Recent advanced in Compiler

Const parameters

The ‘const’ keyword for functions and procedures is now implemented. It protects the parameter from being updated within the procedure or function. If the parameter is one or two bytes long it is passed by value, but if longer it is passed by reference.

The purpose of this construct, though, unlike, say in Delphi, is to minimise Frame usage. Therefore you must treat const in the same way as var (just as it was in early Delphi) for anything other than byte, word, char etc. The only difference is that you are prevented from assigning a value to the parameter. In particular this means that the parameter, if it is bigger than 2 bytes, must always be a variable. So unlike Delphi you cannot pass large (sized) constants to a method using a const parameter.

Default parameters

Example

function MyInc(x : integer; increment : integer = 1) : integer;

The second parameter here defaults to 1. That is the programmer may use the function with one or two parameters. If only one parameter is passed the default value is used for the second parameter.

Not supported for complex parameters like arrays or records, nor for var or const parameters.

Skeletons

These allow ‘late defined’ constants to be used in a structure such as a record.

There are two syntaxes for using a skeleton, the definition and usage. The syntax for these two is slightly different.

The syntax for defining the constants is < *name : type [= default value]* [;…] >

Example

```
TFIFO : record< BuffLen : word >
  Data : array[ 0..BuffLen – 1]  of byte;
  ReadPtr : word;
  WritePtr : word;
end;
```

or

```
TFIFO : record< BuffLen : word = 8>
  Data : array[ 0..BuffLen – 1]  of byte;
  ReadPtr : word;
  WritePtr : word;
end;
```

The advantage of describing it this way (rather than defining the const BuffLen externally) is that if, for example, two FIFO buffers are needed with buffers of different lengths, a single definition is required, and these can be later declared like this

```
var
  USBBuff : TFIFO<BuffLen = 64>;
  SerBuff : TFIFO<BuffLen = 8>;
```

Internally two separate record structures will be created and used, but the benefit to the programmer is that the compiler does this work for him.

Important

A skeleton record must be completely defined, including all functions and procedures before creating any instances. The easiest way to ensure that is to get into the habit of putting all skeleton definitions into a separate unit.

Another, more complex example would be

```
TBiDiFIFO= Record< SendLen :word; RcvLen : word>
  SendBuff : TFIFO< BuffLen = SendLen >;
  RcvBuff : TFIFO< BuffLen = RcvLen >;
…
end;
```

The declaration of a var of this type might be

```
var
  SerBuff : TbiDiFIFO<SendLen = 8; RcvLen=10>;
```

The syntax for using the constants is

< ParmName = ParmValue[;...] >

Examples are shown above.

Defaults in skeletons

Defaults are permitted in skeletons too. So, for example if we decide most buffers are 8 bytes long the above example becomes

```
TFIFO : record< BuffLen : word = 8 >
  Data : array[ 0..BuffLen – 1]  of byte;
  ReadPtr : word;
  WritePtr : word;
end;

var
  USBBuff : TFIFO<BuffLen = 64>;
  SerBuff : TFIFO<>;
```

Inheritance

Skeleton records support both simple inheritance and multiple inheritance. What is more, the inherited records do not need to be the first to be defined, and there is no requirement to put all inherited records at the beginning.

Just as in regular records, though, all field name must be unique within the combined record. This removes most of the issues traditionally associated with multiple inheritance.

The ‘as’ keyword

If you need to restrict the scope of a record to one of its ancestors, for example to pass as a parameter to a function you can use the ‘as’ keyword.

Example:

```
type
  MyRec1<> = Record
    F1 : word;
  end;

  MyRec2<> = Record
    F2 : byte;
  End;

  MyRec3<> = record
    f3 : string[8];
    inherited MyRec1;
    f4 : integer;
    inherited MyRec2;
    f5: byte;
  end;

function V2( const x:MyRec2) : byte;
begin
  return(x.F2);
end;

var
  m : MyRec3;
  i : byte;
begin
   i := V2( m as MyRec2 );
end;
```

Visibility of fields in records

By default all fields are **public**, that is to say, visible and modifiable by all. For skeleton records only it is now possible to hide fields so that those fields can only be modified by the record itself (through record functions, record procedures or record properties – see later) using the keyword **private**. The keyword **public** is also supported to complete the visibility possibilities.

Once a visibility mode is specified it applied to all subsequent fields until another visibility mode is specified.

Fields inherited also inherit the visibility, so private is a little more like protected in classical object orientation. A descendant class cannot make a private field public except through properties.

Record functions and procedures

These are only supported for skeleton records.

These methods have a built in hidden parameter called ‘self’ which is of the record type.

Inheritance is supported for functions and procedures, but please note that this is not conventional inheritance, because conventional inheritance requires extra memory overhead that is not really suitable for this type of microcontroller. You will not really notice the difference unless you start using lists of pointers to records and apply different record types to different methods and expect them to know which version of the function to use. That will not be possible. Also there is no implementation of the ‘**is**’ keyword such as you would see in classes in Delphi. That has no sensible meaning for the implementation of inheritance provided. It is possible to build such structures for yourself, but frankly if you need it you are probably using the wrong type of device.

The simple definitions of functions and procedures is how you might expect. For example

```
TFIFO : record< BuffLen : word >
  Data : array[ 0..BuffLen – 1]  of byte;
  ReadPtr : word;
  WritePtr : word;
  function Empty: boolean;
end;

function TFIFO.Empty : boolean;  // Note that you do not need to repeat the parameter list
begin
  return (ReadPtr = WritePtr);
end;
```

An example that uses the delayed constant BuffLen is

```
TFIFO : record< BuffLen : word >
  Data : array[ 0..BuffLen – 1]  of byte;
  ReadPtr : word;
  WritePtr : word;
  function Full: boolean;
end;

function TFIFO.Full: boolean;
var
  iTest : word;
begin
  iTest := WritePtr + 1;
  if iTest = BuffLen then
    iTest := 0;
  endif;
```

```
  return( iTest = iReadPtr );
end;
```

Default values in record functions and procedures

Record functions and procedures may have defaults, but those defaults must only be defined inside the record definition, so for example:

```
type
  TMyRec : Record<>
    function XX( v : byte = 7) : byte;
  end;
```

the implementation is written as

```
function TmyRec.XX( v : byte ) :byte;
begin
  …
end;
```

Note that the default value is not repeated in the implementation. This makes it easier to change the default value later, because it only need to be changed in one place.

inheritance of record methods

Records and procedures are visible to records that inherit them, but you cannot redeclare them as that would violate the unique field name requirement.

You must tell the compiler that you are going to redeclare (override) a base function.

For example

```
type
  MyRec1<> = Record
    private
      F1 : word;
    public
       virtual function V1: word;
  end;

  MyRec2<> = Record
    private
      F2 : byte;
    public
       virtual function V2: byte;
  End;

  MyRec3<> = record
    f3 : string[8];
    inherited MyRec1;
    f4 : word;
    inherited MyRec2;
```

```
    f5: byte;
    override function V1 : word;
    override function V2 : byte;
  end;

function MyRec3.V1 : word;
begin
  f4 := inherited V1;
  f3 := IntToStr( f4 );
  return( f4 );
end;
```

Record Properties

A little like normal properties a record property has a getter and a setter (**read** and/or **write**). However, unlike normal properties the setter and getter must be defined within the record, either as a field or as a function or procedure with the correct form.

For example:

```
TFIFO : record< BuffLen : word >
  private
    fData : array[ 0..BuffLen – 1]  of byte;
    fReadPtr : word;
    fWritePtr : word;
    function GetNext: byte;
    procedure SetNext( Value : byte );
  public
    property Next : byte read GetNext write SetNext;
    property ReadPtr : word read fReadPtr; // don't allow uncontrolled write to this field
end;
```

Notice how one property definition accesses a function and a procedure, while the other access a field directly. Notice also how everything except the properties are hidden from the programmer so uncontrolled access is forbidden.

The definitions of the functions might be as follows:

```
    function TFIFO.GetNext: byte;
    var
      Result : byte;
    begin
      if fReadPtr <> fWritePtr then
        Result := fData[ fReadPtr ];
        fReadPtr := fReadPtr + 1;
        if fReadPtr = BuffLen then
      else
        return( 0 );
      endif;
    end;

    procedure TFIFO.SetNext( Value : byte );
    var
```

```
    iNext : word;
  begin
    iNext := fWritePtr + 1;
    if iNext <> fReadPtr then // buffer not full
      Data[ fWritePtr ] := Value;
      fWritePtr := iNext;
   endif;
  end;
```

The following is also allowed

```
TFIFO : record< BuffLen : word >
  private
    fData : array[ 0..BuffLen – 1]  of byte;
    fReadPtr : word;
    fWritePtr : word;
    virtual function GetNext< BuffLen >: byte;
    virtual procedure SetNext<BuffLen>( Value : byte );
  public
    property Next : byte read GetNext write SetNext;
    property ReadPtr : word read fReadPtr; // don't allow uncontrolled write to this field
end;
```

So property setters and getters can be overridden.

Other Notes:

1. For those of you familiar with Delphi please note that virtual and override appear before the method definition, rather than after.

2. **Private** is more or less equivalent to Delphi **strict protected**.

3. Chevrons are only required when creating record definitions and creating a record. They are not required for any function or procedure coding.

4. Skeleton records must be completely defined, including all functions and procedures *before* creating any instances of the records.

Constants and types if record functions and procedures

Because there may be multiple physical copies of a record function or procedure, you are not permitted local constants within a function. Use record skeleton constants (late defined constants) or global constants instead. Local types are also currently not implemented.

Making Record Functions and procedures efficient.

Because multiple copied of a record function or procedure exists it makes sense to keep the definitions as short as possible, perhaps by calling an external non-record function to do most of the work. This is good practice anyway.

Restrictions on Record functions and procedures

Certain types of constructs are prohibited in record functions and procedures:

goto (and also labels)
ASM

but of course they can call normal functions and procedures that do contain these constructs.

What could possibly go wrong?

Despite their power there are issues with skeletons. Error messages could be in places that are not intuitively obvious or indeed very helpful. For example suppose we have a TFIFO function

```
function TFIFO.GetPos9 : byte;
begin
  return( fData[9]);
end;
```

Then later we define

```
var
  iBuff1 : TFIFO<Bufflen = 10>;
  iBuff2: TFIFO<BuffLen=8>;
```

Then fData[9] is valid for iBuff1, but not for iBuff2, so the error can only appear on the line defining iBuff2. Narrowing it down from there might not be easy.