example Tdays shown below has 7 enumerated constants (mon to sun) which have values from 0 to 6. Therefore if you called Enumval to make a value of *Tdays* type, use an integer between 0 and 6.

```
type Tdays:enum(mon,tue,wed,thu,fri,sat,sun)
```

Enumval and Ord are related so if you use Ord to get the integer value of an enumerated value, Enumval will convert the integer back to the same enumerated value. For example if *enumvar* shown next is an enumerated value, the following is always true:

```
Enumval(Ord(enumvar)) = enumvar
```

Fix function
The Fix function converts values of type **real** to fixed point. A run time error will occur if the real value

is not in the range of values allowed for **fixed** type. Fix is convenient for using floating point notation with large values. These two lines are equivalent:

```
f <- 1000000000000.00
f <- Fix(1e12)
```

Do not use Fix to convert integers to fixed type. If you did, the integer would be automatically converted to a real first, then Fix would convert the real to a fixed point number! This is unnecessary as the automatic type conversions change integer to fixed type where needed.

Frac function
The Frac function works with real and fixed type, and returns just the fractional part of a value. This has the same sign as the original value and *0≤Frac(v)<1.0*. For example:

```
Frac(1.5) = 0.5
Frac(1.0) = 0.0
Frac(0.5) = 0.5
Frac(0.0) = 0.0
Frac(-0.5) = -0.5
Frac(-1.0) = 0.0
```

Hexstr function
The Hexstr function converts an integer value into the equivalent hexadecimal string. The result of Hexstr will be a string between *0* and *FFFFFFFF*. For example:

```
Hexstr(0) = "0"
Hexstr(255) = "FF"
Hexstr(65535) = "FFFF"
```

Hexval function
The Hexval function converts a string in hexadecimal format into the equivalent integer value. Hexadecimal strings optionally begin with "0x", for example "0xFF" and "FF" both denote the hexadecimal string with the value 255. For example:

```
call Hexval("0", i)      // i = 0
call Hexval("FF", i)     // i = 255
call Hexval("0xFF", i)   // i = 255
call Hexval("0xFFFF", i) // i = 65535
```

Hexval is the inverse of Hexstr, so for any integer value *i* the following is true:

```
Hexval(Hexstr(i)) = i
```

Int function
This function converts values of type real and fixed to integer. It does so by rounding downwards to the next lowest integer, unless the value already is an integer. In the latter case it is unaltered. For example:

```
Int(1.9) = 1
Int(1.1) = 1
Int(1.0) = 1
Int(0.9) = 0
Int(0.1) = 0
Int(0.0) = 0
Int(-0.1) = -1
Int(-0.9) = -1
Int(-1.0) = -1
```

Any fractional part of the number being converted will be lost. Also Int will not work if the resulting integer value is too large or too small (greater than MAXINT or less than -MAXINT).

Ord function
This function converts a value of boolean type, enum type, or the first character of a string into an

integer. Values of boolean type are defined so that Ord(False) = 0 and Ord(True) = 1. Values of enumerated type are defined by the order of their constants, starting at zero. The value of a string is the ASCII value of the first character in the string. For example:

```
// Assume a type Tdays:enum(mon,tue,wed,thu,fri,sat,sun)
Ord(False) = 0
Ord(True) = 1
Ord(mon) = 0
Ord(sun) = 6
Ord(" ") = 32
```

Str function
This function converts scalar types into strings. For example:

```
Str(false) = "False"
Str(true) = "True"
Str(10) = "10"
Str(10*0.25) = "2.5"
Str(1.0e9) = "+1.0E+0009"
Str("Mystring") = "Mystring"
```

An optional format specifier is allowed which specifies the field width, the number of decimal places and whether left or right justification is used:

```
+-----------------------------------------------------------+
|                 ±  field-width . decimal-places           |
|                 ^         ^              ^                |
|                 |         |              |                |
| + or no sign indicates    |     Number of decimal places |
| right justification.      |     for fixed types and real |
| - for left justification. |      types only. This is the |
|                           |             number of digits |
| Specifies field width     |                following the |
| for the result string. ---+               decimal point. |
| This is overridden if                                     |
| a wider result string                                     |
| is needed.                                                |
+-----------------------------------------------------------+
```

Here are some boolean and integer conversions:

```
// left justify, field width of 6
Str(False,-6) = "False "

// right justify, field width of 6
Str(False,6) = " False"

// result widened to 2 spaces
Str(10,-1) = "10"

// left justify, field width of 3
Str(10,-3) = "10 "
```

The next examples show Str used with fixed point and real numbers. With fixed point numbers you can specify the number of decimal places, or specify the field width on its own which causes the fixed point value to be converted using the minimum number of decimals. Real numbers are always in exponential format with a minimum field width of 10, for example *-1.2E-1000*. If a larger field width is used the number is left or right justified within the larger width. The number of decimal places specifies how many digits to the right of the decimal point are shown, subject to a minimum of 1.

```
// right justify fixed point, field width of 6
Str(2.5,6) = "   2.5"

// left justify, field width of 6
Str(-2.5,-6.2) = "-2.50 "

// result widened to accommodate 3 decimal places
Str(-2.5,-4.3) = "-2.500"

// widened to 10 spaces, the minimum width for a real number
Str(-1.25e6,1) = "-1.3E+0006"

// field width of 10, fits exactly
Str(-1.25e6,10) = "-1.3E+0006"

// field width of 11, using default of 1 decimal place
Str(-1.25e6,11) = " -1.3E+0006"

// field width of 20, using default of 1 decimal place
Str(-1.25e6,20) = "          -1.3E+0006"

// result widened to accommodate 4 decimal places
Str(+1.25e6,1.4) = "+1.2500E+0006"
```

Some string conversions are shown next. The second and third examples use a field width greater than the length of the string. The result string is padded with spaces on the left or right hand side, making this similar to Lset and Rset found in Basic.

```
// display widened to 5 spaces
Str("Hello",1) = "Hello"

// right justify, field width of 6
Str("Hello",6) = " Hello"

// left justify, field width of 6
Str("Hello",-6) = "Hello "
```

Val function
The Val function takes a string input parameter and converts it into other types. Val returns the converted value and an optional flag which indicates whether the conversion was successful or not. For example:

```
call Val("false",boolval,status)       // boolval = False, status = True
call Val("999",intval,status)          // intval = 999, status = True
call Val("+1.0E+0009",realval,status)  // realval = 1e9, status = True
```

The boolean status flag returns True in the examples above because the syntax of the string being converted is correct. A string should be in the same form as a literal of the type it is being converted into. If you are sure the string will always be correct you can omit the optional status flag, but if the string is not correct run time error ERR_CONV_BOOLEAN (or other conversion error) will occur. Therefore it is best to use the status flag unless you are sure the input is correct.

In many respects Val is the inverse of the Str function, as Str converts any type into a string and Val converts a string back into the original type. This is not always possible since real numbers may have rounding errors because some binary values cannot be represented exactly with a string. Also the reconversion of a fixed point or real number string loses accuracy if the format specifier used with Str lost some of the accuracy of the original variable. Subject to these limitations, a variable *v* of scalar type may be converted to a string and back to the original value again as follows:

```
Val(Str(v)) = v
```

Structured Expressions

A structured expression consists of values separated by commas and enclosed in curly brackets. The

values can be constants, variables, parameters, or other expressions. Structured expressions are useful for initializing structured types because an entire variable can be initialized at once. For example an array variable *a* can be initialized with three string values as follows:

```
a <- {"Hello", Str(i), "every"+"body"}
```

The next example shows structured expressions used to initialize an array and a record variable:

```
function test()
type
  Tvector[*:*]:array[*:*] of integer(0:MAXINT)
type
  Tperson:record
    name:string[20]
    year:integer(1:9999)
  end record
var
  v:Tvector[1:5]
  p:Tperson
code
  v <- {1,2,3,4,5}
  p <- {"James", 1976}
end function
```

First an array type *Tvector[*:*]* and a record type *Tperson* are declared, then variables *v* and *p* of these types are declared. The variables are initialized at run time with two structured expressions. The number and type of values used in a structured expression must match the type being initialized. The type of the structured expression is determined from its context as follows:

Assignment statement. When structured expressions are used as the right hand side of an assignment, they have the type of the variable being assigned to.

When a structured expression is assigned to an array of fixed maximum size there must be an initializer for each element in the array, otherwise run time error ERR_ARRAY_REF occurs. When assigned to a resizable array the array keeps its lower bound at the original value and changes its upper bound to match the number of initializers. The array must be a one-dimensional array and there must be at least one initializer. Because the lower bound remains unchanged, a structured expression can initialize a resizable array regardless of the lower bound. For example you could use Redim to give the array lower and upper bounds of zero, then assign to the array with a structured expression, then the lower bound would still be zero and the upper bound will change to match the initializers.

When a structured expression is assigned to a list or table variable, each element in the expression is added as a component of the variable. After the assignment, the length of the list or table equals the number of initializers in the structured expression. If the structured expression has too many initializers to fit in a list or table of fixed maximum size, run time errors ERR_LIST_FULL or ERR_TABLE_FULL will occur. These errors will not occur when assigning to a dynamically sized list or table because the variable grows to fit the number of initializers.

Function arguments. In this context the structured expression is treated as **record** type. This is because Ubercode uses polymorphic functions which mean the function being called is determined by its signature - therefore we need the types of the parameters to get the signature to find the function.

Empty lists and tables. Empty structured expressions are type compatible with lists and tables. A useful way of removing all items from a list or table variable is by assigning it an empty structured expression. If *v* is a list or table variable, it will have a length of zero after the following assignment:

```
v <- {}
```

Numeric conversions within structured expressions. Numeric conversions do not occur within structured expressions. This is because structured expressions can be used for initializing generic types, and components of the generic type have unknown types and there is no way of knowing when numeric conversions should be applied. Although numeric conversions could (in theory) be enabled

for structured expressions used with non-generic (user declared type) this would be confusing, since some structured expressions would use numeric conversions and others would not.

Therefore to be consistent, the values within a structured expression must use the exact type required.

2.9 Error Handling

When a program is running, unexpected situations may occur that prevent the program working normally. These events are called *run time errors* or *exceptions* and they cause the program to display an error message and (in most cases) halt. Errors are caused by a lack of resources such as memory or disk space, operating system errors, numeric errors, files not being in the correct format, failure of software verification tests, or failures of algorithms. There is a complete list in the Run Time Errors section.

When an error occurs there are several alternatives: fix the problem, halt the program or continue running. The term error handling or exception handling describes the actions taken when an error occurs. Different applications will take different corrective actions. A numerical application will prefer to terminate the program to find the error so correct results can be ensured, whereas a real-time application will prefer to keep running.

Default error handling

At each point in a class where an error could occur, the compiler inserts checks in the code to make sure the operation succeeded. If the operation failed the default error handler is called. This is function Errorhandler in the run time library, and its source code is shown next:

```
+--------------------------------------------------------------------+
|  function Errorhandler(in errno:integer)                           |
|  code                                                              |
|    select errno                                                    |
|    case ERR_RECOVER_FIRST:ERR_RECOVER_LAST =>                      |
|         call Sound("")                                             |
|         if Msgbox("Error " + Str(errno), Errormessage(),           |
|                   "Continue"+NL+"Halt") /= "Continue" then         |
|           call Halt()                                             |
|         end if                                                     |
|    else =>                                                         |
|         call Sound("")                                             |
|         call Msgbox("Error " + Str(errno), Errormessage())         |
|         call Halt()                                                |
|    end select                                                      |
|  end function                                                      |
+--------------------------------------------------------------------+
```

This function is called automatically whenever an error occurs, and the *errno* parameter is one of the error numbers listed in the Run Time Errors section of the manual. The first **case** option checks whether *errno* is a recoverable error. Recoverable errors are errors that do not cause loss of data or threaten program stability. If the error is recoverable, a message box is shown containing two buttons, "Continue" and "Halt". Pressing "Continue" causes the error handler to return normally and allows the program to continue running. Any other response in the message box calls Halt which ends the program.

If *errno* is non-recoverable (the **else** option) a message box is again used to display the error details. The Errormessage function returns a descriptive string relating to the error, and may include extra text describing the line number and class where the error occurred. This message box does not have a "Continue" button, and when closed Halt is called and the program ends.

The error handler therefore allows the program to continue running after recoverable errors, and halts the program when more serious errors occur. Error handling can be modified. Either the default Errorhandler in the run time library *System* class can be altered, or an Errorhandler can be added to a class to handle errors occurring in just that class. Modified error handlers are described next.

Intercepting errors

Reliable software must continue working in the event of errors. This is called robustness (see [Meyer 1988 p4] in the Bibliography). To achieve this write your own error handler to override the default one. This is possible because of the scope rules. You declare a function Errorhandler at private scope with the same parameters as the default handler, then any errors in the class (except operating system errors) are intercepted by your error handler instead of the default. Your handler can do the following:

- Ignore the error and just return, which continues running the program. This is advisable only for a recoverable error.
- Call other functions to fix the error. Be careful when doing this, however. Some errors are due to low resources, such as the stack or heap running out of memory, or disks and directories becoming full. Processing cannot reliably continue because function calls and string allocation may fail. The recommended action in the case of these errors (which include ERR_IO_DISKFULL, ERR_MEMORY and ERR_STACK_SPACE) is to display the error details then halt. Also system errors cannot be handled.
- Call the Halt function to end the program. This is recommended for all errors, except for recoverable errors where it is safe to continue.

As an example, consider a class *Print* which copies some text to the printer. Function *Errorhandler* replaces the default error handler and *Printtext* does the printing.

```
// Print.cls
Ubercode 1 class Print

function Errorhandler(in errno:integer)
code
  select errno
  case ERR_PRINTER_INIT =>
       call Sound("")
       call Msgbox("Print", "Cannot initialize printer")
  case ERR_PRINTER =>
       call Sound("")
       call Msgbox("Print", "Print job canceled or no disk space")
  case ERR_RECOVER_FIRST:ERR_RECOVER_LAST =>
       call Sound("")
       if Msgbox("Error " + Str(errno), Errormessage(),
                 "Continue"+NL+"Halt") /= "Continue" then
         call Halt()
       end if
  else =>
       call Sound("")
       call Msgbox("Error " + Str(errno), Errormessage())
       call Halt()
  end select
end function

function Printtext(in text:string[*])
var count : integer(0:MAXINT)
code
  call Startprint(Sysprinter)
  for count from 1 to Strcount(text)
    call Drawtext(Sysprinter, Strline(text,count))
  end for
  call Endprint(Sysprinter)
end function

end class
```

The error handler processes the ERR_PRINTER_INIT and ERR_PRINTER errors, and includes the rest of the code from the default error handler to handle other recoverable and non-recoverable errors. All error handlers must handle the full range of errors, since errors do not propagate between handlers. Therefore the blocks of code starting with the lines *case ERR_RECOVER_FIRST :*

ERR_RECOVER_LAST => and *else* => are required and must not be left out.

The purpose of the error handler is to display a helpful message whenever specific errors occur in the *Print* class. ERR_PRINTER_INIT occurs if no printer had been set up for the computer, and ERR_PRINTER occurs if the print job was canceled while spooling or if the computer ran out of disk space while writing the spool file. Both these errors are possible while printing.

Without the error handler, errors in *Print* would use the run time library error handler in the *System* class. This would show a generic error message, and allow the program to continue if a recoverable error occurred or terminate the program otherwise. Therefore by using an error handler, a class is able to control its own errors. This makes it easier to write classes that can be used in other programs, and leads to more reliable software.

Scope of error handlers

The default error handler is function Errorhandler in the run time library, and is in scope to all classes. When writing an error handler local to a class, it must be declared at the start of the class (following any prototypes), and it must be declared using private scope. This means it must use the **private** keyword, or no keyword at all because **private** is the default. Error handlers therefore follow the same scope rules as other functions.

The following example shows three classes using different error handlers. The non main class *Menu* is shown first:

```
Ubercode 1 class Menu

private function X()
code
  errors ... ====> calls default handler in System.cls
end function

// other functions...

end class
```

Class *Util* is another non main class:

```
Ubercode 1 class Util

function Errorhandler(in errno:integer)
code                        ^
  // error handler...       |
end function                |
                            |
private function Y()        | call the
code                        | local
  errors ...  ==============* handler
end function

// other functions...

end class
```

Class *Myprog* is a main class which inherits *Menu* and *Util*:

```
Ubercode 1 class Myprog
uses Menu Util

public function main()
code
  errors ...  ===> calls default handler in System.cls
end function

// other functions...

end class
```

The main program *Myprog* contains all three classes when compiled. If *Myprog* has a run time error, the error handler that takes control depends on the class causing the error. Errors in class *Menu* use the default handler in the run time library, as there is no other handler in scope. Errors in class *Util* use the local handler which has private scope. The local handler must be declared at the start of the class, immediately following any prototypes, so it is in scope through the entire class. Errors in *Myprog* use the default handler from the run time library which is in the System class.

It should be noted this example was deliberately made complex to show the full flexibility of error handling. Error handling can be kept simple by following the guidelines in the next section.

Error Handling made simple

Error handling can be made very simple by following these suggestions:

(1) Use the default handler for most applications. The default handler is function Errorhandler in the run time library (the *System* class). Nothing special needs to be done to follow this recommendation, since it is the default behaviour.

(2) For detecting specific problems, such as the Printtext example shown previously, put the functions that might cause an error into a separate class. Declare a private error handler and put this at the start of the class, immediately following any prototypes. The custom error handler then handles all errors occurring in the class.

The custom handler must handle the full range of errors, since errors do not propagate between handlers. Copy the Errorhandler function from the *System* class and use it as a template. Make sure it is declared **private** and include the handled errors immediately after the *select errno* command. Refer to intercepting errors for an example.

(3) For improving the overall reliability of an application, you can modify the default error handler in the system class so that instead of displaying the error and halting, it continues running. This gives the user a chance to save data and gives the application a chance to shut down gracefully. The modified error handler will be the default for the entire application.

System errors

The term *System error* refers to an error detected by the Windows operating system. These errors (errors ERR_SYS_ACCESS to ERR_SYS_OTHER) are not sent to the error handler and cannot be handled in the normal way. Also they cannot be debugged with the integrated debugger. When system errors occur, program data and the stack area may be corrupt, which prevents the program running normally.

System errors should not normally occur. They are caused by undetected bugs in the run time library, or bugs in system software such as printer drivers and graphics drivers, or bugs in the operating system. System errors are used only to guard against unexpected problems that might cause the computer to lock up, or might cause problems for other applications.

When system errors occur, they are displayed in a message box which includes the error number, the description of the error, and the line number and class details if available. Pressing OK in the message

box will end the program.

2.10 File Input and Output

File input and output is used to read and write data from disk, and for working with the computer's files, directories and disks. Ubercode supports file input and output using external variables which are stored on disk, and by providing file and directory functions suitable for common programming tasks:

- Read and write text files.
- Read and write binary files, by declaring external variables that map the data.
- Read and write data base files using the xbase file format or binary.
- Read and write data base files using the XML file format.
- Read and write bitmap files to and from visual objects.
- Store large structures on disk to reduce memory requirements.
- Store program initialization data that is kept between different runs.
- Create and delete files and directories.
- Copy and rename files.

External variables

External variables store data in disk files. They work by declaring variables of list type or table type and using the **external** keyword in the declaration. The variable works like a normal list or table type variable, except its data is stored on disk instead of in memory. Assuming type *T[*]* shown next is a list or table type, external variables are declared as follows:

```
var x:T[*] (external)
var y:T[*] (external <- name)
```

In the example *x* is stored on disk and is automatically deleted when it falls out of scope. Variable *y* is stored in a file called *name* which is not deleted by the program. The *name* element is a fixed string value such as a string constant or an **in** parameter. *Name* must also be a fully qualified name (including a full directory path starting from the root directory). Variable *y* is known as a *persistent object* because its lifetime extends beyond that of the program.

The **external** keyword is in round brackets because it is a storage constraint and not part of the data type. External variables are type compatible with normal variables of the same type, and the same operators can be used. The assignment operator can be used to copy between external variables and memory variables of the same type. Although the operators are optimized for external variables, operations on disk are slower than equivalent operations in memory. Programs should be designed not to loop through external variables more than is necessary.

As explained *name* must be a fully qualified file name. This allows the program to find the file if the working directory changes, and makes it easier to detect aliasing which occurs if different external variables use the same file. Sharing the same file is not recommended as it may cause the file size to change unexpectedly. Also the file must not be write protected, otherwise run time error ERR_IO_READPERM or ERR_IO_WRITEPERM occurs when the external variable is first opened. This applies even if the program does not actually modify the external variable, and is because the file is opened in read / write mode. Therefore external variables cannot be used on read-only media such as a CD, and functions such as Loadfile must be used instead.

Fully Qualified Name

A fully qualified name (or well formed path) is the full path and name of a disk file. All printable characters (" " to "~") are allowed in the path, except for the following six characters:

```
*   ?   "   <   >   |
```

which are prohibited under Windows. Also the following characters:

```
 :  /  \
```

are used as separators between directory names, not as part of an actual name. Paths may contain spaces, but should not begin or end with a space. Either the backslash or forward slash delimiters may be used to separate the directory names. For example:

```
+-----------------------------------------------------------+
| File name            Means                                |
|-----------------------------------------------------------|
| c:\tmp               File called "tmp" in root directory  |
| \tmp                 As above                             |
| /usr/docs/mydoc.txt  File called "mydoc.txt"              |
+-----------------------------------------------------------+
```

The Filechk function checks a string is a fully qualified name. It returns 1 if it is, 0 if the string denotes a wild card and -1 if it is illegal. You can make up fully qualified names as follows:

- Get an existing directory path using Dirstart or Dirpath and join a filename on the end.
- Create a directory path by concatenating FileGetdrive with FileGetdir, and join a filename on the end.
- Get a directory path and join a filename from the Filelist function on the end. Filelist is passed a pattern string, and returns the matching file names.
- Make up a string and use Filechk to check it is legal.

File and Directory Functions

The following file functions and directory functions are available:

Diradd adds a new subdirectory.

Dirchg changes the logged directory. The new directory path must exist.

Dirchk checks a directory (folder) path is valid. A legal directory path is a chain of directories, starting from the root and ending with a slash or backslash.

Dirdel deletes a subdirectory which must be empty. The root directory cannot be deleted.

Dirlabel returns the volume label and serial number of a disk.

Dirpath returns the current directory path. This will always be a legal directory path and will always exist.

Dirsize detects whether a specified directory exists, and if it does it also finds the free space in the directory. Under most operating systems the free space in a directory is the same as the free space on the disk drive.

Dirstart returns the directory that contains an application' s executable image. This is the start directory, in the sense that it is where the executable image is loaded from.

Filechk checks a fully qualified name to make sure it is legal. A legal name is a chain of directories starting at the root, and ending with a wild card notation or a single file. The directories do not necessarily exist, and you can use the Filesize function to see whether the file exists or not.

Filecopy makes a copy of a file. This works with binary files and text files.

Filedel deletes one or more files.

FileGetdir returns the directory part of a filename (everything following the drive letter, up to and including the final slash or backslash).

FileGetdrive returns the drive letter part of a filename (the drive letter and the following colon character).

FileGetext returns the extension part of a filename (everything between the final full-stop character and the end of the filename, including the full-stop).

FileGetfname returns the name part of a filename. This is the part of the filename following the directory or drive letter, excluding the extension.

Fileinfo returns information about the structure of an xbase file or XML file. The information can be used for reading and writing the file.

Filelist is given a fully qualified name which must specify a wild card. It returns a list of all folders (directories) or files matching this pattern.

Fileren is used to rename a file.

Filesize detects whether a specified file exists, and if it does it also returns the size in bytes of the file. If the size is zero the file exists but is empty.

Filetime returns the date and time of the last modification of a file. The information is returned in the same format as the Time function.

Filextn changes the file extension in a file name string.

Loadfile loads data from a disk file into a program variable. It can be used with a text file, CSV file, binary file, xbase file or XML file.

Savefile saves a program variable into a disk file. It uses the same file types as Loadfile.

Winhelp displays a Windows help file in the Help file format.

Files in Ubercode

Ubercode doesn' t use file data types, file pointers or streams, because the external list type and table type provide equivalent functionality to the binary and text files of other languages. The similarities between lists in Ubercode and files in C and Pascal are shown below. All three routines open a binary file, write ten strings "Henry 1" to "Henry 10" to the file, then close it and leave it on disk:

Ubercode

```
| Function TenNames()                                       |
| const name    : string[*] <-"Henry"                       |
| type Tlist[*] : list[*] of string[20]                     |
| var i    :integer (1:10)                                  |
|     Tfile:Tlist[*] (external<-"\x.dat")                   |
| code                                                      |
|   for i from 1 to 10                                      |
|     call Listadd(name+" "+Str(i), Tfile)                  |
|   end for                                                 |
| end function                                              |
```

Pascal

```
| Procedure TenNames;                                       |
| const name = 'Henry';                                     |
| type Tname = string[20];                                  |
| var Tfile       : file of Tname;                          |
|     i           : integer;                                |
|     tmp, buffer : Tname;                                  |
| begin                                                     |
|   assign(Tfile, '\x.dat');                                |
|   rewrite(Tfile);                                         |
|   for i := 1 to 10 do                                     |
|   begin                                                   |
|     str(i, tmp);                                          |
|     buffer := name + ' ' + tmp;                           |
|     write(Tfile, buffer);                                 |
|   end;                                                    |
|   close(Tfile);                                           |
| end;                                                      |
```

C

```
| void TenNames(void)                                       |
| {                                                         |
|   const char name = "Henry";                              |
|   FILE * Tfile;                                           |
|   int    i;                                               |
|   char   buffer[20];                                      |
|   Tfile = fopen("\x.dat", "wb");                          |
|   for (i=0; i<10; i++)                                    |
|   {                                                       |
|     sprintf(buffer,"%s %d", name,i+1);                    |
|     fwrite(&buffer, sizeof(buffer), 1, Tfile);            |
|   }                                                       |
|   fclose(Tfile);                                          |
| }                                                         |
```

The next three routines save a string to a text file and leave it on disk.

```
Ubercode
+-----------------------------------------------------------+
| Function WriteText()                                      |
| code                                                      |
|   call Savefile("\x.txt", FILE_TEXT,                      |
|        "The quick brown fox"      + NL +                  |
|        "jumps over the lazy dog." + NL)                   |
| end function                                              |
+-----------------------------------------------------------+

Pascal
+-----------------------------------------------------------+
| Procedure WriteText;                                      |
| var Tfile : text;                                         |
| begin                                                     |
|   assign(Tfile, '\x.txt');                                |
|   rewrite(Tfile);                                         |
|   writeln(Tfile, 'The quick brown fox');                  |
|   writeln(Tfile, 'jumps over the lazy dog.');             |
|   close(Tfile);                                           |
| end;                                                      |
+-----------------------------------------------------------+

C
+-----------------------------------------------------------+
| void WriteText(void)                                      |
| {                                                         |
|   FILE * Tfile;                                           |
|   Tfile = fopen("\x.txt", "wt");                          |
|   fputs("The quick brown fox\n"                           |
|         "jumps over the lazy dog.\n", Tfile);             |
|   fclose(Tfile);                                          |
| }                                                         |
+-----------------------------------------------------------+
```

Importing Data Files

Existing files of a known record size can be imported, which allows loading data files from other applications. The only requirement is that you know the size in bytes of the fields and their types. Use the section on Internal Data Formats to find the data types that match the record fields. For fields that have no direct equivalent, use an array of enum type, because enums are stored in one byte. These techniques allow matching any fixed record size file. Refer to Importing Xbase Files and Importing XML Files for specific information for these file types.

After finding the field layout, declare a record to match the external file. Its size must match the records in the external file, otherwise run time error ERR_IO_BADSIZE will occur. Then declare an external list or an external table using this record.

The next example shows an external file "people.dat". The records have a fixed size, and each record has a field for an identifying number, a name, an address and a credit limit.

```
"people.dat"
(File can have any number of records)
+---------------------------------------- - - -+
|      |      |      | one  |      |      |      |
|      |      |      |record|      |      |      |
|      |      |      |  in  |      |      |      |
|      |      |      | file |      |      |      |
+---------------------------------------- - - -+

record size = 72 bytes
+-------------------------------------------------+
| i/d no  | name       | address    | credit limit |
| long    | 20         | 40         | floating     |
| integer | characters | characters | point        |
+-------------------------------------------------+
```

The matching record is as follows:

```
type Tchar    : enum (ch0, ch1, ... ch255)
     Talpha   : array[*:*] of Tchar
     Tdata    : record
                  IdNo    : integer(0:MAXINT)
                  Name    : Talpha[1:20]
                  Address : Talpha[1:40]
                  CrLimit : real(0:1e6)
                end record
     Tfile[*] : list[*] of Tdata
var  People   : Tfile[*] (external <- "\data\people.dat")
```

Note the use of the pseudo string type *Talpha*. An enum type *Tchar* is declared which has 256 different values, one for each ASCII character. A one dimensional array type is then based on this type. This process is needed because when a string is stored in a file, extra bytes are saved at the start for the current length. Therefore a string can' t map onto a character array in a binary file. A variable of type *Talpha* is converted to string type with the following function:

```
function Str(in astr:Talpha[*:*] out s:string[*])
var i:integer(0:MAXINT)
code
  s <- ""
  for i from Lbound(astr) to Ubound(astr)
    s <- s + Chr(Ord(astr[i]))
  end for
end function
```

Reading and writing text files is done using Loadfile and Savefile. Loadfile loads a file from disk into a variable, and Savefile saves the variable back to disk. The following example shows how to read a text file from disk and to write it back out to a different file:

```
call Loadfile("autoexec.bat", FILE_TEXT, memvar)
...
call Savefile("autoexec.bak", FILE_TEXT, memvar)
```

The call to Loadfile reads the "autoexec.bat" file into the *memvar* variable. This variable may be a string, an array of strings or a list of strings. After calling Loadfile you can change the text in the variable, and process it any way you want. Finally the call to Savefile saves the text to the "autoexec.bak" file. This technique using Loadfile and Savefile works with any text file, the only restriction being the maximum size of the string or the maximum number of elements in the list or array.

Importing Xbase Files

An xbase file may also be used by a program. You have to know the structure of the xbase file records, then you must declare a record that matches the file (this can be done automatically as described later). Fileinfo can be used to get the structure of the xbase file record. For example if an

imaginary xbase file "people.dbf" has four fields, *Name* of type CHAR[20], *Achievement* of type CHAR[20], *Dob* of type CHAR[10], and *Status* of type NUMBER[10], Fileinfo would return the following record structure:

```
Tpeople : record
  delete_flag : Tchar
  name : Tchars[1:20]
  achievemen : Tchars[1:20]
  dob : Tchars[1:10]
  status : Tnumeric[1:10]
end record
```

Tpeople is the name of the record type, *delete_flag* is an extra field that occurs at the start of every xbase file record, and the other four fields store the xbase data. The *Tchars* and *Tnumeric* types are part of the run time library. The *Achievement* field has been renamed *Achievemen* to allow for the maximum length of an xbase field name. The record structure could be used in a class as follows:

```
// People.cls
Ubercode 1 class People

type Tchar:Enum(ASCII)

public type TpeopleRec : record
  delete_flag : Tchar
  name : Tchars[1:20]
  achievemen : Tchars[1:20]
  dob : Tchars[1:10]
  status : Tnumeric[1:10]
end record

public type tpeople[*]:table[*] of TpeopleRec
  index 1(name)
end table

end class
```

The class is called *People* to match the xbase file. The xbase file record has been renamed to *TpeopleRec* and *Tpeople[*]* is a table type which allows indexed access. This is more efficient when dealing with large xbase files. To use the xbase file, an external variable is declared using *Tpeople[*]* type. Then the usual table functions such as Tabadd, Tabdel and the iterate loop may be used.

This process can be automated using the Developer environment. Open a main class, then use Program_Add Database File to add the database and a new wrapper class to the program. The new non main class is automatically included by the main class (program). The main class is not altered, apart from having its uses clause extended to inherit the new class.

The section of the manual titled *DBF file type* has more details of the xbase file format.

Importing XML Files

An XML file may also be used by a program. The file structure must be known, then a record and table matching this structure must be declared. Fileinfo can be used to get the XML file structure, as long as the file contains at least one record. Fileinfo works by parsing the XML file, and returning the structure of the elements it finds in the file. For example, the following XML file contains two records:

```
<customers>
  <customer>
    <idno>101</idno>
    <name>Ferdinand Fawcette</name>
    <address>1 High St, Anytown</address>
    <crlimit>100.0</crlimit>
  </customer>
  <customer>
    <idno>102</idno>
    <name>John Doe</name>
    <address>3 the Mansions, Puddletown</address>
    <crlimit>50.0</crlimit>
  </customer>
</customers>
```

This XML file is suitable for importing, since it has a database-style structure consisting of repeated records <*customer*> ... *</customer>*, and each record has the same fields in the same order. Fileinfo returns the following structure:

```
customers : list
  customer : record
    idno : integer
    name : string[18]
    address : string[26]
    crlimit : fixed
  end record
end list
```

This shows that *customers* is a list, *customer* is the record structure of the list elements, and the record has four fields as shown above. This structure can be used in a class as follows:

```
// Customers.cls
Ubercode 1 class Customers

public type TcustomerRec : record
  idno : integer(0:MAXINT)
  name : string[18]
  address : string[26]
  crlimit : fixed(0:MAXFIXED)
end record

public type Tcustomer[*]:table[*] of TcustomerRec
  index 1(idno)
end table

end class
```

The class is called *Customers* to match the file. The *customer* record has been renamed to *TcustomerRec* and the table has been renamed to *Tcustomer[*]*. This follows the convention of using *T* as the first letter of types, and gives the record and table types the same base name. The *Tcustomer[*]* table can be used for indexed access to the XML file by using Loadfile to populate it with the XML data. Then the usual table functions such as Tabadd, Tabdel and the iterate loop may be used. If any table operators modify the table, Savefile should be called to save the table back to the XML file.

This process can be automated using the Developer environment. Open a main class, then use Program_Add Database File to add the database and a new wrapper class to the program. The new non main class is automatically included by the main class (program). The main class is not altered, apart from having its uses clause extended to inherit the new class.

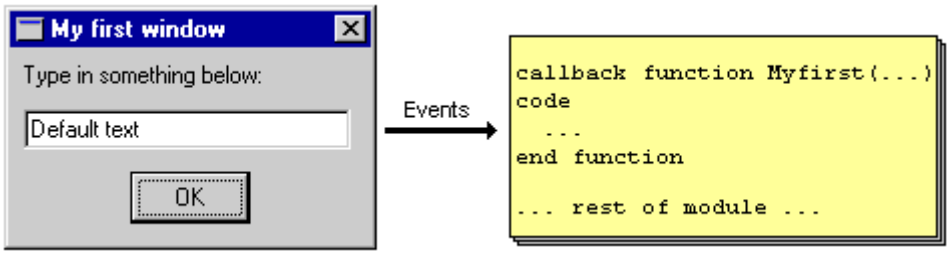
The section of the manual titled *XML file type* has more details of the XML file format.

2.11 Windows and Menus

The Ubercode Graphical User Interface (GUI) is based on Common User Access (see [IBM GUIDE 1991, IBM AIDR 1991] in the Bibliography) as modified by Microsoft Windows. This defines a standard user interface using windows for data entry and menus for navigating through an application.

How Windows work

A window (also called a form or dialog box) is an area on the screen for displaying information. A window usually contain control objects for user interaction. When adding a window to a program you design the window's *appearance* using a visual editor and you write *code* in the form of a function to handle events in the window. When the window is active, it sends events to the window function. The following diagram shows how these are related:



The diagram shows a window titled "My first window" which is included in a class. The class also contains the function *Myfirst* as shown. After the program is compiled and run, and the window is visible, code in *Myfirst* runs in response to user actions in the window. This process is now explained in more detail.

Window appearance

This is designed with the Dialog Editor or an external dialog editor such as the Dlgedit program provided by Microsoft. These visual editors allow you to add controls to the window and to set their properties using a graphical editor. After designing the window it is saved in a resource file.

The window shown above uses the following properties. Most of the properties can be left at their defaults, and the table only shows properties that are set to non-defaults. An important property is the *name* which is needed whenever you refer to a window or a control from code.

```
+-----------------------------------------------------------+
| Caption     = "My first window"                           |
| Name        = Myfirst                                     |
| Position    = 100,100,120,60                              |
| Typeof      = Dialog                                      |
+-----------------------------------------------------------+
+-----------------------------------------------------------+
| Alignment   = 0                                           |
| Caption     = "Type in something below:"                  |
| Name        = Label1                                      |
| Position    = 4,4,110,14                                  |
| Typeof      = Label                                       |
+-----------------------------------------------------------+
+-----------------------------------------------------------+
| Alignment   = 0                                           |
| Borderstyle = 1                                           |
| Name        = Edit1                                       |
| Position    = 4,18,110,14                                 |
| Typeof      = Edit                                        |
+-----------------------------------------------------------+
+-----------------------------------------------------------+
| Caption     = "OK"                                        |
| Default     = 1                                           |
| Name        = Button1                                     |
| Position    = 40,38,40,16                                 |
| Typeof      = Pushbutton                                  |
+-----------------------------------------------------------+
```

The *Dialog* object is the window which contains three control objects. The *Label* object is the text in the window which is called a label because it describes or labels other controls. The *Edit* object is the rectangular area for typing in text. The *Pushbutton* object is the OK button which is pressed to close the window.

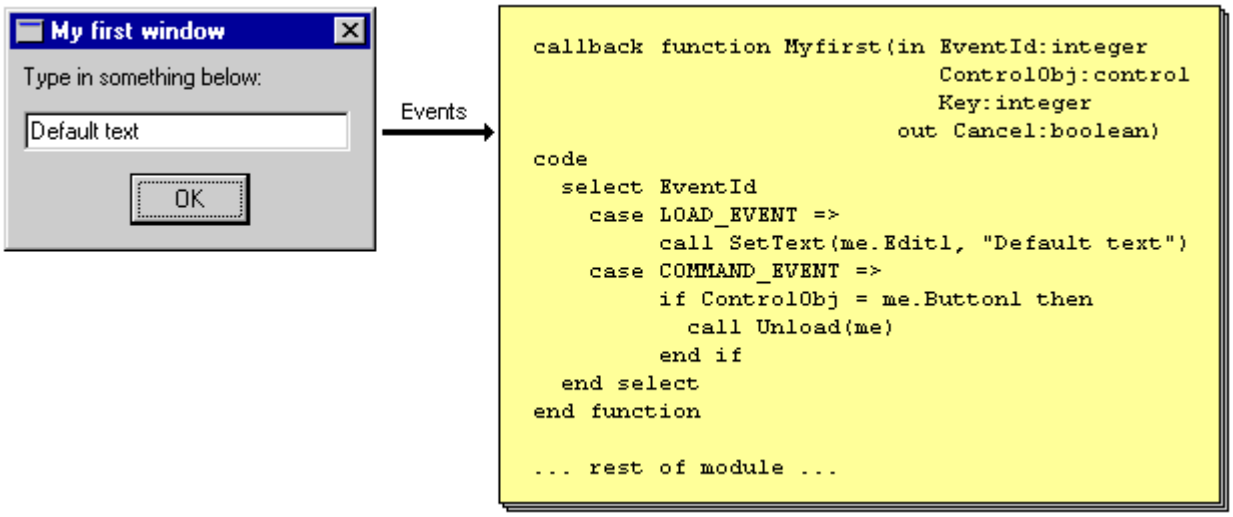
After choosing the properties and designing the window its layout is saved in the resource file. This is a text file with the ".rc" file extension. Normally you don' t need to look at the resource file because it is generated automatically. However you may want to modify resource files from other languages or edit a resource file by hand. Here is the resource file containing the properties just defined:

```
// Windemo.rc
#define Label1  101
#define Edit1   102
#define Button1 103
Myfirst DIALOG 100, 100, 120, 60
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
CAPTION "My first window"
BEGIN
  LTEXT "Type in something below:",Label1,4,4,110,14
  EDITTEXT Edit1,4,18,110,14,WS_TABSTOP
  DEFPUSHBUTTON "OK",Button1,40,38,40,16,WS_TABSTOP
END
```

The Resource statements in the file use Microsoft syntax (see [Microsoft SDK 1992] in the Bibliography).

Window code

The other important part of a window is its window function. This processes graphical events that occur in the window. A window function is similar to a normal function, but it must use the *callback* keyword, it must have the same name as the window, and must have the parameters shown next. The following diagram shows the window function for the *Myfirst* window:



When the user operates the window at run time, events are triggered which are sent to the window function. Events correspond to changes in the controls, pressing buttons, clicking the mouse and pressing keys. Function *Myfirst* above processes the load event and the command event. You can run code or call other functions in response to an event, or you are free to ignore it.

Putting it all together

The window and its event function can easily be combined into a program. The window properties are in "windemo.rc" shown previously. The event function *Myfirst* is part of the main class "Windemo.cls" shown next:

```
// Windemo.cls
Ubercode 1 class Windemo

callback function Myfirst(in EventId:integer
                          ControlObj:control
                          Key:integer
                        out Cancel:boolean)
code
  select EventId
  case LOAD_EVENT =>
       call SetText(me.Edit1, "Default text")
  case COMMAND_EVENT =>
       if ControlObj = me.Button1 then
         call Unload(me)
       end if
  end select
end function

public function main()
code
  call Show(Myfirst, 1)
end function

end class
```

This works as follows. Windemo is a main class which contains window *Myfirst* and its event function. The class starts running at function *main* which calls Show to activate the window. The value of 1 used with Show means Myfirst is a modal window (modal and modeless windows are explained later).

When Show is called, it loads window Myfirst into memory before making it visible. This triggers the Load event which is sent to the window function. The function sets the text in the edit field *(call SetText...)* after which Show resumes control. Show makes the window visible and makes it active

(ready for use).

In this example, the command event is triggered when the user clicks the OK button. When the window function receives the Command event, it checks which button was pressed, in case the window has more than one button. If the OK button was pressed, the window is unloaded which removes it from the screen and from memory. After the window is unloaded the call *Show(Myfirst,1)* returns, function *main* returns and the program ends.

To sum up, all windows use this technique. At design time the Dialog editor sets the properties which are saved in the resource file. The code for the window function is written as part of a class. After compiling and running the program, the Show command makes the window active. When the window is operated it triggers events which are sent to the window function. Eventually the window is closed, and when there are no more windows the program ends.

Visual objects and their properties

The term *object* in Ubercode refers either to a visual part of a program such as a window or control (Visual objects), or a program element which represents part of an application (Program objects). All objects have a name, a type, a hidden internal structure, methods for changing the object' s internal state, properties for reading or writing a single attribute of the object, and the ability to inherit from other objects.

Visual Objects. When designing a program' s graphical interface we are concerned with visual objects because these represent the windows and controls. A visual object is a user interface element provided by Microsoft Windows, such as the Dialog object, Label object, Edit object and Pushbutton object shown in the previous example. All visual objects store values that are read and written with Properties, and have an internal state that is changed using Methods. Visual objects also generate Events when they are used at run time. Refer to the Object list for a complete list of visual objects, their properties, methods and events.

When programming with visual objects the *name* is used to refer to it from code. All visual objects have a name which might be the name of a window, a control, a predefined system object such as the printer, or *me* which refers to the containing window. Visual objects use the following notation:

Notation	Meaning
Dlg	Refers to a window named *Dlg*
Dlg.ctl	Refers to a control named *ctl* in *Dlg*
me	Refers to current window function
me.ctl	Refers to a control named *ctl* in current window function
Sysprinter	Refers to the default printer
Sysclipboard	Refers to the system clipboard
Sysscreen	Refers to the system screen

Program Objects. These represent real-life data and actions being modeled by your application. They are represented by public functions and types in a class. The public functions are the properties and methods of the object, and the function code is the internal behavior of the object. Program objects can be declared as variables in other classes, passed to functions, used as components of other objects (aggregation) and saved to disk. Program objects are discussed in more detail in the Abstract Data Types section of the manual.

Properties. Properties are named attributes of an object that can be read and in most cases modified while the program runs. Design Time Properties are stored in the program' s resource file and are set before compiling the program. All properties are read and modified with the syntax *GetProperty* and *SetProperty*. A property is always read with a function call and is modified with a subroutine call. An example of getting and setting a read/write property of an object *Obj* is:

```
value = Get Property(Obj)
call Set Property(Obj,value)
```

Visual objects are provided "ready-made" and have properties available for most programming tasks. Program objects are written by you as the program develops, so if you want to be able to modify the object' s properties you need to include Get and Set functions when programming the object.

Although properties are used for getting and setting data in both types of object there is an important difference in the lifetime of data stored in the two objects. Data in a visual object exists for the lifetime of the window containing the object. Therefore if a window is loaded in one function and unloaded in another, the data will exist between the two points. In the case of a program object the data exists for the lifetime of the local variable that originally declared the object.

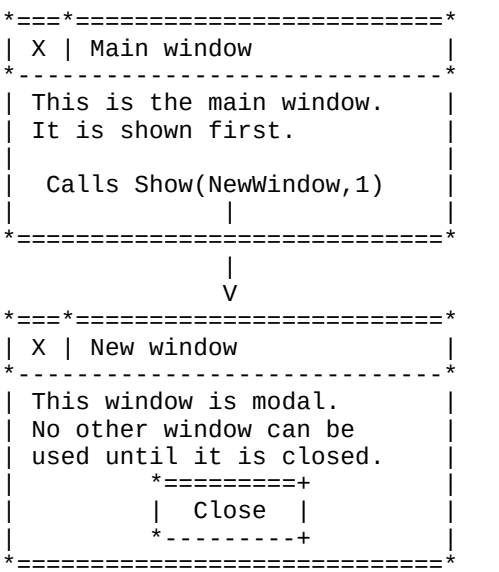
Methods. Methods are functions that operate on the internal state of an object. They carry out a significant amount of processing upon its internal state and may change the appearance of a visual object. Methods are always invoked with a call statement because they are represented by public functions.

Modal and Modeless Windows

An application with more than one window needs to decide if its windows are modal or modeless. A modal window disables all other windows in the application while it is active, thus only one modal window can be used at a time. This is useful where an application needs to get some information before continuing, as the user has to enter the data then close the window before continuing. A modeless window does not disable any other windows and any number of modeless windows can be used at one time. This is useful when an application allows the user to work in several windows at once.

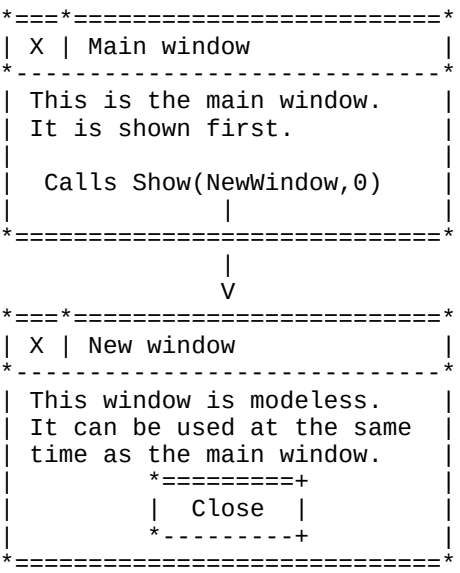
Whether a window is modal or not is determined by the call to Show that displayed the window. The call *Show(Dlg,1)* displays Dlg as a modal window and *Show(Dlg,0)* displays a modeless window. A window' s modal or modeless state cannot be changed while the window is actually visible. To test whether a window is modal or not, use the Ismodal function.

The next diagram shows a modal window. An application starts by showing a main window, which is the window at the top of the diagram. The main window then calls *Show(NewWindow,1)* which shows the new window as a modal window. After the new window appears, the main window is locked out until the new window is closed or hidden. When this happens the new window disappears from the screen and the main window can be used again.



The next diagram shows a modeless window. The application starts by displaying a main window which calls *Show(NewWindow,0)* to show the new window as a modeless window. After calling Show the new window becomes the active window. Both windows can be activated by clicking with the

mouse, and events in each window are sent to the corresponding window function.



When an application has several windows on display, the following rules explain how Show works when you want to display different windows. The purpose of these rules is to make sure modal windows work properly, and that when a modal window is active all the other application windows are disabled.

(1) If a modeless window is already on display, then Show(Modeless) can be used to reactivate it. Show(Modeless) is equivalent to clicking the window with the mouse. You cannot use Show(Modal) to reactivate the modeless window as a modal window. Instead you must Unload or Hide the window before showing it modally.

(2) A modal window may show another modal window but not a modeless window. If a modal window could show modeless windows, it would not be modal.

(3) A disabled window cannot be activated with Show. Although in theory Show could first enable the window, then activate and show it, this does not happen because the window may have been disabled by a different modal window. If Show was allowed to enable other windows, then calling Show from a modal window on one of the previous windows would stop the window being modal.

Main window and Main function

All programs start running from function *main*, which can be an ordinary function or a window function. Function *main* is declared using public scope in a main class. All applications consist of a main class and any number of non-main classes. Two examples of a main class are shown next. In the first example *main* is a normal function and in the second example *main* is a window function. Here is the first example:

```
Ubercode 1 class MainClass
function main()

public function main() <---------------------- Execution starts here.
code
  // This is the first line of code run
  ...
end function           ----------------------> Execution ends here when
                                               all the application windows
end class                                      are unloaded.
```

In this first example, main is an ordinary (non-window) function. The program starts at function main. Main is called a *startup function* because it contains the code that runs first. The code in main may

show modal and modeless windows which operate as usual. Calls to Show(modeless) return immediately after display and calls to Show(modal) do not return until the modal window is hidden or closed. When execution reaches the end of main, all windows stay loaded until the user closes them or they are unloaded by code (the Unload function). The program does not end until all application windows have been unloaded.

In the second example, main is a window function. There must also be a window called *main* designed with the dialog editor, although only the class is shown below:

```
                                          Execution starts here
                                          as the application always
                                          shows the main window first
Ubercode 1 class MainClass                              |
                                                        |
public callback function Main(in EventId:integer <----+
                                 ControlObj:control
                                 Key:integer
                              out Cancel:boolean)
code
  select EventId
  case LOAD_EVENT =>
       // This is the first line of code run
       ...
  case UNLOAD_EVENT =>  ------------------> This is the last event
       ...                                  fired, unless the main
  end select                                window opened other
end function                                windows that have not
                                            yet been closed.
end class
```

In this example, *main* is a public window function which is automatically loaded and shown when the program starts. Main is known as a *startup window* because it contains the code that runs first. The first line of code is the Load event in the window function.

After loading and showing the main window, code in the window function runs in response to events. For example double clicking the window runs code under the Dblclick event and changing the text in an Edit object runs code under the Change event. The main window is able to show further modeless and modal windows which operate as usual. The program continues running until all windows are unloaded, then the application terminates.

By allowing main to be a window function or a normal function, it is possible to use a mixture of event driven programming and procedural programming styles. Code that runs in response to user input is put under the appropriate event in the window function, and more complex code that runs without interaction is put in separate functions.

Since the program does not end until all windows are unloaded, it is possible to create a zombie process that cannot be ended if windows hide themselves but don't unload. This is undesirable because the program can't end while windows are still loaded in memory, and you can't use the program because its windows are hidden. The only way of ending the program is to use the operating system to kill the process.

To avoid this, keep track of any windows that have been hidden and unload them when the program's main window is closed. A useful technique is for a modal window to hide itself when it is closed, making its properties available when the call to Show that showed the window returned. If you use this technique of hiding windows instead of unloading them, remember to unload the window after getting the properties.

How the Owner property works

An application may consist of a top level window that manages several modeless windows displaying similar information. This is known as an MDI application (multiple document interface) and the Owner property is used for the top level window. An application with a top level owner window works as

follows:

(1) When the top level window is minimized, all the other windows in the application are automatically hidden. This is useful when running several applications at once, because when you want to switch to another application you only need to minimize the top level window. You don' t need to minimize all the modeless windows as well.

(2) The top level window is displayed behind all the other windows. This prevents it getting in the way when you switch between the modeless windows.

(3) When the top level window is closed, the other windows are automatically unloaded. This makes it easier when the application ends, because the user only need close the top level window. If the modeless windows need to save information before being unloaded, they need code under the Unload event in the window function to save their data.

The owner property is set at design time and only one window can have the property. All the other windows in the application are owned windows which causes the effects described above. The owner window should be the first window loaded by the application, because the owned windows expect to find their owner when they are loaded.

How to draw graphics

Ubercode has graphics methods for drawing graphics, text and bitmaps on the printer or in a window. When drawing in a window the graphics are drawn behind the controls so generally graphics are drawn in a window without controls. This is not essential though and you can combine graphics and controls in the same window for interesting visual effects.

When drawing graphics call Startprint to initialize the printer, or Startgraph if drawing on a window background. Then call any of the graphics methods which include functions such as Drawline for lines, Drawpicture for icons and bitmaps, and Drawtext for text. When drawing is finished call Endprint to close the printer or Endgraph to clear the graphics from the window background.

Startgraph and Endgraph are optional when drawing on a window, because if needed they are called automatically. However Startprint and Endprint are required when using the printer, because they open and close the print job and display a printer progress window during the print job.

The next example shows a simple graphics program which displays a window and draws a circle in the middle. The program uses the visual objects in the following table:

```
+-----------------------------------------------------------+
| Caption     = "Graphics window"                           |
| Name        = Main                                        |
| Position    = 100,100,120,100                             |
| Typeof      = Dialog                                      |
+-----------------------------------------------------------+
```

The resource file containing these properties is shown next, followed by the main class.

```
// Graphics.rc
MAIN DIALOG 100, 100, 120, 100
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
CAPTION "Graphics window"
BEGIN
END

// Graphics.cls
Ubercode 1 class Graphics
public callback function Main(in  EventId:integer
                                  ControlObj:control
                                  Key:integer
                              out Cancel:boolean)
code
  select EventId
  case LOAD_EVENT =>
       call Startgraph(me)
       call Drawshape(me, 3, 1, 10, 10,
            GetPagewidth(me)-10, GetPageheight(me)-10)
  end select
end function
end class
```

The main class *Graphics* does the actual drawing. There is a single window function *Main* which initializes the graphics and draws a shape under the Load event. The parameters of Drawshape determine the shape that is drawn and its size - the second parameter specifies the shape and 3 means a circle. The Pagewidth and Pageheight properties get the size of the graphics area of the window. All the graphics are drawn under the Load event so they are complete before the window becomes visible (load is triggered when the window is in memory but before it is visible). The graphics are drawn in an off screen bitmap so they are cached and refreshed from the bitmap whenever the window needs to be redrawn.

Graphics and Window redrawing

All graphics methods (Drawtext, Drawpicture etc.) draw in an off screen bitmap which refreshes the window when necessary. For example, the bitmap is used if the graphics window is reactivated, after being covered by another window. Redrawing occurs when Windows regains control after processing an event, not during each call to a graphics method. This is more efficient and avoids flickering, since all redrawing is merged and occurs at one time. This system of drawing is internal to the graphics system, and normally has no effect on a program.

However, this system affects graphics code that takes many seconds to run, and affects code that mixes graphics with calls to the Delay function. The effect is the graphics appear after the event code that called the graphics has finished running. For example:

```
case DBLCLICK_EVENT =>
     call Drawtext(me, "Some text" + NL)
     call Delay(1.0)
     call Drawtext(me, "More text" + NL)
     call Delay(1.0)
case OTHER_EVENT =>
```

In this example the double click event uses graphics methods to draw text, mixed with calls to Delay. The text only appears when the event code has finished, which is after the second call to Delay. The result is a delay of two seconds, then the two lines of text "Some text" and "More text" will both appear. Also, if the calls to Drawtext and Delay were replaced by complex graphics that took many seconds to run, the graphics would only appear when the event code finished.

If this causes problems, Show can be used to redraw the graphics immediately without waiting for the event code to finish. In addition to making a window active, Show updates the window graphics from the off screen bitmap. The example just shown could be modified by adding *call Show(me,0)* after each call to Drawtext. The graphics window must be modeless for this to work, as modal windows cannot re-show themselves.

How to detect keystrokes

Programs sometimes need to detect the keys pressed in a window. For example a data base application may use the page up and page down keys for moving through data, or a game may use the arrow keys to control the movement of an object on the screen.

Whenever a key is pressed in a window it sends the Keypress event to the window function, so you can put code under this event to carry out the desired actions. When the event is triggered, the *ControlObj* parameter of the window function is the control object in which the key was pressed, and the *Key* parameter is the actual key that was pressed. The following key constants are available:

KEY CONSTANT	DESCRIPTION
KUP,KDOWN	Cursor keys (up/down arrow)
KLEFT,KRIGHT	Cursor keys (left/right arrow)
KCTRLA to KCTRLZ	Command keys (Ctrl+A to Ctrl+Z)
KDEL	Delete key
KEND	End key
KENTER	Enter key
KESC	Escape key
KF1 to KF12	Function keys (F1 to F12)
KHOME	Home key
KPGDN	Page down key
KPGUP	Page up key

The Active Window

The active window is the window currently being operated by the user. It is always drawn with a highlighted caption bar. The flow of control remains in the operating system while the window is active. The normal sequence of events is the user carries out actions in the window, the window sends any events that were triggered to the window function which then runs code to process the event. Typically the processing code runs quickly, taking just a few seconds, and when the processing is complete the window function returns. This returns control to the operating system which then awaits further user actions in the window.

This continues until the window is closed. What happens next depends on whether the window was shown as a modal or modeless window. When a modal window is closed program execution continues from the call to Show that activated the window. When a modeless window is closed the flow of control remains with the operating system which activates another modeless window in the same application, sending the events to its window function. If the last modeless window was closed there are no more events for the application, which will terminate if there are no windows loaded.

Using Menus

Any window can have a menu attached to it. The menu is drawn as a menu bar just below the caption bar of the window. Whenever the window is active its menu can be used, and choices made from the menu sends a Command event to the window function.

To attach a menu to a window, use the Dialog editor to edit the window and click the Add Menu button in the Toolbox. This starts the Menu Editor, which allows you to set the properties of the menu items, and to arrange their order. The next example shows a program with a single window having four menu items. The program uses the following visual objects:

```
+-----------------------------------------------------------+
| Caption     = "Menu window"                               |
| Name        = Main                                        |
| Position    = 100,100,120,80                              |
| Typeof      = Dialog                                      |
+-----------------------------------------------------------+
+-----------------------------------------------------------+
| Caption     = "&Open..."                                  |
| Name        = FILE_OPEN                                   |
| Typeof      = Menuitem                                    |
+-----------------------------------------------------------+
+-----------------------------------------------------------+
| Caption     = "E&xit"                                     |
| Name        = FILE_EXIT                                   |
| Typeof      = Menuitem                                    |
+-----------------------------------------------------------+
+-----------------------------------------------------------+
| Caption     = "&1st choice"                               |
| Name        = OTHER_CHOICE1                               |
| Typeof      = Menuitem                                    |
+-----------------------------------------------------------+
+-----------------------------------------------------------+
| Caption     = "&2nd choice"                               |
| Name        = OTHER_CHOICE2                               |
| Typeof      = Menuitem                                    |
+-----------------------------------------------------------+
```

The resource file containing these properties is shown next, followed by the main class.

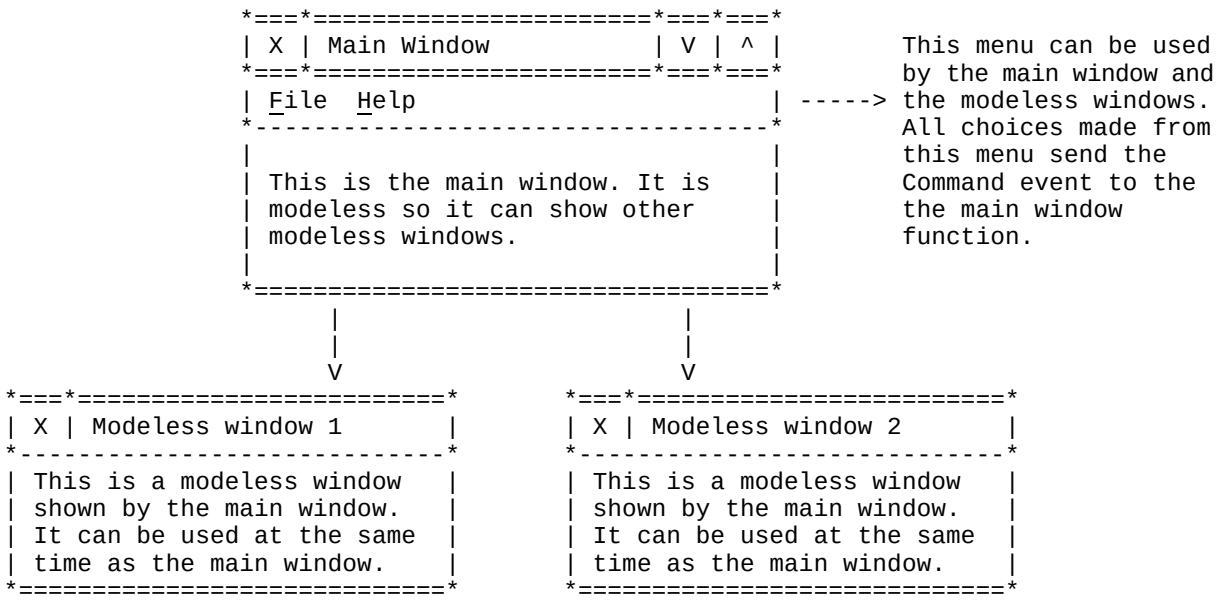
```
// Menudemo.rc
#define FILE_OPEN               101
#define FILE_EXIT               102
#define OTHER_CHOICE1           103
#define OTHER_CHOICE2           104
MainMenu MENU
BEGIN
  POPUP "&File"
  BEGIN
    MENUITEM "&Open...",          FILE_OPEN
    MENUITEM "E&xit",             FILE_EXIT
  END
  POPUP "&Other"
  BEGIN
    MENUITEM "&1st choice",       OTHER_CHOICE1
    MENUITEM "&2nd choice",       OTHER_CHOICE2
  END
END
MAIN DIALOG 100, 100, 120, 80
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
CAPTION "Menu window"
MENU MainMenu
BEGIN
END

// Menudemo.cls
Ubercode 1 class Menudemo
public callback function Main(in  EventId:integer
                                  ControlObj:control
                                  Key:integer
                              out Cancel:boolean)
code
  select EventId
  case COMMAND_EVENT =>
       select ControlObj
       case me.FILE_EXIT =>
            call Unload(me)
       else =>
            call Msgbox("Menu", GetCaption(ControlObj))
       end select
  end select
end function
end class
```

The main class shows a single window called Main. The main window has a menu attached with four menu items having the properties defined previously. When the program runs, any choice made from the menu triggers a command event which is sent to the window function. The function can tell which choice was made by examining the *ControlObj* parameter of the window function. There is code under the choices to show a short message, apart from File-Exit which calls Unload to unload the main window and end the program.

Applications normally have more than one window, and the way a menu works depends on the window it is attached to. A menu can be attached to a main window in which case it is available to all modeless windows shown subsequently. A menu attached to a non-main window is available only while the non-main window is active. These situations are discussed in more detail below.

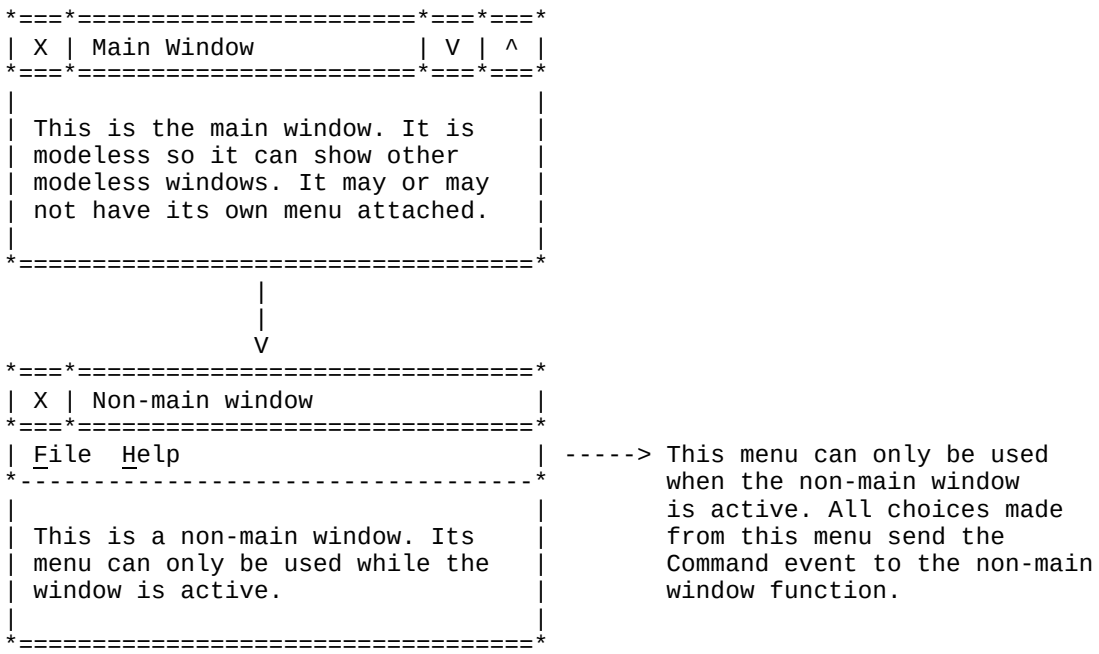
Menu attached to main window

This is the most common situation. The application' s main window is an owner window with a menu attached to it. As the application runs it displays one or more modeless windows. This is known as an MDI application (multiple document interface) because the application consists of a top level window with multiple document windows available below it. The application' s menu can be used either from the main window or from any of the modeless windows. Whenever a menu choice is made the command event is sent to the window function of the main window. This is shown next:



Menu attached to non-main window

Menus can also be attached to a non-main window, in which case the menu can be used only while the window is active. An application could attach different menus to different windows but this is unusual. When a choice is made from the menu of a non-main window the command event is sent to the window function of the non-main window. This applies regardless of whether the non-main window is a modeless window or a modal window. A menu attached to a non-main window is shown next:



2.12 Printing

Ubercode can copy text, bitmaps, graphical shapes and window images to the printer. You can use the default printer or you can get the list of attached printers and choose which to use. The basic sequence of code for printing is:

```
call Startprint(printer)
// call graphics methods for printing
call Endprint(printer)
```

Startprint is called at the start of every print job to initialize the printer and to display a temporary window that shows the progress of the print job. The temporary window is known as a printer progress dialog. Graphics methods such as Drawtext and Drawpicture do the actual printing. Newpage can be used to make the printer eject the current page and start a new page. Page breaks also happen automatically if the printed data is too large to fit on a page. Finally Endprint is called to close the printer, send the print job to the print spooler and to remove the printer progress window. Endprint must always be called when printing is finished.

Using the default printer

The quickest way of printing is using the default printer as you don' t need code to enumerate the connected printers and choose which to use. The default printer is the Sysprinter object. This is set up under Windows using the command Start - Settings - Printers, then by clicking the right mouse button on the printer you want to be the default, then by choosing "Set as Default" from the popup menu.

The following print loop shows how to print the *lines* array, which is assumed to be an array of strings. The print loop starts the printer, loops through all the strings in the array and uses Drawtext to copy each to the printer. After printing the strings it calls Endprint to close the print job:

```
call Startprint(Sysprinter)
for i from Lbound(lines) to Ubound(lines)
  exit when GetPrinterstatus(Sysprinter) /= 1
  call Drawtext(Sysprinter,lines[i])
end for
call Endprint(Sysprinter)
```

The call to Startprint initializes the printer, displays a printer progress window and disables all the application windows to avoid re-entrancy problems while the progress window is active.

The **for** loop prints out all the strings in the array. The call to Printerstatus checks the printer is still active each time round the loop, and quits the loop if Cancel is pressed in the progress window or if an error occurred during printing. This is useful for large print jobs as it avoids repeated unnecessary calls to Drawtext if the print job was canceled or if an error occurred, and means the application regains control more quickly. For smaller print jobs of only a few pages, checking the printer status is unnecessary because the print job will quickly finish regardless of whether there is an error or if the print job is canceled.

The operating system often speeds up printing by spooling the entire print job to a temporary file, then scheduling the printing of the file for later on. This means a print loop that copies out 10 or 20 pages may complete in a matter of seconds, in terms of the time spent executing the program statements.

After the **for** loop finishes normally, or finishes because the printer status indicated an error or a canceled print job, you must call the Endprint function. Endprint closes the printer which tells the operating system the print job has finished spooling, also it removes the printer progress window and re-enables any application windows that were previously disabled. Endprint must always be called after finishing the print job, otherwise the printer progress window will not be removed.

The next example shows a complete program that prints to the default printer:

```
// Print1.cls
Ubercode 1 class Print1

function Printtext(in text:string[*])
var i:integer(0:MAXINT)
code
  call Startprint(Sysprinter)
  for i from 1 to Strcount(text)
    call Drawtext(Sysprinter, Strline(text,i))
  end for
  call Endprint(Sysprinter)
end function

public function main()
var filename:string[*]
    textstr:string[*]
code
  filename <- Openfiledialog("Print", 0, "*.txt")
  if filename /= "" then
    call Loadfile(filename, FILE_TEXT, textstr)
    call Printtext(textstr)
  end if
end function

end class
```

In this example the actual printing is done by *Printtext* which calls Startprint, loops through the lines of text calling Drawtext, then calls Endprint when the printing is finished. Function *main* calls Openfiledialog which prompts for the name of a text file to print, then if a name was entered Loadfile is called to read the text file into a string. The Printtext function is then called to print the string.

Using a non default printer

A program may need to use a printer other than the default. To do this use the Printers function which enumerates all the printers, then choose a printer to use. The Printers function is one of the iterator functions normally used with a For each loop.

The next example shows how to do this. The most important function below is *Chooseprinter* which enumerates all the printers, allows one to be selected from a list, then returns the printer object that was selected. Function *main* calls *Chooseprinter* to select a printer and prints a short message.

```
// Print2.cls
Ubercode 1 class Print2

function Chooseprinter(out result:printer)
var prn:printer
    prnlist:string[*]
    prname:string[*]
code
  prnlist <- ""
  for each prn in Printers()
    prnlist <- prnlist + GetName(prn) + NL
  end for
  prname <- Listbox("Print", "", prnlist)
  if prname /= "" then
    for each prn in Printers()
      if GetName(prn) = prname then
        result <- prn
      end if
    end for
  end if
end function

public function main()
var prn:printer
code
  prn <- Chooseprinter()
  if GetName(prn) /= "" then
    call Startprint(prn)
    call Drawtext(prn, "Hello printer" + NL)
    call Endprint(prn)
  end if
end function

end class
```

Chooseprinter has a single **out** parameter *result* which is the printer object selected by the user. The returned printer object has an empty Name property if no printer was selected.

Chooseprinter first builds a list of printers using the **for each** iterator to enumerate all the connected printers. The Name property obtains the printer name from the printer objects returned by the iterator. After the printer names have been obtained they are displayed in a Listbox window which allows the user to pick a name. If a name was picked we have to find the corresponding printer object by enumerating all the printers with another **for each** loop and by storing the details of the printer object with a matching name. Printers always have unique names so it is safe to enumerate all the connected printers in the search.

Chooseprinter then returns the chosen printer object to function main. Main checks the returned printer is valid by making sure it has a valid name property, then it opens the printer, writes out a short message and closes the printer again.

Chooseprinter is a useful function to add to a utility class, because you can use it whenever the user wants to choose a printer. Another useful function is Printtext shown in the earlier example which copies a string to the default printer. Printtext could easily be modified to take a printer object as a parameter instead of always using the default printer. These handy functions are shown next:

```
function Chooseprinter(out result:printer)
var prn:printer
    prnlist:string[*]
    prname:string[*]
code
  prnlist <- ""
  for each prn in Printers()
    prnlist <- prnlist + GetName(prn) + NL
  end for
  prname <- Listbox("Print", "", prnlist)
  if prname /= "" then
    for each prn in Printers()
      if GetName(prn) = prname then
        result <- prn
      end if
    end for
  end if
end function

function Printtext(in prn:printer text:string[*])
var i:integer(0:MAXINT)
code
  call Startprint(prn)
  for i from 1 to Strcount(text)
    call Drawtext(prn, Strline(text,i))
  end for
  call Endprint(prn)
end function
```

Printing a window

Sometimes a program needs to print out a window as it appears on the screen. This makes it possible for a window to have a "Print" button. To do this use Startprint to initialize the printer in the usual way, Drawwindow which prints the image of a window and Endprint when printing has finished.

The next example shows how to print a window. The program has a multi line edit object for entering text, a Print button that prints the window and a Close button. The program uses the visual objects in the following table:

```
+-----------------------------------------------------------+
| Caption     = "Print window"                              |
| Name        = Main                                        |
| Position    = 100,100,120,120                             |
| Typeof      = Dialog                                      |
+-----------------------------------------------------------+
+-----------------------------------------------------------+
| Caption     = "Type in something below:"                  |
| Name        = Label1                                      |
| Position    = 4,4,110,14                                  |
| Typeof      = Label                                       |
+-----------------------------------------------------------+
+-----------------------------------------------------------+
| Name        = Edit1                                       |
| Multiline   = 1                                           |
| Position    = 4,18,110,74                                 |
| Typeof      = Edit                                        |
+-----------------------------------------------------------+
+-----------------------------------------------------------+
| Caption     = "Print"                                     |
| Name        = Button1                                     |
| Position    = 10,98,40,16                                 |
| Typeof      = Pushbutton                                  |
+-----------------------------------------------------------+
+-----------------------------------------------------------+
| Caption     = "Close"                                     |
| Name        = Button2                                     |
| Position    = 65,98,40,16                                 |
| Typeof      = Pushbutton                                  |
+-----------------------------------------------------------+
```

The resource file containing these properties is shown next, followed by the main class.

```
// Print3.rc
#define Label1  101
#define Edit1   102
#define Button1 103
#define Button2 104
MAIN DIALOG 100, 100, 120, 120
STYLE WS_POPUP | DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
CAPTION "Print window"
BEGIN
  LTEXT "Type in something below:",Label1,4,4,110,14
  EDITTEXT Edit1,4,18,110,74,ES_MULTILINE|ES_WANTRETURN
  PUSHBUTTON "Print",Button1,10,98,40,16,WS_TABSTOP
  DEFPUSHBUTTON "Close",Button2,65,98,40,16,WS_TABSTOP
END

// Print3.cls
Ubercode 1 class Print3
public callback function Main(in EventId:integer
                                 ControlObj:control
                                 Key:integer
                              out Cancel:boolean)
code
  select EventId
  case COMMAND_EVENT =>
       select ControlObj
       case me.Button1 =>
            call Startprint(Sysprinter)
            call Drawwindow(Sysprinter, Gethandle(me))
            call Endprint(Sysprinter)
       case me.Button2 =>
            call Unload(me)
       end select
  end select
end function
end class
```

This works as follows. Function main is displayed as a window with an MLE control at the top, and with a Print button and a Close button near the bottom of the window. When the program starts, the window is loaded and the user can type text into the edit area and tab round the controls in the usual way.

If the Print button is pressed this sends the Command event to the window function, with the *ControlObj* parameter equal to Button1 (the Print button). This calls Startprint to initialize the default printer, then calls Drawwindow to draw the image of the current window function (Main) onto the printer. The keyword *me* is used only in window functions - it is shorthand for the name of the window, and the Handle property returns the window handle (HWND) needed for printing the window. Drawwindow has a HWND argument which specifies the window to be printed, which allows you to print windows from other applications if you know the window handle. After the window is printed, Endprint is called to close the printer and finish the print job.

Finally when the Close button is pressed this sends a Command event with the *ControlObj* parameter equal to Button2. This calls Unload to unload the main window, after which the program has no windows left in memory and will end.