SDK™ VRT FOR WINDOWS

REFERENCE MANUAL

VERSION 5.71

Superscape Inc.
3945 Freedom Circle
Suite 1000
Santa Clara
CA 95054
USA

Tel: (408) 969 0500
Fax: (408) 969 0510

Email: info@us.superscape.com

Superscape Ltd.
Cromwell House
Bartley Wood Business Park
Hook
Hampshire RG27 9XA
UK

Tel: +44 (0)1256 745745
Fax: +44 (0)1256 745777

Email: info@superscape.com

Fifth Edition 20 March 2000

Contents

Preface

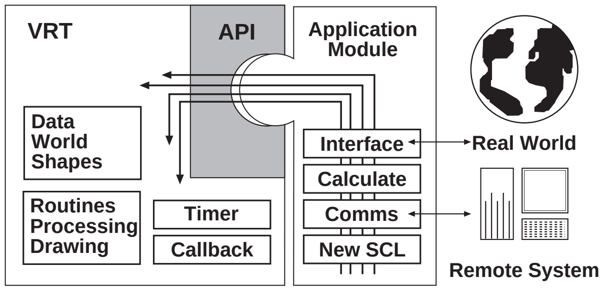
The Superscape Developers Kit (SDK)

The Superscape Developers Kit (SDK) lets you create your own program modules that can be added into VRT, and Visualiser. These applications may perform any task, and communicate their results back to the Superscape system through the Application Program Interface (API). You can load up to 100 different applications at once, although three or four is most usual.

The three main type of application are:

- Interface to specific hardware—device drivers.
- Transfer of Superscape data to and from different file formats—data convertors.
- User-defined tasks within the Superscape system—user applications.

Examples of user applications include: overlaying data from the Superscape representation of the world; performing complex calculations quickly; creating new SCL commands; controlling objects within the world.



Relationship Between VRT, API and Application Module

At its most basic level, the SDK consists of a C module which acts as startup code, and header files to define functions and variables included in the API. It also includes example applications which demonstrate how to use the major features of the API, from registering new SCL commands to intercepting interrupt vectors and communicating with external hardware.

The API interface

Each application that you write is compiled as a 32-bit dynamic link library (DLL). The interface to the three different types of application is very similar, consisting of the following main parts:

- Initialization
- Functional capability
- Exit

When the application is loaded, control is transferred to the API initialization routine in the startup code. This performs various tasks before calling the application initialisation routine, `App_Init`.

Your initialization routine must inform the Superscape system of any other code to be called and when. This can be done in one of three ways:

i. The most common method is to register a new SCL function, using the `RegisterSCL` call. This sets up a new SCL function that, when executed, calls the C routine you have written. This routine has access to any parameters passed to the SCL function, and returns a value if required. Each application module can register several new SCL functions. A similar mechanism exists for registering functions for device drivers.

ii. The second method is to intercept vectors. At various points within the visualisation process, VRT calls code pointed to by vectors. By default, these point to a piece of code that does nothing, but which can be diverted. Since several application modules can be loaded, it is good practise to call the old vector after the code has been executed, to give any other applications a chance at intercepting the vector. Data convertors intercept several vectors during initialisation.

   Vectors may be called every frame, or only after specific events. A complete list of vectors can be found in “Chapter 3: Vectors”.

iii. The third method is to use the timer device to call code at regular intervals. The address of the code and the delay between calls is passed to the timer device, which calls the registered code at the appropriate times. There are some restrictions on the type of code that can be called from the timer routine—see the timer device driver in “Chapter 5: Device drivers” for further details.

Note that when `App_Init` is called, other data—such as world, shape, palette—may not have been loaded. Therefore, any initialization that needs access to this data should be placed on the vector `VecB4Init`, which is called just before the world is initialized. At that point, all the data has been loaded and may be accessed by the application.

Just before your application is erased from memory, its exit routine `App_Exit` is called. This should deregister new SCL commands, remove timer calls, deallocate any memory used and close any open files before returning to the main program. This routine is only called once.

User routines may call other routines in the application or in the main program. Header files are provided to make the precise nature of the calls to VRT less ‘visible’ in the application source code, and should be used at all times.

Sometimes you may find it convenient to use SCL functions to perform certain tasks. This has the advantage that you are likely to be familiar with the operation of the function. Some SCL functions are also quite sophisticated, and can be used to replace a significant amount of C code.

One way of doing this is to store SCL source code, compile it, and call the SCL executor to execute the compiled code. This is quite slow, and passing values to and from routines written in SCL can be a problem. It is possible, however, to call inbuilt SCL functions directly from C code without having to deal with the SCL executor. For further details on using SCL functions directly see “Chapter 2: SCL functions”.

Getting data to VRT

There are many ways of getting data from your application module to VRT, but the two most common methods are by writing new SCL commands, and direct modification of data.

Writing SCL commands takes the pain out of modifying objects. The SCL command can simply calculate the values required (by monitoring an external device, or doing calculations for which SCL itself is unsuitable), and pass them back as a returned value. The SCL program uses the value returned by the new function to alter the world.

For example, reading a thermometer module can typically be a complex process, involving setting up and reading external devices, but the end result is simply one value. However, if the application registers a new SCL function, `gettemp`, it can be used on an object in any of the following ways:

```
/* Change height with temperature                */
ysize(me)=gettemp*10;

/* Animate with temperature                      */
ccel(me)=gettemp-20;

/* Rotate the dial on the thermometer object     */
xrot(‘Thermometer-dial’)=gettemp*18-180;
```

In each of these cases, the way in which the data is represented is immaterial to the application defined code, and can easily be changed from within VRT. This is particularly useful when the programmer of the application module is working separately from the world designers. By giving them access to the value and letting them interpret it as they wish, the application is made more general and the turnaround time for changes is reduced.

The second method of direct alteration of data, is useful for items that cannot be directly accessed by SCL, or for timer-related functions. The API vector table contains a set of pointers to all the important data in VRT such as the world, shapes, and palette. You can, therefore, locate specific items and alter them directly. This is usually much more closely linked to a particular world than use of SCL.

For further details on getting data to VRT see “Chapter 9: SDK examples”.

Conventions

The following conventions are used by the SDK when defining functions, variables, typedefs, and in dealing with indentation of loops.

You do not have to follow these conventions when writing your own applications, but a consistent style makes reading and debugging programs a lot easier.

- Function names are case-sensitive, with upper case characters introducing new words in the name. For example:

  ```
  void StartNewApplication(void);
  ```

- Variables and elements of structures are defined as for functions, unless they are single letters or abbreviations. For example:

  ```
  short           Number,i;
  long            x,y,z,Position;
  FILE            *fp,*OutputFile;

  typedef struct
  {
      short       ChkType,Length;
      long        XPos,YPos,ZPos;
      long        XSize,YSize,ZSize;
  } T_SIZEPOS;
  ```

- Constants are always upper case, and preceded by `E_`. For example:

  ```
  #define E_MAXAPPS   8
  #define E_MINAPPS  (E_MAXAPPS-8)
  ```

- Macros follow the same rules as for functions. For example:

  ```
  #define Sin (a) (C_SinTable[(unsigned)a>>6])
  #define FreeMem (p)free(p)
  ```

- Hex constants always use digits `0–9` and `A–F`, not `a–f`. For example:

  ```
  0xFEC0 0x7FFF 0x8000
  ```

- Typedefs are in upper case, preceded by `T_`. For example:

  ```
  typedef struct
  {
      short    x, y, z;
  }   T_3DVECTOR;
  typedef short T_MATRIX[3][3];
  ```

- Statements grouped in curly brackets—for example in `if` statements, structure definitions, or loops—are indented by 4 spaces, with the surrounding curly brackets on lines by themselves. For example:

```
for(i=0;i<E_MAXAPPS;i++)
{
    TerminateApplication[i]();
    UnloadApplication(i);
    FreeApplicationMemory(i);
}
```

- Blank lines are left between logical blocks in the program. For example:

```
XSize=0;
YSize=0;
ZSize=0;

NumApps=0;
for(i=0;i<E_MAXAPPS;i++)
    AppSpace[i]=NULL;

CurrentVP=InitialVP;
VPFrame=0;
```

- Statements split over two lines have the second part indented. For example:

```
if(C_Console->Corner[1].x > C_Console ->
    Corner[2].x + C_Console->Corner[2].y)
    {
                    /* ... */
    }
```

- `Switch` statements are indented by a convenient amount, usually enough to allow the `case` statements (which are not indented) to fit. For example:

```
switch(ChunkType)
{
case E_CTTEXT:  /* ... */
                       break;
default:             /* ... */
                       break;
}
```

- Function prototypes are always used, and functions defined using the ANSI standard, with arguments defined, not omitted. For example:

  ```
  short main(short argc,char **argv)
  {
                    /* ... */
  }

  void NoArguments(void)
  {
                    /* ... */
  }
  ```

- The type `int` is never used. Integer variables are always specified `long`, `short` or `char`. The internal representation of type `int` can vary not only on which compiler is being used, but on the options set in the command line of the compiler.

Chapter 1 - Installing the SDK and compiler

To use SDK 5.70, you must be running VRT 5.70 and Microsoft Visual C++ 5.0 with Service Pack 2 or later.

Microsoft Visual C++ is the only compiler that Superscape supports directly for SDK 5.70.

This chapter explains how to install the SDK and your compiler, and explains any compatibility issues that exist between the compiler and the SDK.

Installing the SDK

Superscape VRT and SDK are supplied separately.

To install the VRT software follow the instructions in the VRT "Getting Started" booklet.

The SDK is supplied on a 3.5" floppy disk, with a Setup program that must be run from within Windows. We assume that drive A: is your floppy drive, and C: is your hard disk. Substitute the correct drive letter for your computer if necessary.

The Setup program includes a copy of the "Superscape Software License and Limited Warranty Agreement". Please make sure that you read the agreement and agree with the terms and conditions before proceeding.

⇒ To install the SDK

1. Start Windows.
2. Place the SDK floppy disk in drive A: of your computer.
3. Choose Start>Run.
4. At the command line type:

   A:\SETUP.EXE
5. Click OK.
6. Follow the instructions on the screen. The Setup program guides you through the installation process.
7. When the installation is complete, install the compiler and configure it to work with the SDK.

Installing the compiler

Follow the instructions supplied with the compiler to install it, taking particular note of the following sections which detail compatibility issues between the SDK and Microsoft Visual C++ compiler.

Using the compiler runtime library - considerations

You can link with the runtime library supplied with the compiler, or direct functions to the C runtime library in VRT. We recommend that you use the compiler runtime library (by defining USE_STD_CRT in the compiler options) which automatically disables the vectoring of the VRT library.

`USE_STD_CRT` disables vectoring from the user application to VRT, while other vectors such as `VecVisEntry` are still available. You cannot disable vectoring from VRT to the user application.

If you switch off vectored access, you can use any version of the CRT you choose—static or dynamic. Note, however, that if you are using the dynamic (DLL) version of the CRT, you must make sure that the relevant DLL is available. If you cannot guarantee that it is on your target machine, build the application with the static library.

Take care if you disable vectored access to the runtime library in conjunction with memory allocation, as the memory allocated by your application may be freed elsewhere (for example, by Visualiser or VRT).

Setting up the compiler to build an application using USE_STD_CRT

The SDK is supplied with a workspace SDK.DSW (in the EXAMPLES directory where you installed the software) with all the settings you need. To create a new application, open the workspace and then create a new project within it. Alternatively, you can set up the compiler yourself with a new workspace by following the instructions below.

1. Create a new project, and set the Project Type field to Win32 Dynamic-Link Library.

Make sure that there are no spaces in the name of the DLL as this may cause VRT to work incorrectly.

---

2. Add your SDK source code files, and the file APP_LIB.C from your SDK directory.

APP_LIB.C is a file that lists the source files you need to create your SDK application.

3. Select Win32 release as the Target at the top of the project window.

4. Choose Projects>Settings, and set the following options:

Tab	Category	Setting	Set to
C/C++	General	Preprocessor definitions	USE_STD_CRT WINVIS
	Code Generation	Use run-time library	Single Threaded (recommended) or Multi Threaded
		Struct Alignment	1 byte
	Preprocessor	Additional include dirs	SDK directory
Link	Output	Entry-point symbol	DllEntryPoint

5. Set the special type definitions for Visual C++:

```
/* __vrtcall is standard calling convention so it is not needed */
#define __vrtcall

/* __vrtexport symbols are defined using a declspec extension   */
#define __vrtexport __declspec(dllexport)

/* far vectors are not supported so define a dummy type         */
typedef void (*T_DOSVECTOR)(void);
```

When you are ready to build a project, choose Build >Update All Dependencies, and then Build>Rebuild All. Errors notwithstanding, your SDK module is built and can be loaded into VRT.

Errors and warnings

Addition and other arithmetic operations may promote arguments to functions to a longer integer form, which will not work with the SDK. For example:

```
short f(short Param);

short Result;
short x=0;

Result=f(x+1);
```

This produces a level-1 warning, as `x+1` is promoted to a 32-bit int which does not fit within the expected `short` parameter to the function. The function call needs an additional cast, as in:

```
Result=f((short)(x+1));
```

A related problem involves assignment to variables. For example if you try to pop a short value from the SCL stack using:

```
short x;
x+PopN(E_SSINTEGER);
```

PopN always returns a long, and the assignment of a long value to a short variable result in alevel-2 warning. Again you can use a cast to solve the problem:

```
x+(short)PopN(E_SSINTEGER);
```

Although you can overcome these problems by suppressing all warnings or treating warnings as errors, it is generally bad practice to ignore compiler and linker warnings. If you get any sort of warning, try to eliminate it.

Care with C functions

Although the majority of C functions are available from within the SDK with no additional problems or side effects, there are a few which should be used with caution.

Take care with the `bioskey` functions. Since VRT controls the keyboard, the key presses are not passed through to the keyboard handler, and `bioskey(0)` in particular (wait for a key and report it) may lock up. Instead, use `KbReadKey` or `KbKeyReady` (see “Chapter 5: Device drivers”).

Memory allocation is another potential problem area. The standard functions `malloc`, `free`, `calloc` and `realloc` are all passed through to VRT automatically, so these can be used without modification. However, it is not permissible to bypass these and use the lower level functions, such as `_nmalloc`, directly.

Adding an application to a world

1. Start VRT and open the world which you want to add the application. Make sure that you are in the World Editor.

2. Choose File>Other Files>Open Applications.

The Open File dialog box is displayed.

3. Select the application you want to include, and click OK.

The selected application is loaded and started.

Application modules must be saved as part of the world using File>Save or File>Save As, so that it contains the entire world description including the extensions specified using the application module.

Updating an existing application

Since application modules are copied and included in a world when you choose File>Save As, follow the instructions below to include a newer version of the application.

1. Go to the World Editor and choose File>Other Files>Clear Applications.

All application modules are terminated and cleared from memory.

2. Choose Files>Other Files>Open Applications.
3. Load the new version of the application as above.

The world will use the latest version of the application.

Chapter 2 - SCL functions

Introduction

To write application code that can be executed as an SCL function you need to understand certain aspects of how SCL works internally.

The C-like source code is compiled into an intermediate code, whose format is not important to the application. Values are passed between functions on a stack, which serves as a store for intermediate results.

There are two stacks in SCL, the data stack which stores values and intermediate results, and the return stack which stores where procedures were called from and loop counters.

Arguments to an SCL function are placed (‘pushed’) on the data stack such that the last argument is at the top. When these arguments are retrieved (‘popped’) from the stack, they appear in reverse order.

Since other intermediate results may be on the data stack, the application code must remove its own arguments and only those arguments, and must return the correct number of results. Failure to do this leads to the stack containing incorrect data, and the SCL code cannot work as expected.

Writing an SCL function

The sequence of operations for an SCL instruction is as follows:

1. Pop arguments from the stack.
2. Check and manipulate these arguments as required.
3. Push return value back onto the stack.

A function with no input arguments omits step 1, and if it has no return value step 3.

Generally, the values on the stack are of two main classes: numeric expressions and pointers. You can retrieve them from the stack using the functions `PopN`, `PopF` and `PopP`, which get the next item from the stack as, respectively, an integral number, a floating point number or a pointer. The converse operations `PushN`, `PushF` and `PushP` are also defined.

Each item on the stack has a type (as used by the SCL command `settype`), which you should specify when attempting to pop an item from the stack. If the item is not of the correct type, an error is generated. When popping items from the stack, whole classes of values may match a given type, so you should use the following values:

Using	Allows	Returns
`PopN(E_SSINTEGER)`	Integer, fixed or float	Integer
`PopN(E_SSOBJNUM)`	Object number	Integer
`PopF(E_SSFLOAT)`	Integer, fixed or float	Float
`PopP(E_SSPCHAR)`	Pointer to char	Pointer
`PopP(E_SSPSHORT)`	Pointer to short	Pointer
`PopP(E_SSPANGLE)`	Pointer to angle	Pointer
`PopP(E_SSPLONG)`	Pointer to long	Pointer
`PopP(E_SSPFIXED)`	Pointer to fixed point value	Pointer
`PopP(E_SSPFLOAT)`	Pointer to floating point value	Pointer
`PopP(E_SSPOBJNUM)`	Pointer to object number	Pointer

`PopP` does not, by default, distinguish between writable pointers (such as variable addresses) and constant pointers (such as string constants). To make sure that a pointer is writable, add `E_SSWRITE` to the type value passed to `PopP`:

	Pop instruction	**Pop instruction**
Argument	`PopP(E_SSPCHAR)`	`PopP(E_SSPCHAR+E_SSWRITE)`
`&CharVar`	OK	OK
`"String"`	OK	Error

If a `pop` instruction fails—due to a wrong type or the stack being empty, for instance—the variable `*C_SCLError` is set to a negative value (it is normally positive). This lets you see if the retrieval of the arguments was performed correctly.

A pointer to the SCL variable `null` is identified as a NULL pointer in C. This gives a quick and easy way of determining whether to store a value at a passed variable address. If it is NULL, do not read or write to the address that the pointer points to.

`PopP` returns a pointer of type `void *`, which may be assigned to a pointer of any type. It is important to be aware of the type of the pointer, however, since writing a 4 byte `long` value to a pointer which was passed as a pointer to a `short` will almost certainly cause problems.

After the arguments have been popped from the stack, the SCL function performs some action based on the arguments. If this leads to an error—by attempting to open a file that does not exist, for instance—set the variable `C_SCLError` to `E_ERROR` and return immediately. This is caught as an “Unhelpful general error” by the SCL error reporting mechanism, and displayed like any other SCL error.

If the function executes successfully, it may return a value, which is pushed back onto the stack. When pushing an item back onto the stack, the format of the push instruction is:

`PushN(Type, Value);`	for integer values, *or*
`PushF(Type, Value);`	for floating point values *or*
`PushP(Type, Pointer);`	for pointers.

The exact type must be specified, as follows:

`E_SSINTEGER`	Integer
`E_SSOBJNUM`	Object number (returned as an integer).
`E_SSFIXED`	Fixed point number (returned as an integer).
`E_SSFLOAT`	Floating point number.
`E_SSPCHAR`	Pointer to `char` value.
`E_SSPSHORT`	Pointer to `short` value.
`E_SSPANGLE`	Pointer to angle value (`short` value in brees, converted to and from degrees when read or written by SCL. 1 bree = $^1/_{65536}$ of a circle).
`E_SSPLONG`	Pointer to `long` value.
`E_SSPFIXED`	Pointer to `fixed` value (`long` value, top 16 bits before the point, bottom 16 bits after the point).
`E_SSPFLOAT`	Pointer to `float` value.
`E_SSUNSIGNED`	Add to `E_SSPCHAR` or `E_SSPSHORT` to give unsigned operation.
`E_SSWRITE`	Add to any pointer type to make pointer writable. Otherwise, writing to pointer address using SCL gives error (for example, during assignment).
`E_SSINDIRECT`	Add to any pointer for extra level of indirection (for example, add to `E_SSPCHAR` to get pointer to pointer to char).

A complete SCL function is shown below, taken from one of the example files. It prints a text string at a given position on the screen which is passed the x and y positions, the string address and the foreground and background colors in that order.

```
static void SCL_PrintString(void)
{

    /* Define variables for holding arguments /*

    short x,y,FGColour,BGColour;
    char *String;

    /*  Get them from the stack,              /*
    /*  last argument first.                  /*

    BGColour=PopN(E_SSINTEGER);
    FGColour=PopN(E_SSINTEGER);
    String  =PopP(E_SSPCHAR);
    y       =PopN(E_SSINTEGER);
    x       =PopN(E_SSINTEGER);

    /* Now check the pointer is non-NULL      /*
    /* and if so call the Text routine        /*
    /* to print this string.                  /*

    if(String!=NULL)
    {
        Text(x,y,                /* Position */
             FGColour,BGColour,  /* Colors  */
             6,              /* Font 6 (6x8)  */
             -1,             /* No max length */
             0,              /* Align=topleft */
             3,              /* Both screens  */
             String);        /* Print string  */
    }
}
```

This function starts by popping all the arguments from the stack in reverse order, and storing them in temporary variables. It then checks to see if the string address is non-NULL, and if so prints the string on the screen using the internal VRT routine `Text`. As there is no return value, there is no push before the end of the function.

The following table shows the type of record that can be returned by each function type, which version of `Push` you must use, and all valid data types.

Function type	Returned type	Push	Valid data type
E_PROCEDURE	E_SSINTEGER	PushN	E_SSINTEGER
		PushF	E_SSFIXED
			E_SSFLOAT
	E_SSOBJNUM	PushN	E_SSOBJNUM
	E_SSNONE		
E_VARIABLE	E_SSPOINTER+E_SSWRITE	PushP	E_SSPCHAR+E+SSWRITE
			E_SSPSHORT+E_SSWRITE
			E_SSPANGLE+E_SSWRITE
			E_SSPLONG+E_SSWRITE
			E_SSPFIXED+E_SSWRITE
			E_SSPFLOAT+E_SSWRITE
			E_SSPCHAR+E_SSUNSIGNED +E_SSWRITE
			E_SSPSHORT+E_SSUNSIGNED +E_SSWRITE
E_CONSTANT	E_SSPOINTER	PushP	E_SSPCHAR
			E_SSPSHORT
			E_SSPANGLE
			E_SSPLONG
			E_SSPFIXED
			E_SSPFLOAT
			E_SSPCHAR+E_SSUNSIGNED
			E_SSPSHORT+E_SSUNSIGNED
	E_SSNONE		
E_POINTER	E_SSRPOINTER	PushP	E_SSPCHAR
			E_SSPSHORT
			E_SSPANGLE
			E_SSPLONG
			E_SSPFIXED
			E_SSPFLOAT
			E_SSPCHAR+E_SSUNSIGNED
			E_SSPSHORT+E_SSUNSIGNED

While `E_VARIABLE` and `E_CONSTANT` type functions return a pointer, these pointers are resolved during compilation and so act more like a variable. `E_POINTER` type functions return a real pointer.

The compiler record

To let the SCL compiler and decompiler know the name of the SCL command, how many arguments to expect, and what type they are, you need to specify the data in a structure known as a compiler record (type `T_COMPILEREC`), which is passed to VRT when new SCL commands are registered. T_COMPILREC has the following structure:

```
typedef struct
{
  char                   Name[10];
  short                  OpCode;
  char                   InOut,Action;
  unsigned short         Type;
  unsigned char          RetType;
  unsigned char          ArgType[10];
}  T_COMPILEREC;
```

It consists of the following elements:

Name	The name of the SCL command, up to 8 characters long.
OpCode	Filled in by the system; set it to `0x400`.
InOut	Number of input and output values. If specified as a 2-digit hex number, the first digit is the number of input values (arguments) from `0–A`, and the second digit is the number of output values from this function, 0 or 1.
Action	Used by the system; set it to `0`.
Type	Type of function. This should be one of the following:

`E_PROCEDURE`	Returns a number (or nothing).
`E_VARIABLE`	Returns a pointer to variable, and is treated like a variable.
`E_CONSTANT`	Returns a pointer to constant, and is treated like one.
`E_POINTER`	Returns a pointer.

RetType Return value type. This should be one of the following:

`E_SSINTEGER`	Any numeric value (integer, fixed or float).
`E_SSOBJNUM`	An Object number.
`E_SSPOINTER`	A constant address.
`E_SSPOINTER+E_SSWRITE`	A variable address.
`E_SSNONE`	None.
`E_SSRPOINTER`	A real pointer value. Instead of being treated as a variable, the pointer value itself is returned.

ArgType Array of argument types, in order. These should be one of:

`E_SSINTEGER`	Any (non-object) number.
`E_SSOBJNUM`	Object number.
`E_SSPOINTER`	Any pointer.
`E_SSPOINTER+E_SSWRITE`	A variable address.
`E_SSANY`	Anything at all.

For example, here is the compiler record for the previous sample function.

```
static T_COMPILEREC    NewSCL=
  {
    “sprint”,       /* Name of new instruction */
    0x400,          /* Function code           */
    0x50,           /* 5 inputs, no output     */
    0,              /* Used by compiler        */
    E_PROCEDURE,    /* It’s a procedure        */
    E_SSNONE,       /* Returning nothing       */
    E_SSINTEGER,    /* Taking... x position    */
    E_SSINTEGER,    /* y position              */
    E_SSPOINTER,    /* string pointer          */
    E_SSINTEGER,    /* foreground color       */
    E_SSINTEGER,    /* background color       */
  };
```

It takes as arguments the x and y positions (integers), a pointer to the string (pointer to character), and the foreground and background colors (integers).

Registering SCL functions

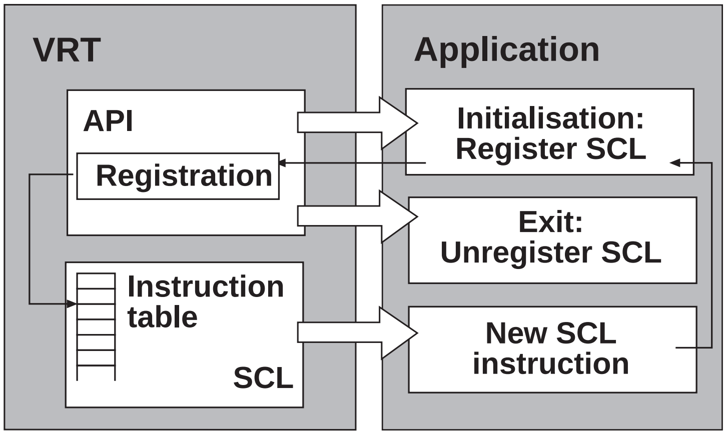
To register the new SCL function, which should be done during the initialization phase of the application, you call the `RegisterSCL` function with the addresses of the compiler record and the actual code to execute. Assuming that you have defined the function and compiler record as above, the call to `RegisterSCL` looks like this:

```
short SCLCode;
SCLCode=RegisterSCL(&NewSCL,SCL_PrintString);
```

`SCLCode` is a `short` variable which is used to store the 'handle' of the new SCL instruction. It is used later by `UnRegisterSCL` to free the SCL function when the application is terminated. It is essential that you unregister SCL commands otherwise they cause problems.

You can now use the new SCL function like any other.

The diagram below shows how the application works.



Registering a new SCL function

For further examples of how to use SCL functions, see the "Chapter 10: SDK examples."

Inside the stacks

It is occasionally useful to read data directly from the stack. Although the `push` and `pop` functions are sufficient in most cases, a knowledge of how the data is organized on the stack is necessary.

The data stack consists of two parallel arrays, one holding the type (`C_SCLType`) and one holding the value (`C_SCLStack`) for the item on the stack. Each stack value can be an integer value (`long`), a floating point value, or a pointer. You can do this by defining a union which holds all these values and simply referring to the appropriate one for the type. The members of the union are referred to as `.l` for long, `.f` for float and `.p` for pointer.

The top of the stack is marked by the stack pointer (`*C_SCLSP`), which is an index into the arrays. It refers to the first empty entry on the stack.

For example, to access the integer value on the top of the stack, use:

```
Value=C_SCLStack[*C_SCLSP-1].l;
```

The diagram below shows the construction of the data stack with sample contents:

Index	Type	Value	Contents
6			Empty
*C_SCLSP ⇨ 5			Empty
4	E_SSINTEGER	.1=2345	Integer 2345
3	E_SSPCHAR	.p="Hello"	Pointer to "Hello"
2	E_SSOBJNUM	.1=22	Object 22
1	E_SSFLOAT	.f=3.14159	Float 3.14159
0	E_SSFIXED	.1=65536	Fixed point 1.0000

If possible, you should always use the functions `PopN`, `PopP`, `PopF`, `PushN`, `PushP` and `PushF` for manipulating the data stack.

The return stack is simpler, consisting of a single array (`C_SCLReturnStack`). It is much rarer that you need to access this stack directly. It stores return addresses and loop counters, which can be determined from context. If the SCL being executed is inside a `repeat` loop, the top item on the stack is the number of remaining iterations (the SCL instruction `inloop` merely retrieves this value). Otherwise it is a return address. If the return stack is empty when SCL returns or terminates, that SCL program is finished. The return stack pointer (`*C_SCLRSP`) operates in the same way as the data stack pointer.

Parameters are passed to procedures on the data stack, and the variable `*C_SCLParamIndex` is set to refer to the first of them. When a `return` instruction is executed, the stack pointer is set back to this value, the return value (if any) pushed, and control returns to the main program. The diagram below shows a typical data stack during a procedure call:

	Index	Type
	6	Empty
*C_SCLSP ⇨	5	Empty
	4	Intermediate results (procedure)
	3	Parameter 2
*C_SCLParamIndex ⇨	2	Parameter 1
	1	Intermediate results (main program)
	0	Intermediate results (main program)

For example to access the value of integer parameter 2 you could use:

```
Value=C_SCLStack[*C_SCLParamIndex+1].l;
```

Direct access to inbuilt SCL functions

Sometimes you may find it convenient to use SCL functions to perform certain tasks. VRT lets you call inbuilt SCL functions directly from C code without having to deal with the SCL executor. There is a list of SCL functions and their function codes at the end of the APP_DEFS.H file.

An inbuilt SCL function, like those registered from the SDK, takes no arguments and returns nothing. Communication with it is done using the SCL stack, with functions such as `PushN` and `PopN`.

The table of inbuilt SCL functions is defined as an array called `C_SCLFunctions`. Indices into this array are defined as `E_SCL` followed by the capitalized name of the SCL instruction as it appears in a SCL program. For example, the index for SCL function `moveto` is `E_SCLMOVETO`.

Where the name contains ‘awkward’ characters (for example `vis?` or `!=`) the name is constructed by substituting the following characters:

For	Substitute
?	Q
+	PLUS
-	MINUS
=	EQ
<	LESS
>	GREAT
!	NOT
~	TILDE
&	AND
|	OR
^	XOR
*	STAR
/	SLASH
%	MOD

For example, the index for `vis?` is `E_SCLVISQ`, for `<<` the index is `E_SCLLESSLESS`, and for `!=` the index is `E_SCLNOTEQ`.

There are two cases where an ambiguity arises, namely `*` and `-`. These have special cases as follows:

i. `*` as in `*Pointer` is `E_SCLSTAR`, as expected, but
`*` as in `Width*Height` is `E_SCLMULTIPLY`.

ii. `-` as in `Credit-Debit` is `E_SCLMINUS`, as expected, but
`-` as in `-Debit` is `E_SCLNEGATE`.

Some SCL functions are deliberately not made available. These are:

i. Functions that exist solely as aliases for constants, such as `true` and `false`.

ii. Functions that affect flow of control, such as `if...else...then`.

iii. Functions that define variables or other storage, such as `alloc` and variable declarations.

iv. `resume`

It is possible to use `waitf` (and `waitfs`, `stopf`, `resetf` and `waitmsg`), but their effect of splitting execution over several frames is not evident within the directly called function sequence. If they are called within a registered SCL function, their effect becomes apparent after the execution of that whole new instruction. The reason for this is that the SCL executor handles the flow-of-control issues in these cases, and this does not have a chance to act on the effects of the functions until control returns to it.

To pass arguments to an SCL function, you must push them onto the SCL stack. If you are calling the function from anywhere except inside a registered SCL function, you must first clear the SCL stack by setting the stack pointer to 0 as follows:

```
*C_SCLSP=0;
```

Arguments can then be pushed in the same order in which they appear in the SCL code, before calling the function itself. Any return value can be popped from the stack when the function returns.

This example simulates the SCL code `moveby(0,0,1000,me);`

```
PushN(E_SSINTEGER,0);
PushN(E_SSINTEGER,0);
PushN(E_SSINTEGER,1000);
C_SCLFunctions[E_SCLME]();      /* Remember the () brackets */
C_SCLFunctions[E_SCLMOVEBY]();
```

Note that all the checking for stack overflow and underflow and other errors is done by the SCL executor, not by the individual functions. It is therefore up to you to check for these conditions. The lack of these checks, and of the overhead of decoding the SCL instructions means that direct calls to SCL are considerably faster than the execution of compiled SCL code. However, they are still slower than calls to 'real' C code, due to the overheads of pushing and popping arguments and return values.

For registered SCL functions, there is a separate table, `C_RegSCLFunctions`. This is indexed using the lower 8 bits of the code returned by `RegisterSCL`. For example, if the registration function return code was stored in `SCLCode`, you could call the registered SCL function directly using:

```
C_RegSCLFunctions[SCLCode&0xFF]();
```

In this case, however, you would almost certainly be able to call the function by other means. This would be clearer, and is much preferred over the above example.

Chapter 3 - Vectors

The vector table

The application is passed the address of the vector table contained in the main VRT program as it enters the system. The pointer to this table is called `API_Vectors` and is accessible by an application's code.

The application header file APP_DEFS.H contains many useful definitions which you should use in preference to accessing the vector table directly, as they aid readability in the application source code. For example, you can set the initialization vector using:

```
VecB4Init=MyCode;
```

or:

```
(API_Vectors->_VecB4Init)=MyCode;
```

The first form is preferable, if only to save the amount of typing involved in the second.

The standard memory management routines `calloc`, `malloc`, `realloc` and `free` have been redirected through the vector table. Therefore, you should use only these routines for memory management (as opposed to using `_nfree` directly, for instance). In most applications this will be entirely invisible.

VRT vectors

The application may intercept a set of vectors called by the main program (by default they point to a routine which does nothing), installing its own code to be executed by the main program. The overall system can be represented as follows—vectors are shown in bold:

```
Cold start VRT program
VecB4Init
Initialize data
VecAFInit
while(select browser - Visualiser or Viscape)
{
  VecVisEntry
  while(not exited from browser)
  {
        VecB4GetFunc
        Get functions from control devices
        VecAFGetFunc
        for(each set of functions)
        {
            VecB4DoFunc
            Do control functions
            VecAFDoFunc
        }
        VecB4Update
        Update world
        VecB4SCL
        Execute SCL programs
        VecB4Render
        for(each window on the screen)
        {
            VecB4Process
            Process objects into drawing list
            VecB4Sort
            Sort drawing list
            VecB4Instrs
            Draw instruments for this window
            if(time to swap screens)
            {
                VecB4ScreenSwap
                Swap screens
            }
            VecB4Draw
            Draw drawing list
            VecAFDraw
        }
        VecAFRender
  }
  VecVisExit
}
Exit VRT program
```

In addition to the main vectors, there are some expansion vectors which are called when the main program comes across something it does not recognize such as an undefined instrument type. The application attempts to recognize and deal with these undefined items.

A copy of all these vectors is made by the APP_LIB module, allowing the routine which intercepts the vector to chain to its previous owner. These are identified by the word ‘Old’ added to the front of the vector name. For example, the old vector `VecB4Init` is called `OldVecB4Init`.

The syntax for the vector functions on the following pages shows the short form of the function as defined in APP_DEFS.H, as if it were a function prototype. The actual name of the element within the structure is the same as the function name preceded by an underscore. For example `VecB4Init` has the corresponding element `_VecB4Init`.

The vectors that VRT supports are listed on the following pages.

VecAFDoFunc

Syntax:	**`void __vrtcall VecAFDoFunc(void);`**
Description:	Called just after functions from the control devices are acted upon. This allows modification of the state of the world before it is actually updated.
Old vector:	`OldVecAFDoFunc`
See also:	`VecB4DoFunc`

VecAFDraw

Syntax:	**`void __vrtcall VecAFDraw(void);`**
Description:	Called just after the sorted drawing list has been drawn on the background screen. This is part of the rendering process and is called once for each window on the screen.
Old vector:	`OldVecAFDraw`
See also:	`VecB4Draw`

VecAFInit

Syntax:	**`void __vrtcall VecAFInit(void);`**
Description:	Called just after initializing the world data, for example when F12 is pressed.
Old vector:	`OldVecAFInit`
See also:	`VecB4Init`

VecAFGetFunc

Syntax:	**`void __vrtcall VecAFGetFunc(void);`**
Description:	Called just after functions from the control devices are gathered. This vector may be used to modify or filter the functions that arrive from the control devices before they are acted upon.
Old vector:	`OldVecAFGetFunc`
See also:	`VecB4GetFunc`

VecAFRender

Syntax:	**`void __vrtcall VecAFRender(void);`**
Description:	Called just after the entire rendering process is complete.
Old vector:	`OldVecAFRender`
See also:	`VecB4Render`

VecB4DoFunc

Syntax:	**`void __vrtcall VecB4DoFunc(void);`**
Description:	Called just before (B4) functions from the control devices are acted upon.
Old vector:	`OldVecB4DoFunc`
See also:	`VecAFDoFunc`

VecB4Draw

Syntax:	**`void __vrtcall VecB4Draw(void);`**
Description:	Called just before (B4) the sorted drawing list is drawn onto the background screen. For additional graphics within the window, it is more efficient to insert information into the drawing list (which is drawn all in one go) rather than to use the graphics device to draw them directly. This is part of the rendering process, and is called once for each window on the screen.
Old vector:	`OldVecB4Draw`
See also:	`VecAFDraw`

VecB4Init

Syntax:	**`void __vrtcall VecB4Init(void);`**
Description:	Called just before (B4) initializing the world data, for example when F12 is pressed.
Old vector:	`OldVecB4Init`
See also:	`VecAFInit`

VecB4GetFunc

Syntax:	**`void __vrtcall VecB4GetFunc(void);`**
Description:	Called just before (B4) functions from the control devices are gathered.
Old vector:	`OldVecB4GetFunc`
See also:	`VecAFGetFunc`

VecB4Instrs

Syntax:	**`void __vrtcall VecB4Instrs(void);`**
Description:	Called just before (B4) the instruments are drawn. This vector is part of the rendering process, and is called once for each window on the screen.
Old vector:	`OldVecB4Instrs`

VecB4Process

Syntax:	**`void __vrtcall VecB4Process(void);`**
Description:	Called just before (B4) the world is processed into an unsorted drawing list. This is part of the rendering process, and is performed once for each window on the screen.
Old vector:	`OldVecB4Process`

VecB4Render

Syntax:	**`void __vrtcall VecB4Render(void);`**
Description:	Called just before (B4) the screen is actually rendered. This allows last-minute changes to the world after the update routine and SCL have processed it.
Old vector:	`OldVecB4Render`
See also:	`VecAFRender`

VecB4SCL

Syntax: **`void __vrtcall VecB4SCL(void);`**

Description: Called just before (B4) the SCL programs on all the objects are executed. This allows modification of the updated attributes (such as positions, velocities) before the SCL programs have to look at them.

Old vector: `OldVecB4SCL`

VecB4ScreenSwap

Syntax: **`void __vrtcall VecB4ScreenSwap(void);`**

Description: Called just before (B4) the background screen (the one the rendered picture is drawn onto) and the foreground screen (the one being displayed) are swapped over, or the background screen is copied onto the foreground.

Old vector: `OldVecB4ScreenSwap`

VecB4Sort

Syntax: **`void __vrtcall VecB4Sort(void);`**

Description: Called just before (B4) the drawing list is sorted. This is part of the rendering process, and is called once for each window on the screen.

Old vector: `OldVecB4Sort`

VecB4Update

Syntax: **`void __vrtcall VecB4Update(void);`**

Description: Called just before (B4) the world data is updated for this frame. The update consists of actually moving the objects, taking care of animations, angular velocities, and any other attributes.

Old vector: `OldVecB4Update`

VecVisEntry

Syntax:	**`void __vrtcall VecVisEntry(void);`**
Description:	Called as Visualiser is entered, allowing Visualiser specific routines to be triggered.
Old vector:	`OldVecVisEntry`
See also:	`VecAFInit`,`VecVisExit`

VecVisExit

Syntax:	**`void __vrtcall VecVisExit(void);`**
Description:	Called as Visualiser is exited, allowing Visualiser specific routines to be cleared.
Old vector:	`OldVecVisExit`
See also:	`VecVisEntry`

UnknownFunction

Syntax:	**`void __vrtcall UnknownFunction (long FuncNum);`**
Description:	Called when a control function is not recognized. It may be used to trigger an action from the application on a key press or icon depression. Unrecognized functions include those in the range 0x4600F000 to 0x4600FFFF, and any which have a top byte not equal to `0x46` (‘F’ for function). Placing one of these function numbers on a key alerts the application through this vector. The actual function number is passed in `FuncNum`. If this application does not recognize the unknown function number, it should call the old vector, allowing any other applications to check if they recognize it.
Old vector:	`OldUnknownFunction`
See also:	`UnknownInstr`, `UnknownDialItem`, `UnknownWChunk`, `UnknownSChunk`

UnknownInstr

Syntax:	**`void __vrtcall UnknownInstr (T_INSTRUMENT *Ins);`**
Description:	Called when an instrument type is not recognized. A pointer to the instrument in question is passed in `Ins`, allowing the application to read its value and draw an appropriate representation on the screen. If this application does not recognize the unknown instrument type, it should call the old vector, allowing any other applications to check if they recognize it.
Old vector:	`OldUnknownInstr`
See also:	`UnknownFunction`, `UnknownDialItem`, `UnknownWChunk`, `UnknownSChunk`

UnknownDialItem
UnknownSChunk
UnknownWChunk

Reserved for future expansion.

Chapter 4 - VRT functions

The application is passed the address of the vector table contained in the main VRT program as it enters the system. The pointer to this table is `API_Vectors` and is accessible to the application's code. It includes pointers to many useful VRT functions.

The application header file APP_DEFS.H contains many useful definitions which you should use in preference to accessing the vector table directly, as they aid readability in the application source code. For example, you can get the address of an object's data using:

```
ChunkAdd(Object,E_CTSTANDARD);
```

or:

```
(API_Vectors->_ChunkAdd)(Object,E_CTSTANDARD);
```

The first form is preferable, if only to save the amount of typing involved in the second.

The standard memory management routines `calloc`, `malloc`, `realloc` and `free` have all been redirected through the vector table. Therefore, you should only use these routines for memory management (as opposed to using `_nfree` directly, for instance). In most applications this is entirely invisible.

VRT functions

The functions that point to the main program are listed on the following pages. The syntax shows the short form as defined in APP_DEFS.H, as if it were a function prototype. The actual name of the element within the structure is the same as the function name preceded by an underscore. For example, for `ClearScreen` the corresponding element is `_ClearScreen`).

Some functions defined in APP_DEFS (such as `PopupDialogue()` and `EditScreen()`) have been retained for backwards compatibility with earlier versions of VRT and are not documented on the following pages.

The functions `bsearch`, `qsort` and `ProcessFunctions` are not declared as `__vrtcall` in the vector definitions. You can access these routines directly by linking with the compiler runtime library rather than through the vectors.

AbsPosition

Syntax: **`void AbsPosition(long *x, long *y, long *z, short Object, char Lock, short *xr, short *yr, short *zr);`**

Description: Works out the absolute position and rotation of a given point within an object. Pointers `x`, `y` and `z` point to `long` variables containing the offset of the required point from the origin of the object. `Object` is the object number of the object.

`Lock` sets which rotations are to be ignored (like viewpoint locks):

bit 0: Ignore Z

bit 1: Ignore Y

bit 2: Ignore X

The absolute position of the point is filled in `x, y, z` and the totalled rotations of the object are filled in `xr, yr, zr`.

See also: `FindBearing`, `FindOffset`

Example: `AbsPosition(&x, &y, &z, 222, 0, &xr, &yr, &zr);`

AddDrawChunk

Syntax: **`void AddDrawChunk(short Type, void *Data, short Length, char Flush, short Buffer);`**

Description: Adds `Length` bytes of data at `Data` to the current drawing list as chunk type `Type`. If `Flush` is non-zero, the list is sent to the graphics card and restarted.

`Buffer` The data can be stored in one of three buffers that dictate when it is drawn on screen:

`E_DRAWBUFFER` The main buffer that holds all object data.

`E_PREDRAWBUFFER` The data is drawn first before that in `E_DRAWBUFFER`.

`E_POSTDRAWBUFFER` The data is drawn after that in `E_DRAWBUFFER`.

See also: `AddRectangle`, `AddText`

Example: `AddDrawChunk(E_DCFACET,&FacData,20,0);`

AddRectangle

Syntax: **`void AddRectangle(short x1, short y1, short x2, short y2, unsigned char Colour, unsigned char Type);`**

Description: Adds a rectangle to the current drawing list. Its position and size are specified by the top left corner (`x1`, `y1`) and the bottom right corner (`x2`, `y2`). `Colour` is the color in which to draw the rectangle, and `Type` is the rectangle type to be used, either `E_DCRECT` or `E_DCPALRECT`.

See also: `AddDrawChunk`, `AddText`

Example:

```
AddRectangle(0,0,639,479,E_COLWHITE,E_DCRECT);
```

AddText

Syntax:

```
void AddText(short x, short y, unsigned char FGColour,
  unsigned char BGColour, short FontID, short MaxLen,
  short Align, unsigned char Flags, char *String);
```

Description: Adds some text to the current `Drawing` list. Draws the (null terminated) text in `String` on to the screen at position (`x`, `y`). Foreground and background colors are specified in `FGColour` and `BGColour` respectively, and the maximum number of characters to display is in `MaxLen`. If this is -1, all characters in the string are displayed.

`FontID` specifies the number of the font to use:

0	8x8 (wide) font
2	16x16 (large) font
4	6x6 (small) font
6	6x8 (normal) font
10	6x9 (standard) font

`Flags` specifies which screen(s) to display the string on:

0	Neither
1	Foreground only
2	Background only
3	Both

`Align` sets the relative alignment of the string to the start position (`x`,`y`) as follows:

0	Top left	1	Top center	2	Top right
4	Center left	5	Center	6	Center right
8	Bottom left	9	Bottom center	10	Bottom right

See also: `AddDrawChunk, AddRectangle, Text`

Example:

```
AddText(320,256,E_COLWHITE,E_COLBLACK,0,-1,5,2,
    ”Middle of background screen”);
```

Alert

Syntax: **`long Alert(char *String, short Type, short Buttons);`**

Description: Brings up an alert box displaying the string `String`. `Type` is a value specifying the type of alert box and can be one of the following:

`E_ALERTINFO`	An information box
`E_ALERTWARNING`	A warning box
`E_ALERTERROR`	An error box

`Buttons` determines which buttons to display, and can be one of the following:

`E_ALERTOK`	OK
`E_ALERTCANCEL`	Cancel
`E_ALERTOKDEFCAN`	OK + Cancel, OK default
`E_ALERTOKCANDEF`	OK + Cancel, Cancel default
`E_ALERTOKCANCON`	OK + Cancel + Continue, OK default
`E_ALERTSAVE`	Save + Discard + Cancel, Save default
`E_ALERTREMAP`	Remap + Leave + Cancel, Remap default

The return value may be one of the following:

`E_DROK`	OK selected
`E_DRCANCEL`	Cancel selected
`E_DRCONTINUE`	Continue selected

ArcCos

Syntax:	**`short ArcCos(short Value);`**
Description:	Returns the angle (in brees) for which `Value` is the normalized cosine (cosine*16384).
See also:	`ArcSin`, `ArcTan`, `Sin`, `Cos`
Example:	`a=ArcCos(0x2000);`

ArcSin

Syntax:	**`short ArcSin(short Value);`**
Description:	Returns the angle (in brees) for which `Value` is the normalized sine (sine*16384).
See also:	`ArcCos`, `ArcTan`, `Sin`, `Cos`
Example:	`a=ArcSin(0x2000);`

ArcTan

Syntax:	**`short ArcTan(long Num, long Denom);`**
Description:	Returns the angle (in brees) for which `Num`/`Denom` is the tangent.
See also:	`ArcSin`, `ArcCos`, `Sin`, `Cos`
Example:	`a=ArcTan(YDiff,XDiff);`

AscToLong

Syntax: **`char *AscToLong(long *n, char *String, short Base, char MaxChar);`**

Description: Converts ASCII representation of number at `String` into a number in base `Base`. This is placed at `n`, and `MaxChar` is the maximum character allowed in the number—for example ‘F’ for base 16. Returns a pointer to the first invalid character in the string.

See also: `BinToLong`, `DecToLong`, `HextoLong`, `LongToAsc`

Example: `p=AscToLong(&Num,”1F4E”,16,’F’);`

BinToLong

Syntax: **`char *BinToLong(long *n, char *String);`**

Description: Converts ASCII representation of binary number at `String` into a number. This is placed at `n`. Returns a pointer to the first invalid character in the string.

See also: `AscToLong`, `DecToLong`, `HexToLong`, `LongToBin`

Example: `p=BinToLong(&Num,”101101001");`

BLoad

Syntax: **`long BLoad(char *FileName, void *LoadAddress, long MaxLength);`**

Description: Loads a binary file, called `FileName`, to `LoadAddress`, loading at most `MaxLength` bytes. Returns the actual number of bytes loaded, or a negative error number. These errors are passed back directly from C’s file handling routines.

See also: `BSave`, `GetFileLength`, `GetFileList`

Example: `Len=BLoad(“DATA.DAT”,Buffer,10000);`

BSave

Syntax: **`long BSave(char *FileName, void *SaveAddress, long Length);`**

Description: Saves a binary file, called `FileName`, from `SaveAddress`, saving at most `Length` bytes. Returns the actual number of bytes saved, or a negative error number.

See also: `BLoad`, `GetFileLength`, `GetFileList`

Example: `Len=BSave("DATA.DAT",Buffer,10000);`

CheckRepeat

Syntax: **`void CheckRepeat(long Function, short Param);`**

Description: Checks if the function `Function` can be repeated with the center mouse button, and if so enters it and `Param` (the proportional device parameter) so that this is in fact done.

Example: `CheckRepeat(FuncNum,PropParam);`

ChoosePrinter

Syntax: **`void ChoosePrinter(void);`**

Description: Allows the user to choose from a list of available printers.

See also: `PrintScreen`

Example: `ChoosePrinter();`

ChunkAdd

Syntax: **`T_WORLDCHUNK *ChunkAdd(short Object, unsigned short ChunkType);`**

Description: Returns a pointer to the chunk of type `ChunkType` on object `Object`, or `NULL` if it does not exist. Valid values for `ChunkType` are:

E_CTANGVELS	E_CTLIGHTSOURCE
E_CTANIMATIONS	E_CTLITCOLS
E_CTANIMCOLS	E_CTROTATIONS
E_CTATTACHMENTS	E_CTSCL
E_CTBENDING	E_CTSCLGLOBAL
E_CTBUBBLE	E_CTSCLLOCAL
E_CTCOLLISION	E_CTSHOOTVEC
E_CTCOLOURS	E_CTSNDTRANS
E_CTDEFCOLS	E_CTSORTING
E_CTDEFLITCOLS	E_CTSPRTRANS
E_CTDISTANCE	E_CTSTANDARD
E_CTDYNAMICS	E_CTTEXTINFO
E_CTINITPOS	E_CTTEXTURES
E_CTINITSIZE	E_CTVIEWPOINT
E_CRPROPERTIES	E_CTTEXTCOORDS
E_CTAUTOSOUND	E_CTENTITY
E_CTORIGINALCOL	

Example: `p=ChunkAdd(222,E_CTROTATIONS);`

ClearMessages

Syntax: **`long ClearMessages(void);`**

Description: Clears the current message file from memory.

See also: `LoadMessages`, `SaveMessages`, `ClearShape`, `ClearWorld`, `ClearUserSprites`, `ClearUserRsrc`, `ClearPalette`, `ClearSounds`

Example: `Error=ClearMessages();`

ClearPalette

Syntax:	**`long ClearPalette(void);`**
Description:	Replaces the current palette file in memory with the default.
See also:	`LoadPalette`, `SavePalette`, `ClearShape`, `ClearWorld`, `ClearUserSprites`, `ClearUserRsrc`, `ClearMessages`, `ClearSounds`
Example:	`Error=ClearPalette();`

ClearScreen

Syntax:	**`void ClearScreen(void);`**
Description:	Clears the background screen, ready to be drawn on by `Draw`.
See also:	`Draw`, `ClearWindow`
Example:	`ClearScreen();`

ClearShape

Syntax:	**`long ClearShape(void);`**
Description:	Replaces the current shape file in memory with the default.
See also:	`LoadShape`, `SaveShape`, `ClearWorld`, `ClearPalette`, `ClearUserSprites`, `ClearUserRsrc`, `ClearMessages`, `ClearSounds`
Example:	`Error=ClearShape();`

ClearSounds

Syntax:	**`long ClearSounds(void);`**
Description:	Clears the current sound file from memory.
See also:	`LoadSounds`, `SaveSounds`, `ClearShape`, `ClearWorld`, `ClearUserSprites`, `ClearUserRsrc`, `ClearMessages`, `ClearPalette`
Example:	`Error=ClearPalette();`

ClearUserSprites

Syntax:	**`long ClearUserSprites(void);`**
Description:	Clears the current user image file from memory.
See also:	`LoadUserSprites`, `SaveUserRsrc`, `ClearShape`, `ClearPalette`, `ClearWorld`, `ClearMessages`, `ClearSounds`
Example:	`Error=ClearUserSprites();`

ClearUserRsrc

Syntax:	**`long ClearUserRsrc(void);`**
Description:	Clears the current user resource file from memory.
See also:	`LoadUserRsrc`, `SaveUserRsrc`, `ClearShape`, `ClearPalette`, `ClearWorld`, `ClearUserSprites`, `ClearMessages`, `ClearSounds`
Example:	`Error=ClearUserRsrc();`

ClearWindow

Syntax:	**`void ClearWindow(void);`**
Description:	Clears the currently selected window on the background screen.
See also:	`Draw`, `ClearScreen`
Example:	`ClearWindow();`

ClearWorld

Syntax:	**`long ClearWorld(void);`**
Description:	Replaces the current world file in memory with the default.
See also:	`LoadWorld`, `SaveWorld`, `ClearShape`, `ClearPalette`, `ClearUserSprites`, `ClearUserRsrc`, `ClearMessages`, `ClearSounds`
Example:	`Error=ClearWorld();`

CloseDialogue

Syntax: **short CloseDialogue(char * DialName);**

Description: Since resident dialogs persist until an exit code is returned from them, it is sometimes necessary to close them down if an exceptional circumstance occurs. `CloseDialogue` closes any open dialog with the name `DialName`. It returns non-zero if successful, or 0 if no dialog of that name could be found.

Example:

```
Dialogue("MY-DIALOGUE");
If(Error)
{
   CloseDialogue("MY-DIALOGUE");
    Return(E-ERROR);
}
```

Compile

Syntax:

```
long Compile(char *Source, unsigned char *SCL,
     T_OBJSYM *Comments, T_OBJSYM *VarNames);
```

Description: Compiles SCL program at `Source` into compiled code at `SCL`, storing comments at `Comments` and variable names at `VarNames`. Returns length of compiled code, or negative error number.

See also: `DeCompile`, `Execute`

Example: `i=Compile(Source,Buffer,NULL,NULL);`

Conv_CheckBreak

Syntax: **short Conv_CheckBreak (void);**

Description: Checks if the user has pressed the ESC key to cancel the current conversion. This function must be called if you wish to allow the user to cancel the data conversion. It should be called within tight loops to prevent long delays before a cancel request is acknowledged. This function also handles the display of the Mr Twirly image.

See also: `Conv_ElementAllocMem`, `Conv_ElementReallocMem`, `Conv_ElementDeallocMem`, `Conv_CleanUpElement`, `Conv_InitElement`

Example:

```
if(ConvCheckBreak()==E_ERROR)
  return;
```

Conv_CleanUpElement

Syntax: **`void Conv_CleanUpElement(void);`**

Description: Cleans up any element specific memory buffers allocated using `Conv_ElementAllocMem` and `Conv_ElementReallocMem`. This function should be called after `Conv_Element` to ensure that all buffers are cleared and any memory allocated is freed. This function is also called in the event of a fatal error, or if the user cancels the current conversion during element processing.

See also: `Conv_ElementAllocMem`, `Conv_ElementReallocMem`, `Conv_ElementDeallocMem`, `Conv_InitElement`

Example:

```
Conv_Element(pElement);
Conv_CleanUpElement();
```

Conv_Element

Syntax: **`long Conv_Element (T_ELEMENT *pElement);`**

Description: Converts the facet list at `C_CNV3DFace_List` into a series of shapes according to the cut flags. During this processing, the facet direction calculations are performed on the resulting shapes and a check for likely associations within the individual shapes is preformed. Returns `E_OK` if the conversion is successful, or `E_ERROR` if it fails.

Example:

```
Conv_Element (pElement)
```

Conv_ElementAllocMem

Syntax: **`void *Conv_ElementAllocMem(size_t size, long Length);`**

Description: Allocates a memory buffer which is related to the element currently being processed. This memory buffer is automatically deallocated upon a fatal error, a user cancel request or upon calling `Conv_CleanUpElement` at the end of element processing.

See also: `Conv_ElementReallocMem`, `Conv_ElementDeallocMem`, `Conv_CleanUpElement`, `Conv_InitElement`

Example:

```
if((Buffer=Conv_ElementAllocMem (sizeof(char),200))==NULL)
    return(E_ERROR);
```

Conv_ElementDeallocMem

Syntax: **void Conv_ElementDeallocMem(void *p);**

Description: Deallocates a memory buffer which was allocated by Conv_ElementAllocMem.

See also: Conv_ElementAllocMem, Conv_ElementReallocMem, Conv_ElementDeallocMem, Conv_CleanUpElement, Conv_InitElement

Example:

```
Buffer=Conv_ElementAllocMem (sizeof(short), 100);
if(Buffer!=NULL)
  {
    .....
    .....
    Conv_CheckBreak();
    .....
    .....
    Conv_ElementDeallocMem(Buffer);
  }
```

Conv_ElementReallocMem

Syntax: **void *Conv_ElementReallocMem(void *p, size_t size);**

Description: Reallocate a buffer previously allocated using Conv_ElementAllocMem. Use this to reallocate any buffers allocated with Conv_ElementAllocMem as it updates the state of the buffer record in VRT allowing a clean deallocation upon calling Conv_CleanUpElement. Note if this function is called with a NULL pointer, it calls Conv_ElementAllocMem.

See also: Conv_ElementAllocMem, Conv_ElementDeallocMem, Conv_CleanUpElement, Conv_InitElement

Example:

```
if((Buffer=Conv_ElementReallocMem(NULL,
   sizeof(char)*200))==NULL)
     return;
```

Conv_Fatal

Syntax: **`void Conv_CheckBreak(char *Error);`**

Description: Report a fatal error to the user. VRT deallocates any current element specific memory buffers, and any known conversion specific memory buffers. Then it returns to the calling function to handle any cleanup of its own before returning to VRT by returning `E_ERROR`.

See also: `Conv_ElementAllocMem`, `Conv_ElementReallocMem`, `Conv_ElementDeallocMem`, `Conv_CleanUpElement`, `Conv_InitElement`

Example:

```
Conv_Fatal(“Out of Memory”);
  return(E_ERROR);
```

Conv_FindElement

Syntax: **`T_ELEMENT *Conv_FindElement(char *Name);`**

Description: Gets a pointer to a `T_ELEMENT` structure in the current list. Takes a valid element name, and returns a pointer to the element, or `NULL` if it is not found.

Example:

```
if((pElement=Conv_FindElement(“FREE”))==NULL)
  return;
```

Conv_InitElement

Syntax: **`void Conv_InitElement(void);`**

Description: Prepares the VRT side of the converter for the processing of a new element. Clears any existing facet list pointed to by `C_CNV3DFace_List` and prepares the memory allocation functions `Conv_ElementAllocMem` and `Conv_ElementReallocMem`.

See also: `Conv_ElementAllocMem`, `Conv_ElementReallocMem`, `Conv_ElementDeallocMem`, `Conv_CleanUpElement`

Example:

```
Conv_InitElement();
```

Conv_MakeWorldObject

Syntax: **T_STANDARD *Conv_MakeWorldObject (T_ELEMENT *Element);**

Description: Creates an instance in the world of a complete and processed element. This must not be called before `Conv_Element` is called for the same element structure. Before this function is called you must fill in the `Insert_X/Y/Z`, `Scale_X/Y/Z`, `Rotation_X/Y/Z` and if required the `ECS_X/Y/Z` values in the `T_ELEMENT` structure. If the layer to be used for coloring is different to the standard layer then the `Layer` entry must be filled in too. You may also select to change the `Colour` entry in the `T_ELEMENT` structure so that any facets contained within the element that are to be colored `BYELEMENT`, adopt the new color. This function creates a complete instance including any subobject tree, and children. To create the children `Conv_MakeWorldObject` actually calls itself, therefore it is not advisable to fill the instance information in any but the current element as it may be overwritten by this function.

Example:

```
if(Conv_Element(Element)==E_OK)
  {
    Element->Insert_X=0;
    Element->Insert_Y=100;
    Element->Insert_Z=0;
    Element->Scale_X=
    Element->Scale_Y=
    Element->Scale_Z=1;
    Element->Rotation_X=0;
    Element->Rotation_Y=5;
    Element->Rotation_Z=0;
    Element->ECS_X=Element- >ECS_Y=0;
    Element->ECS_Z=1;
    Element->Colour=256;
    strcpy(Element->Layer,“Standard”);
    Conv_MakeWorldObject(Element);
  }
```

Conv_Message

Syntax: **void Conv_Message(char *Message);**

Description: Prints a message about the conversion in the Message field of the progress dialog. Useful for warning the user.

Example:

```
Conv_Message(“Entity not supported”);
```

Conv_NewChild

Syntax: **`T_CHILD *Conv_NewChild(char *ElementName);`**

Description: Adds a new `T_CHILD` structure to the end of the list on the specified element. Automatically updates the `NumChildren` entry on the element as necessary. Returns a pointer to the child structure.

See also: `Conv_RemoveChild`

Example: `Child=Conv_NewChild(“FREE”);`

Conv_NewElement

Syntax: **`T_ELEMENT *Conv_NewElement(void);`**

Description: Creates a new `T_ELEMENT` structure in the list pointed to by `C_CNVElement_List`. The element counter `C_CNVElements` is automatically updated as necessary and a pointer to the new element is returned, or `NULL` if out of memory.

See also: `Conv_RemoveElement`

Example: `Element=Conv_NewElement();`

Conv_NewFacet

Syntax: **`T_CNVFACET *Conv_NewFacet(void);`**

Description: Adds a facet to the list at `C_CNV3DFace_List`. The facet counter `C_CNV3DFace_Cnt` is automatically updated, and a pointer to the new facet returned, or `NULL` if out of memory.

See also: `Conv_RemoveFacet`

Example:

```
T_CNVFACET *Facet;
Facet=Conv_NewFacet();
if(Facet!=NULL)
{
    Facet->NumPoints=4;
    ......
}
```

Conv_NewLayer

Syntax: **`T_LAYER *Conv_NewLayer(void);`**

Description: Adds a layer to the list at `C_CNVLayers`. The layer counter `C_CNVLayer_Cnt` is automatically updated, and a pointer to the new layer returned, or `NULL` if out of memory.

See also: `Conv_RemoveLayer`

Example:

```
T_CNVLAYER *Layer;
Layer=Conv_NewLayer();
```

Conv_RemapColour

Syntax:

```
long Conv_RemapColour(unsigned char r, unsigned char g,
    unsigned char b, unsigned char a);
```

Description: Returns the palette index of the nearest color to the Red, Green, Blue Alpha value specified. Red, green and blue values are between 0–255; alpha is a transparency value, 0 is totally opaque, 255 is totally transparent.

Example: `Element->Colour=Conv_RemapColour(255,255,255,0);`

Conv_RemoveChild

Syntax: **`void Conv_RemoveChild(char *ElementName, long Index);`**

Description: Removes a child from the list and the specified element. Takes the name of the element on which to remove the child and an array index into the list of children. Automatically updates the `NumChildren` entry on the element as necessary.

See also: `Conv_NewChild`

Example:

```
Conv_RemoveChild(“FREE”, Element->NumChildren);
  /*Remove last*/
```

Conv_RemoveElement

Syntax: **void Conv_RemoveElement(char *ElementName);**

Description: Removes an element from the list. Takes a pointer to the name of the element. Automatically updates the element counter `C_CNVElements` as necessary. Any instances of the specified element as children of other existing elements can be eliminated using `Conv_RemoveChild`.

See also: `Conv_NewElement`

Example:
```
Conv_RemoveElement(“FREE”);
  /*Remove FREE element*/
```

Conv_RemoveFacet

Syntax: **void Conv_RemoveFacet(long Index);**

Description: Removes a facet from the list. Takes the array index into the list of the facet. Automatically updates the facet counter `C_CNV3DFace_Cnt` as necessary.

See also: `Conv_NewFacet`

Example:
```
Conv_RemoveFacet(*C_CNV3DFace_Cnt);
 /*Remove last*/
```

Conv_RemoveLayer

Syntax: **void Conv_RemoveLayer(long Index);**

Description: Removes a layer from the list. Takes the array index into the list of the layer. Automatically updates the layer counter `C_CNVLayer_Cnt` as necessary.

See also: `Conv_NewLayer`

Example:
```
Conv_RemoveLayer(*C_CNVLayer_Cnt);
  /*Remove last*/
```

Conv_UpdateProgress

Syntax: **`void Conv_UpdateProgress(void);`**

Description: Updates the details on the progress report dialog. The values relevant to the converter module are `C_CNVElement_NAME`, which contains the name of the item currently being processed, `C_CNVPercentage_Complete`, which is the value to show in the percentage complete slider, and `C_CNV3DFace_Cnt` which is the number of three dimensional faces in the item currently being processed. If you are using `Conv_NewFacet` and `Conv_RemoveFacet` the latter should be automatically handled.

Example:

```
*C_CNVPercentage_Complete=(Total/100)*Current;
  Conv_UpdateProgress();
```

CreateTextBlock

Syntax:

```
void * CreateTextBlock(char *Title, char *Text, long MaxLen,
    unsigned short Flags);
```

Description: Creates an editable text page control. `Title` sets the title of the dialog to display (the title of the control is not actually displayed). `Text` is a pointer to the text buffer. `MaxLen` specifies the maximum length of the string which must be zero-terminated. Leave `Flags` as 0.

See also: `EditTextBlock`, `SetTextBlockCursor`, `SetTextBlockErrorStrings`, `DestroyTextBlock`

Example:

```
Handle=CreateTextBlock(“GetText”,Buffer,32000,0);
```

CreateVRTDial

Syntax: **`short CreateVRTDial(char *pBuffer, char *pName);`**

Description: Creates a new user dialog or updates an existing one. `pBuffer` points to an area of memory containing a complete dialog box definition. `pName` is not used in VRT 4-00 or later—set to NULL. Returns a positive value if successful, or a negative error number.

See also: `CreateVRTIcon`, `CreateVRTInstrument`, `CreateVRTObject`, `CreateVRTPalette`, `CreateVRTShape`, `CreateVRTSound`, `CreateVRTSprite`, `CreateVRTWindow`, `DeleteDial`, `DeleteVRTDial`

CreateVRTIcon

Syntax:

```
short CreateVRTIcon(T_ICON *pBuffer, short IconTable,
    short IconNum);
```

Description: Creates a new icon or replaces an existing one. `pBuffer` points to a `T_ICON` structure containing a complete icon definition. `IconTable` is the number of the icon table in which to insert the new icon. `IconNum` is the number of the icon to replace, or -1 if a new one is to be created. Returns the number of the created icon, or a negative error number.

See also: `CreateVRTDial`, `CreateVRTInstrument`, `CreateVRTObject`, `CreateVRTPalette`, `CreateVRTShape`, `CreateVRTSound`, `CreateVRTSprite`, `CreateVRTWindow`, `DeleteVRTIcon`

CreateVRTInstrument

Syntax:

```
short CreateVRTInstrument(T_INSTRUMENT *pBuffer,
  short Console,short InstNum);
```

Description: Creates a new instrument or replaces an existing one. `pBuffer` points to a `T_INSTRUMENT` structure containing a complete instrument definition. `Console` is the number of the console (window) in which to insert the new instrument. `InstNum` is the number of the instrument to replace, or `-1` if a new one is to be created. Returns the number of the created instrument, or a negative error number.

See also: `CreateVRTDial`, `CreateVRTIcon`, `CreateVRTObject`, `CreateVRTPalette`, `CreateVRTShape`, `CreateVRTSound`, `CreateVRTSprite`, `CreateVRTWindow`, `DeleteVRTInstrument`

CreateVRTObject

Syntax: **`short CreateVRTObject(char *pBuffer, char *pName, short ObjNum);`**

Description: Creates a new object or replaces an existing one. `pBuffer` points to a buffer containing a complete object definition. `pName` points to a name for the object. No checking is done to ensure that names are unique. `ObjNum` is the number of the object to replace, or `-1` if a new object is to be created. Returns the number of the created object, or a negative error number. The new object is placed in the world according to the position in its standard attributes chunk, and then relinked. If replacing an existing object, the existing object is unlinked first.

See also: `CreateVRTDial`, `CreateVRTIcon`, `CreateVRTInstrument`, `CreateVRTPalette`, `CreateVRTShape`, `CreateVRTSound`, `CreateVRTSprite`, `CreateVRTWindow`, `DeleteVRTObject`

CreateVRTPalette

Syntax: **`short CreateVRTPalette(char *pBuffer, short PalNum);`**

Description: Creates a new palette or replaces an existing one. `pBuffer` points to a buffer containing a palette chunk, a stipples chunk, and a ranges chunk, terminated by `0xFFFF`. `PalNum` is the palette number to replace, or `-1` to create a new palette. Returns the index number of the palette created, or a negative error number.

See also: `CreateVRTDial`, `CreateVRTIcon`, `CreateVRTInstrument`, `CreateVRTObject`, `CreateVRTShape`, `CreateVRTSound`, `CreateVRTSprite`, `CreateVRTWindow`, `DeleteVRTPalette`

CreateVRTShape

Syntax: **`short CreateVRTShape(char *pBuffer, char *pName, short ShpNum);`**

Description: Creates a new shape or replaces an existing one. `pBuffer` points to a buffer containing a complete shape definition. `pName` points to a name for the shape. No checking is done to ensure that names are unique. `ShpNum` is the number of the shape to replace, or -1 if a new shape is to be created. Returns the number of the created shape, or a negative error number.

See also: `CreateVRTDial`, `CreateVRTIcon`, `CreateVRTInsturument`, `CreateVRTObject`, `CreateVRTPalette`, `CreateVRTSound`, `CreateVRTSprite`, `CreateVRTWindow`, `DeleteVRTShape`

CreateVRTSound

Syntax: **`short CreateVRTSound(T_SOUNDREC *pBuffer, char *pName, short SndNum);`**

Description: Creates a new sound or replaces an existing one. `pBuffer` points to a `T_SOUNDREC` structure containing a complete sound definition. `pName` points to a name for the sound. No checking is done to ensure that names are unique. `SndNum` is the number of the sound to replace, or `-1` if a new sound is to be created. Returns the number of the created sound, or a negative error number.

See also: `CreateVRTDial`, `CreateVRTIcon`, `CreateVRTInstrument`, `CreateVRTObject`, `CreateVRTPalette`, `CreateVRTShape`, `CreateVRTSprite`, `CreateVRTWindow`, `DeleteVRTSound`

CreateVRTSprite

Syntax: **`short CreateVRTSprite(T_GRSPRINST *pBuffer, char *pName, short SprNum);`**

Description: Creates a new sprite or replaces an existing one. `pBuffer` points to a `T_GRSPRINST` structure containing a complete sprite definition. `pName` points to a name for the sprite. No checking is done to ensure that names are unique. `SprNum` is the number of the user sprite to replace, or `-1` if a new one is to be created. Returns the number of the created sprite, or a negative error number.

Note: The sprite is automatically placed in the first (lowest) available slot, unless there are no free slots between 1 and <SprNum> and the specified slot already exists. If there is a free slot the sprite is placed in it.

See also: `CreateVRTDial`, `CreateVRTIcon`, `CreateVRTInstrument`, `CreateVRTObject`, `CreateVRTPalette`, `CreateVRTShape`, `CreateVRTSound`, `CreateVRTWindow`, `DeleteVRTSprite`

CreateVRTSpriteWithPalette

Syntax: **short CreateVRTSpriteWithPalette(T_GRSPRINST *pBuffer, char *pName, short SprNum, unsigned char *pPal);**

Description: Creates a new sprite or replaces an existing one - the sprite will use the specified palette. `pBuffer` points to a `T_GRSPRINST` structure containing a complete sprite definition. `pName` points to a name for the sprite. No checking is done to ensure that names are unique. `SprNum` is the number of the user sprite to replace or -1 is a new one is to be created. `pPal` is a pointer to the palette data. Returns the number of the created sprite, or a negative error number.

See also: `CreateVRTSprite`, `DeleteVRTSprite`

Example:

```
char            *pFileName,*pName;
short           SprNum;
T_IMAGEINFO     Image;
T_GRSPRINST     Sprite;

ImageLoadLibrary();
if(ImageLoad(pFileName,&Image)==0)
{
     Sprite.Width=Image.Width|E_SPRNOSAVE;
     Sprite.Height=Image.Height;
     Sprite.SpriteData=Image.Data;
     SprNum=CreateVRTSpriteWithPalette
          (&Sprite,pName,SprNum,Image.Palette);
     ImageFree(&Image);
}
```

CreateVRTWindow

Syntax: **short CreateVRTWindow(T_CONSOLE *pBuffer, short WinNum);**

Description: Creates a new window. `pBuffer` points to a `T_CONSOLE` structure containing a complete window definition. `WinNum` is the number of the console (the number of the first window in the console) in which to insert the new window. Returns the number of the created window, or a negative error number.

See also: `CreateVRTDial`, `CreateVRTIcon`, `CreateVRTInstrument`, `CreateVRTObject`, `CreateVRTPalette`, `CreateVRTShape`, `CreateVRTSound`, `CreateVRTSprite`, `DeleteVRTWindow`

Decompile

Syntax: **`long Decompile(char *Object, char *Source, char *EndSource, T_OBJSYM *Comments, T_OBJSYM *VarNames);`**

Description: Decompiles SCL object code at `Object` into SCL source code at `Source`, using comments at `Comments` and variable names at `VarNames`. `EndSource` should point to the last byte of the buffer to decompile into. Returns 0 if successful, or a negative error number. If either `Comments` or `VarNames` is `NULL`, they are ignored.

See also: `Compile`, `Execute`

Example: `i=Decompile(Object,Buffer,Buffer+1024,NULL,NULL);`

DecToLong

Syntax: **`char *DecToLong(long *n, char *String);`**

Description: Converts ASCII representation of decimal number at `String` into a number. This is placed at `n`. Returns a pointer to the first invalid character in the string.

See also: `AscToLong`, `BinToLong`, `HexToLong`, `LongToDec`

Example: `p=DecToLong(&Num,”12456");`

DeleteVRTDial

Syntax: **`short DeleteVRTDial(char *pName);`**

Description: Deletes an existing user dialog box. `pName` is the name of the dialog box to delete. Returns 0 if successful, or a negative error value.

See also: `CreateVRTDial`, `DeleteVRTIcon`, `DeleteVRTInstrument`, `DeleteVRTObject`, `DeleteVRTPalette`, `DeleteVRTShape`, `DeleteVRTSound`, `DeleteVRTSprite`, `DeleteVRTWindow`

DeleteVRTIcon

Syntax:	**`short DeleteVRTIcon(short IconTable, short IconNum);`**
Description:	Deletes an existing icon. `IconTable` is the icon table number in which the icon exists, and `IconNum` is the icon number to delete. Returns `0` if successful, or a negative error number.
See also:	`CreateVRTIcon`, `DeleteVRTDial`, `DeleteVRTInstrument`, `DeleteVRTObject`, `DeleteVRTPalette`, `DeleteVRTShape`, `DeleteVRTSound`, `DeleteVRTSprite`, `DeleteVRTWindow`

DeleteVRTInstrument

Syntax:	**`short DeleteVRTInstrument(short Console, short InstNum);`**
Description:	Deletes an existing instrument. `Console` is the console (window) number to which the instrument is attached, and `InstNum` is the instrument number to delete. Returns 0 if successful, or a negative error number.
See also:	`CreateVRTInstrument`, `DeleteVRTDial`, `DeleteVRTIcon`, `DeleteVRTObject`, `DeleteVRTPalette`, `DeleteVRTShape`, `DeleteVRTSound`, `DeleteVRTSprite`, `DeleteVRTWindow`

DeleteVRTObject

Syntax:	**`short DeleteVRTObject(short ObjNum);`**
Description:	Deletes an existing object. `ObjNum` is the object number to delete. Returns 0 if successful, or a negative error number. If an object with children is deleted, all its children are also deleted.
See also:	`CreateVRTObject`, `DeleteVRTDial`, `DeleteVRTIcon`, `DeleteVRTInstrument`, `DeleteVRTPalette`, `DeleteVRTShape`, `DeleteVRTSound`, `DeleteVRTSprite`, `DeleteVRTWindow`

DeleteVRTPalette

Syntax: **`short DeleteVRTPalette(short PalNum);`**

Description: Deletes an existing palette. `PalNum` is the index number of the palette to delete. Returns 0 if successful, or a negative error number. Subsequent palettes are shuffled down by one index number (for example, deleting palette 2 means that palette 3 becomes palette 2, and palette 4 becomes palette 3. Palette 1 is unaffected.)

See also: `CreateVRTPalette`, `DeleteVRTDial`, `DeleteVRTIcon`, `DeleteVRTInstrument`, `DeleteVRTObject`, `DeleteVRTShape`, `DeleteVRTSound`, `DeleteVRTSprite`, `DeleteVRTWindow`

DeleteVRTShape

Syntax: **`short DeleteVRTShape(short ShpNum);`**

Description: Deletes an existing shape. `ShpNum` is the shape number to delete. Returns 0 if successful, or a negative error number. Other shapes in the list are not affected (they keep their existing index numbers).

See also: `CreateVRTShape`, `DeleteVRTDial`, `DeleteVRTIcon`, `DeleteVRTInstrument`, `DeleteVRTObject`, `DeleteVRTPalette`, `DeleteVRTSound`, `DeleteVRTSprite`, `DeleteVRTWindow`

DeleteVRTSound

Syntax: **`short DeleteVRTSound(short SndNum);`**

Description: Deletes an existing sound. `SndNum` is the sound number to delete.

The value you enter for `SndNum` must be the sound number you want to delete +1. For example, if you want to delete sound number 3, `SndNum` must be 4.

Returns 0 if successful, or a negative error number. Other sounds in the list are not affected (they keep their existing index numbers).

See also: `CreateVRTSound`, `DeleteVRTDial`, `DeleteVRTIcon`, `DeleteVRTInstrument`, `DeleteVRTObject`, `DeleteVRTPalette`, `DeleteVRTShape`, `DeleteVRTSprite`, `DeleteVRTWindow`

DeleteVRTSprite

Syntax: **`short DeleteVRTSprite(short SprNum);`**

Description: Deletes an existing user sprite. `SprNum` is the sprite number to delete. Returns 0 if successful, or a negative error number. Other sprites in the list are not affected (they keep their existing index numbers).

See also: `CreateVRTSprite`, `DeleteVRTDial`, `DeleteVRTIcon`, `DeleteVRTInstrument`, `DeleteVRTObject`, `DeleteVRTPalette`, `DeleteVRTShape`, `DeleteVRTSound`, `DeleteVRTWindow`

DeleteVRTWindow

Syntax: **`short DeleteVRTWindow(short WinNum);`**

Description: Deletes an existing window. `WinNum` is the window number to delete. Returns 0 if successful, or a negative error number.

See also: `CreateVRTWindow`, `DeleteVRTDial`, `DeleteVRTIcon`, `DeleteVRTInstrument`, `DeleteVRTObject`, `DeleteVRTPalette`, `DeleteVRTShape`, `DeleteVRTSound`, `DeleteVRTSprite`

DestroyTextBlock

Syntax: **`void DestroyTextBlock(void *Handle);`**

Description: Removes an editable text block.

See also: `CreateTextBlock`, `EditTextBlock`, `SetTextBlockCursor`, `SetTextBlockErrorStrings`

Example:

```
DestroyTextBlock(Handle);
```

Dialogue

Syntax: **`long Dialogue(char *DialName);`**

Description: Allows the user to edit information, using dialog box `DialName`. The data to be edited should be placed in a buffer, then global variable `*C_EditBuffer` should be modified to point to the buffer. On return, the modified data is in the buffer, and the return value reflects which button was pressed to exit from the dialog box. Typically, if this was the OK button (return `E_DROK`), the data would be copied back from the buffer into main memory.

See also: `FindDial`

Example: `if(Dialogue(“ALERT_SURE”)==E_DROK)`

Draw

Syntax: **`void Draw(void *Tree);`**

Description: Draws the drawing tree whose root node is at `Tree`. This is usually the root node returned from the last call to `Sort`. The procedure to draw a Superscape rendering of the current world into the current console window is to call `Process` with the address of the root object, then call `Sort` and finally call `Draw` to draw the sorted tree into the current window. The draw list is drawn to the background screen, which is displayed by calling `ScreenDraw`.

See also: `Process`, `Sort`, `ScreenDraw`

Example: `Draw(DrawRoot);`

DrawInstruments

Syntax: **`void DrawInstruments(short WindowNumber);`**

Description: Draws instruments for window `WindowNumber` onto the screen. This routine performs all the checks for update required to maintain the instruments with minimal processor load.

See also: `Draw`, `DrawRectangle`, `DrawPolygon`, `Text`

Example: `DrawInstruments(1);`

DrawPolygon

Syntax: **`void DrawPolygon(T_GRPOLYGON *Polygon);`**

Description: Draws a convex polygon, defined by the structure pointed to by `Polygon`, onto the background screen.

See also: `Draw`, `DrawRectangle`

Example: `DrawPolygon(Polygon);`

DrawPropMenu

Syntax: **`void DrawPropMenu(char *DialName);`**

Description: Draws dialog box `DialName` as the proportional device menu, such as the displayed Spacemouse buttons. Proportional menus must be turned on for this to have any effect.

See also: `Dialogue`

Example: `DrawPropMenu(“USER_PROP_MENU”);`

DrawRectangle

Syntax: **`void DrawRectangle(short x1, short y1, short x2, short y2, unsigned char Colour);`**

Description: Draws a rectangle whose diagonally opposite corners are (`x1`,`y1`) and (`x2`,`y2`) on the background screen in color `Colour`.

See also: `Draw`, `DrawPolygon`, `AddRectangle`

Example: `DrawRectangle(0,0,639,511,E_COLBLACK);`

EditTextBlock

Syntax: **`long EditTextBlock(void* Handle);`**

Description: Replaces the current value in the text block. Returns the return value from the editable text box.

See also: `CreateTextBlock`, `DestroyTextBlock`, `SetTextBlockCursor`, `SetTextBlockErrorStrings`

Example: `ReturnValue=EditTextBlock(Handle);`

Execute

Syntax: **`short Execute(unsigned char *Code);`**

Description: Executes previously compiled SCL object code at `Code`. Returns with 0 if successful, or a negative value if an error occurred.

See also: `ExecuteSCL`, `Compile`

Example: `Execute(ChunkAdd(22,E_CTSCL));`

ExecuteSCL

Syntax: **`void ExecuteSCL(T_WORLDCHUNK *Tree);`**

Description: Executes all SCL programs on the object pointed to by `Tree`, its subsequent siblings, and all its children.

See also: `Execute`, `UpdateWorld`, `UpdateViewPoint`

Example: `ExecuteSCL(ChunkAdd(0,E_CTSTANDARD));`

FFT

Syntax: **`short FFT(float *Real, float *Imag, short Pwr, short Dir);`**

Description: Performs a forward or inverse FFT (Fast Fourier Transform) on linear arrays of data. The real part of the data is presented in an array pointed to by `Real`, and the imaginary part at `Imag` (for real world signals like sound, this imaginary part is usually all 0s). The size of the array must be a power of two, and less than or equal to 1024. `Pwr` is the power-of-two size of the array (array size is 2^{Pwr}), and `Dir` is 1 for a forward transform, or -1 for an inverse transform. Returns a negative error number if the input values are invalid, or 0 with the transformed data stored in the supplied `Real` and `Imag` arrays.

VRT uses the FFT function to filter sounds. It converts blocks of the sound into the frequency domain, alters the amplitude of the frequency data, and converts it back again. Each block overlaps the previous one in order to prevent unwanted noise at the joins.

Information about the theory behind Fourier transforms can be found in the book “Information Theory for Information Technologists” by M.J.Usher (Macmillan, 1984, ISBN 0-333-36703-0).

FileSelector

Syntax:

```
char *FileSelector(char *Path, char *FileName, char *Title,
  long FileTypes);
```

Description: Displays the Windows Open File dialog box, containing all files matched by `Path`, a default filename of `FileName`, and a title of `Title`. `FilesTypes` is a bit vector specifying which file types should appear in the File Types list. It is a combination of the following:

`E_MTYPEALL`	*.* all files
`E_MTYPEBMP`	BMP files
`E_MTYPEDXF`	DXF files
`E_MTYPEGIF`	GIF files
`E_MTYPEINF`	.INF files
`E_MTYPEJPG`	JPEG files
`E_MTYPEPCX`	PCX files
`E_MTYPESCRIPT`	Script files
`E_MTYPESMP`	SMP files
`E_MTYPETGA`	TGA files
`E_MTYPETIF`	TIFF files
`E_MTYPEVCA`	VCA files
`E_MTYPEWAV`	WAV files

Returns a complete filename with path of the selected file, or `NULL` if Cancel pressed.

See also: `GetFileList`

Example:

```
FileName=FileSelector  (“C:\DATA\*.DAT”,“DEFAULT.DAT”,
    ”Load Data File”, E_MTYPEALL);
```

FindDial

Syntax: **`T_DIALCHUNK *FindDial(char *DialName);`**

Description: Returns a `T_DIALCHUNK` pointer to the root item of the dialog box whose name is `DialName`.

See also: `Dialogue`

Example: `pDial=FindDial("ALERT_NO_MEM");`

FindBearing

Syntax: **`short FindBearing(long x, long y, long z, short *XRot);`**

Description: Finds the angles required to look along the vector `x`, `y`, `z`. Returns the y rotation directly, and fills in the x rotation in `XRot`.

See also: `AbsPosition`, `FindOffset`

Example: `yr=FindBearing(XDiff,YDiff,ZDiff,&xr);`

FindFacet

Syntax:

```
void FindFacet(short x, short y, T_OBJFAC *pObjFac,
    unsigned short Exclude);
```

Description: Finds the object and facet number under the point (`x`,`y`) on the screen. These are filled into the `T_OBJFAC` structure pointed to by `pObjFac`. `Exclude` is the number of an object to ignore, or -1 if all objects should be included.

Example: `FindFacet(XPos,YPos,&FacFound,0);`

FindOffset

Syntax:

```
void FindOffset(short Object, short PointNum, long *x,
    long *y, long *z);
```

Description: Finds the offset from the origin of point `PointNum` in object `Object`. The position is filled in at `x`, `y`, `z`.

See also: `AbsPosition`, `FindBearing`

Example: `FindOffset(222,12,&x,&y,&z);`

FlushUndoBuf

Syntax: **`void FlushUndoBuf(void);`**

Description: Flushes all undo information from the undo buffer. This should be done when changing the length or position of any chunk in the world, shape, palette, sound, image or resource buffers.

Example: `FlushUndoBuf();`

GetDosVector

Syntax: **`T_DOSVECTOR *GetDosVector(short VecNum);`**

Description: Returns a pointer to the handler function on DOS interrupt vector `VecNum`. Use this in preference to the WATCOM library function `_dos_getvect`.

This function is included for compatibility with earlier versions of the SDK.

See also: `SetDosVector`

GetFileLength

Syntax: **`long GetFileLength(char *FileName);`**

Description: Gets the physical length, in bytes, of file `FileName`. If an error occurs (such as file not found), returns a negative error code.

See also: `BSave`, `BLoad`, `GetFileList`

Example: `Len=GetFileLength(“DEMO.VRT”);`

GetFileList

Syntax:

```
char ** GetFileList(char *PathName, char *Buffer,
    short BufLen);
```

Description: Returns an array of the names of all files matching `PathName`. Uses `Buffer` as workspace, up to a maximum length of `BufLen`. If the buffer space is exhausted, only a partial list of files is returned.

This function has not been implemented in Windows Visualiser. It is included for backward compatibility.

GetFileName

Syntax:	**`char *GetFileName(char *FileName, char *Type);`**
Description:	Returns the full path and name of `FileName`. `Type` points to a four character file type as follows:

"`.VRT`"	VRT file
"`CNFG`"	Configuration file
"`DDRV`"	Device driver file
"`FONT`"	Font file
"`MESS`"	Message file
"`PALT`"	Palette file
"`PRNT`"	Printer driver
"`RSRC`"	Resource file (version 3.60 or earlier)
"`RESC`"	Resource file (version 4.00 or later)
"`SHAP`"	Shape file
"`SOUN`"	Sound file
"`SPRT`"	Image (sprite) file
"`WRLD`"	World file

If `FileName` is a .VRT file and another type has been specified, returns the name of the first file with that type in the specified .VRT file. If the filename does not contain any path, this tries the same directory as the currently loaded .CFG file, and then the default VRT directory.

Compressed worlds in .SVR and XVR format are decompressed and stored in memory as .VRT files.

See also: `BSave`, `BLoad`, `GetFileLength`

Example: `FileName=GetFileName("DEMO.VRT", "CNFG");`

GetFunctions

Syntax:	**`void GetFunctions(void);`**
Description:	Interrogates all the devices that can generate control functions (for example, mouse, keyboard, proportional devices), and places the functions into the functions pending buffer.
See also:	`WriteFunc`, `ProcessFunctions`, `ObeyFunction`
Example:	`GetFunctions();`

GetSCLLength

Syntax:	**`short GetSCLLength(char *SCL);`**
Description:	Returns the length, in bytes, of the SCL instruction pointed to by SCL. This can be used to step through SCL programs looking for specific operation codes.
Example:	`Next=SCL+GetSCLLength(SCL);`

GetLenSS

Syntax:	**`long GetLenSS(char *Filename, char *Type, char Mode);`**
Description:	Returns the length of the Superscale file `Filename`, as it will be when loaded into memory. `Type` is the filetype (as in `LoadSS`). `Mode` should be set to 1.

GetShapeSym

Syntax: **`T_OBJSYM *GetShapeSym(short ShpNum, short Chk);`**

Description: Returns a pointer to symbols chunk type `Chk` for shape `ShpNum`. If no such chunk is found, returns `NULL`. Valid chunk types are:

`E_SYMSHPNAME`	Shape name
`E_SYMVARNAMES`	SCL variable names
`E_SYMCOMMENTS`	SCL comments
`E_SYMSHAPESIZE`	Shape size (no longer used)

See also: `GetSym`, `GetSymList`

Example: `Name=GetShapeSym(MyShape,E_SYMSHPNAME);`

GetShare

Syntax: **`T_APISHARE *GetShare(char *Name);`**

Description: Attempts to find a registered application called `Name`. If one is not found, it returns `NULL`, otherwise a pointer to the named application's shared information structure is returned. The value of the `CallBack` field in this structure must always be checked before calling, since a `NULL` value is legal (and disastrous if called).

See also: `RegisterShare`

GetSym

Syntax: **`T_OBJSYM *GetSym(short ObjNum, short Chk);`**

Description: Returns a pointer to symbols chunk type `Chk` for object `ObjNum`. If no such chunk is found, returns `NULL`. Valid chunk types are:

`E_SYMOBJNAME`	Object name
`E_SYMVARNAMES`	SCL variable names
`E_SYMCOMMENTS`	SCL comments
`E_SYMLOCVARS`	Locally triggered SCL variables
`E_SYMLOCCOMM`	Locally triggered SCL comments
`E_SYMGLOVARS`	Globally triggered SCL variables
`E_SYMGLOCOMM`	Globally triggered SCL comments

See also: `GetShapeSym`, `GetSymList`

Example: `Name=GetSym(Object,E_SYMOBJNAME);`

GetSymList

Syntax: **`T_OBJSYM *GetSymList(short Number, short Chk, T_OBJSYM *s);`**

Description: Returns a pointer to symbols chunk type `Chk` for item `Number` from list of symbols at `s`. If no such chunk is found, returns `NULL`. Valid chunk types are:

`E_SYMOBJNAME`	Object name
`E_SYMSHPNAME`	Shape name
`E_SYMVARNAMES`	SCL variable names
`E_SYMCOMMENTS`	SCL comments
`E_SYMSHAPESIZE`	Shape size (no longer used)
`E_SYMLAYERNAMES`	Layer names
`E_SYMSNDNAME`	Sound name
`E_SYMSPRNAME`	Image (sprite) name
`E_SYMLOCVARS`	Locally triggered SCL variables
`E_SYMLOCCOMM`	Locally triggered SCL comments
`E_SYMGLOVARS`	Globally triggered SCL variables
`E_SYMGLOCOMM`	Globally triggered SCL comments

See also: `GetShapeSym`, `GetSym`

Example: `Name=GetSymList(SoundNum,E_SYMSNDNAME,  SoundSyms);`

HexToLong

Syntax: **`char *HexToLong(long *n, char *String);`**

Description: Converts ASCII representation of hex number at `String` into a number. This is placed at `n`. Returns a pointer to the first invalid character in the string.

See also: `AscToLong`, `BinToLong`, `DecToLong`, `LongToHex`

Example: `p=HexToLong(&Num,”1F4E”);`

Horizon

Syntax: **`void Horizon(void);`**

Description: Fills in the background according to the current state of `C_BufferClear`.

See also: `Process`, `Sort`, `Draw`

Example: `Horizon();`

ImageAlloc

Syntax: **`long ImageAlloc(T_IMAGEINFO *ImageInfo);`**

Description: Creates space for a new image, based on the `Width`, `Height`, and `BitsPerPixel` information passed in the `T_IMAGEINFO` structure. The other fields in the structure are filled in by the function. Returns 0 if successful, or a negative error number if enough memory could not be found.

See also: `ImageLoadLibrary`, `ImageFree`, `ImageLoad`, `ImagePalette`, `ImageSave`

ImageFree

Syntax: **`long ImageFree(T_IMAGEINFO *ImageInfo);`**

Description: Frees space taken by an image. The `T_IMAGEINFO` structure should have been successfully filled in by `ImageAlloc` or `ImageLoad`. Returns 0 if successful, or a negative error number if `ImageInfo` is invalid.

See also: `ImageLoadLibrary`, `ImageAlloc`, `ImageLoad`, `ImagePalette`, `ImageSave`

ImageLoad

Syntax: **`long ImageLoad(char *FileName, T_IMAGEINFO *ImageInfo);`**

Description: Allocates space for an image and loads it from file `FileName`. The `T_IMAGEINFO` structure is filled in with the attributes of the image loaded. Returns 0 if successful, or a negative error number.

See also: `ImageLoadLibrary`, `ImageAlloc`, `ImageFree`, `ImagePalette`, `ImageSave`

ImageLoadLibrary

Syntax: **`short ImageLoadLibrary();`**

Description: Checks if the default image library is available; if not, it attempts to load it. If this fails, it returns a negative error number. If the default library is successfully loaded or is already available, it returns a value of 0.

See also: `ImageAlloc`, `ImageFree`, `ImageLoad`, `ImagePalette`, `ImageSave`

ImagePalette

Syntax: **`long ImagePalette(char *FileName, char *Palette);`**

Description: Loads a palette from file `FileName`, storing it at `Palette`. `Palette` must point to enough space to hold the complete palette. This is organized as up to 256 RGB triples, so 768 bytes of space is required. Returns `0` if successful, or a negative error number.

See also: `ImageLoadLibrary`, `ImageAlloc`, `ImageFree`, `ImageLoad`, `ImageSave`

ImageSave

Syntax:

```
long ImageSave(char *FileName, T_IMAGEINFO *ImageInfo,
    char Type);
```

Description: Saves an image to disk as file `FileName`. The `T_IMAGEINFO` structure should have been successfully filled by `ImageAlloc` or `ImageLoad`. `Type` is the required image format to save. The standard library recognizes the following types:

`E_IMAGEBMP`	BMP image
`E_IMAGEGIF`	GIF image
`E_IMAGEJPG`	JPEG image
`E_IMAGEPCX`	PCX image
`E_IMAGETGA`	TGA image
`E_IMAGETIF`	TIFF image

Returns 0 if successful, or a negative error number.

See also: `ImageLoadLibrary`, `ImageAlloc`, `ImageFree`, `ImageLoad`, `ImagePalette`

Init

Syntax: **`void Init(T_WORLDCHUNK *Root, T_SHAPECHUNK *Shape);`**

Description: Reinitializes the world and shape data. `Root` should point to the root object in the world, and `Shape` should point to the first shape definition. This is equivalent to pressing F12 when running VRT.

See also: `VecB4Init`, `VecAFInit`

Example: `Init(ChunkAdd(0,E_CTSTANDARD),ShapeList);`

InitSCL

Syntax: **`void InitSCL(T_WORLDCHUNK *SCL);`**

Description: Reinitializes the SCL in the chunk whose address is supplied. This call sets up all the variables, and resets the storage in `resume` statements. It performs the same job as the SCL instruction `resetf`.

See also: `Compile`, `RegisterSCL`

Example: `InitSCL(Chk);`

InitViewPoints

Syntax: **`void InitViewPoints(T_WORLDCHUNK *p);`**

Description: Reinitializes the viewpoints in the chunk whose address is supplied. This should be called after any changes made to viewpoint paths to make sure they are registered by the main program. Note that viewpoints should only be stored on object 0, the root object.

Example: `InitViewPoints(ChunkAdd(0,  E_CTVIEWPOINTS));`

LoadBackdrop

Syntax:	**`long LoadBackdrop(short Number);`**
Description:	Loads a new graphics file as a backdrop, displaying a dialog box and getting the filename from the user. This is loaded into backdrop buffer `Number`. Returns a negative error number if the backdrop was not loaded correctly, otherwise 0.
See also:	`LoadShape`, `LoadWorld`, `LoadUserSprites`, `LoadUserRsrc`, `LoadMessages`, `LoadSounds`, `LoadPalette`
Example:	`Error=LoadBackdrop();`

LoadMessages

Syntax:	**`long LoadMessages(void);`**
Description:	Loads a new message file, displaying a dialog box and getting the filename from the user. Returns a negative error number if the message file was not loaded correctly, otherwise 0.
See also:	`ClearMessages`, `SaveMessages`, `LoadShape`, `LoadWorld`, `LoadUserSprites`, `LoadUserRsrc`, `LoadPalette`, `LoadSounds`, `LoadBackdrop`
Example:	`Error=LoadMessages();`

LoadPalette

Syntax:	**`long LoadPalette(void);`**
Description:	Loads a new palette file, displaying a dialog box and getting the filename from the user. Returns a negative error number if the palette was not loaded correctly, otherwise 0.
See also:	`ClearPalette`, `SavePalette`, `LoadShape`, `LoadWorld`, `LoadUserSprites`, `LoadUserRsrc`, `LoadMessages`, `LoadSounds`, `LoadBackdrop`
Example:	`Error=LoadPalette();`

LoadShape

Syntax: **`long LoadShape(void);`**

Description: Loads a new shape file, displaying a dialog box and getting the filename from the user. Returns a negative error number if the shape file was not loaded correctly, otherwise 0.

See also: `ClearShape`, `SaveShape`, `LoadWorld`, `LoadPalette`, `LoadUserSprites`, `LoadUserRsrc`, `LoadMessages`, `LoadSounds`, `LoadBackdrop`

Example: `Error=LoadShape();`

LoadSounds

Syntax: **`long LoadSounds(void);`**

Description: Loads a new sound file, displaying a dialog box and getting the filename from the user. Returns a negative error number if the sound file was not loaded correctly, otherwise 0.

See also: `ClearSounds`, `SaveSounds`, `LoadShape`, `LoadWorld`, `LoadUserSprites`, `LoadUserRsrc`, `LoadMessages`, `LoadPalette`, `LoadBackdrop`

Example: `Error=LoadSounds();`

LoadSS

Syntax:
```
long LoadSS(char *FileName, void *LoadAddress,
    long MaxLength, char *Type, char Mode);
```

Description: Loads a Superscape file, called `FileName`, to `LoadAddress`, loading at most `MaxLength` bytes. The file type must match `Type`, one of:

“`.VRT`”	VRT file
“`CNFG`”	Configuration file
“`DDRV`”	Device driver file
“`FONT`”	Font file
“`MESS`”	Message file
“`PALT`”	Palette file
“`PRNT`”	Printer driver
“`RSRC`”	Resource file (version 3.60 or earlier)
“`RESC`”	Resource file (version 4.00)
“`SHAP`”	Shape file
“`SOUN`”	Sound file
“`SPRT`”	Image (sprite) file
“`WRLD`”	World file

`Mode` should be set to 1. Returns the actual number of bytes loaded, or a negative error number. This call loads any Superscape file, converting from a script file to a binary file if necessary. It also deals with finding files within .VRT files.

Compressed worlds in .XVR and .SVR format are decompressed and stored in memory as .VRT files.

See also: `SaveSS`, `GetFileList`

Example: `Len=LoadSS(“DEMO.WLD”,WorldBuffer,10000,“WRLD”,1);`

LoadUserSprites

Syntax: **`long LoadUserSprites(void);`**

Description: Loads a new user image file, displaying a dialog box and getting the filename from the user. Returns a negative error number if the image file was not loaded correctly, otherwise 0.

See also: `ClearUserSprites`, `SaveUserSprites`, `LoadShape`, `LoadPalette`, `LoadWorld`, `LoadUserSprites`, `LoadMessages`, `LoadSounds`, `LoadBackdrop`

Example: `Error=LoadUserSprites();`

LoadUserRsrc

Syntax: **`long LoadUserRsrc(void);`**

Description: Loads a new user resource file, displaying a dialog box and getting the filename from the user. Returns a negative error number if the resource file was not loaded correctly, otherwise 0.

See also: `ClearUserRsrc`, `SaveUserRsrc`, `LoadShape`, `LoadPalette`, `LoadWorld`, `LoadUserRsrc`, `LoadMessages`, `LoadSounds`, `LoadBackdrop`

Example: `Error=LoadUserRsrc();`

LoadWorld

Syntax: **`long LoadWorld(void);`**

Description: Loads a new world file, displaying a dialog box and getting the filename from the user. Returns a negative error number if the world was not loaded correctly, otherwise 0.

See also: `ClearWorld`, `SaveWorld`, `LoadShape`, `LoadPalette`, `LoadUserSprites`, `LoadUserRsrc`, `LoadMessages`, `LoadSounds`, `LoadBackdrop`

Example: `Error=LoadWorld();`

LongToAsc

Syntax: **`char *LongToAsc(long n, char *String, short Length, long *Power, short NumPowers);`**

Description: Converts number `n` to an ASCII representation in any base, placing `Length` characters at `String`. Returns pointer to the character after the last converted character. If number of characters is less than `Length`, the number is right justified in the available `Length` characters, padded with spaces. If the top bit of Length is set (`0x8000`), the bottom 15 bits are used as the length and the padding is done with zeroes. An array of powers of the chosen base should be set up and pointed to by `Power`, and the number of powers represented should be placed in `NumPowers`. The power table should include all powers below 2^{32}. For example, the decimal power table would be 1,10,100,1000... 10^{8}, with `NumPowers` set at 9.

See also: `AscToLong`, `LongToBin`, `LongToDec`, `LongToHex`

Example: `*LongToAsc(Num,Buffer,10,DecPwrs,9)=0;`

LongToBin

Syntax: **`char *LongToBin(long n, char *String, short Length);`**

Description: Converts number `n` to an ASCII representation in binary, placing `Length` characters at `String`. Returns pointer to the character after the last converted character. If number of characters is less than `Length`, the number is right justified in the available `Length` characters, padded with spaces. If the top bit of `Length` is set (`0x8000`), the bottom 15 bits are used as the length and the padding is done with zeroes.

See also: `BinToLong`, `LongToAsc`, `LongToDec`, `LongToHex`

Example: `*LongToBin(Num,Buffer,0x8010)=0;`

LongToDec

Syntax: **`char *LongToDec(long n, char *String, short Length);`**

Description: Converts number `n` to an ASCII representation in decimal, placing `Length` characters at `String`. Returns pointer to the character after the last converted character. If number of characters is less than `Length`, the number is right justified in the available `Length` characters, padded with spaces. If the top bit of `Length` is set (`0x8000`), the bottom 15 bits are used as the length and the padding is done with zeroes.

See also: `DecToLong`, `LongToAsc`, `LongToBin`, `LongToHex`

Example:

```
/* Convert Num to decimal, store      */
/* in buffer, and Null terminate      */

*LongToDec(Num,Buffer,10)=0;
```

LongToHex

Syntax: **`char *LongToHex(long n, char *String, short Length);`**

Description: Converts number n to an ASCII representation in hex, placing `Length` characters at `String`. Returns pointer to the character after the last converted character. If number of characters is less than `Length`, the number is right justified in the available `Length` characters, padded with spaces. If the top bit of `Length` is set (0x8000), the bottom 15 bits are used as the length and the padding is done with zeroes.

See also: `HexToLong`, `LongToAsc`, `LongToBin`, `LongToDec`

Example:

```
/* Convert Num to hex, store in       */
/* buffer, and Null terminate         */

*LongToHex(Num,Buffer,0x8004)=0;
```

MatMake

Syntax:

```
void MatMake(short XRot, short YRot, short ZRot,
    T_MATRIX Mat);
```

Description: Makes a ‘forward’ rotation matrix from `XRot`, `YRot` and `ZRot` (all specified in brees), placing the result in 3x3 matrix `Mat`.

See also: `MatMult`, `MatMakeInv`, `RotatePoint`

Example:

```
MatMake(xr,yr,zr,Mat);
```

MatMakeInv

Syntax: **`void MatMakeInv(short XRot, short YRot, short ZRot, T_MATRIX Mat);`**

Description: Makes a 'backward' rotation matrix from `XRot`, `YRot` and `ZRot` (all specified in brees), placing the result in 3x3 matrix `Mat`. Using the same rotations in `MatMake` and `MatMakeInv` result in two inverse matrices.

See also: `MatMult`, `MatMake`, `RotatePoint`

Example: `MatMakeInv(xr,yr,zr,Mat);`

MatMult

Syntax: **`void MatMult(T_MATRIX m1,T_MATRIX m2, T_MATRIX Result);`**

Description: Multiplies two 3x3 matrices `m1` and `m2`, putting the result in `Result`.

See also: `MatMake`, `MatMakeInv`, `RotatePoint`

Example: `MatMult(RotM,RotN,RotR);`

MouseOff

Syntax: **`void MouseOff(void);`**

Description: Turns the mouse pointer off. If the graphics board has a hardware mouse cursor this has no effect.

This function has no effect under Windows. It has been included for compatibility with earlier versions of the SDK.

MouseOn

Syntax: **`void MouseOn(void);`**

Description: Turns the mouse pointer on. If the graphics board has a hardware mouse cursor this has no effect. If multiple calls to `mouseoff` are made, the same number of calls to `mouseon` are required to restore the mouse pointer.

This function has no effect under Windows. It has been included for compatibility with earlier versions of the SDK.

ObeyFunction

Syntax:	**`void ObeyFunction(long FuncNum);`**
Description:	Obeys control function code `FuncNum` immediately. This is equivalent to the SCL instruction `obey`.
See also:	`WriteFunc`, `ProcessFunctions`, `GetFunctions`
Example:	`ObeyFunction(0x46000001);`

Orientation

Syntax:	**`void Orientation(short Object,short *xr,short *yr,short *zr);`**
Description:	Get the absolute (world) orientation of Object.
See also:	`AbsPosition`, `RelPosition`, `RelRotation`
Example:	`Orientation(ObjNum,&xr,&yr,&zr);`

Pick

Syntax:

```
T_PICKITEM *Pick(char *Title, T_PICKITEM *PickList,
    short Mode, T_PICKITEM *(*FilterFunction)(T_PICKITEM*));
```

Description: Allows the user to pick one or more items from the presented list of alternatives. `Title` is a string to display as the title of the displayed dialog box. `PickList` is a pointer to an array of `T_PICKITEM` structures, initialized appropriately and ending with an item whose Name field is `NULL` (see “Chapter 7: Data structures). `Mode` determines whether the user may pick just one or several items from the list, and may be a combination of the following values (add them together):

`E_PICKONE`	Pick only one item.
`E_PICKSOME`	Pick several items.
`E_PICKSORT`	Sort the list alphabetically.
`E_PICKNOSORT`	Do not sort the list.
`E_PICKFILT`	Show ‘Filter’ button on pick box.

The returned pointer points to an identically laid out list of `T_PICKITEM` structures, or is `NULL` if the user selected Cancel. When the list is no longer needed, its memory should be freed using a call to free.

`FilterFunction` is called when the Filter button is pressed, and is passed the complete pick item list. It returns a pointer to the filtered list.

See also: `FileSelector`

Example:

```
List=Pick("Select object type",PList,
    E_PICKONE+E_PICKSORT);
```

PopF

Syntax: **`float PopF(short t);`**

Description: Used within a registered SCL instruction to pop a floating point number from SCL's internal stack. If the item popped is not of type `t`, and cannot be converted to it, an SCL runtime error is generated. If any type will do, call with `t` set to `0xFF`.

See also: `PushF`, `PopN`, `PopP`

See "Chapter 2: SCL functions" for full details of how to use this function.

Example:

```
f=PopF(E_SSFLOAT);
```

PopN

Syntax: **`long PopN(short t);`**

Description: Used within a registered SCL instruction to pop a number from SCL's internal stack. If the item popped is not of type `t`, and cannot be converted to it, an SCL runtime error is generated. If any type will do, call with `t` set to `0xFF`.

See also: `PushN`, `PopF`, `PopP`

See "Chapter 2: SCL functions" for full details of how to use this function.

Example:

```
i=PopN(E_SSINTEGER);
```

PopP

Syntax: **`void *PopP(short t);`**

Description: Used within a registered SCL instruction to pop a pointer from SCL's internal stack. If the item popped is not of type `t`, and cannot be converted to it, an SCL runtime error is generated. If any type will do, call with `t` set to `0xFF`.

See also: `PushP`, `PopF`, `PopN`

See "Chapter 2: SCL functions" for full details of how to use this function.

Example: `q=PopP(E_SSPOINTER);`

Process

Syntax: **`void Process(T_WORLDCHUNK *Tree);`**

Description: Processes all children and siblings of the object pointed to by `Tree`, creating a drawing list to be sorted and drawn. `Tree` usually points to the root object (object 0), thus processing the whole world.

See also: `Sort`, `Draw`

Example: `Process(ChunkAdd(0,E_CTSTANDARD));`

ProcessFunctions

Syntax: **`long ProcessFunctions(void (*Func)(long));`**

Description: Takes each control function in turn from the functions pending buffer and executes it. If it is not a recognized function code, the code is passed through to the C function pointed to by `Func`. If `Func` is `NULL`, unrecognized function codes are simply ignored. Returns a non-zero value if an exit function has been processed.

`ProcessFunctions` is not declared as a `__vrtcall` in the vector definitions. You can access the routine directly under Windows by linking with the runtime DLL rather than through the vectors.

See also: `WriteFunc`, `GetFunctions`, `ObeyFunction`

Example: `ProcessFunctions(DoUnknownFunction);`

PushF

Syntax: **`void PushF(short t, float v);`**

Description: Used within a registered SCL instruction to push a floating point number on to SCL's internal stack. Pushes the value `v`, with type `t`.

See also: `PushN`, `PushP`, `PopF`

See "Chapter 2: SCL functions" for full details of how to use this function.

Example:

```
PushF(E_SSFLOAT,3.1415927);
```

PushN

Syntax: **`void PushN(short t, long v);`**

Description: Used within a registered SCL instruction to push a number on to SCL's internal stack. Pushes the value `v`, with type `t`.

See also: `PushF`, `PushP`, `PopN`

See "Chapter 2: SCL functions" for full details of how to use this function.

Example:

```
short i;
PushN(E_SSINTEGER,i);
```

PushP

Syntax: **`void PushP(short t, void *v);`**

Description: Used within a registered SCL instruction to push a pointer on to SCL's internal stack. Pushes the pointer `v`, with type `t`.

See also: `PushF`, `PushN`, `PopP`

See "Chapter 2: SCL functions" for full details of how to use this function.

Example:

```
short i;
PushP(E_SSPSHORT,&i);
```

Random

Syntax: **`long Random(long Seed);`**

Description: If `Seed` is non-zero, reseeds VRT's random number generator with the new seed, otherwise returns the next number in the pseudo-random sequence generated. The return is a full 32-bit number from -2147483648 to +2147483647.

Example: `n=Random(0);`

RedrawDialogue

Syntax: **`short RedrawDialogue(char * DialName);`**

Description: Since resident dialogs persist until an exit code is returned from them, it is sometimes necessary to redraw them immediately if an exceptional circumstance occurs. `RedrawDialogue` redraws any open dialog with the name `DialName`. It returns non-zero if successful, or 0 if no dialog of that name could be found.

Example: `RedrawDialogue ("MY_TOOLBAR");`

RegisterSCL

Syntax:

```
short RegisterSCL(T_COMPILEREC *CRec,
  void __vrtcall (*Function)(void));
```

Description: Registers a new SCL instruction with the main program. `CRec` is a pointer to a compiler record, which contains the name of the instruction, a preferred opcode number and various information about its parameters (see "Chapter 7: Data structures"). `Function` is the address of the function to execute when the SCL instruction is interpreted at runtime. The return value is either an opcode allocated to that instruction, or a negative error number.

See also: `UnRegisterSCL`

Example: `OpCode=RegisterSCL(ComRec[1],Func[1]);`

RegisterShare

Syntax: **`short RegisterShare(T_APISHARE *ShareData)`**

Description: Attempts to register the shared data pointed to by `ShareData`. If successful, returns 0, otherwise a negative error value is returned. This may only be called within the application's `App_Init` function.

See also: `Getshare`

ReInsert

Syntax:

```
short ReInsert(char *Ptr, long Length, long PrevLen,
    char *End, char *Buff, long BufUsed);
```

Description: Reinserts a record of `Length` characters into a buffer at `Ptr`. `PrevLen` is the previous length of the record at `Ptr`, and `End` points to the end of the buffer. `Buff` points to the record to be inserted. `BufUsed` should be set to 0. Returns 0 if successful, or -1 if there was not enough space in the buffer.

Example:

```
Err=ReInsert(BufBase+x, 12, 0, BufBase+E_BUFSIZE,
     C_EditBuffer, 0);
```

RelPosition

Syntax: **`void RelPosition(long *x,long *y,long *z,short Object);`**

Description: Given an absolute position, convert it to a position relative to Object.

See also: `AbsPosition`, `RelRotation`, `Orientation`

Example: `RelPosition(&x,&y,&z,ObjNum);`

RelRotation

Syntax: **`void RelRotation(short *xr,short *yr,short *zr,short Object);`**

Description: Given an absolute (world) rotation, convert it to a rotation relative to Object.

See also: `AbsPosition`, `RelPosition`, `Orientation`

Example: `RelRotation(&xr,&yr,&zr,ObjNum);`

RemoveSpriteColour

Syntax: **`int RemoveSpriteColour(unsigned int Colour, int SprNum, int bUseFirst16)`**

Description: Remaps the specified color in a sprite to the closest match in the palette. `Colour` is the color index to remove (0-255). `SprNum` is the number of the user sprite to affect. If `bUseFirst16` is zero the first sixteen (system) colors will not be considered during the remapping, otherwise the whole palette will be considered. Returns 0 if succesful.

See also: `CreateVRTSprite`, `CreateVRTSpriteWithPalette`, `DeleteVRTSprite`

Example:

```
T_GRSPRSINGLE    SprSingle;
int              SprNum=2;
// Remove transparency (0) from sprite 2 using whole
palette
RemoveSpriteColour(0,SprNum,1);

// Make sure graphics device knows about the change
SprSingle.Sprite=SprNum;
SprSingle.Flags=E_SPRUSER;
GrSetOneSprite(&SprSingle);
```

ResizeScreen

Syntax: **`short ResizeScreen(char Resolution);`**

Description: Switches the resolution of the screen to that specified in `Resolution`. Returns non-zero value if unable to set to that resolution for some reason (for example, already running at that resolution, or resolution not supported).

Example:

```
Error=ResizeScreen(0);
```

RotatePoint

Syntax: **`void RotatePoint(T_POINTREC3D Pnt, T_MATRIX Mat, short Type);`**

Description: Rotates three dimensional point `Pnt` about the origin using rotation matrix `Mat`. `Mat` is usually made using `MatMake` or `MatMakeInv`. `Type` is the type of matrix usually -1 for a general rotation matrix.

See also: `MatMult`, `MatMakeInv`, `MatMake`, `RotatePointLong`

Example:

```
RotatePoint(Pnt,Mat);
```

RotatePointLong

Syntax: **`void RotatePointLong(T_LONGVECTOR Pnt, T_MATRIX Mat, short Type);`**

Description: Rotates 3-D long point `Pnt` about the origin using rotation matrix `Mat`. `Mat` is usually made using `MatMake` or `MatMakeInv`. `Type` is the type of matrix usually -1 for a general rotation matrix.

See also: `MatMult`, `MatMakeInv`, `MatMake`

Example: `RotatePointLong(Vec,Mat);`

SaveConfig

Syntax: **`long SaveConfig(void);`**

Description: Saves the configuration file, displaying a dialog box and getting the filename from the user. Returns a negative error number if the onfiguration file was not saved, otherwise 0.

See also: `SaveShape`, `SavePalette`, `SaveUserSprites`, `SaveUserRsrc`, `SaveMessages`, `SaveSounds`, `SaveWorld`, `SavePrefs`, `SaveVRT`

Example: `Error=SaveConfig();`

SaveMessages

Syntax: **`long SaveMessages(void);`**

Description: Saves the user messages, displaying a dialog box and getting the filename from the user. Returns a negative error number if the message file was not saved, or 0.

See also: `ClearMessages`, `LoadMessages`, `SaveShape`, `SavePalette`, `SaveWorld`, `SaveUserRsrc`, `SaveUserSprites`, `SaveSounds`, `SaveConfig`, `SavePrefs`, `SaveVRT`

Example: `Error=SaveMessages();`

SavePalette

Syntax: **`long SavePalette(void);`**

Description: Saves the palette, displaying a dialog box and getting the filename from the user. Returns a negative error number if the palette was not saved, otherwise 0.

See also: `ClearPalette`, `LoadPalette`, `SaveShape`, `SaveWorld`, `SaveUserSprites`, `SaveUserRsrc`, `SaveMessages`, `SaveSounds`, `SaveConfig`, `SavePrefs`, `SaveVRT`

Example: `Error=SavePalette();`

SavePrefs

Syntax: **`long SavePrefs(void);`**

Description: Saves the preferences file, displaying a dialog box and getting the filename from the user. Returns a negative error number if the preferences file was not saved, otherwise 0.

See also: `SaveShape`, `SavePalette`, `SaveUserSprites`, `SaveUserRsrc`, `SaveMessages`, `SaveSounds`, `SaveWorld`, `SaveConfig`, `SaveVRT`

Example: `Error=SavePrefs();`

SaveScreen

Syntax:	**`short SaveScreen(short Format);`**
Description:	Saves the currently displayed screen to disk, using the next available filename. Picture format is determined by `Format` which can take one of the following values:

`E_IMAGEBMP`	BMP file
`E_IMAGEGIF`	GIF file
`E_IMAGEJPG`	JPEG file
`E_IMAGEPCX`	PCX file
`E_IMAGETGA`	TGA file
`E_IMAGETIF`	TIFF file

Returns 0 if successful or a negative error number.

See also:	`SaveSS`, `SaveVRT`
Example:	`SaveScreen(0);`

SaveShape

Syntax:	**`long SaveShape(void);`**
Description:	Saves the shape file, displaying a dialog box and getting the filename from the user. Returns a negative error number if the shapes were not saved, otherwise 0.
See also:	`ClearShape`, `LoadShape`, `SaveWorld`, `SavePalette`, `SaveUserSprites`, `SaveUserRsrc`, `SaveMessages`, `SaveSounds`, `SaveConfig`, `SavePrefs`
Example:	`Error=SaveShape();`

SaveSounds

Syntax:	**`long SaveSounds(void);`**
Description:	Saves the sounds, displaying a dialog box and getting the filename from the user. Returns a negative error number if the sound file was not saved, otherwise 0.
See also:	`ClearSounds`, `LoadSounds`, `SaveShape`, `SavePalette`, `SaveUserSprites`, `SaveUserRsrc`, `SaveMessages`, `SaveWorld`, `SaveConfig`, `SavePrefs`, `SaveVRT`
Example:	`Error=SaveSounds();`

SaveSS

Syntax:

```
long SaveSS(char *FileName, char *SaveAddress, long Length,
    char *Type, char Mode);
```

Description: Saves a Superscape file, called `FileName`, from `SaveAddress`, saving `Length` bytes. The file type is set to `Type`, one of:

“`.VRT`”	VRT file
“`CNFG`”	Configuration file
“`DDRV`”	Device driver file
“`FONT`”	Font file
“`MESS`”	Message file
“`PALT`”	Palette file
“`PRNT`”	Printer driver
“`RESC`”	Resource file
“`SHAP`”	Shape file
“`SOUN`”	Sound file
“`SPRT`”	Image (sprite) file
“`WRLD`”	World file

`Mode` should be set to 1. Returns the number of bytes saved, or a negative error number. This call saves any Superscape file, converting to a script file from a binary file if necessary. It also deals with finding files within .VRT files.

See also: `LoadSS`, `GetFileList`

Example: `Len=SaveSS(“DEMO.WLD”,WorldBuffer,10000,“WRLD”,1);`

SaveUserSprites

Syntax: **`long SaveUserSprites(void);`**

Description: Saves the user images, displaying a dialog box and getting the filename from the user. Returns a negative error number if the image file was not saved, otherwise 0.

See also: `ClearUserSprites`, `LoadUserSprites`, `SaveShape`, `SavePalette`, `SaveWorld`, `SaveUserRsrc`, `SaveMessages`, `SaveSounds`, `SaveConfig`, `SavePrefs`, `SaveVRT`

Example: `Error=SaveUserSprites();`

SaveUserRsrc

Syntax: **`long SaveUserRsrc(void);`**

Description: Saves the user resources (dialog boxes), displaying a dialog box and getting the filename from the user. Returns a negative error number if the resource file was not saved, otherwise 0.

See also: `ClearUserRsrc`, `LoadUserRsrc`, `SaveShape`, `SavePalette`, `SaveWorld`, `SaveUserSprites`, `SaveMessages`, `SaveSounds`, `SaveConfig`, `SavePrefs`, `SaveVRT`

Example: `Error=SaveUserRsrc();`

SaveVRT

Syntax: **`long SaveVRT(char *FileName, short Flags, char Mode);`**

Description: Saves a VRT file, called `FileName`, containing the files flagged by `Flags` (for individual flag definitions see entry for `C_FilesChanged`). Mode should be set to 1. Returns length of file saved, or a negative error number.

To save a file in Viscape or Visualiser format, set the file extension to .SVR or .XVR for the corresponding file.

See also: `SaveSS`, `GetFileList`

Example:
```
Len=SaveVRT("DAT.VRT",  E_MTYPECONFIG+E_MTYPEWORLD+
     E_MTYPESHAPE+E_MTYPEPALETTE,1);
```

SaveWorld

Syntax: **`long SaveWorld(void);`**

Description: Saves the world, displaying a dialog box and getting the filename from the user. Returns a negative error number if the world was not saved, otherwise 0.

See also: `ClearWorld`, `LoadWorld`, `SaveShape`, `SavePalette`, `SaveUserSprites`, `SaveUserRsrc`, `SaveMessages`, `SaveSounds`, `SaveConfig`, `SavePrefs`, `SaveVRT`

Example: `Error=SaveWorld();`

ScreenDraw

Syntax: **`void ScreenDraw(void);`**

Description: Swaps the foreground and background screens or copies the background screen to the foreground, depending on the state of `C_CopyFlag`, so that whatever was drawn on the background screen becomes immediately visible. It does not clear the background screen.

See also: `Draw`, `ClearWindow`, `ClearScreen`

Example: `ScreenDraw();`

SetDosVector

Syntax: **`void SetDosVector(short VecNum, T_DOSVECTOR FuncPtr);`**

Description: Sets the pointer to the handler function on DOS interrupt vector `VecNum` to `FuncPtr`. Use this in preference to WATCOM library function `_dos_setvect`. A combination of this and `GetDosVector` can be used to install handlers on any DOS interrupt routine and restore them on exit from the application.

This function does not work under Windows. It is included for compatibility with earlier versions of the SDK.

See also: `GetDosVector`

SetTDVInteger

Syntax: **`short SetTDVInteger(T_DIALCHUNK * Chunk, short TVal, long NewVal);`**

Description: Sets the integer value specified by one of the `T_DIALVALUE` structures in the `DIALOGUE` chunk pointed to by `Chunk` to the value of `NewVal`. The index of the `T_DIALVALUE` structure is specified in `TVal`. These indices are given along with the description of each field in the individual chunk structure definitions. The target value may have been specified immediately, be in a buffer, or be the destination of a pointer in the buffer; only the actual value itself is set. If the value cannot be set as an integer, this function returns a 0, otherwise it returns 1.

Note: There is no `SetTDVString` function, since this requires moving string data in memory; to do this, place the string at the required destination and set up the `T_DIALVALUE` structure directly.

See also: `SetTDVFloat`

Example: `SetTDVInterger (pDial, E_IFXPOS, 200);`

SetTDVFloat

Syntax: **`short SetTDVFloat(T_DIALCHUNK * Chunk, short TVal, float NewVal);`**

Description: Sets the floating point value specified by one of the `T_DIALVALUE` structures in the chunk pointed to by `Chunk` to the value of `NewVal`. The index of the `T_DIALVALUE` structure is specified in `TVal`. These indices are given along with the description of each field in the individual chunk structure definitions. The target value may have been specified immediately, be in a buffer, or be the destination of a pointer in the buffer; only the actual value itself is set. If the value cannot be set as a floating point number, this function returns a 0, otherwise it returns 1.

Note: There is no `SetTDVString` function, since this requires moving string data in memory; to do this, place the string at the required destination and set up the `T_DIALVALUE` structure directly.

See also: `SetTDVInteger`

Example: `SetTDVFloat (pDial, E_IFVALUE, 1.0);`

SetTextBlockCursor

Syntax: **`void SetTextBlockCursor(void *Handle, long CursorPos, long BlockStart, long BlockEnd);`**

Description: Sets the position of the cursor in the text block, and a selected text block. The cursor position, set by `CursorPos`, is measured in characters from the start of the buffer. `BlockStart` and `BlockEnd` indicate the first and last characters in a selected block; set to -1 if no block is selected. A line break is represented by one character.

See also: `EditTextBlock`, `CreateTextBlock`, `SetTextBlockErrorStrings`, `DestroyTextBlock`

SetTextBlockErrorStrings

Syntax: **`void SetTextBlockErrorStrings(void *Handle, char *ErrorString1, char *ErrorString2, char *ErrorString3);`**

Description: Defines three error stings for an editable text block.

See also: `EditTextBlock`, `CreateTextBlock`, `SetTextBlockCursor`, `DestroyTextBlock`

SetViewpoint

Syntax: **`void SetViewpoint(short VPNum);`**

Description: Sets the current viewpoint to be `VPNum`.

Example: `SetViewpoint(1);`

SortTree

Syntax: **`T_DCOBJECT *SortTree(void);`**

Description: Sorts the currently active drawing tree (as produced by the last call to `Process`), returning a pointer to the root node of the tree.

See also: `Process`, `Draw`

Example: `DrawRoot=SortTree();`

SplitPath

Syntax: **`void SplitPath(char *FileName, char *Path, char *File);`**

Description: Splits a complete path `FileName` into path and file information, placing these at `Path` and `File` respectively. If either of these is `NULL`, that part is not stored.

See also: `GetFileList`, `FileSelector`

Example:

```
SplitPath(“C:\DATA\DATA.DAT”,Path,File);
```

SquareRoot

Syntax: **`long SquareRoot(long Value);`**

Description: Returns the integer square root of `Value`.

Example:

```
Dist=SquareRoot(dx*dx+dy*dy+dz*dz);
```

StackNextDial

Syntax: **`void StackNextDial(void);`**

Description: Signals that the next dialog is to be stacked above any dialogs which are currently open, rather than replacing them. This occurs when a resident dialog box needs to bring up a sub-dialog. The `Alert` function already does this, so there is no need to call `StackNextDial` before displaying an alert box.

Example:

```
StackNextDial();
Return=Dialogue("STACKED_DIAL");
```

Terminate

Syntax: **`void _Terminate(short ErrorCode);`**

Description: Terminates VRT, with error code `ErrorCode`.

Do not use the C library routine `exit` which does not perform additional necessary procedures.

Example:

```
_Terminate(0);
```

Text

Syntax: **`void Text(short x,short y, unsigned char FGColour, unsigned char BGColour, short FontID, short MaxLen, short Align, unsigned char Flags, char *String);`**

```
void Text(short x,short y, unsigned char FGColour,
    unsigned char BGColour, short FontID, short MaxLen,
    short Align, unsigned char Flags, char *String);
```

Description: Draws the (null terminated) text in `String` on to the screen at position (`x`,`y`). Foreground and background colors are specified in `FGColour` and `BGColour` respectively, and the maximum number of characters to display is in `MaxLen`. If this is -1, all characters in the string are displayed. `FontID` is the number of the font to use, as specified in the font file:

0	8x8 (standard) font
2	16x16 (large) font
4	6x6 (small) font
6	6x8 (condensed) font
10	6x9 (condensed) font

`Flags` specifies which screen(s) to display the string on (0=Neither, 1=Foreground only, 2=Background only, 3=Both). `Align` is the relative alignment of the string to the start position (`x`,`y`) as follows:

0	Top left	1	Top center	2	Top right
4	Center left	5	Center	6	Center right
8	Bottom left	9	Bottom center	10	Bottom right

See also: `DrawInstruments`, `AddText`, `VisError`

Example:

```
Text(320,256,E_COLWHITE,E_COLBLACK,0,-1,5,3,
    ”Middle of both screens”);
```

UnRegisterSCL

Syntax: **`short UnRegisterSCL(short OpCode);`**

Description: Removes a registered SCL instruction from the registration list. `OpCode` is the opcode returned by `RegisterSCL` when the instruction was registered with the main program.

Registered SCL instructions must be unregistered before the application terminates.

See also: `RegisterSCL`

Example: `UnRegisterSCL(OpCode);`

UpdateDialogue

Syntax: **void UpdateDialogue(char *DialName, short Item, short MaxKids);**

Description: Updates a portion of dialog box DialName, starting at item Item, drawing at maximum MaxKids items. This is usually used for updating progress displays without having to redraw the entire dialog box at each stage.

See also: FindDial, Dialogue

Example: UpdateDialogue(“PROGRESS”,12,1);

UpdateWorld

Syntax: **void UpdateWorld(T_WORLDCHUNK *Tree);**

Description: Updates the information in the object pointed to by Tree, its subsequent siblings and all its children. Calling this with the address of the root object updates the whole world—moving all moving objects, advancing animations, performing angular velocities, and so on. It does not, however, execute SCL programs. This is a separate process, ExecuteSCL.

See also: ExecuteSCL, UpdateViewPoint

Example: UpdateWorld(ChunkAdd(0,E_CTSTANDARD));

UpdateViewPoint

Syntax: **void UpdateViewPoint(void);**

Description: Updates the information in the current viewpoint, moving it on, if necessary, to the next point on its path, ‘tweening its position and rotations.

See also: UpdateWorld, ExecuteSCL

Example: UpdateViewPoint();

VisError

Syntax: **`void VisError(char *String);`**

Description: Writes `String` to the screen as an error message in Visualiser. The last four error messages are displayed in the top right corner of the currently active window onto the world. To clear the error messages, pass a `NULL` pointer to this function.

See also: `Text`

Example: `VisError("ERROR: External device failure");`

WriteFunc

Syntax: **`void WriteFunc(long Function, short Parameter);`**

Description: Writes control function code `Function` into the functions pending buffer, with proportional parameter `Parameter`. `Parameter` is only used for such control functions as viewpoint movement, so for other control functions set it to 0.

See also: `GetFunctions`, `ProcessFunctions`, `ObeyFunction`

Example: `WriteFunc(0x46000001,0);`

Chapter 5 - Device drivers

Introduction

VRT uses a set of device drivers to control the system hardware, for example driving the screen, reading the keyboard, or outputting sounds. This allows the VRT code to be independent of the actual hardware; the device driver translates data between what the device expects and what VRT expects.

Applications have access to all device driver functions, which are defined as usual in APP_DEFS.H. They are called like normal C routines, and have standardized inputs and outputs.

All device drivers have their own Install, Setup and Deinstall routines, which generally are not needed by an application. The device drivers are loaded, installed and set up before the application's initialization code is called. They are unloaded when the program is terminated.

There are eight device drivers:

Keyboard	Kb	Handles inputs from the keyboard.
Mouse	Ms	Handles inputs from the mouse.
Graphics	Gr	Handles output to the screen.
Proportional	Pr	Handles inputs from the proportional device.
Sound	Sd	Handles outputs to the sound/MIDI device.
Timer	Tm	Handles interrupt-driven timers and fatal error exceptions.
Network	Nt	Handles communication between machines running the same world.
Serial	Sr	Handles serial communications to external devices.

It is very important that a device is not installed twice.

Each device belongs to a family which can have several device drivers loaded (up to `E_MAXPERDEV` drivers for each device type). For example, the proportional device family includes the Spacemouse, Spaceball, Polhemus Fastrak, and Flock of Birds. These are all loaded as the Superscape system starts, and may be enabled independently of one another. Each instance of a device within a device family is known as a 'slot'. By default, the first graphic device slot is occupied by the SVGA device driver.

Each device driver within each family must return a unique ID code, which distinguishes it from other device drivers. The ID code is not related to slot numbers. The currently used ID codes are listed below:

Family	Device	ID Code	Value
Keyboard	Standard	**`E_KBIDSTANDARD`**	1
Mouse	Microsoft Compatible	**`E_MSIDMICROSOFT`**	1
Graphics	TIGA	**`E_GRIDTIGA`**	1
	SVGA	**`E_GRIDSVGA`**	2
	MCGA	**`E_GRIDMCGA`**	3
	Direct3D	**`E_GRIDD3D`**	4
Timer	Standard	**`E_TMIDSTANDARD`**	1
Proportional	Spaceball	**`E_PRIDSBALL`**	1
	Joystick	**`E_PRIDJOYSTICK`**	2
	Flock of Birds	**`E_PRIDFLOB`**	3
	Fastrak	**`E_PRIDFASTRAK`**	4
	Spacemouse	**`E_PRIDSMOUSE`**	5
	Mouse movement	**`E_PRIDMOUSE`**	6
	Logitech 3D Mouse	**`E_PRID6MOUSE`**[1]	7
Sound	Music Quest MIDI	**`E_SDIDMIDI`**	1
	Ad Lib Gold	**`E_SDIDADLIB`**	2
	Sound Blaster /16	**`E_SDIDSOUNDBL`**	3
Network	Packet Driver	**`E_NTIDCLARKSON`**	1
	NetBios	**`E_NTIDNETBIOS`**	2
Serial	Standard	**`E_SRIDDEFAULT`**	1

[1] The original name for the device was 6D Mouse.

Drivers with a gray background are no longer supported, but do not reuse these ID values. We recommended that you give any new device drivers an ID starting from 100, to avoid conflicts with directly supported devices in future releases.

The device driver itself consists of a set of functions which are called by the main program. In order that the interface is completely generic, all device driver functions take a `void *` pointer and return a `void *` pointer. If no argument is required, a `NULL` pointer is passed. Similarly, a `NULL` pointer should be returned if no return value is required.

These functions are registered with the main program during `App_Init`. They are passed in the form of an array which always consists of the same number of entries, `E_MAXDEVFUNCS`. The index in the array defines what the function at that address is expected to do.

Unused entries should point to a function that simply returns its input value, for example:

```
void * __vrtcall DeviceStub(void *Parameter)
{
    return(Parameter);
}
```

The interface to each function in the device must be exactly the same as for other devices of the same family. The following table lists the device driver families, a description of each function, the corresponding array indices, and the prototype function which they mimic.

Family / Index	**Description**	**Array Index**	**Prototype**
Keyboard	Install	**`E_KBINSTALL`**	**`KbInstall`**
`E_DEVKEYBOARD`	Setup	**`E_KBSETUP`**	**`KbSetup`**
0	DeInstall	**`E_KBDEINSTALL`**	**`KbDeinstall`**
	Keyboard Ready	**`E_KBKEYREADY`**	**`KbKeyReady`**
	Read Key	**`E_KBREADKEY`**	**`KbReadKey`**
Mouse	Install	**`E_MSINSTALL`**	**`MsInstall`**
`E_DEVMOUSE`	Setup	**`E_MSSETUP`**	**`MsSetup`**
1	DeInstall	**`E_MSDEINSTALL`**	**`MsDeinstall`**
	Update Position	**`E_MSUPDATE`**	**`MsUpdate`**
Graphics	Install	**`E_GRINSTALL`**	**`GrInstall`**
`E_DEVGRAPHICS`	Setup	**`E_GRSETUP`**	**`GrSetup`**
3	DeInstall	**`E_GRDEINSTALL`**	**`GrDeinstall`**
	Draw Scan Buffer	**`E_GRDRAWSCAN`**	**`GrDrawScan`**
	Draw Line	**`E_GRDRAWLINE`**	**`GrDrawLine`**
	Draw Text	**`E_GRDRAWTEXT`**	**`GrDrawText`**
	Draw Backdrop	**`E_GRDRAWBACKDROP`**	**`GrDrawBackdrop`**

Family / Index	Description	Array Index	Prototype
Graphics	Draw Horizon	**E_GRDRAWHORIZON**	**GrDrawHorizon**
E_DEVGRAPHICS	Draw Sprite	**E_GRDRAWSPRITE**	**GrDrawSprite**
3	Undraw Sprite	**E_GRUNDRAWSPRITE**	**GrUndrawSprite**
	Copy Screen	**E_GRSCREENDONE**	**GrScreenDone**
	Set Palette - VRT	**E_GRSETPALETTE**	**GrSetPalette**
	Pick Object	**E_GRPICKOBJECT**	**GrPickObject**
	Draw Polygon	**E_GRDRAWPOLYGON**	**GrDrawPolygon**
	Draw Sorted List	**E_GRDRAWSORTED**	**GrDrawSorted**
	Set Stipple Table	**E_GRSETSTIPPLES**	**GrSetStipples**
	Set Sprite Table	**E_GRSETSPRITES**	**GrSetSprites**
	Precopy Operation	**E_GRPRECOPY**	**GrPreCopy**
	Process Object Tree	**E_GRDRAWMODEL**	**GrDrawModel**
	Set Palette - Windows	**E_GRPALCHANGED**	**GrPalChanged**
	Get Back Buffer DC	**E_GRGETBACKBUFFDC**	**GrGetBackBuffDC**
	Release Back Buffer DC	**E_GRRELBACKBUFFDC**	**GrRelBackBuffDC**
	Set Driver Mode	**E_GRSETCONFIG**	**GrSetConfig**
Timer	Install	**E_TMINSTALL**	**TmInstall**
E_DEVTIMER	Setup	**E_TMSETUP**	**TmSetup**
4	DeInstall	**E_TMDEINSTALL**	**TmDeinstall**
	Add List Node	**E_TMADDLIST**	**TmAddList**
	Remove List Node	**E_TMREMLIST**	**TmRemList**
	Fast Timer	**E_TMFASTTIMER**	**TmFastTimer**
	Slow Timer	**E_TMSLOWTIMER**	**TmSlowTimer**
Proportional	Install	**E_PRINSTALL**	**PrInstall**
E_DEVPROP	Setup	**E_PRSETUP**	**PrSetup**
5	DeInstall	**E_PRDEINSTALL**	**PrDeinstall**
	Get Position	**E_PRGETPOS**	**PrGetPos**

Family / Index	Description	Array Index	Prototype
Sound	Install	**`E_SDINSTALL`**	**`SdInstall`**
`E_DEVSOUND`	Setup	**`E_SDSETUP`**	**`SdSetup`**
6	DeInstall	**`E_SDDEINSTALL`**	**`SdDeinstall`**
	Play Sound	**`E_SDPLAYSOUND`**	**`SdPlaySound`**
	Record Sound	**`E_SDRECORD`**	**`SdRecord`**
	Modify Sound	**`E_SDMODIFY`**	**`SdModify`**
	Shut Up	**`E_SDSHUTUP`**	**`SdShutUp`**
	Send Sounds	**`E_SDSENDBLOCK`**	**`SdSendBlock`**
Network	Install	**`E_NTINSTALL`**	**`NtInstall`**
`E_DEVNETWORK`	Setup	**`E_NTSETUP`**	**`NtSetup`**
8	DeInstall	**`E_NTDEINSTALL`**	**`NtDeinstall`**
	Exchange Data	**`E_NTNETWORK`**	**`NtNetwork`**
	Resign	**`E_NTRESIGN`**	**`NtResign`**
Serial	Install	**`E_SRINSTALL`**	**`SrInstall`**
`E_DEVSERIAL`	Setup	**`E_SRSETUP`**	**`SrSetup`**
9	DeInstall	**`E_SRDEINSTALL`**	**`SrDeinstall`**
	Set Up Port	**`E_SRSETUPPORT`**	**`SrSetUpPort`**
	Release Port	**`E_SRRELEASEPORT`**	**`SrReleasePort`**
	Send Bytes	**`E_SRSENDBYTES`**	**`SrSendBytes`**
	Get Bytes	**`E_SRGETBYTES`**	**`SrGetBytes`**
	Set Call Back	**`E_SRSETCALLBACK`**	**`SrSetCallBack`**
	Clear Call Back	**`E_SRCLRCALLBACK`**	**`SrClrCallBack`**
	Flush Buffer	**`E_SRFLUSHBUF`**	**`SrFlushBuf`**
	Get Port Info	**`E_SRGETPORTINFO`**	**`SrGetPortInfo`**

Note: Device families 2 and 7 no longer exist and should not be used.

Registering the function array

The function array can be registered with the main code in two ways.

It can be passed as an argument to `RegisterDev`. This effectively registers a whole new device, in addition to the ones which are already there. For example:

```
short Slot;
STDFUNCPTR  MyFunctions[E_MAXDEVFUNCS];

/* Clear function array for proportional device              */

for(FuncNum=0;FuncNum<E_MAXDEVFUNCS;FuncNum++)
  MyFunctions[FuncNum]=DeviceStub;

/* Fill in actual function calls                             */

MyFunctions[E_PRINSTALL]   =MyInstall;
MyFunctions[E_PRSETUP]     =MySetup;
MyFunctions[E_PRDEINSTALL]=MyDeInstall;
MyFunctions[E_PRGETPOS]    =MyGetPos;

/* Register functions and check for error                    */

Slot=RegisterDev(MyFunctions,E_DEVPROP);
if(Slot<0)
{
  /* Error - no free slots for proportional devices         */
}
```

This device is allocated a slot number in a table internal to VRT. The slot number is returned from the `RegisterDev` call if successful, or a negative error number is returned if there are no free slots.

Alternatively, to replace a device which already exists and whose slot number you know, you can use the function call `ReRegisterDev`. For example:

```
short Slot;
T_STDFUNCPTR  MyFunctions[E_MAXDEVFUNCS];

/* Clear function array for proportional device            */
for(FuncNum=0;FuncNum<E_MAXDEVFUNCS;FuncNum++)
    MyFunctions[FuncNum]=NULL;

/* Fill in actual function calls                            */
MyFunctions[E_PRINSTALL]   =MyInstall;
MyFunctions[E_PRSETUP]     =MySetup;
MyFunctions[E_PRDEINSTALL]=MyDeInstall;
MyFunctions[E_PRGETPOS]    =MyGetPos;

/* Find slot number to replace, and put in Slot            */
Slot=ReRegisterDev(MyFunctions,E_DEVPROP,Slot);
if(Slot<0)
{
  /* Error - incorrect slot for proportional device         */
}
```

In this case, any `NULL` entries in the function table do not overwrite existing device functions. It is therefore possible to make a hybrid device, where some functions are handled by one driver, and some by another.

Another way of doing this is using `RegisterDevFunc`, which registers a single device function:

```
void *MyFunction(void *);
short Slot;

/* Slot is slot number to replace                            */
Slot=RegisterDevFunc(MyFunction,E_DEVPROP,Slot,E_PRGETPOS);
if(Slot<0)
{
  /* Error - incorrect slot for proportional device          */
}
```

This replaces the proportional get position function with `MyFunction`.

Building hybrid device drivers should be avoided if possible. Making a whole device driver may be more effort, but you have control at all times over the hardware and other resources that it uses.

When you build a device driver file, it should be included in the VRT preferences file. This can be done by editing the preferences script directly, or by using the Add button in Settings>Setup>Device. Using the Add option is preferable, since it allows the main program to make all the necessary changes to the preferences and system resource files. To add a new device you need to create a .INF file, which contains all the information VRT needs to register it.

⇒ To add the new device to your VRT

An .INF file is a simple text file that you can create with any text editor. Here is an example .INF file for a new proportional device—the .INF file must be in precisely this format (no additional blank lines) since line breaks are significant:

`NEWDEV.DEV`	*Name of device driver file*
`NEWDEV.RSC`	*Name of file for dialog box (or blank line)*
`5`	*Device family (decimal) Proportional family=5*
`8`	*Device ID (decimal). Must match ID in device driver file*
`0x03 0x01 0x00 0x20`	*Device setup data*

Under Windows you cannot use a dialog box (.RSC file) to configure the device; you must install the device and configure it manually. If you do define a configuration dialog box, it is ignored by Windows, so you can leave the line blank.

The device setup data is different for each new device and typically contains information such as I/O addresses, IRQ and DMA lines. The first byte defines the number of subsequent data bytes (in this case 3). These may be specified in hex (as above) or decimal.

⇒ To add the new device to your VRT

1. Start VRT.
2. Choose Settings>Setup [E].

The Setup dialog box is displayed.

3. Click Add.

The file selector is displayed.

4. Select the .INF file, and click OK.

The device is added to your configuration.

5. Choose File>Exit.

An alert box warns you that the configuration file has been changed.

6. Click OK until you exit VRT.
7. Restart VRT.

The device is registered and shown alongside all the other devices in the Setup dialog box.

8. Activate it in the same way as any other proportional device.

When the Superscape system is restarted, the devices are loaded. After the devices have been loaded, the Install function is called for each device in turn with a `NULL` pointer as an argument. The order is: Timer, Keyboard, Graphics, Serial, Proportional, Mouse, Sounds, Network. The device should not attempt to install itself, but return a pointer to the appropriate `T_??INSTALLRET` structure (such as `T_PRINSTALLRET` for a proportional device), with the first four fields filled in as follows:

`Error`	0
`ID`	A unique ID code. Must match the one given in the .INF file.
`Name[16]`	The name of the device, for example 'Spacemouse'
`DialName[16]`	The name of the configuration dialog box, for example 'EDIT_SMSE'. As you cannot use a dialog to configure the device under Windows, set this field to 0.

If the device has been marked as active, the install routine is called again, this time with a pointer to a `T_DEVINSTALL` structure. The most important part of this is the `Config` field, which points to a `T_??INSTALL` structure that contains the information required to install the device correctly and configure it under Windows.

The Setup function may be called to reinitialize the driver, or to pass additional parameters to the device. Refer to the individual setup function descriptions for more information on the subsequent options.

The DeInstall function is called as the driver is unloaded. The driver should free any resources it was using (such as memory, I/O addresses, interrupt lines) and restore its device to a default state before returning. Deinstall functions are called in reverse order to the Install functions, so that chained interrupts can be freed in the correct order.

Other device calls are made at various times by the main program. Their input and output data are defined by the prototype functions listed in this chapter. New device drivers should precisely mimic the operation of these calls.

Since the call to register a device is made during `App_Init`, you can build several arrays of functions, and register several different device drivers from within a single SDK application. Each set of functions defines the interface to a particular device. This is a useful technique for reducing the number of modules loaded. All the default device drivers, for instance, are defined in a single SDK application module.

Device driver functions

The device driver calls are presented as a set of function prototypes, although they are, as usual, passed through the vector table. The shorter forms are defined in APP_DEFS.H. Do not use the long forms.

The Install, Setup and Deinstall procedures are included for completeness—the application should not need to use them.

In addition to the function calls, there are a few support functions and global variables which are also defined.

All structures passed to and returned from the device drivers are specified in full in the “Chapter 7: Data structures.”

ActiveDevice

Syntax:

```
short ActiveDevice(short DeviceType,
    unsigned short DeviceIndex);
```

Description: Returns the device number of an active device. `DeviceType` is the type of device to interrogate. It is one of the following:

`E_DEVGRAPHICS`	Graphics device
`E_DEVKEYBOARD`	Keyboard device
`E_DEVMOUSE`	Mouse device
`E_DEVNETWORK`	Network device
`E_DEVPROP`	Proportional device
`E_DEVSERIAL`	Serial device
`E_DEVSOUND`	Sound device
`E_DEVTIMER`	Timer device

`DeviceIndex` is an index from 0 to the number of devices minus one of the type which are active (0 is the first active device, 1 is the second, and so on). It returns the actual device driver number (this is not related to the device driver ID) which is active. This is in the range 0 to the number of devices of that type installed.

See also: `DeviceEnabled`, `C_DeviceActive`, `C_NumDevsActive`

Example:

```
/* Get primary proportional device if any */

if(C_NumDevsActive[E_DEVPROP]>0)
   Active=ActiveDevice(E_DEVPROP,0);
```

C_DevInstall

Syntax: **`T_DEVINSTALL *C_DevInstall;`**

Description: This variable points to a device install structure. The fields in this are already set up apart from the `Config` field which should be set to point to the device's install configuration data before installing the device.

Note: Do not change the values of the other fields in this structure.

See also: `GrInstall`, `KbInstall`, `MsInstall`, `NtInstall`, `PrInstall`, `SdInstall`, `SrInstall`, `TmInstall`

Example:

```
/* Install graphics device */

T_GRINSTALL  InstallData;
C_DevInstall->Config=&InstallData;
GrInstall(C_DevInstall);
```

C_GraphicsDD C_KeyboardDD C_MouseDD C_NetworkDD C_PropDD C_SerialDD C_SoundDD C_TimerDD

Syntax:

```
short *C_GraphicsDD;
short *C_KeyboardDD,
short *C_MouseDD;
short *C_NetworkDD;
short *C_PropDD;
short *C_SerialDD;
short *C_SoundDD;
short *C_TimerDD;
```

Description: These variables point to the numbers of the currently active device of each type. These are automatically used when calling the device drivers, and as such can quite safely be ignored.

These may be changed temporarily in order to use a different device from the default, but they *must* be set back to their original value before returning to the main program. Even in this manner, it is usually unwise to change the value of `*C_NetworkDD`, `*C_KeyboardDD`, and especially `*C_TimerDD`.

See also: `ActiveDevice`

Example:

```
/* Play sound on secondary sound device */

OldSoundDev=*C_SoundDD;
*C_SoundDD=ActiveDevice(E_DEVSOUND,1);
SdPlaySound(Sound);
*C_SoundDD=OldSoundDev;
```

DeviceEnabled

Syntax:

```
unsigned char DeviceEnabled
  (short DeviceType, short DeviceNum);
```

Description: Returns the enabled flags for device number `DeviceNum`, of type `DeviceType`. The returned value is a bitvector defined as follows:

Bit 0	Set if device currently enabled
Bits 1–7	Reserved

See also: `ActiveDevice`, `C_DeviceActive`, `C_DeviceEnabled`

Example:

```
if(DeviceEnabled(E_DEVGRAPHICS,1))
    /* We are GO for Secondary device */
```

RegisterDev

Syntax:

```
short RegisterDev(T_STDFUNCPTR *Functions, short Type);
```

Description: Registers a set of functions as being a device driver. `Functions` points to an array of pointers to the device driver functions. Each of these functions must be of type `void * __vrtcall (void *)`. `Type` is an indexed device type (such as `E_DEVGRAPHICS`, or `E_DEVSERIAL`). Each type of device has several ‘slots’ available for individual devices of that type. A positive slot number is returned from this function, or a negative error number if the registration failed.

See also: `ReRegisterDev`; `RegisterDevFunc`

ReRegisterDev

Syntax: **`short ReRegisterDev(T_STDFUNCPTR *Functions, short Type, short Slot);`**

Description: Reregisters a set of functions as being a specific device driver. Functions points to an array of pointers to the device driver functions, as in `RegisterDev`. NULL pointers in the table do not replace the existing function pointers. It is thus possible to reregister only a subset of the functions in a device with this call. `Type` is an indexed device type (such as `E_DEVGRAPHICS`, or `E_DEVSERIAL`). `Slot` is the slot index of the device to replace. A negative error number is returned if the registration failed.

See also: `RegisterDev`, `RegisterDevFunc`

RegisterDevFunc

Syntax: **`short RegisterDevFunc(T_STDFUNCPTR Function, short Type, short Slot,short EntryNum);`**

Description: Registers a single function as being tied to a specific device driver. Function is a pointer to the device driver function. `Type` is an indexed device type (such as `E_DEVGRAPHICS`, or `E_DEVSERIAL`). `Slot` is the slot index of the device to replace. `EntryNum` is the index of the function to replace in this device. A negative error number is returned if the registration failed.

See also: `RegisterDev`, `ReRegisterDev`

Graphics devices

`GrDeInstall`

Syntax:	**`void GrDeInstall(NULL);`**
Description:	Removes the graphics device.
See also:	`GrInstall`, `GrSetUp`
Example:	`GrDeInstall(NULL);`

`GrDrawBackDrop`

Syntax:	**`void GrDrawBackDrop(T_GRBACKDROP *);`**
Description:	Draws a previously loaded backdrop on the screen. Takes a pointer to a backdrop structure which specifies the address and length of the backdrop data.
See also:	`GrDrawSorted`, `GrDrawUnsorted`
Example:	`GrDrawBackDrop(PCXBuffer);`

`GrDrawHorizon`

Syntax:	**`void GrDrawHorizon(NULL);`**
Description:	Draws the horizon facets into the current.
	This function is not supported in all device drivers—check the `Abilities` flags returned from `GrSetUp`.
See also:	`GrDrawSorted`, `GrSetUp`
Example:	`GrDrawHorizon(NULL);`

GrDrawLine

Syntax: **`void GrDrawLine(T_GRLINE *);`**

Description: Draws a single pixel thick line on the background screen. Takes a pointer to a graphic line structure which specifies the color and position of the line.

See also: `GrDrawScan`, `GrDrawText`, `GrDrawBackDrop`, `GrDrawSprite`, `GrDrawPolygon`, `GrDrawSorted`, `GrDrawUnsorted`

Example: `GrDrawLine(&Line);`

GrDrawModel

Syntax: **`void *GrDrawModel(T_GRPROCINFO *);`**

Description: Processes the object tree pointed to by the `T_GRPROCINFO` structure, which also contains information about how to process the data.

This function is not supported in all device drivers—check the `Abilities` flags returned from `GrSetUp`.

Example: `GrDrawModel(&Tree);`

GrDrawPolygon

Syntax: **`T_GRPOLYGON *GrDrawPolygon(T_GRPOLYGON *);`**

Description: Draws a polygon on the background screen. Takes a pointer to a polygon structure, which specifies the number of vertices, the color, the facet number and various flags as well as the position of each vertex on the screen. If not supported, returns a NULL pointer, else returns non-NULL.

This function is not supported in all device drivers—check the Abilities flags returned from `GrSetUp`.

See also: `GrDrawLine`, `GrDrawScan`, `GrDrawText`, `GrDrawBackDrop`, `GrDrawSprite`, `GrDrawSorted`, `GrDrawUnsorted`

Example: `GrDrawPolygon(&Poly);`

GrDrawScan

Syntax: **`void GrDrawScan(T_GRSCAN *);`**

Description: Draws a scan buffer on the background screen. Takes a pointer to the scan buffer structure which specifies the length and position of each horizontal line segment, the start position to draw at, and its color.

See also: `GrDrawLine`, `GrDrawText`, `GrDrawBackDrop`, `GrDrawSprite`, `GrScreenDone`, `GrDrawSorted`, `GrDrawUnsorted`

Example: `GrDrawScan(&ScanBuf);`

GrDrawSorted

Syntax: **`T_DRAWLIST *GrDrawSorted(T_DRAWLIST *);`**

Description: Draws a sorted drawing list all in one go, on the background screen. Takes a pointer which points to the drawing list, as produced by the sorting routine. If not supported, returns a `NULL` pointer, else returns non-`NULL`.

This function is not supported in all device drivers—check the flags returned from `GrSetUp`.

See also: `GrDrawUnsorted`

Example: `GrDrawSorted(DrawRoot);`

GrDrawSprite

Syntax: **`char *GrDrawSprite(T_GRSPRITE *);`**

Description: Draws a sprite to the foreground, background or both screens. Takes a pointer to a sprite record specifying the address of the sprite data, its x and y position on the screen, its size, flags for which screens to draw it to, and a pointer to a background save buffer (NULL if background not to be saved). Returns a pointer to the background save area, or NULL if none.

See also: `GrDrawLine`, `GrDrawScan`, `GrDrawText`, `GrDrawBackDrop`, `GrUndrawSprite`, `GrDrawPolygon`, `GrDrawSorted`, `GrDrawUnsorted`

Example: `SvPtr=GrDrawSprite  (C_Sprite[*C_GraphicsDD]+2);`

GrDrawText

Syntax: **`T_GRRECT *GrDrawText(T_GRTEXT *);`**

Description: Draws a text string onto the foreground, background or both screens. Takes a pointer to a text structure, which specifies the position, font, color and alignment of the text string. Returns a pointer to a rectangle which just covers the extent of the text on the screen.

See also: `GrDrawLine`, `GrInstall`, `GrSetUp`, `GrDeInstall`, `GrDrawScan`, `GrDrawBackDrop`, `GrDrawSprite`, `GrUndrawSprite`, `GrScreenDone`, `GrSetPalette`, `GrSetStipples`, `GrDrawPolygon`, `GrDrawSorted`, `GrDrawUnsorted`, `GrSetStipples`, `GrSetSprites`

Example:

```
Size=GrDrawText(&Text);
```

GrDrawUnsorted

Syntax: **`T_DRAWLIST *GrDrawUnsorted(T_DRAWLIST *);`**

Description: Draws an unsorted drawing list all in one go, on the background screen. Takes a pointer which points to the drawing list, as produced by the processing routine. If not supported, returns a NULL pointer, else returns non-NULL.

This function is not supported in all device drivers—check the Abilities flags returned from `GrSetUp`.

See also: `GrDrawLine`, `GrDrawScan`, `GrDrawBackDrop`, `GrDrawSprite`, `GrUndrawSprite`, `GrDrawPolygon`, `GrDrawText`, `GrDrawSorted`

Example:

```
GrDrawUnsorted(DrawRoot);
```

GrGetBackBuffDC

Syntax: **`HDC GrGetBackBuffDC(NULL);`**

Description: Returns the device context for the back buffer, or `NULL` if not available. The device context should be released after use.

See also: `GrRelBackBuffDC`

Example:

```
DC=(HDC)GrGetBackBuffDC(NULL);
...
GrRelBackBuffDC(DC);
```

GrInstall

Syntax: **`T_GRINSTALLRET *GrInstall(T_DEVINSTALL *);`**

Description: Installs the graphics device. Takes a pointer to a device install structure, which in turn contains a pointer to a graphics install structure, which specifies the addresses of the palette, stipples, and font data, along with the drawing buffer size. Returns a pointer to a graphics return structure, containing the device ID, name and dialog box name, along with width and height of the screen, and a pointer to the internal sprite table.

Passing a NULL pointer fills in the ID, name and dialog box name fields and returns without installing the device.

See also: `GrDeInstall`

Example:

```
C_DevInstall->Config=&InstallInfo;
C_ScreenParams=GrInstall(C_DevInstall);
```

GrPalChanged

Syntax: **`void *GrPalChanged(HWND *);`**

Description: Notifies the driver that Windows has changed the palette. Takes a pointer to the handle of the current window.

See also: `GrSetPalette`

GrPickObject

Syntax: **`T_OBJFAC *GrPickObject(T_GRPICKOBJ *);`**

Description: Returns the object and facet at the specified x, y position in the specified drawing tree. If not supported returns a NULL pointer, else returns non-NULL.

This function is not supported in all device drivers—check the `Abilities` flags returned from `GrSetUp`.

See also: `GrDrawSorted`, `GrSetUp`

Example:

```
ObjFac=GrPickObject(PickObject);
```

GrPreCopy

Syntax: **`void GrPreCopy(char *);`**

Description: This is always used in conjunction with `GrScreenDone`, and takes a pointer to a flag. If the flag is non-0, copies the current background screen to the graphics card.

See also: `GrScreenDone`

Example:

```
GrPreCopy(NULL);
```

GrRelBackBuffDC

Syntax: **`HDC GrRelBackBuffDC(HDC);`**

Description: Releases the device context for the back buffer. The device context is that returned by a call to `GrGetBackBuffDC`.

See also: `GrGetBackBuffDC`

Example:

```
DC=(HDC)GrGetBackBuffDC(NULL);
...
GrRelBackBuffDC(DC);
```

GrScreenDone

Syntax: **`void GrScreenDone(char *);`**

Description: This is always used in conjunction with `GrPreCopy`. Displays the current background screen. Takes a pointer to a flag. If the pointer is NULL or the flag is 0, the screens are swapped if possible, otherwise the background screen is copied by `GrPreCopy`. This precopy/screen done mechanism is to ensure synchronous screen updates on multiple double-buffered graphics cards.

See also: `GrPreCopy`

Example:

```
GrScreenDone(NULL);
```

GrSetConfig

Syntax:	**`T_GRSETUP *GrSetConfig(short *);`**
Description:	Sets the driver mode. Takes a pointer to the driver mode required and returns the resulting setup information. Available modes are described by a `T_GRSETUP` structure returned from `GrSetUp`.
See also:	`GrSetUp`

Example:	`Short Mode=1;`
	`T_GRSETUP* Setup;`
	`Setup = GrSetConfig (&Mode);`

GrSetOneSprite

Syntax:	**`void GrSetOneSprite(T_GRSPRSINGLE *);`**
Description:	Sends altered sprite data to the graphics device. Takes a pointer to a structure defining the sprite number and some flags. The changed data must be at exactly the same memory address as it was when first sent to the graphics device by `GrSetSprites`.
See also:	`GrSetSprites`
Example:	`GrSetOneSprite(&OneSpriteSet);`

GrSetPalette

Syntax:	**`void GrSetPalette(T_GRPALETTE *);`**
Description:	Sets the color representation of several entries in the palette. Takes a pointer to a palette set structure, which specifies the RGB values to set, and which palette entries to assign them to.
See also:	`GrDrawSprite`, `GrUndrawSprite`, `GrSetStipples`, `GrSetSprites`, `GrPalChanged`
Example:	`GrSetPalette(&PalSet);`

GrSetSprites

Syntax: **`T_GRSPRITERET * GrSetSprites(T_GRSPRINSTALL *);`**

Description: Sends any altered sprite data to the graphics device. Takes a pointer to a structure defining the number of sprites, the address of the sprite table, and some flags.

See also: `GrDrawSprite`, `GrUndrawSprite`

Example: `GrSetSprites(&SpriteSet);`

GrSetStipples

Syntax: **`void GrSetStipples(T_GRSTIPPLES *);`**

Description: Sets the color representation of several entries in the stipple table. Takes a pointer to a stipple set structure, which specifies the values of the stipples to set, and which table entries to assign them to.

See also: `GrSetPalette`, `GrSetStipples`, `GrSetSprites`

Example: `GrSetStipples(&StipSet);`

GrSetUp

Syntax: **`T_GRSETUP *GrSetUp(T_GROPTIONS *);`**

Description: Sets up the graphics device. Takes a pointer to a graphics setup stucture containing various options, and returns a setup structure containing screen size, color resolution, and abilities of the device driver.

See also: `GrInstall`, `GrDeInstall`, `GrSetConfig`

Example: `GrSetUp(&Gr_Setup);`

GrUndrawSprite

Syntax: **`void GrUndrawSprite(char *);`**

Description: Restores to the screen the background saved (by `GrDrawSprite`) behind a previously drawn sprite. Do not attempt to undraw a sprite before it has been drawn. Takes a pointer to the save buffer.

See also: `GrDrawSprite`, `GrDrawSorted`, `GrDrawUnsorted`, `GrSetSprites`

Example: `GrUndrawSprite(SavePtr);`

GrUpdateObject

Syntax:	**`void GrUpdateObj(NULL);`**
Description:	Tells the device to update its internal object buffer state. This is called when objects are changed by adding or deleting data. This function is not supported in all device drivers—check the `Abilities` flags returned from `GrSetUp`.
See also:	`GrSetup`
Example:	`GrUpdateObj(NULL);`

Keyboard devices

KbDeInstall

Syntax:	**`void KbDeInstall(NULL);`**
Description:	Removes the keyboard device.
See also:	`KbInstall`, `KbSetUp`, `KbKeyReady`, `KbReadKey`
Example:	`KbDeInstall(NULL);`

KbInstall

Syntax: **`T_KBINSTALLRET *KbInstall(T_DEVINSTALL *);`**

Description: Installs the keyboard device. Takes a pointer to a device install structure, which in turn contains a NULL pointer. Returns a pointer to information about the keyboard, containing the device ID and name, a dialog box name, a matrix showing the status of each key, pointers into the keyboard buffer, a pointer to the keyboard scan code to ASCII lookup table, and the current shift and lock states.

Passing a NULL pointer fills in the ID, name and dialog box name fields and returns without installing the device.

See also: `KbDeInstall`, `KbSetUp`, `KbKeyReady`, `KbReadKey`

Example:

```
C_DevInstall->Config=NULL;
KbInfo=KbInstall(C_DevInstall);
```

KbKeyReady

Syntax: **`char *KbKeyReady(char *);`**

Description: Tests to see if a key is ready. Takes a pointer to a flag to be filled in, and returns the address of the same filled-in flag. Flag is 0 if no key is ready, otherwise non-0.

See also: `KbDeInstall`, `KbInstall`, `KbSetUp`, `KbReadKey`

Example:
```
if(*KbKeyReady(&c))
   k=KbReadKey(NULL);
```

KbReadKey

Syntax: **`T_KBKEYREC *KbReadKey(NULL);`**

Description: Reads a key from the keyboard. If no key is ready, waits until there is one. Returns a pointer to a key record structure, which includes the ASCII code of the key, shift status, and scan code.

See also: `KbDeInstall`, `KbInstall`, `KbSetUp`, `KbKeyReady`

Example:
```
c=KbReadKey(NULL)->Ascii;
```

KbSetUp

Syntax: **`T_TIMENODE *KbSetUp(T_KBSETUP *);`**

Description: Sets up the keyboard device. Takes a pointer to setup information for the keyboard (repeat rate). Returns a pointer to a timer node (see “Timer devices”) to handle keyboard repeats.

See also: `KbDeInstall`, `KbInstall`, `KbKeyReady`, `KbReadKey`

Example:
```
TmAddList(KbSetUp(&SetUpInfo));
```

Mouse devices

The mouse is controlled by Windows in VRT 5.50. The following device drivers are included in case you wish to write your own mouse device driver.

`MsDeInstall`

Syntax:	**`void MsDeInstall(NULL);`**
Description:	Removes the mouse device.
See also:	`MsInstall`, `MsSetUp`
Example:	`MsDeInstall(NULL);`

`MsInstall`

Syntax: **`T_MSINSTALLRET *MsInstall(T_DEVINSTALL *);`**

Description: Installs the mouse device. Takes a pointer to a device install structure, which in turn contains a NULL pointer. Returns a pointer to a structure containing the device ID, name and dialog box name, and an error flag, which is zero if installation was successful.

Passing a NULL pointer simply fills in the ID, name and dialog box name fields and returns without installing the device.

See also: `MsDeInstall`, `MsSetUp`

Example:

```
C_DevInstall->Config=NULL;
MsInfo=MsInstall(C_DevInstall);
```

MsSetUp

Syntax: **`T_MSPOS *MsSetUp(T_MSSETUP *);`**

Description: Sets up the mouse device. Takes a pointer to setup structure containing min and max x and y coordinates allowed, together with initial x and y mouse position. Returns a pointer to a mouse position structure, containing the current mouse position, button states, flags for movement, and an optional address for a timer node. If this is not `NULL`, it points to a timer node which must be added to the timer chain. The position itself is updated by the device driver under interrupt, and always reflects the current state of the mouse.

See also: `MsDeInstall`, `MsInstall`

Example:

```
C_MousePos=MsSetUp(&SetUpInfo);
if(C_MousePos->MouseTime!=NULL)
     TmAddList(C_MousePos->MouseTime);
```

MsUpdate

Syntax: **`void MsUpdate (Null);`**

Description: Updates the current mouse position under Windows. Polled each frame to update the position.

Network devices

NtDeInstall

Syntax: **`void NtDeInstall(NULL);`**

Description: Removes the network device.

See also: `NtInstall`, `NtSetUp`, `NtNetwork`, `NtResign`

Example: `NtDeInstall(NULL);`

NtInstall

Syntax: **`T_NTINSTALLRET *NtInstall(T_DEVINSTALL *);`**

Description: Installs the network device. Takes a pointer to a device install structure, which in turn contains a pointer to the description of the network parameters required, and returns a pointer to a structure containing the device ID, name and dialog box name, along with an error flag, which is zero if installation was successful.

Passing a NULL pointer simply fills in the ID, name and dialog box name fields and returns without installing the device.

See also: `NtDeInstall`, `NtSetUp`, `NtNetwork`, `NtResign`

Example:

```
C_DevInstall->Config=&NetParams;
Err=NtInstall(C_DevInstall)->Error;
```

NtNetwork

Syntax: **`T_NTOUTPUT *NtNetwork(T_NTINPUT *);`**

Description: Performs the networking function. Takes a pointer to input structure containing the user number, length of function buffer and a pointer to the function buffer. This is transmitted to all the other machines, and it returns with an output structure containing the number of users and all the transmitted input structures. This ensures all the information about what is happening is available to all machines on the network.

See also: `NtDeInstall`, `NtInstall`, `NtSetUp`, `NtResign`

Example:

```
Out=NtNetwork(Input);
```

NtResign

Syntax: **`void NtResign(NULL);`**

Description: Resigns this machine from the network. It now becomes a stand-alone machine, and all the other machines on the network continue unaffected.

See also: `NtDeInstall`, `NtInstall`, `NtNetwork`, `NtSetup`

Example:

```
NtResign(NULL);
```

NtSetUp

Syntax: **`void NtSetUp(T_NTSETUP *);`**

Description: Sets up the network device. Takes a pointer to setup structure containing the random number seed. This is transmitted from machine 1 and filled in on the others.

See also: `NtDeInstall`, `NtInstall`, `NtNetwork`, `NtResign`

Example:

```
SetUpInfo.Seed=0x12345678;
NtSetUp(&SetUpInfo);
Random(SetupInfo.Seed);
```

Proportional devices

PrDeInstall

Syntax: **`void PrDeInstall(NULL);`**

Description: Removes the proportional device.

See also: `PrInstall`, `PrSetUp`, `PrGetPos`

Example: `PrDeInstall(NULL);`

PrGetPos

Syntax: **`T_PRPOS *PrGetPos(NULL);`**

Description: Gets the position of the proportional device. Returns a pointer to a structure containing the state of the devices's buttons and the position of each of its axes.

See also: `PrDeInstall`, `PrInstall`, `PrSetUp`

Example: `Pos=PrGetPos(NULL);`

PrInstall

Syntax: **`T_PRINSTALLRET *PrInstall(T_DEVINSTALL *);`**

Description: Installs the proportional device. Takes a pointer to a device install structure, which in turn contains a pointer to the configuration information for the device. This is device specific and stored in the preferences file. Returns a pointer to a structure containing the device ID, name and dialog box name, and an error flag which is zero if installation was successful.

Passing a `NULL` pointer fills in the ID, name and dialog box name fields and returns without installing the device.

See also: `PrDeInstall`, `PrSetUp`, `PrGetPos`

Example:

```
C_DevInstall->Config=PropTable[1];
Err=PrInstall(C_DevInstall)->Error;
```

PrSetUp

Syntax:	**`T_PRSETUP *PrSetUp(T_PROPTABLE *);`**
Description:	Sets up the proportional device. Takes either a pointer to configuration information or a `NULL` pointer in which case the device is reset using the last configuration values. Returns a pointer to the number of axes and buttons on the device.
See also:	`PrDeInstall`, `PrInstall`, `PrGetPos`
Example:	`SetupInfo=*PrSetUp(NULL);`

Sound devices

SdDeInstall

Syntax:	**`void SdDeInstall(NULL);`**
Description:	Removes the sound device.
See also:	`SdInstall`, `SdSetUp`, `SdPlaySound`
Example:	`SdDeInstall(NULL);`

SdInstall

Syntax:	**`T_SDINSTALLRET *SdInstall(T_DEVINSTALL *);`**
Description:	Installs the sound device. Takes a pointer to a device install structure, which in turn contains a NULL pointer. Returns a pointer to a structure containing the device ID, name and dialog box name, along with an error flag, which is zero if installation was successful.
	Passing a NULL pointer fills in the ID, name and dialog box name fields and returns without installing the device.
See also:	`SdDeInstall`, `SdSetUp`, `SdPlaySound`
Example:	`C_DevInstall->Config=NULL;` `Err=SdInstall(C_DevInstall)->Error;`

SdPlaySound

Syntax:	**`long *SdPlaySound(char *);`**
Description:	Plays a sound. Takes a pointer to a string of extended MIDI data (first byte is length), which is interpreted by the sound device. Returns a pointer to a `long` handle that you can use as an argument to the following function. If the handle is `-1`, the device cannot play (for example, sounds may currently be playing).
	The handle is stored in the buffer until the `SdPlaySound` function is called, when it is overwritten by any new data. If you want to use the original handle, you must extract and store it elsewhere.
See also:	`SdDeInstall`, `SdInstall`, `SdSetUp`
Example:	`pHandle=SdPlaySound("\x03\x90\x3C\x7F");`

SdSendBlock

Syntax:	**`void SdSendBlock (Null);`**
Description:	Sends any sound data in the sound buffer to the current sound device. Called automatically by the system.

SdSetUp

Syntax:	**`void SdSetUp(char *SndBuffer);`**
Description:	Sets up a pointer for the sound driver to the sound buffer.
See also:	`SdDeInstall`, `SdInstall`, `SdPlaySound`
Example:	`SdSetUp(C_SoundBuffer+256);`

SdShutUp

Syntax:	**`void SdShutUp(NULL *);`**
Description:	Stops the sound driver. All sounds are immediately stopped. It should be passed a `NULL` pointer.
See also:	`SdDeInstall`, `SdInstall`, `SdPlaySound`
Example:	`SdShutUp(NULL);`

SdModify

Syntax: **`long *SdModify(T_SOUNDMODIFY *Modify);`**

Description: Modifies or queries the status of a sound which is currently playing or recording. It is passed a pointer to a `T_SOUNDMODIFY` structure which contains a sound handle to modify or query, command and a value. The command specifies what to do with the parameter. If the command is a query command, the value is ignored, and the sound is not modified, otherwise the value is used to modify the sound appropriately. A pointer to a `long` containing the current value of the requested aspect of the sound is returned (if the sound has been modified the new value is returned), or `NULL` if the sound handle is invalid (for example, the sound has stopped playing).

See also: `SdRecord`, `SdPlaySound`

Example:

```
Modify.Handle=*pHandle;
Modify.Command=E_SMODQUERY+E_SMODLENGTH;
while(*SdModify(&Modify)>0);
  /*Wait for end of sound*/
```

SdRecord

Syntax: **`long *SdRecord(T_SOUNDRECORD *Record);`**

Description: Starts recording from the current sound device. The `T_SOUNDRECORD` structure contains the type of the sound (`E_STSAM8` or `E_STSAM16` for 8 or 16 bit respectively), the required length (in samples), and a pointer to the buffer in which to store the sound. Returns `NULL` if the device does not support recording, or a pointer to a `long` handle. If the handle is `-1`, the device cannot record (for example, sounds may currently be playing), otherwise, it may be used as an argument to the following function.

See also: `SdModify`, `SdPlaySound`

Example:

```
pHandle=SdRecord(&Record);
if(Handle==NULL||*pHandle<0)
  return(E_ERROR);
```

Serial devices

For an example of how to get information from the serial port, see the example SER1.C (supplied with the SDK) which registers new SCL functions to read bytes from serial port COM2.

SrClrCallBack

Syntax: **`T_SRRETURN *SrClrCallBack(T_SRRELPORT *);`**

Description: Removes the routine to be called when a byte is received on a given port. Takes a pointer to a structure containing the port number. The routine is passed the received character as an argument. Returns a non-zero value if an error occurred.

See also: `SrInstall`, `SrDeInstall`, `SrSetUp`, `SrGetBytes`, `SrSendBytes`, `SrClrCallBack`, `SrFlushBuf`, `SrGetPortInfo`, `SrSetUpPort`, `SrReleasePort`

Example: `Err=SrClrCallBack(&ThisPort)->Error;`

SrDeInstall

Syntax: **`void SrDeInstall(NULL);`**

Description: Removes the serial device.

See also: `SrInstall`, `SrSetUp`, `SrGetBytes`, `SrSendBytes`, `SrSetCallBack`, `SrClrCallBack`, `SrFlushBuf`, `SrGetPortInfo`, `SrSetUpPort`, `SrReleasePort`

Example: `SrDeInstall(NULL);`

SrFlushBuf

Syntax: **`T_SRRETURN *SrFlushBuf(T_SRRELPORT *);`**

Description: Flushes all characters from the receive buffer on a given port. Takes a pointer to a structure containing the port number. Returns a non-zero value if an error occurred.

See also: `SrInstall`, `SrDeInstall`, `SrSetUp`, `SrGetBytes`, `SrSendBytes`, `SrClrCallBack`, `SrFlushBuf`, `SrGetPortInfo`, `SrSetUpPort`, `SrReleasePort`

Example: `Err=SrFlushBuf(&ThisPort)->Error;`

SrGetBytes

Syntax: **`T_SRRETURN *SrGetBytes(T_SRDATA *);`**

Description: Gets a number of bytes from a serial port. Takes a pointer to a structure containing the port number, number of bytes to receive, and buffer address. Returns a non-zero value if an error occurred.

See also: `SrInstall`, `SrDeInstall`, `SrSetUp`, `SrSetCallBack`, `SrClrCallBack`, `SrFlushBuf`, `SrGetPortInfo`, `SrSetUpPort`, `SrReleasePort`, `SrSendBytes`,

Example:

```
Err=SrGetBytes(&MyBuffer)->Error;
```

SrGetPortInfo

Syntax: **`T_SRPORTRET *SrGetPortInfo(T_SRRELPORT *);`**

Description: Gets various information about a serial port. Takes a pointer to a structure containing the serial port number. Returns a pointer to a structure containing an error code, port address, IRQ number, interrupt vector number and IRQ mask for this serial port.

See also: `SrInstall`, `SrDeInstall`, `SrSetUp`, `SrSendBytes`, `SrSetCallBack`, `SrClrCallBack`, `SrFlushBuf`, `SrGetPortInfo`, `SrSetUpPort`, `SrReleasePort`

Example:

```
ThisAddr=SrGetPortInfo
   (&ThisPort)->PortAddress;
```

SrInstall

Syntax: **`T_SRINSTALLRET *SrInstall(T_DEVINSTALL *);`**

Description: Installs the serial device. Takes a pointer to a device install structure, which in turn contains a pointer to the configuration information for the device. This is device specific and stored in the configuration file. Returns a pointer to a structure containing the device ID, name and dialog box name, along with an error flag, which is zero if installation was successful.

Passing a NULL pointer fills in the ID, name and dialog box name fields and returns without installing the device.

See also: `SrDeInstall`, `SrSetUp`, `SrGetBytes`, `SrSendBytes`, `SrSetCallBack`, `SrClrCallBack`, `SrFlushBuf`, `SrGetPortInfo`, `SrSetUpPort`, `SrReleasePort`

Example:

```
C_DevInstall->Config=SerInfo;
Err=SrInstall(C_DevInstall)->Error;
```

SrReleasePort

Syntax: **`T_SRRETURN *SrReleasePort(T_SRRELPORT *);`**

Description: Releases a port in the serial device. Takes a pointer to a structure containing the port number, and returns a pointer to an error flag. This is non-0 if the port was not already in use.

See also: `SrInstall`, `SrDeInstall`, `SrSetUp`, `SrGetBytes`, `SrSendBytes`, `SrSetCallBack`, `SrClrCallBack`, `SrFlushBuf`, `SrGetPortInfo`, `SrSetUpPort`

Example:

```
Err=SrSetUp(&MyPort)->Error;
```

SrSendBytes

Syntax: **`T_SRRETURN *SrSendBytes(T_SRDATA *);`**

Description: Sends a number of bytes to a serial port. Takes a pointer to a structure containing the port number, number of bytes to send, and buffer address. Returns a non-zero value if an error occurred.

See also: `SrInstall`, `SrDeInstall`, `SrSetUp`, `SrGetBytes`, `SrSetCallBack`, `SrClrCallBack`, `SrFlushBuf`, `SrGetPortInfo`, `SrSetUpPort`, `SrReleasePort`

Example: `Err=SrSendBytes(&MyBuffer)->Error;`

SrSetCallBack

Syntax:
```
T_SRRETURN *SrSetCallBack
  (T_SRSETCALLBACK *);
```

Description: Sets up a routine to be called when a byte is received on a given port. Takes a pointer to a structure containing the port number and the address of the routine. The routine is passed a pointer to a structure of type `T_SRCALLBACK` as an argument. Returns a non-zero value if an error occurred.

See also: `SrInstall`, `SrDeInstall`, `SrSetUp`, `SrGetBytes`, `SrSendBytes`, `SrClrCallBack`, `SrFlushBuf`, `SrGetPortInfo`, `SrSetUpPort`, `SrReleasePort`

Example: `Err=SrSetCallBack(&MyCallBack)->Error;`

SrSetUp

Syntax: **`void SrSetUp(NULL);`**

Description: Sets up the serial device. Takes either a pointer to configuration information, or a NULL pointer in which case the device is reset using the last configuration values.

See also: `SrInstall`, `SrDeInstall`, `SrGetBytes`, `SrSendBytes`, `SrSetCallBack`, `SrClrCallBack`, `SrFlushBuf`, `SrGetPortInfo`, `SrSetUpPort`, `SrReleasePort`

Example: `SrSetUp(NULL);`

SrSetUpPort

Syntax: **`T_SRRETURN *SrSetUpPort(T_SRSETPORT *);`**

Description: Sets up a port in the serial device. Takes a pointer to a structure containing the port number, baud rate and line control register contents, and returns a pointer to an error flag. This is non-0 if the port is already in use.

See also: `SrInstall`, `SrDeInstall`, `SrSetUp`, `SrGetBytes`, `SrSendBytes`, `SrSetCallBack`, `SrClrCallBack`, `SrFlushBuf`, `SrGetPortInfo`, `SrReleasePort`

Example: `Err=SrSetUp(&MyPort)->Error;`

Timer devices

TmAddList

Syntax: **`void TmAddList(T_TIMENODE *);`**

Description: Adds a timer node to the timer chain (the node must be static). Takes a pointer to the node, which contains the rate at which to call a routine, the address of the routine to call, and a pointer to the next node, which is filled in by `TmAddList`. The node must be static.

See also: `TmDeInstall`, `TmInstall`, `TmSetUp`, `TmRemList`

Example: `TmAddList(KbSetUp(&SetUpInfo));`

TmDeInstall

Syntax: **`void TmDeInstall(NULL);`**

Description: Removes the timer device.

See also: `TmInstall`, `TmSetUp`, `TmAddList`, `TmRemList`

Example: `TmDeInstall(NULL);`

TmInstall

Syntax: **`T_TMINSTALLRET *TmInstall(T_DEVINSTALL *);`**

Description: Installs the timer device. Takes a pointer to a device install structure, which in turn contains a pointer to the routine `GrDeInstall`, for use in case of fatal errors. Returns a pointer to a structure containing the device ID, name and dialog box name, along with an error flag, which is zero if installation was successful.

Passing a `NULL` pointer simply fills in the ID, name and dialog box name fields and returns without installing the device.

See also: `TmDeInstall`, `TmSetUp`, `TmAddList`, `TmRemList`

Example:

```
C_DevInstall->Config=GrDeInstall;
Err=*TmInstall(C_DevInstall);
```

TmRemList

Syntax: **`void TmRemList(T_TIMENODE *);`**

Description: Removes a timer node to the timer chain. Takes a pointer to the node, which should have been added to the chain by `TmAddList`.

See also: `TmDeInstall`, `TmInstall`, `TmSetUp`, `TmAddList`

Example: `TmRemList(&PrevNode);`

TmSetUp

Syntax: **`long *TmSetUp(NULL);`**

Description: Sets up the timer device, clearing all timer chains, and resetting all counters. Returns a pointer to the monotonic time, in milliseconds.

See also: `TmDeInstall`, `TmInstall`, `TmAddList`, `TmRemList`

Example: `C_MonoTime=TmSetUp(NULL);`

Chapter 6 - Data pointers

Introduction

On entry, the application is passed the address of the vector table contained in the main VRT program. The pointer to this table is accessible by the user’s application code and is called `API_Vectors`. It includes a set of pointers which point to useful common data.

The application can access all of the most important data areas in VRT which are available as pointers from the vector table. Macros have been defined in APP_DEFS.H to make the existence of the vector table less intrusive. All data which may be accessed in this way is referred to as ‘common data’, and all variable names defined as common data begin with `C_`.

C_AbsTime

Syntax: **`unsigned short *C_AbsTime;`**

Description: Points to the number of milliseconds between subsequent world updates when in `AbsTime` mode.

Example: `if(*C_AbsTime<100)    /* Too fast ! */`

C_Activated

Syntax: **`short *C_Activated;`**

Description: Points to the number of the last object activated by the mouse.

Example: `LastObj=*C_Activated;`

C_Ambient

Syntax: **`unsigned char *C_Ambient;`**

Description: Points to the ambient light contribution to the lighting. Ranges from 0 to 255 (100%).

See also: `C_IAmbient`

Example: `*C_Ambient=128;     /* 50% */`

C_AngleSize

Syntax: **`short *C_AngleSize;`**

Description: Points to the turning angle step used when turning objects or the viewpoint from the keyboard. This is measured in brees (1 bree = 1/65536 of a circle).

See also: `C_StepSize`

Example: `*C_AngleSize=1024;     /* 11¼° */`

C_BackBuffer

Syntax: **`char **C_BackBuffer;`**

Description: Points to an array of pointers to backdrop file buffers. For format of data in the buffers, see `T_PCXHEADER` in “Chapter 7: Data structures”.

See also: `C_BackBufLength`

Example: `BackPtr=C_BackBuffer[0];`

C_BackBufLength

Syntax: **`long *C_BackBufLength;`**

Description: Points to an array of lengths of backdrop file buffers, in bytes.

See also: `C_BackBuffer`

Example: `BackPtr+= *C_BackBufLength-768;`

C_BackFile

Syntax: **`char *C_BackFile;`**

Description: Points to the name of the last loaded backdrop file. This does not include the path.

See also: `C_BackPath`

Example:

```
strcpy(FileName,C_BackPath);
strcat(FileName,C_BackFile);
```

C_BackLen

Syntax: **`long *C_BackLen;`**

Description: Points to an array of lengths of the data in the backdrop file buffers, in bytes.

See also: `C_BackBuffer`

Example: `BackPtr+= *C_BackLen-1;`

C_BackName

Syntax:	**`char **C_BackName;`**
Description:	Points to an array of pointers to the names of each loaded Backdrop file. This does not include the path.
See also:	`C_BackPath`, `C_BackFile`
Example:	`strcpy(FileName,C_BackName[0]);`

C_BackPath

Syntax:	**`char *C_BackPath;`**
Description:	Points to the path for the last loaded backdrop file. This does not include the file name.
See also:	`C_BackFile`

Example:	`strcpy(FileName,C_BackPath);`
	`strcat(FileName,C_BackFile);`

C_BackPos

Syntax:	**`T_POINTREC2D *C_BackPos;`**
Description:	Points to an array of backdrop positions. Each record in the array specifies the x and y position of the top left of the backdrop relative to the top left of the screen.
See also:	`C_BackBuffer`

Example:	`C_BackPos[0].x=100;`
	`C_BackPos[0].y=50;`

C_BaseAddress

Syntax: **`T_FUNCPTR **C_BaseAddress;`**

Description: Points to the array containing the entry points for the device drivers.

It is not wise to use this array directly—there are macros defined for all device driver functions. For a full discussion see “Chapter 5: Device drivers”.

See also: `C_DevLen`, `C_DevsInstalled`

Example:
```
GrDrawLine(&Line);    /* Indirect use */
```

C_BufferClear

Syntax: **`unsigned char *C_BufferClear;`**

Description: Points to the mode in which the drawing buffer is cleared. This can be one of the following:

`E_HZNONE`	Not cleared.
`E_HZSOLID`	Solid color (from `C_ClearColour`)
`E_HZHORIZON`	Horizon drawn.
`E_HZBACKDROP`	Backdrop drawn (if any).
`E_HZTEXFULL`	Fully textured horizon drawn.
`E_HZTEXHALF`	Half textured horizon drawn.
`E_HZTEXSTRIPE`	Texture strip horizon drawn.

See also: `C_SkyCol`, `C_GndCol`, `C_ClearColour`

Example:
```
*C_BufferClear=E_HZHORIZON;
```

C_ClearColour

Syntax: **`unsigned char *C_ClearColour;`**

Description: Points to the current background clear color for use when the horizon is on "solid".

See also: `C_SkyCol`, `C_GndCol`, `C_BufferClear`

Example:

```
if(*C_ClearColour==E_COLTRANSPARENT)
   *C_ClearColour=E_COLBLACK;
```

C_ClipBoard

Syntax: **`char **C_ClipBoard;`**

Description: Points to the address of the VRT clipboard, or NULL if none. If there is no clipboard, one may be allocated using `malloc`. If the clipboard is not large enough, it may be extended using `realloc`. Remember to set the `owner` and `length` to appropriate values. The data placed in the clipboard can be any format whatsoever.

See also: `C_ClipOwner`, `C_ClipLen`

Example:

```
memcpy(*C_ClipBoard,MyData,Length);
```

C_ClipLen

Syntax: **`long *C_ClipLen;`**

Description: Points to the size of the VRT clipboard, in bytes.

See also: `C_ClipOwner`, `C_ClipBoard`

Example:

```
memcpy(*C_ClipBoard,MyData,Length);
C_ClipLen=Length;
```

C_ClipOwner

Syntax: **`char *C_ClipOwner;`**

Description: Points to a flag indicating the owner of the VRT clipboard. This is set up so that one type of data cannot be pasted in an inappropriate place. If placing data into the clipboard, the format of the data and the owner should be set according to where this data will be pasted. The possible owners (not all of which are used at present) are:

`E_UOWORLDED`	World Editor
`E_UOSHAPEED`	Shape Editor
`E_UOVISUALISER`	Visualiser
`E_UOCOLOURED`	Palette Editor
`E_UOTEXTED`	Text Editor (dialogs)
`E_UOSPRITEED`	Image Editor
`E_UOUSRRSCED`	Resource Editor
`E_UOSOUND`	Sound Editor

See also: `C_ClipBoard`, `C_ClipLen`

Example:

```
strcpy(*C_ClipBoard,
  ”Text in clipboard”);
*C_ClipOwner=E_UOTEXTED;
*C_ClipLen=18;
```

C_ColRanges

Syntax: **`char **C_ColRanges;`**

Description: Points to the address of the currently active range table.

See also: `C_Stipples`, `C_Palette`

Example:

```
while(i>0 && C_ColRanges[i - 1]>
  C_ColRanges[i])i--;
```

C_ConfigBuffer

Syntax:	**`char **C_ConfigBuffer;`**
Description:	Points to the address of the configuration file buffer. For format of data in the buffer, refer to “Chapter 7: Data structures”.
See also:	`C_ConfigBufLength`, `C_ConfigLen`
Example:	`CfgPtr=*C_ConfigBuffer+sizeof(T_FILEHEADER);`

C_ConfigBufLength

Syntax:	**`long *C_ConfigBufLength;`**
Description:	Points to the length of the configuration file buffer, in bytes. For format of data in the buffer, refer to “Chapter 7: Data structures”.
See also:	`C_ConfigBuffer`, `C_ConfigLen`
Example:	`CfgPtr+= *C_ConfigBufLength-1;`

C_ConfigFile

Syntax:	**`char *C_ConfigFile;`**
Description:	Points to the name of the configuration file. This does not include the path.
See also:	`C_ConfigPath`
Example:	`strcpy(FileName,C_ConfigPath);` `strcat(FileName,C_ConfigFile);`

C_ConfigLen

Syntax:	**`long *C_ConfigLen;`**
Description:	Points to the length of the data in the configuration file buffer, in bytes.
See also:	`C_ConfigBuffer`, `C_ConfigBufLength`
Example:	`CfgPtr+= *C_ConfigLen-1;`

C_ConfigPath

Syntax: **`char *C_ConfigPath;`**

Description: Points to the path for the configuration file. This does not include the file name.

See also: `C_ConfigFile`

Example:
```
strcpy(FileName,C_ConfigPath);
strcat(FileName,C_ConfigFile);
```

C_Console

Syntax: **`T_CONSOLE *C_Console;`**

Description: Points to the currently active console.

Example:
```
CentreX=C_Console->WindXMid;
```

C_CopyFlag

Syntax: **`char *C_CopyFlag;`**

Description: Points to a flag. If this flag is set (non-zero) then the background screen is copied onto the foreground screen during a screen swap rather than using screen switching. In zero, screen switching is used if at all possible.

Example:
```
if(*C_CopyFlag)
  ClearScreen();
```

C_Counters

Syntax: **`long *C_Counters;`**

Description: Points to the array containing the counter values used by SCL. The size of the array is `E_MAXCOUNTERS` elements.

See also: `C_Markers`

Example:
```
C_Counters[21]=200000;
```

C_CurrentPalette

Syntax: **unsigned char **C_CurrentPalette;**

Description: Points to the address of the currently displayed palette.

Example:
```
/* Alter the ‘R’ component of color 16 */
*C_CurrentPalette[16*3+0]=255;
```

C_CurrentVP

Syntax: **char *C_CurrentVP;**

Description: Points to the number of the currently selected viewpoint.

See also: C_VPType

Example:
```
if(*C_CurrentVP==1)
  /* Still on the default viewpoint */
```

C_Day

Syntax: **char *C_Day;**

Description: Points to the day portion of VRT’s copy of the realtime clock. This is the day number within the month, from 1 to 31.

See also: C_Year, C_Month, C_Hour, C_Minute, C_Second

Example:
```
if(*C_Day==23 && *C_Month==5)
  /* Happy birthday Sean */
```

C_DebugSCL

Syntax: **`long *C_DebugSCL;`**

Description: Points to the SCL debug flags. This is a bit vector which consists of combinations of the following bits:

`E_DEBUGSOME`	Debug only selected objects.
`E_DEBUGALL`	Debug all objects.
`E_DEBUGTHISONE`	Set when debugging an object.
`E_DEBUGSTRINGS`	Display string pointers as strings.
`E_DEBUGCHARS`	Display chars as characters.
`E_DEBUGHEX`	Display values in hexadecimal.

Example:
```
if(*C_DebugSCL&E_DEBUGTHISONE)
  /* We’re being debugged */
```

C_DefResolution

Syntax: **`unsigned char *C_DefResolution;`**

Description: Points to the initial (default) resolution for the screen. This is normally 0, the default resolution for the graphics card. Different cards have different resolution values.

See also: `C_Resolution`

Example: `ResizeScreen(*C_DefResolution);`

C_DetailLevel

Syntax: **`short *C_DetailLevel;`**

Description: Points to the current detail level setting, as set up in the Setup dialog box. Normally ranges from +10 to -10.

See also: `C_TextureScale`

Example: `*C_DetailLevel=0;`

C_DeviceActive

Syntax: **`char *C_DeviceActive;`**

Description: Points to an array of active device numbers. This is effectively a two dimensional array. Each device type (in order) has `E_MAXPERDEV` entries in this array. The values are the actual device driver numbers.

The access to this array is simplified by using the defined macro `ActiveDevice` (see “Chapter 5: Device drivers”).

See also: `C_DeviceEnabled`, `ActiveDevice`

Example:

```
/* Get number of primary graphics device */
*C_GraphicsDD=C_DeviceActive
  [E_MAXPERDEV*E_DEVGRAPHICS+0];
/* ...Or do it with the macro */
*C_GraphicsDD=ActiveDevice
  (E_DEVGRAPHICS,0);
```

C_DeviceDial

Syntax: **`char **C_DeviceDial;`**

Description: Points to an array of device driver dialog box names. This is effectively a two dimensional array. Each device type (in order) has `E_MAXPERDEV` entries in the array. Each entry is a pointer to the dialog box used to edit its configuration information.

You cannot use VRT resources to configure proportional devices under Windows. You need to configure the device manually.

See also: `C_DeviceName`

Example:

```
/* Get current sound device setup box name */
SoundDial=C_DeviceDial
   [E_MAXPERDEV*E_DEVSOUND+*C_SoundDD];
```

C_DeviceEnabled

Syntax:	**`char *C_DeviceEnabled;`**

Description: Points to an array of device enabled flags. This is effectively a two dimensional array. Each device type (in order) has `E_MAXPERDEV` entries in this array. The values are bitvectors defined as follows:

Bit 0	Set if device enabled
Bits 1-7	Reserved

The access to this array is simplified by using the defined macro `DeviceEnabled` (see “Chapter 5: Device drivers”).

See also: `C_DeviceActive`, `ActiveDevice`

Example:

```
/* See if secondary graphics enabled */
if(*C_DeviceEnabled)
   [E_MAXPERDEV*E_DEVGRAPHICS+1]&1)
/* Yes it is */;
/* ...Or do it with the macro */
if(DeviceEnabled(E_DEVGRAPHICS,1)&1)
  /* Still is */;
```

C_DeviceID

Syntax: **`unsigned char *C_DeviceID;`**

Description: Points to an array of device ID numbers. This is effectively a two dimensional array. Each device type (in order) has `E_MAXPERDEV` entries in this array. The values are the device ID fields for the devices and may take the following:

Type	**ID**	**Description**
`E_DEVGRAPHICS`	`E_GRIDTIGA`	TIGA
	`E_GRIDSVGA`	SVGA
	`E_GRIDMCA`	MCGA
	`E_GRIDD3D`	Direct3D
`E_DEVKEYBOARD`	`E_KBIDSTANDARD`	IBM keyboard
`E_DEVPROP`	`E_PRODSBALL`	Spaceball
	`E_PRIDJOYSTICK`	Analog joystick
	`E_PRIDFLOB`	Flock of Birds
	`E_PRIDFASTRAK`	Polhemus FASTRAK
	`E_PRIDSMOUSE`	Spacemouse
	`E_PRIDMOUSE`	Proportional mouse
	`E_PRID6MOUSE`	Logitech 6-D mouse
	`E_PRIDJOYSTICK2`	2nd Joystick
`E_DEVMOUSE`	`E_MSIDMICROSOFT`	Microsoft mouse
`E_DEVNETWORK`	`E_NTIDCLARKSON`	Clarkson/Ethernet
	`E_NTIDNETBIOS`	NetBios
`E_DEVSOUND`	`E_SDIDMIDI`	Music Quest MIDI
	`E_SDIDADLIBGOLD`	AdLib Gold 1000
	`E_SDIDSOUNDBL`	SoundBlaster
	`E_SDIDSOUNDBL16`	SoundBlaster 16 ASP
`E_DEVTIMER`	`E_TMIDSTANDARD`	Standard timer device
`E_DEVSERIAL`	`E_SRIDSTANDARD`	Standard serial device

Note: Additional device drivers may be issued from time to time.

See also: C_DeviceEnabled, ActiveDevice, DeviceEnabled

Example:

```
/* Find SVGA graphics device, if any */
for(i=0;i<E_MAXPERDEV;i++)
if(C_DeviceID[E_MAXPERDEV*E_DEVGRAPHICS+i]==E_GRIDSVGA)
/* SVGA is device number "i" */
```

C_DeviceName

Syntax: **char **C_DeviceName;**

Description: Points to an array of device driver names. This is effectively a two dimensional array. Each device type (in order) has E_MAXPERDEV entries in the array. Each entry is a pointer to the name of that particular device driver. This is the name displayed in the device driver setup dialog box.

See also: C_DeviceDial

Example:

```
/* Get current sound device name */
SoundName=C_DeviceName
   [E_MAXPERDEV* E_DEVSOUND+*C_SoundDD];
```

C_DeviceSetup

Syntax: **T_PROPSETUP **C_DeviceSetup;**

Description: Points to an array of addresses of device driver setup data. This is effectively a two dimensional array. Each device type (in order) has E_MAXPERDEV entries in the array. Each entry is a pointer to the data used when setting up the device driver. This data in turn consists of a length byte, followed by up to 255 bytes of device dependant setup data.

See also: C_DeviceDial

Example:

```
/* Get pointer to sound device setup data */
SoundSet=C_DeviceSetup
   [E_MAXPERDEV* E_DEVSOUND+*C_SoundDD];
```

C_DevInstall

Syntax: **`T_DEVINSTALL *C_DevInstall;`**

Description: Points to the device information chunk, `T_DEVINSTALL`.

C_DevLen

Syntax: **`long *C_DevLen;`**

Description: Points to the length of the system device driver file, in bytes.

Example: `l= *C_DevLen;`

C_DevPath

Syntax: **`char *C_DevPath;`**

Description: Points to the path for the system device driver file. This does not include the file name.

Example: `strcpy(FileName,C_DevPath);`

C_DevsInstalled

Syntax: **`short *C_DevsInstalled;`**

Description: Points to the array containing the number of each type of device driver installed. For a full discussion, see “Chapter 5: Device drivers”.

See also: `C_DeviceActive`, `C_DeviceEnabled`

Example:

```
if(C_DevsInstalled[E_DEVTIMER]==0)
  printf(“Timer not installed”);
```

C_DialSaveBuffer

Syntax: **`char **C_DialSaveBuffer;`**

Description: Included for backwards compatibility.

C_DiskFlags

Syntax: **`long *C_DiskFlags;`**

Description: Points to a set of flags describing the disks attached to the VRT system. Bit 0 is set if drive A: exists, bit 1 for drive B:, bit 2 for drive C: and so on.

Example:

```
if(*C_DiskFlags&2)
  /* Drive B: exists */
```

C_DllNumber

Syntax: **`short *C_DllNumber;`**

Description: The number of the slot into which a DLL has been loaded. The first loaded DLL has slot 0, the next has slot 1, and so on.

C_DrawAdd

Syntax: **`unsigned char **C_DrawAdd;`**

Description: Points to the next free address in the buffer used for storing the drawing list. If any additional information is to be added to the drawing list, this is the place to put it.

See also: `C_DrawRoot`, `C_DrawList`

Example:

```
Draw=(T_DCOBJECT *)*C_DrawAdd;
*Draw++=&NewInfo;
*C_DrawAdd=(char *)Draw;
```

C_DrawList

Syntax: **`unsigned char **C_DrawList;`**

Description: Points to the address of the buffer used for storing the drawing list. This is the information that is generated from the world, sorted, and drawn onto the screen by `GrDrawSorted`. It is a tree of information about the rendering of objects (see “Chapter 7: Data structures”).

See also: `C_DrawRoot`, `C_DrawAdd`

Example: `Draw=*C_DrawList;`

C_DrawListSize

Syntax: **`long *C_DrawListSize;`**

Description: Points to the length of the buffer used for storing the drawing list.

See also: `C_DrawRoot`, `C_DrawAdd`, `C_DrawList`

Example:

```
if(*C_DrawListSize<100000)
  /* It’s a bit small */
```

C_DrawRoot

Syntax: **`unsigned char **C_DrawRoot;`**

Description: Points to the address of the root object in the buffer used for storing the drawing list. This is known only after the drawing list has been sorted.

See also: `C_DrawList`, `C_DrawAdd`

Example: `Draw=*C_DrawRoot;`

C_EditorPrefs

Syntax: **`T_EDITCONFIG **C_EditorPrefs;`**

Description: Points to the editor preferences chunk `T_EDITCONFIG`. For the format of this chunk see “Chapter 7: Data structures”.

See also: `C_ExtraConfig`

Example: `Collision_cuboid_col=(*C_EditorPrefs)->CollCubeCol;`

C_EditBuffer

Syntax: **`char **C_EditBuffer;`**

Description: Points to the address of the buffer used for storing information during the dialog box routine. Set this address to point to your own buffer before calling the dialog box routine, and restore it afterward.

Example:

```
OldEdBuf=*C_EditBuffer;
*C_EditBuffer=String;
Dialogue(“EDIT_STRING”);
*C_EditBuffer=OldEdBuf;
```

C_ErrorAddress

Syntax: **`void **C_ErrorAddress;`**

Description: Points to the address of the offending character after an SCL compile-time error. This is, for instance, used to position the cursor in the SCL dialog box after an error.

Example:

```
strncpy(MyError,*C_ErrorAddress,16);
```

C_ErrorMessage

Syntax: **`char **C_ErrorMessage;`**

Description: Points to an array containing the most error messages. These are pointers to the actual text in the `C_SysMessage` array. If an error occurs, the error return is usually a negative number. In the case of SCL runtime and compile-time errors indexing into this table by the absolute value of the error number gives you a (hopefully) descriptive error message.

Example:

```
Error=Execute(SCLCode);
if(Error<0)
   printf(C_ErrorMessage [-Error]);
```

C_ErrorString

Syntax: **`char **C_ErrorString;`**

Description: Points to the address of the string to be displayed on exit from the program. Normally points to “Program terminated normally”, but may be set immediately before calling `_Terminate` to display a different message.

Example:

```
*C_ErrorString=”ERROR: Database corrupt”;
  _Terminate();
```

C_ExtraConfig

Syntax: **`T_EXTRACONFIG **C_ExtraConfig;`**

Description: Points to the address of the extra configuration chunk in the current configuration buffer. For the format of this chunk, see “Chapter 7: Data structures”.

See also: `C_PrefsExtraConfig`

Example:

```
ambient_red=(*C_ExtraConfig)->AmbinetR;
```

C_FacetCount

Syntax: **`long *C_FacetCount;`**

Description: Points to the number of facets drawn in the last window onto the world.

Example:

```
Average+= *C_FacetCount;
```

C_Failed

Syntax: **`char *C_Failed;`**

Description: Points to a flag which is set (non-zero) if `Reinsert` could not reinsert the specified data into its buffer.

Example:

```
if(*C_Failed)
/* Error - cannot reinsert data */
```

C_FilesChanged

Syntax: **`short *C_FilesChanged;`**

Description: Points to a short bitvector which indicates which files have changed since they were last saved or loaded.

The bits are defined as follows:

`E_MTYPECONFIG`	Config file changed
`E_MTYPEWORLD`	World file changed
`E_MTYPESHAPE`	Shape file changed
`E_MTYPEPALETTE`	Palette file changed
`E_MTYPEFONT`	Font file changed
`E_MTYPERESOURCE`	Resource file changed
`E_MTYPESPRITE`	Image (sprite) file changed
`E_MTYPEDEVICES`	Device driver file changed
`E_MTYPEMESSAGE`	Message file changed
`E_MTYPEBACKDROP`	Backdrop file changed
`E_MTYPEPREFS`	Preferences file changed
`E_MTYPEUSERRSC`	User resource file changed
`E_MTYPEVRT`	VRT file changed
`E_MTYPEUSERSPR`	User image file changed
`E_MTYPEAPPCODE`	Application program file changed
`E_MTYPESOUNDS`	Sound file changed

Example:

```
c=C_FilesChanged;
```

C_FirstPal

Syntax: **`unsigned char *C_FirstPal;`**

Description: Points to the index of the first palette item to be updated. All the palette entries in `C_Palette` from `C_FirstPal` to `C_LastPal` are set when the next frame is drawn, and `C_FirstPal` and `C_LastPal` are set to invalid values. `C_FirstPal` must be less than or equal to `C_LastPal` for any colors to be updated.

See also: `C_LastPal`

Example:

```
/* Update colors 16-31 */
C_FirstPal=16;
C_LastPal=31;
```

C_FontBuffer

Syntax: **`char **C_FontBuffer;`**

Description: Points to the address of the font file buffer. For format of data in the buffer, refer to “Chapter 7: Data structures”.

See also: `C_FontBufLength`

Example:

```
FntPtr=*C_FontBuffer+sizeof(T_FILEHEADER);
```

C_FontBufLength

Syntax: **`long *C_FontBufLength;`**

Description: Points to the length of the font file buffer, in bytes. For format of data in the buffer, refer to “Chapter 7: Data structures”.

See also: `C_FontBuffer`

Example:

```
FntPtr+= *C_FontBufLength-1;
```

C_FontFile

Syntax: **`char *C_FontFile;`**

Description: Points to the name of the font file. This does not include the path.

See also: `C_FontPath`

Example:
```
strcpy(FileName,C_FontPath);
strcat(FileName,C_FontFile);
```

C_FontLen

Syntax: **`long *C_FontLen;`**

Description: Points to the length of the data in the font file buffer, in bytes.

See also: `C_FontBuffer`

Example:
```
FntPtr+= *C_FontLen-1;
```

C_FontPath

Syntax: **`char *C_FontPath;`**

Description: Points to the path for the font file. This does not include the file name.

See also: `C_FontFile`

Example:
```
strcpy(FileName,C_FontPath);
strcat(FileName,C_FontFile);
```

C_FuncBuf

Syntax: **`T_FUNCTION *C_FuncBuf;`**

Description: Points to the function buffer. This is a circular buffer where the function numbers from the keyboard and other input devices are stored before they are obeyed. Writing functions into this buffer is best done using `WriteFunc`.

See also: `C_FuncBufHead`, `C_FuncBufTail`

Example:
```
FuncPtr=C_FuncBuf;
```

C_FuncBufHead

Syntax: **`short *C_FuncBufHead;`**

Description: Points to the head index of the function buffer. This is where the next function to be placed in the function buffer would go. This can be used to add functions to the buffer if required, but it is neater and safer to use `WriteFunc`.

See also: `C_FuncBuf`, `C_FuncBufTail`

Example:

```
C_FuncBuf[(*C_FuncBufHead)]=SubstituteFunction;
```

C_FuncBufTail

Syntax: **`short *C_FuncBufTail;`**

Description: Points to the tail index of the function buffer. This is where the next function to be obeyed in the function buffer would be read from. If this is the same as `*C_FuncBufHead`, the buffer is empty.

See also: `C_FuncBuf`, `C_FuncBufHead`

Example:

```
if(*C_FuncBufHead==*C_FuncBufTail)
  /* Buffer empty */;
```

C_FuncNames

Syntax: **`T_FUNCNAME *C_FuncNames;`**

Description: Points to an array of function names. Each function name record consists of the function number, its name and some flags (see “Chapter 7: Data structures” for more details). Finding the name of a function thus involves looking through this table until the function numbers match, then grabbing the function name from the same place.

See also: `C_FuncNameSize`

Example:

```
for(i=0;i<*C_FuncNameSize;i++)
   if(C_FuncNames[i].Function==FuncNum)
  {
     Name=C_FuncNames[i].Name;
    break;
  }
```

C_FuncNameSize

Syntax: **`short *C_FuncNameSize;`**

Description: Points to the number of function names in `C_FuncNames`.

See also: `C_FuncNames`

Example:

```
for(i=0;i<*C_FuncNameSize;i++)
   if(C_FuncNames[i].Function==FuncNum)
  {
     Name=C_FuncNames[i].Name;
    break;
  }
```

C_GndBands

Syntax: **`unsigned char *C_GndBands;`**

Description: Points to the number of bands displayed on the ground by the horizon routine.

See also: `C_SkyBands`, `C_GndCol`, `C_GndGrad`

Example: `*C_GndBands=0;`

C_GndCol

Syntax: **`unsigned char *C_GndCol;`**

Description: Points to the color of the ground.

See also: `C_SkyCol`, `C_GndBands`, `C_GndGrad`

Example: `*C_GndCol=E_COLBLACK;`

C_GndGrad

Syntax: **`unsigned char *C_GndGrad;`**

Description: Points to the color increment between successive bands on the ground (usually between -2 and 2).

See also: `C_SkyGrad`, `C_GndBands`, `C_GndCol`

Example: `*C_GndGrad=1;`

C_GraphicsDD

Syntax: **`short *C_GraphicsDD;`**

Description: This variable points to the number of the currently active graphics device. This is automatically used when calling the device drivers, and as such can quite safely be ignored.

This may be changed temporarily in order to use a different device from the default, but it must be set back to its original value before returning to the main program.

See also: `C_KeyboardDD`, `C_MouseDD`, `C_NetworkDD`, `C_PropDD`, `C_SoundDD`, `C_TimerDD`, `C_SerialDD`

Example:

```
/* Draw text on secondary   */
/* graphics device          */

OldGraphicsDev=*C_GraphicsDD;
*C_GraphicsDD=ActiveDevice
  (E_DEVGRAPHICS,1);
GrDrawText(&Text);
*C_GraphicsDD=OldGraphicsDev;
```

C_Hour

Syntax: **`char *C_Hour;`**

Description: Points to the hour portion of VRT’s copy of the realtime clock. This is in 24 hour format running from 0 to 23.

See also: `C_Year`, `C_Month`, `C_Day`, `C_Minute`, `C_Second`

Example:

```
if(*C_Hour==0 && *C_Minute==0)
  /* Midnight */
```

C_IAmbient

Syntax: **`unsigned char *C_IAmbient;`**

Description: Points to the initial ambient light contribution to the lighting. Ranges from 0 to 255 (100%).

See also: `C_Ambient`

Example:

```
*C_Ambient=*C_IAmbient;
```

C_InsVal

Syntax:	**`T_LONGORPTR *C_InsVal;`**
Description:	Points to the array containing the primary instrument values.
See also:	`C_InsVal2`
Example:	`(*C_InsVal)[12].l=1000;` `(*C_InsVal)[13].p=”This is a door”;`

C_InsVal2

Syntax:	**`T_LONGORPTR *C_InsVal2;`**
Description:	Points to the array containing the secondary instrument values.
See also:	`C_InsVal`
Example:	`(*C_InsVal2)[12].l=0x0F020500;`

C_Keyboard

Syntax:	**`volatile T_KBINSTALLRET **C_Keyboard;`**
Description:	Points to the address of the structure returned after installing the keyboard driver. This includes various useful information about the current state of the keyboard, and is updated under interrupt.
See also:	`C_KeyTable`
Example:	`if((*C_Keyboard)->LockFlags&E_KBSCROLLLOCK)` `/* Scroll lock is active */`

C_KeyboardDD

Syntax: **`short *C_KeyboardDD;`**

Description: This variable points to the number of the currently active keyboard device. This is automatically used when calling the device drivers, and as such can quite safely be ignored.

This may be changed temporarily in order to use a different device from the default, but it must be set back to its original value before returning to the main program.

See also: `C_GraphicsDD`, `C_MouseDD`, `C_NetworkDD`, `C_PropDD`, `C_SoundDD`, `C_TimerDD`, `C_SerialDD`

Example:

```
/* Get type of keyboard we are using */
KbType=C_DeviceID
   [E_MAXPERDEV*E_DEVKEYBOARD+*C_KeyboardDD];
```

C_KeyTable

Syntax: **`char **C_KeyTable;`**

Description: Points to the address of the key table in the preferences file, if any. If no keyboard table is defined, this is NULL.

See also: `C_Keyboard`

Example:

```
if(*C_KeyTable==NULL)
  /* Using default (UK) keyboard table */
```

C_LastDialName

Syntax: **`char **C_LastDialName;`**

Description: Points to the address of the name of the last dialog box used when switching between two dialog boxes (such as the Edit SCL dialog box).

Example: `RetVal=Dialogue(C_LastDialName);`

C_LastPal

Syntax: **`unsigned char *C_LastPal;`**

Description: Points to the index of the last palette item to be updated. All the palette entries in `C_Palette` from `C_FirstPal` to `C_LastPal` are set when the next frame is drawn, and `C_FirstPal` and `C_LastPal` are set to invalid values. `C_FirstPal` must be less than or equal to `C_LastPal` for any colors to be updated.

See also: `C_LastPal`

Example:

```
/* Update colors 16-31 */
C_FirstPal=16;
C_LastPal=31;
```

C_LayerFlags

Syntax: **`unsigned short *C_LayerFlags;`**

Description: Points to the array containing the status for each layer. Each layer has one bit which is set if it is visible. Layer 0 is bit 0 of the first element of this array, layer 1 is bit 1, and so on, then layer 16 is bit 0 of the second element, up to layer 255.

Example:

```
C_LayerFlags[6]|=0x0010;
  /* Layer 100 = ON */
```

C_Light

Syntax: **`T_LIGHT *C_Light;`**

Description: Points to an array of converted light sources. These are the light sources that are actually used in the lighting calculations.

See also: `C_SunLight`, `C_Ambient`, `C_NumLights`

Example:

```
C_Light[0].Brightness=10000;
```

C_LongAppPath
C_LongBackPath
C_LongConfigPath
C_LongConvPath
C_LongConvModPath
C_LongFontPath
C_LongMessPath
C_LongPalPath
C_LongPrefsPath
C_LongResourcePath
C_LongShapePath
C_LongSoundPath
C_LongUserRsrcPath
C_LongUserSprPath
C_LongWorldPath

Syntax:

```
char *C_LongAppPath
char *C_LongBackPath
char *C_LongConfigPath
char *C_LongConvPath
char *C_LongConvModPath
char *C_LongFontPath
char *C_LongMessPath
char *C_LongPalPath
char *C_LongPrefsPath
char *C_LongResourcePath
char *C_LongShapePath
char *C_LongSoundPath
char *C_LongUserRsrcPath
char *C_LongUserSprPath
char *C_LongWorldPath
```

Desription: Points to the path of the corresponding file (`App` for application, `Back` for Background and so on). Each path can be up to `E_MAXPATH` characters long, and must not exceed 256 characters. This does not include the filename.

Example:

```
strcpy(Filename,  C_LongConfigPath);
strcat(Filename,  C_LongConfigFile);
```

C_LongAppFile
C_LongBackFile
C_LongConfigFile
C_LongConvFile
C_LongConvModFile
C_LongFontFile
C_LongMessFile
C_LongPalFile
C_LongPrefsFile
C_LongResourceFile
C_LongShapeFile
C_LongSoundFile
C_LongUserRsrcFile
C_LongUserSprFile
C_LongWorldFile

Syntax:

```
char *C_LongAppFile
char *C_LongBackFile
char *C_LongConfigFile
char *C_LongConvFile
char *C_LongConvModFile
char *C_LongFontFile
char *C_LongMessFile
char *C_LongPalFile
char *C_LongPrefsFile
char *C_LongResourceFile
char *C_LongShapeFile
char *C_LongSoundFile
char *C_LongUserRsrcFile
char *C_LongUserSprFile
char *C_LongWorldFile
```

Desription: Points to the name of the corresponding file (`App` for application, `Back` for Background and so on) that can be up to `E_MAXFILE` characters long with a limit of 256 characters. This does not include the path.

Example:

```
strcpy(Filename, C_LongConfigPath);
strcat(Filename, C_LongConfigFile);
```

C_Markers

Syntax: **`char *C_Markers;`**

Description: Points to the array containing the marker values used by SCL. The size of the array is `E_MAXMARKERS` elements.

See also: `C_Counters`

Example: `C_Markers[12]=-1;`

C_Message

Syntax: **`char **C_Message;`**

Description: Points to an array containing pointers to each message. Messages off the end of the message buffer have NULL addresses.

See also: `C_MessBuffer`

Example: `MessPtr=C_Message[12];`

C_MessBuffer

Syntax: **`char **C_MessBuffer;`**

Description: Points to the address of the message file buffer. For format of data in the buffer, refer to “Chapter 7: Data structures”.

See also: `C_MessBufLength`

Example: `MessPtr=*C_MessBuffer+sizeof(T_FILEHEADER);`

C_MessBufLength

Syntax: **`long *C_MessBufLength;`**

Description: Points to the length of the message file buffer, in bytes. For format of data in the buffer, refer to “Chapter 7: Data structures”.

See also: `C_MessBuffer`

Example: `MessPtr+=  *C_MessBufLength-1;`

C_MessFile

Syntax:	**`char *C_MessFile;`**
Description:	Points to the name of the message file. This does not include the path.
See also:	`C_MessPath`

Example:	`strcpy(FileName,C_MessPath);`
	`strcat(FileName,C_MessFile);`

C_MessLen

Syntax:	**`long *C_MessLen;`**
Description:	Points to the length of the data in the message file buffer, in bytes.
See also:	`C_MessBuffer`
Example:	`MessPtr+= *C_MessLen-1;`

C_MessPath

Syntax:	**`char *C_MessPath;`**
Description:	Points to the path for the message file. This does not include the file name.
See also:	`C_MessFile`

Example:	`strcpy(FileName,C_MessPath);`
	`strcat(FileName,C_MessFile);`

C_Minute

Syntax:	**`char *C_Minute;`**
Description:	Points to the minute portion of VRT's copy of the realtime clock.
See also:	`C_Year`, `C_Month`, `C_Day`, `C_Hour`, `C_Second`

Example:	`if(*C_Hour==0 && *C_Minute==0)`
	`  /* Midnight */`

C_MonoTime

Syntax: **volatile long **C_MonoTime;**

Description: Points to the address of the monotonic timer. The timer always increases, measuring time in milliseconds and is updated under interrupt from the timer device driver. Although you may write to this variable, it is not advisable, since several system functions rely on the fact that it always increases at a constant rate.

Example:

```
First=**C_MonoTime;
TimedRoutine();
sprintf(String,”Time:%ldms”,  **C_MonoTime-First);
```

C_Month

Syntax: **char *C_Month;**

Description: Points to the month portion of VRT’s copy of the realtime clock. This is the month number 1 to 12.

See also: C_Year, C_Day, C_Hour, C_Minute, C_Second

Example:

```
if(*C_Day==26 && *C_Month==2)
  /* Happy birthday Huw  */
```

C_MouseDD

Syntax: **short *C_MouseDD;**

Description: This function is not used in VRT 5.50 as the mouse is controlled by Windows. The function is included for compatibility with earlier versions of VRT.

Points to the number of the currently active mouse device. This is automatically used when calling the device drivers, and as such can quite safely be ignored. This may be changed temporarily in order to use a different device from the default, but it must be set back to its original value before returning to the main program.

Example:

```
/* Get name of mouse device we are using */
MsName=C_DeviceName
   [E_MAXPERDEV*E_DEVMOUSE+*C_MouseDD];
```

C_MousePos

Syntax: **`volatile T_MSPOS **C_MousePos;`**

Description: This function is not used in VRT 5.50 as the mouse is controlled by Windows. The function is included for compatibility with earlier versions of VRT.

Points to the address of the mouse position structure, which contains the mouse's position and button state. It is updated under interrupt from the mouse driver.

Example:

```
x=(*C_MousePos)->XPos;
```

C_MouseMode
C_MouseMode2
C_MouseHomePos

Syntax:

```
short * C_MouseMode;
short * C_MouseMode2;
T_POINTREC2D * C_MouseHomePos;
```

Description: This function is not used in VRT 5.50 as the mouse is controlled by Windows. The function is included for compatibility with earlier versions of VRT.

Variables used by the mouse movement device driver. `C_MouseMode` is the current mouse movement mode, a combination of flag values (definitions begin `E_MSCTL`). `C_MouseMode2` is set up in the main code and is copied to `C_MouseMode` on installation of the device. This allows the correct synchronization between turning mouse movement on and the rest of the system. `C_MouseHomePos` is the position of the mouse's 'home base'—the center of the small box on the screen.

C_MsgFree

Syntax: **`T_CONMSG **C_MsgFree;`**

Description: Points to the address of the next free space in the SCL inter-object message buffer. For a description of the layout of the buffer, see "Chapter 7: Data structures".

See also: `C_MsgHead`, `C_MsgTail`

Example:

```
if(*C_MsgFree==NULL)
  /* No free space */
```

C_MsgHead

Syntax: **`T_CONMSG **C_MsgHead;`**

Description: Points to the address of where the next message should be placed in the SCL inter-object message buffer.

See also: `C_MsgTail`, `C_MsgFree`

Example:

```
(*C_MsgHead)=Message;
*C_MsgHead=*C_MsgFree;
*C_MsgFree=(*C_MsgFree)->Next;
```

C_MsgTail

Syntax: **`T_CONMSG **C_MsgTail;`**

Description: Points to the address of the next message in the SCL inter-object message buffer.

See also: `C_MsgHead`, `C_MsgFree`

Example:

```
if((*C_MsgTail)->Destination==Me)
  /* Message is for me */
```

C_NetworkDD

Syntax: **`short *C_NetworkDD;`**

Description: This variable points to the number of the currently active network device. This is automatically used when calling the device drivers, and as such can quite safely be ignored.

Do not change this device number, as it will cause extreme problems for other users on the network.

See also: `C_GraphicsDD`, `C_KeyboardDD`, `C_MouseDD`, `C_PropDD`, `C_SoundDD`, `C_TimerDD`, `C_SerialDD`

Example:

```
/* Get name of network device */

NtName=C_DeviceName
   [E_MAXPERDEV*E_DEVNETWORK+*C_NetworkDD];
```

C_NumDevsActive

Syntax:	**`unsigned char *C_NumDevsActive;`**
Description:	Points to an array containing the number of active devices for each device type.
See also:	`C_DeviceActive`

Example:

```
if(C_DeviceActive[E_DEVNETWORK]==0)
  /* We're on our own */
```

C_NumDialogues

Syntax:	**`short *C_NumDialogues;`**
Description:	Points to the number of system dialog boxes defined.
See also:	`C_NumUserDials`

Example:

```
if(C_NumDialogues<10)
  /* We must have lost some! */
```

C_NumLights

Syntax:	**`short *C_NumLights;`**
Description:	Points to the number of active lights which have been placed in the `C_Light` array.
See also:	`C_SunLight`, `C_Ambient`, `C_Light`

Example:

```
if(C_NumLights==0)  /* Dark in here, isn't it ? */
```

C_NumPlayers

Syntax:	**`short *C_NumPlayers;`**
Description:	Points to the number of users on the system.
See also:	`C_PlayerView`, `C_ThisPlayer`, `C_Player`

Example:

```
if (*C_NumPlayers<2)    /* I am alone */
```

C_NumSprites

Syntax: **short *C_NumSprites;**

Description: Points to the number of system images (sprites) defined.

See also: `C_Sprite`, `C_NumUserSprites`

Example:

```
GrDrawSprite(*C_Sprite+*C_NumSprites-1);
```

C_NumUserDials

Syntax: **short *C_NumUserDials;**

Description: Points to the number of user dialog boxes defined.

See also: `C_NumDialogues`

Example:

```
if(C_NumUserDials==0)
  /* None of our own */
```

C_NumUserSprites

Syntax: **short *C_NumUserSprites;**

Description: Points to the number of user images (sprites) defined.

See also: `C_UserSprite`, `C_NumSprites`

Example:

```
GrDrawSprite
   (*C_UserSprite+*C_NumUserSprites-1);
```

C_ObjChanged

Syntax: **unsigned char *C_ObjChanged;**

Description: Points to the array of object changed flags. The array is indexed by object number and each element is set to 0xFF on reset or when its object changes. The interpretation and use of these flags is up to the graphics device.

Example:

```
if(ObjChanged[ObjNum]==0xFF)
{
    /*  remake object  */
    ObjChanged[ObjNum]=0;
}
```

C_OldPalette

Syntax: **`char **C_OldPalette;`**

Description: Points to the address of the palette which was last copied to `C_Palette`.

See also: `C_Stipples`, `C_ColRanges`, `C_Palette`

Example: `memcpy(*C_Palette,*C_OldPalette,768);`

C_PalBuffer

Syntax: **`char **C_PalBuffer;`**

Description: Points to the address of the palette file buffer. For format of data in the buffer, refer to "Chapter 7: Data structures".

See also: `C_PalBufLength`

Example:

```
PalPtr=*C_PalBuffer+
   sizeof(T_FILEHEADER);
```

C_PalBufLength

Syntax: **`long *C_PalBufLength;`**

Description: Points to the length of the palette file buffer, in bytes. For format of data in the buffer, refer to "Chapter 7: Data structures".

See also: `C_PalBuffer`

Example: `PalPtr+=  *C_PalBufLength-1;`

C_Palette

Syntax: **`char **C_Palette;`**

Description: Points to the address of a copy of the currently active palette. Changing this palette does not change the colors on the screen unless a `GrSetPalette` call is executed.

See also: `C_Stipples`, `C_ColRanges`, `C_OldPalette`

Example: `(*C_Palette)[Col*3+1]=NewGreen;`

C_PalFile

Syntax:	**`char *C_PalFile;`**
Description:	Points to the name of the palette file. This does not include the path.
See also:	`C_PalPath`

Example:	`strcpy(FileName,C_PalPath);`
	`strcat(FileName,C_PalFile);`

C_PalLen

Syntax:	**`long *C_PalLen;`**
Description:	Points to the length of the data in the palette file buffer, in bytes.
See also:	`C_PalBuffer`
Example:	`PalPtr+= *C_PalLen-1;`

C_PalPath

Syntax:	**`char *C_PalPath;`**
Description:	Points to the path for the palette file. This does not include the file name.
See also:	`C_PalFile`

Example:	`strcpy(FileName,C_PalPath);`
	`strcat(FileName,C_PalFile);`

C_Plane

Syntax:	**`short *C_Plane;`**
Description:	Indicates current elevation view when *C_ZPlane is non-zero:

0	Plan view
1	North view
2	South view
3	East view
4	West view
5	Underside view

See also: `C_Zplane`, `C_ZPlaneScale`

Example:

```
if((*C_ZPlane) && *C_Plane==0)
/* We are in plan view */
```

C_Player

Syntax: **`short *C_Player;`**

Description: Points to the number of the user being processed, indexed from 0. His viewpoint, or controlled object, are all set with reference to `C_PlayerView`. For a single user system this is always 0.

See also: `C_PlayerView`,`C_ThisPlayer`

Example: `*C_Player=0;`

C_PlayerView

Syntax: **`short *C_PlayerView;`**

Description: Points to an array of viewpoint numbers, one for each user. For a single user system, only the first item in this list is valid.

See also: `C_Player`,`C_ThisPlayer`

Example: `View=C_PlayerView[*C_Player];`

C_PrefsBuffer

Syntax: **`char **C_PrefsBuffer;`**

Description: Points to the address of the preferences file buffer. For format of data in the buffer (which is the same as the configuration data), refer to “Chapter 7: Data structures”.

See also: `C_PrefsBufLength`

Example: `PrfPtr=*C_PrefsBuffer+sizeof(T_FILEHEADER);`

C_PrefsBufLength

Syntax: **`long *C_PrefsBufLength;`**

Description: Points to the length of the preferences file buffer, in bytes. For format of data in the buffer (which is the same as the configuration data), refer to “Chapter 7: Data structures”.

See also: `C_PrefsBuffer`

Example: `PrfPtr+= *C_PrefsBufLength-1;`

C_PrefsExtraConfig

Syntax: **`T_EXTRACONFIG **C_PrefsExtraConfig;`**

Description: Points to the address of the extra configuration chunk in the preferences buffer. For the format of this chunk see “Chapter 7: Data structures”.

See also: `C_ExtraConfig`

Example: `default_near_clip=(*C_PrefsExtraConfig)->NearClip;`

C_PrefsFile

Syntax: **`char *C_PrefsFile;`**

Description: Points to the name of the preferences file. This does not include the path.

See also: `C_PrefsPath`

Example:

```
strcpy(FileName,C_PrefsPath);
strcat(FileName,C_PrefsFile);
```

C_PrefsLen

Syntax: **`long *C_PrefsLen;`**

Description: Points to the length of the data in the preferences file buffer, in bytes.

See also: `C_PrefsBuffer`

Example:

```
PrfPtr+= *C_PrefLen-1;
```

C_PrefsPath

Syntax: **`char *C_PrefsPath;`**

Description: Points to the path for the preferences file. This does not include the file name.

See also: `C_PrefsFile`

Example:

```
strcpy(FileName,C_PrefsPath);
strcat(FileName,C_PrefsFile);
```

C_PrinterBuffer

Syntax: **`char **C_PrinterBuffer;`**

Description: This function is not used in VRT 5.50, as printers are controlled by Windows. The function is included for compatibility with earlier versions of VRT.

Points to the address of the printer driver buffer. The printer driver is loaded using the routine `ChoosePrinter`.

C_PrinterBufLength

Syntax: **`long *C_PrinterBufLength;`**

Description: This function is not used in VRT 5.50, as printers are controlled by Windows. The function is included for compatibility with earlier versions of VRT.

Points to the length of the printer driver buffer, in bytes.

Example:

```
if(*C_PrinterBufLength==0)
  /* No printer driver loaded */
```

C_ProductCode

Syntax: **`char *C_ProductCode`**

Description: Points to a null-terminated string which is either `VIS` if running under Visualiser, `NET` if running under Viscape, or `VRT` if running under VRT.

Note: Also returns `VIS` if running under DOS Visualiser, or `VIS_SGI` if running under Visualiser for Silicon Graphics.

C_PropAxes

Syntax: **`short *C_PropAxes;`**

Description: Points to an array containing the number of axes that each proportional device has (usually 6). Only valid if the proportional device is actually installed.

See also: `C_PropReturn`, `C_PropDD`, `C_PropButtons`

Example:

```
if(C_PropAxes[*C_PropDD]<6)
  /* Not a full 3D+3D controller */
```

C_PropButtons

Syntax: **`short *C_PropButtons;`**

Description: Points to an array containing the number of buttons that each proportional device has. Only valid if the proportional device is actually installed.

See also: `C_PropReturn`, `C_PropDD`, `C_PropAxes`

Example:

```
if(C_PropButtons[*C_PropDD]<1)
  /* No buttons on this device */
```

C_PropDD

Syntax: **`short *C_PropDD;`**

Description: Points to the number of the first active proportional device. This is automatically used when calling the device drivers, and as such can quite safely be ignored.

This may be changed temporarily in order to use a different device from the default, but it must be set back to its original value before returning to the main program.

See also: `C_GraphicsDD`, `C_KeyboardDD`, `C_MouseDD`, `C_NetworkDD`, `C_SoundDD`, `C_TimerDD`, `C_SerialDD`

Example:

```
/* Read values from 2nd proportional device */
OldPropDev=*C_PropDD;
*C_PropDD=ActiveDevice(E_DEVPROP,1);
Position=PrGetPos(NULL);
*C_PropDD=OldPropDev;
```

C_PropMenu

Syntax: **`char **C_PropMenu;`**

Description: Points to the address of the name of a dialog boxes to display for the currently installed proportional device. If proportional device menus are turned on, then this dialog box is displayed at the bottom of the screen in the Visualiser.

See also: `C_PropDD`

Example:

```
CurrentName=C_PropMenu;
```

C_PropParam

Syntax: **`short *C_PropParam;`**

Description: Points to the value of the proportional parameter. This can be set before using `ObeyFunction` to perform an action, or interrogated within a user function.

See also: `C_FuncBuf`

Example:
```
*C_PropParam=1024;
ObeyFunction(0x46000001);
```

C_PropReturn

Syntax: **`T_PRINSTALLRET **C_PropReturn;`**

Description: Points to an array of addresses of the structures returned by the installation of the proportional devices, or `NULL` if none. Each contains various useful information about the device.

See also: `C_PropAxes`, `C_PropDD`

Example:
```
DeviceName=
   (C_PropReturn[C_PropDD])->Name;
```

C_RangeOn

Syntax: **`char *C_RangeOn;`**

Description: Points to a flag which is set to a non-zero value if the color ranges are to be displayed on color selectors in a dialog box. It should be set just before the call to `Dialog`, and reset immediately after.

Example:
```
*C_RangeOn=1;
Return=Dialogue(“CHOOSE_COLOUR”);
*C_RangeOn=0;
```

C_RegSCLFunctions

Syntax: **`T_SCLFUNCPTR *C_RegSCLFunctions;`**

Description: See “Chapter 2: SCL functions” for further details.

C_Resolution

Syntax: **`unsigned char *C_Resolution;`**

Description: Points to the current resolution for the screen. This is normally 0, the default resolution for the graphics card. Different cards have different resolution values.

See also: `C_Resolution`

Example: `ResizeScreen(*C_Resolution+1);`

C_ResourceBuffer

Syntax: **`char **C_ResourceBuffer;`**

Description: Points to the address of the resource file buffer. For format of data in the buffer, refer to “Chapter 7: Data structures”.

See also: `C_ResourceBufLength`

Example: `RPtr=*C_ResourceBuffer+sizeof(T_FILEHEADER);`

C_ResourceBufLength

Syntax: **`long *C_ResourceBufLength;`**

Description: Points to the length of the resource file buffer, in bytes. For format of data in the buffer, refer to “Chapter 7: Data structures”.

See also: `C_ResourceBuffer`

Example: `RPtr+=  *C_ResourceBufLength-1;`

C_ResourceFile

Syntax: **`char *C_ResourceFile;`**

Description: Points to the name of the system resource file. This does not include the path.

See also: `C_ResourcePath`

Example:

```
strcpy(FileName,C_ResourcePath);
strcat(FileName,C_ResourceFile);
```

C_ResourceLen

Syntax: **`long *C_ResourceLen;`**

Description: Points to the length of the data in the resource file buffer, in bytes.

See also: `C_ResourceBuffer`

Example:

```
RPtr+= *C_ResourceLen-1;
```

C_ResourcePath

Syntax: **`char *C_ResourcePath;`**

Description: Points to the path for the system resource file. This does not include the file name.

See also: `C_ResourceFile`

Example:

```
strcpy(FileName,C_ResourcePath);
strcat(FileName,C_ResourceFile);
```

C_SCLChkType

Syntax: **`short *C_SCLChkType;`**

Description: Points to the type of SCL chunk being executed. This is one of the following:

`E_CTSCL`	“Normal” SCL on an object
`E_CTSCLLOCAL`	Locally triggered SCL
`E_CTSCLGLOBAL`	Globally triggered SCL
`0`	User functions

This is only valid within a registered SCL instruction.

Example:

```
if(*C_SCLChkType==E_CTSCLLOCAL)
  /* I've been triggered locally */
```

C_SCLDefVarAdd

Syntax: **`unsigned char **C_SCLDefVarAdd;`**

Description: Points to the address of the SCL variable definitions. These may be read or modified by SCL commands.

See also: `C_SCLResumeAdd`

This is only valid within a registered SCL instruction.

Example:

```
*C_SCLDefVarAdd=NULL;
```

C_SCLError

Syntax: **`short *C_SCLError;`**

Description: Points to the SCL error. This is normally positive, but can be set to `E_OK` to terminate the SCL with no error, or the value `E_ERROR` to terminate with a "General error" message.

This is only valid within a registered SCL instruction.

Example: `*C_SCLError=E_ERROR;`

C_SCLFiles

Syntax: **`FILE **C_SCLFiles;`**

Description: Points to an array of file pointers, each of which either represents an open SCL file, or is `NULL`. The handle returned from SCL function `fopen` is the index into this array.

Example: `fread(Buffer,128,1,C_SCLFiles[Handle]);`

C_SCLFileFlags

Syntax: **`T_SCLFILE *C_SCLFileFlags;`**

Description: Points to a structure which determines whether the file is a URL or not, and if the browser is open.

C_SCLFunctions

Syntax: **`T_SCLFUNCPTR *C_SCLFunctions;`**

Description: See "Chapter 2: SCL functions" for further details.

C_SCLParamIndex

Syntax: **`short *C_SCLParamIndex;`**

Description: Points to the SCL parameter pointer. This is the index of the first parameter passed to a procedure, which is stored on the stack in the `C_SCLStack` and `C_SCLType` arrays.

See also: `C_SCLStack`, `C_SCLType`, `C_SCLSP`

This is only valid within a registered SCL instruction.

Example: `C_SCLStack[*C_SCLParamIndex].l=0x4000;`

C_SCLPC

Syntax: **`unsigned char **C_SCLPC;`**

Description: Points to the SCL program counter. This is the address of the currently executing SCL command, and may be modified by SCL instructions (with care!).

This is only valid within a registered SCL instruction.

Example: `*C_SCLPC+=1;`

C_SCLResumeAdd

Syntax: **`unsigned char **C_SCLResumeAdd;`**

Description: Points to the address of the SCL resume statement. This may be set to NULL to terminate the SCL at the end of this frame.

See also: `C_SCLDefVarAdd`

This is only valid within a registered SCL instruction.

Example: `*C_SCLResumeAdd=NULL;`

C_SCLReturnStack

Syntax: **T_LONGORPTR *C_SCLReturnStack;**

Description: Points to an array of values for items stored on the stack. The values may be of type `long` or a pointer, so the union suffix `.l` or `.p` is used to select which type of value to read according to the type. It is good practice only to look at this array by indexing with the stack pointer, `C_SCLRSP`.

See also: `C_SCLSP`, `C_SCLRSP`

This is only valid within a registered SCL instruction.

Example:
```
if(C_SCLReturnStack[*C_SCLRSP].l==0)
  /* Loop has finished */
```

C_SCLRSP

Syntax: **short *C_SCLRSP;**

Description: Points to the SCL return stack pointer. This is the index of the top item on the return address stack in the `C_SCLReturnStack` array.

See also: `C_SCLReturnStack`, `C_SCLSP`, `C_SCLParamIndex`

This is only valid within a registered SCL instruction.

Example:
```
C_SCLReturnStack[*C_SCLRSP].p=LoopAddress;
```

C_SCLSP

Syntax: **short *C_SCLSP;**

Description: Points to the SCL stack pointer. This is the index of the top item on the stack in the `C_SCLStack` and `C_SCLType` arrays.

See also: `C_SCLStack`, `C_SCLType`, `C_SCLRSP`, `C_SCLParamIndex`

This is only valid within a registered SCL instruction.

Example:
```
C_SCLStack[*C_SCLSP].l=0x4000;
```

C_SCLStack

Syntax: **`T_LONGORPTR *C_SCLStack;`**

Description: Points to an array of values for items stored on the stack. The values may be of type `long`, `float` or `pointer`, so the union suffix `.l`, `.f` or `.p` is used to select which type of value to read according to the type. It is good practice only to look at this array by indexing with the stack pointer, `C_SCLSP`.

See also: `C_SCLType`, `C_SCLSP`, `C_SCLRSP`, `C_SCLParamIndex`

This is only valid within a registered SCL instruction.

Example:

```
if(C_SCLType[*C_SCLSP]==0x40)
   Pointer=C_SCLStack[*C_SCLSP].p;
```

C_SCLType

Syntax: **`short *C_SCLType;`**

Description: Points to an array of types for items stored on the stack. The types are those used when pushing items on the stack. It is good practice only to look at this array by indexing with the stack pointer, `C_SCLSP`.

See also: `C_SCLStack`, `C_SCLSP`, `C_SCLRSP`, `C_SCLParamIndex`

This is only valid within a registered SCL instruction.

Example:

```
if(C_SCLType[*C_SCLSP]==0x40)
  /* It’s a char * pointer */
```

C_ScreenParams

Syntax: **`T_GRSETUP **C_ScreenParams;`**

Description: Points to an array of addresses of the structures returned by the graphics setup. This contains various useful data about each screen. See “Chapter 7: Data structures” for more information about its precise contents.

See also: `C_Console`

Example: `ScrWidth=(C_ScreenParams[*C_GraphicsDD])->Width;`

C_ScreenSave

Syntax:	**`T_GRRECT *C_ScreenSave;`**
Description:	Points to an array of rectangle structures which define the area of screen to be saved by the `SaveScreen` function. By default, covers the whole screen.
Example:	`C_ScreenSave[*C_GraphicsDD]->Width=100;`

C_Second

Syntax:	**`char *C_Second;`**
Description:	Points to the seconds portion of VRT's copy of the realtime clock.
See also:	`C_Year`, `C_Month`, `C_Day`, `C_Hour`, `C_Minute`

Example:

```
if(*C_Second==0)
  /* Another minute gone by */
```

C_SerialDD

Syntax:	**`short *C_SerialDD;`**
Description:	This variable points to the number of the currently active serial device. This is automatically used when calling the device drivers, and as such can quite safely be ignored.
See also:	`C_GraphicsDD`, `C_KeyboardDD`, `C_MouseDD`, `C_NetworkDD`, `C_PropDD`, `C_SoundDD`, `C_TimerDD`
Example:	`i=C_SerialDD;`

C_ShapeAdd

Syntax:	**`void ***C_ShapeAdd;`**
Description:	Points to the address of an array of pointers to each of the shapes. The index into this list is the shape type as used by the object. By default, the group is shape -1, the cube is shape 0. The pointer to a shape is NULL if it does not exist.
See also:	`C_ShapeBuffer`
Example:	`ShpPtr=(*C_ShapeAdd)[Object->Std.Type];`

C_ShapeBuffer

Syntax: **`char **C_ShapeBuffer;`**

Description: Points to the address of the shape file buffer. For format of data in the buffer, refer to “Chapter 7: Data structures”.

See also: `C_ShapeBufLength`

Example:

```
ShpPtr=*C_ShapeBuffer+
   sizeof(T_FILEHEADER);
```

C_ShapeBufLength

Syntax: **`long *C_ShapeBufLength;`**

Description: Points to the length of the shape file buffer, in bytes. For format of data in the buffer, refer to “Chapter 7: Data structures”.

See also: `C_ShapeBuffer`

Example:

```
ShpPtr+=*C_ShapeBufLength-1;
```

C_ShapeFile

Syntax: **`char *C_ShapeFile;`**

Description: Points to the name of the shape file. This does not include the path.

See also: `C_ShapePath`

Example:

```
strcpy(FileName,C_ShapePath);
strcat(FileName,C_ShapeFile);
```

C_ShapeLen

Syntax: **`long *C_ShapeLen;`**

Description: Points to the length of the data in the shape file buffer, in bytes.

See also: `C_ShapeBuffer`

Example:

```
ShpPtr+= *C_ShapeLen-1;
```

C_ShapePath

Syntax: **`char *C_ShapePath;`**

Description: Points to the path for the shape file. This does not include the file name.

See also: `C_ShapeFile`

Example:

```
strcpy(FileName,C_ShapePath);
strcat(FileName,C_ShapeFile);
```

C_ShapeSym

Syntax: **`char **C_ShapeSym;`**

Description: Points to the address of the start of symbol information in the shape buffer.

See also: `C_ShapeBuffer`

Example: `SymPtr=*C_ShapeSym;`

C_SinTable

Syntax: **`short *C_SinTable;`**

Description: Points to the array containing the values of the normalized sines (sin(x)*16384) of angles 0–65535 brees, in 8192 steps.

Example:

```
x=CentreX+(C_SinTable[Theta]*Radius>>14);
y=CentreY+(C_SinTable[Theta+2048]*Radius>>14);
```

C_SkyBands

Syntax: **`unsigned char *C_SkyBands;`**

Description: Points to the number of bands displayed in the sky by the horizon routine.

See also: `C_GndCol`, `C_SkyCol`, `C_SkyGrad`

Example: `*C_SkyBands=6;`

C_SkyCol

Syntax:	**`unsigned char *C_SkyCol;`**
Description:	Points to the color of the sky.
See also:	`C_GndCol`, `C_SkyBands`, `C_SkyGrad`
Example:	`*C_SkyCol=E_COLBLACK;`

C_SkyGrad

Syntax:	**`unsigned char *C_SkyGrad;`**
Description:	Points to the color increment between successive bands in the sky (usually between -2 and 2).
See also:	`C_GndCol`, `C_SkyBands`, `C_SkyCol`
Example:	`*C_SkyGrad=1;`

C_SoundBuffer

Syntax:	**`char **C_SoundBuffer;`**
Description:	Points to the address of the sound file buffer. For format of data in the buffer, refer to "Chapter 7: Data structures".
See also:	`C_SoundBufLength`
Example:	`SoundPtr=*C_SoundBuffer+sizeof(T_FILEHEADER);`

C_SoundBufLength

Syntax:	**`long *C_SoundBufLength;`**
Description:	Points to the length of the sound file buffer, in bytes. For format of data in the buffer, refer to "Chapter 7: Data structures".
See also:	`C_SoundBuffer`
Example:	`SoundPtr+= *C_SoundBufLength-1;`

C_SoundDD

Syntax: **`short *C_SoundDD;`**

Description: This variable points to the number of the currently active sound device. This is automatically used when calling the device drivers, and as such can quite safely be ignored.

This may be changed temporarily in order to use a different device from the default, but it must be set back to its original value before returning to the main program.

See also: `C_GraphicsDD`, `C_KeyboardDD`, `C_MouseDD`, `C_NetworkDD`, `C_PropDD`, `C_SerialDD`, `C_TimerDD`

Example:

```
/* Play sound on all active sound devices   */

    OldSoundDev=*C_SoundDD;
    for(i=0;i<C_NumDevsActive[E_DEVSOUND];i++)
    {
       *C_SoundDD=ActiveDevice(E_DEVSOUND,i);
        SdPlaySound(&Sound);
    }
    *C_SoundDD=OldSoundDev;
```

C_SoundFile

Syntax: **`char *C_SoundFile;`**

Description: Points to the name of the sound file. This does not include the path.

See also: `C_SoundPath`

Example:

```
strcpy(FileName,C_SoundPath);
strcat(FileName,C_SoundFile);
```

C_SoundLen

Syntax: **`long *C_SoundLen;`**

Description: Points to the length of the data in the sound file buffer, in bytes.

See also: `C_SoundBuffer`

Example:

```
SoundPtr+= *C_SoundLen-1;
```

C_SoundPath

Syntax: **`char *C_SoundPath;`**

Description: Points to the path for the sound file. This does not include the file name.

See also: `C_SoundFile`

Example:

```
strcpy(FileName,C_SoundPath);
strcat(FileName,C_SoundFile);
```

C_SoundReturn

Syntax: **`T_SDINSTALLRET **C_SoundReturn`**

Description: Points to an array that contains pointers to the return values from each installed sound device.

See also: `C_SoundDD`

Example:

```
pName=C_SoundReturn[*C_SoundDD]->Name;
```

C_Spr0SaveBuffer

Syntax: **`char **C_Spr0SaveBuffer;`**

Description: Points to an array containing the addresses of sprite 0's background save area for each installed graphics device. Sprite 0 is reserved for system use.

See also: `C_SprSaveBuffer`

Example:

```
GrUndrawSprite(C_Spr0SaveBuffer[*C_GraphicsDD]);
```

C_SprSaveBuffer

Syntax: **`char **C_SprSaveBuffer;`**

Description: Points to an array containing the addresses of the background save area for system sprites other than sprite 0, one for each possible graphics device.

See also: `C_Spr0SaveBuffer`, `C_UserSprSaveBuf`

Example:

```
GrUndrawSprite(C_SprSaveBuffer[C_GraphicsDD]);
```

C_Sprite

Syntax: **T_GRSPRITE **C_Sprite;**

Description: Points to an array (one entry per graphics device) containing the addresses of arrays of sprite records, one for each system sprite. `Sprite 0` is blank, and is used solely to save the background behind areas of the screen. Other sprites are for system use only.

See also: `C_UserSprite`

Example: `GrDrawSprite(C_Sprite[*C_GraphicsDD]+2);`

C_SpriteBuffer

Syntax: **char **C_SpriteBuffer;**

Description: Points to the address of the sprite file buffer. For format of data in the buffer, refer to “Chapter 7: Data structures”.

See also: `C_SpriteBufLength`

Example: `SprPtr=*C_SpriteBuffer+sizeof(T_FILEHEADER);`

C_SpriteBufLength

Syntax: **long *C_SpriteBufLength;**

Description: Points to the length of the sprite file buffer, in bytes. For format of data in the buffer, refer to “Chapter 7: Data structures”.

See also: `C_SpriteBuffer`

Example: `SprPtr+=*C_SpriteBufLength-1;`

C_SpriteFile

Syntax: **char *C_SpriteFile;**

Description: Points to the name of the system image (sprite) file. This does not include the path.

See also: `C_SpritePath`

Example:
```
strcpy(FileName,C_SpritePath);
strcat(FileName,C_SpriteFile);
```

C_SpriteLen

Syntax: **`long *C_SpriteLen;`**

Description: Points to the length of the data in the image (sprite) file buffer, in bytes.

See also: `C_SpriteBuffer`

Example:
```
SprPtr+= *C_SpriteLen-1;
```

C_SpritePath

Syntax: **`char *C_SpritePath;`**

Description: Points to the path for the system image (sprite) file. This does not include the file name.

See also: `C_SpriteFile`

Example:
```
strcpy(FileName,C_SpritePath);
strcat(FileName,C_SpriteFile);
```

C_Stipples

Syntax: **`char **C_Stipples;`**

Description: Points to the address of a copy of the currently active stipple table. Changing this table does not change the colors on the screen unless a `GrSetStipples` call is executed.

See also: `C_Palette`, `C_ColRanges`

Example:
```
(*C_Stipples)[Col*8+2]=NewBotLeftCol;
```

C_StepSize

Syntax: **`long *C_StepSize;`**

Description: Points to the step used when moving objects or the viewpoint from the keyboard, in basic units.

See also: `C_AngleSize`

Example:
```
*C_StepSize=10000;
```

C_SunLight

Syntax: **`T_LIGHTSOURCE **C_SunLight;`**

Description: Points to the address of a default sun-like light source. This is usually only used during the `Light Object` function in the World Editor, but could be useful in adding an additional sun into a world.

See also: `C_NumLights`, `C_Ambient`, `C_Light`

Example: `*C_SunLight=MyLightSource;`

C_SuppressFileErr

Syntax: **`char *C_SuppressFileErr;`**

Description: Points to a byte that indicates whether file errors should be automatically reported to the user (using an alert box) or not. If `*C_SuppressFileErr` is 0, errors are reported; if it is non-0, errors are not reported to the user. This does not effect error codes returned from file functions, allowing the SDK application to report its own errors.

Example: `*C_SuppressFileErr=0`

C_SysMess

Syntax: **`char **C_SysMess;`**

Description: Points to an array containing pointers to each system message. Messages off the end of the system message buffer have NULL addresses.

See also: `C_SysMessBufLength`

Example: `MessPtr=C_SysMess[1212];`

C_SysMessBuffer

Syntax: **`char **C_SysMessBuffer;`**

Description: Points to the address of the system message file buffer. For format of data in the buffer, refer to “Chapter 7: Data structures”.

See also: `C_SysMessBufLength`

Example: `MessPtr=*C_SysMessBuffer+sizeof(T_FILEHEADER);`

C_SysMessBufLength

Syntax: **`long *C_SysMessBufLength;`**

Description: Points to the length of the system message file buffer, in bytes. For format of data in the buffer, refer to “Chapter 7: Data structures”.

See also: `C_SysMessBuffer`

Example:

```
MessPtr+= *C_SysMessBufLength-1;
```

C_SysMessFile

Syntax: **`char *C_SysMessFile;`**

Description: Points to the name of the system message file. This does not include the path.

See also: `C_SysMessPath`

Example:

```
strcpy(FileName,C_SysMessPath);
strcat(FileName,C_SysMessFile);
```

C_SysMessLen

Syntax: **`long *C_SysMessLen;`**

Description: Points to the length of the data in the system message file buffer, in bytes.

See also: `C_SysMessBuffer`

Example:

```
MessPtr+= *C_SysMessLen-1;
```

C_SysMessPath

Syntax: **`char *C_SysMessPath;`**

Description: Points to the path for the system message file. This does not include the file name.

See also: `C_SysMessFile`

Example:

```
strcpy(FileName,C_SysMessPath);
strcat(FileName,C_SysMessFile);
```

C_TextureScale

Syntax:	**`char *C_TextureScale;`**
Description:	Points to the overall texture detail level, which can be one of the following values:

`E_TEXOFF`	Textures off
`E_TEX4X4`	}
`E_TEX3X3`	} No longer used. Displays textures as `E_TEX1X1`.
`E_TEX2X2`	}
`E_TEX1X1`	Textures displayed with 1x1 pixel blocks

See also:	`C_DetailLevel`
Example:	`*C_TextureScale=E_TEXOFF;`

C_ThisPlayer

Syntax:	**`short *C_ThisPlayer;`**
Description:	Points to the user number assigned to this machine in a multi user world. For a single user system, this is always 0.
See also:	`C_PlayerView`, `C_Player`
Example:	`MyUser=*C_ThisPlayer;`

C_TimerA
C_TimerB
C_TimerC
C_TimerD

Syntax:

```
T_TIMENODE **C_TimerA;
T_TIMENODE **C_TimerB;
T_TIMENODE **C_TimerC;
T_TIMENODE **C_TimerD;
```

Description:	Points to the address of the timer node structure used to register each of the standard SCL timers with the timer device driver. These may be modified by changing their rates.
See also:	`C_Triggers`, `C_Triggers2`
Example:	`(*C_TimerA)->Rate=1000;`

C_TimerDD

Syntax: **`short *C_TimerDD;`**

Description: This variable points to the number of the currently active timer device. This is automatically used when calling the device drivers, and as such can quite safely be ignored.

Do not change this device number, as it will cause the system to crash.

See also: `C_GraphicsDD`, `C_KeyboardDD`, `C_MouseDD`, `C_NetworkDD`, `C_PropDD`, `C_SoundDD`, `C_SerialDD`

Example:

```
/* Get name of timer device */
TmName=C_DeviceName
   [E_MAXPERDEV*E_DEVTIMER+*C_TimerDD];
```

C_Triggers

Syntax: **`long *C_Triggers;`**

Description: Points to a set of 32 bits which indicates that various global events have taken place. This should only be read, and is zeroed after each frame's SCL has been done. The defined bits are:

`E_GTFIRSTFRAME`	Set if 1st frame after reset.
`E_GTTIMERA`	Set if timer A timed out.
`E_GTTIMERB`	Set if timer B timed out.
`E_GTTIMERC`	Set if timer C timed out.
`E_GTTIMERD`	Set if timer D timed out.

See also: `C_Triggers2`

Example:

```
if(*C_Triggers&E_GTTIMERA)
  /* Timer A has gone off */
```

C_Triggers2

Syntax:	**`long *C_Triggers2;`**
Description:	Points to a set of 32 bits which indicates that various global events have taken place. This is written to by the events, and transferred into `C_Triggers` to prevent double sensing of events. Any global triggers should be indicated by writing to this variable, not `C_Triggers`. The defined bits are:

`E_GTFIRSTFRAME`	Set if 1st frame after reset.
`E_GTTIMERA`	Set if timer A timed out.
`E_GTTIMERB`	Set if timer B timed out.
`E_GTTIMERC`	Set if timer C timed out.
`E_GTTIMERD`	Set if timer D timed out.

See also:	`C_Triggers`
Example:	`*C_Triggers2|=E_GTTIMERD;`

C_UndoBuffer

Syntax:	**`char **C_UndoBuffer;`**
Description:	Points to the address of the undo buffer. Under no circumstances alter the data in this buffer.
See also:	`C_UndoBufLength`
Example:	`UndoPtr=*C_UndoBuffer;`

C_UndoBufLength

Syntax:	**`long *C_UndoBufLength;`**
Description:	Points to the length of the undo buffer, in bytes.
See also:	`C_UndoBuffer`
Example:	`UndoPtr+=*C_UndoBufLength-1;`

C_UndoHead

Syntax: **`char **C_UndoHead;`**

Description: Points to the address of the next free space in the undo buffer. Under no circumstances alter the data in this buffer.

See also: `C_UndoBuffer`, `C_UndoBufLength`

Example: `UndoPtr=*C_UndoHead;`

C_UserFunc

Syntax: **`unsigned char **C_UserFunc;`**

Description: Points to the array containing the addresses of the SCL object code for the user functions. A user function with a `NULL` address is not defined.

Example: `ReturnVal=Execute((*C_UserFunc)[12]);`

C_UserRsrcBuffer

Syntax: **`char **C_UserRsrcBuffer;`**

Description: Points to the address of the user resource file buffer. For format of data in the buffer, refer to “Chapter 7: Data structures”.

See also: `C_UserRsrcBufLength`

Example: `UPtr=*C_UserRsrcBuffer+sizeof(T_FILEHEADER);`

C_UserRsrcBufLength

Syntax: **`long *C_UserRsrcBufLength;`**

Description: Points to the length of the user resource file buffer, in bytes. For format of data in the buffer, refer to “Chapter 7: Data structures”.

See also: `C_UserRsrcBuffer`

Example: `UPtr+=*C_UserRsrcBufLength-1;`

C_UserRsrcFile

Syntax: **`char *C_UserRsrcFile;`**

Description: Points to the name of the user resource file. This does not include the path.

See also: `C_UserRsrcPath`

Example:

```
strcpy(FileName,C_UserRsrcPath);
strcat(FileName,C_UserRsrcFile);
```

C_UserRsrcLen

Syntax: **`long *C_UserRsrcLen;`**

Description: Points to the length of the data in the user resource file buffer, in bytes.

See also: `C_UserRsrcBuffer`

Example:

```
UPtr+= *C_UserRsrcLen-1;
```

C_UserRsrcPath

Syntax: **`char *C_UserRsrcPath;`**

Description: Points to the path for the user resource file. This does not include the file name.

See also: `C_UserRsrcFile`

Example:

```
strcpy(FileName,C_UserRsrcPath);
strcat(FileName,C_UserRsrcFile);
```

C_UserSprite

Syntax: **`T_GRSPRITE **C_UserSprite;`**

Description: Points to an array of addresses (one per graphics device) of arrays of sprite records, one for each user sprite. The representation of the sprites is entirely up to the user.

See also: `C_Sprite`

Example:

```
GrDrawSprite(C_UserSprite[*C_GraphicsDD]+2);
```

C_UserSprBuffer

Syntax: **`char **C_UserSprBuffer;`**

Description: Points to the address of the user sprite file buffer. For format of data in the buffer, refer to “Chapter 7: Data structures”.

When a sprite file is loaded, the Offset value in `T_SPRITEINFO` is converted from an offset from the start of the file, to a pointer to the sprite data.

See also: `C_UserSprBufLength`

Example: `UsPtr=*C_UserSprBuffer+sizeof(T_FILEHEADER);`

C_UserSprBufLength

Syntax: **`long *C_UserSprBufLength;`**

Description: Points to the length of the user sprite file buffer, in bytes. For format of data in the buffer, refer to “Chapter 7: Data structures”.

See also: `C_UserSprBuffer`

Example: `UsPtr+=  *C_UserSprBufLength-1;`

C_UserSprFile

Syntax: **`char *C_UserSprFile;`**

Description: Points to the name of the user image (sprite) file. This does not include the path.

See also: `C_UserSprPath`

Example:

```
strcpy(FileName,C_UserSprPath);
strcat(FileName,C_UserSprFile);
```

C_UserSprLen

Syntax:	**`long *C_UserSprLen;`**
Description:	Points to the length of the data in the user sprite file buffer, in bytes.
See also:	`C_UserSprBuffer`
Example:	`UsPtr+= *C_UserSprLen-1;`

C_UserSprPath

Syntax:	**`char *C_UserSprPath;`**
Description:	Points to the path for the user image (sprite) file. This does not include the file name.
See also:	`C_UserSprFile`
Example:	`strcpy(FileName,C_UserSprPath);` `strcat(FileName,C_UserSprFile);`

C_UserSprSaveBuf

Syntax:	**`char **C_UserSprSaveBuf;`**
Description:	Points to an array of addresses of the background save areas for user sprites (one per device driver).
See also:	`C_Spr0SaveBuffer`, `C_SprSaveBuffer`
Example:	`GrUndrawSprite(C_UserSprSaveBuf*C_GraphicsDD]);`

C_VersionMajor
C_VersionMinor
C_VersionRev

Syntax:

```
unsigned short * C_VersionMajor;
unsigned short * C_VersionMinor;
char * C_VersionRev;
```

Description: These three variables define the version number of VRT under which the application is running (v3.50 or above only). On pre-3.50 the pointers `C_VersionMajor` and `C_VersionMinor` are the same. `C_VersionRev` points to a character which may be added to the version number for minor updates. This is usually `0x20` (space).

See also: `C_VersionDate`

Example:

```
/* Put version string into String */
if(C_VersionMajor==C_VersionMinor)
    sprintf(String,"Pre 3.50 version");
else
    sprintf(String,"Version %d.%02d%c",
      *C_VersionMajor, *C_VersionMinor, *C_VersionRev);
```

C_VersionDate

Syntax:

```
char * C_VersionDate;
```

Description: Points to a date string relating to the version of VRT under which the application is being run (version 3.50 and above only). On pre-3.50 the pointers `C_VersionDate` and `C_VersionRev` are the same.

See also: `C_VersionMajor`, `C_VersionMinor`, `C_VersionRev`

Example:

```
/* Put date string into String */

if(C_VersionDate==C_VersionRev)
    strcpy(String,"Pre 3.50 version");
else
    strcpy(String,C_VersionDate);
```

C_ViewList

Syntax:	**`T_VIEW *C_ViewList;`**
Description:	Points to an array of viewpoint descriptions.
See also:	`C_PlayerView`, `C_Player`
Example:	`Attached=C_ViewList[C_PlayerView[*C_Player]];`

C_ViewMatrix

Syntax:	**`T_MATRIX *C_ViewMatrix;`**
Description:	Points to the matrix which is used to transform the world into the viewpoint reference frame. This effectively rotates the world around the viewpoint.
See also:	`C_ScreenParams`
Example:	`Point=RotatePoint(Point,C_ViewMatrix);`

C_ViewPosition

Syntax:	**`T_LONGVECTOR **C_ViewPosition;`**
Description:	For a viewpoint with a path, points to the address of an array of position lists. Each entry in the array points to a list of `T_LONGVECTOR` structures, one for each frame in the viewpoint path, or is `NULL` if there is no path for that viewpoint.
See also:	`C_ViewPtBuffer`

Example:	`p=C_ViewPosition[0];`
	`  /* Path for vp 1 */`

C_ViewPtBuffer

Syntax:	**`char **C_ViewPtBuffer;`**
Description:	Points to the address of the viewpoint position buffer. This is set up on reset, and is indexed into by `C_ViewPosition`.
See also:	`C_ViewPtBufLength`, `C_ViewPosition`
Example:	`VpPtr=*C_ViewPtBuffer+sizeof(T_FILEHEADER);`

C_ViewPtBufLength

Syntax:	**`long *C_ViewPtBufLength;`**
Description:	Points to the length of the viewpoint position buffer, in bytes.
See also:	`C_ViewPtBuffer`
Example:	`VpPtr+= *C_ViewPtBufLength-1;`

C_VPType

Syntax: **`char *C_VPType`**

Description: Points to the control type of the currently selected viewpoint.

See also: `C_CurrentVP`

Example:

```
if(*C_VPType!=LastVPType)
  /* Changed control type! */
```

C_WhereAmI

Syntax: **`char *C_WhereAmI;`**

Description: Points to a flag showing which part of VRT you are in. This is only in an editor if executing a user function with registered SCL instructions in, or when executing registered user functions in the World or Shape Editors. It may be one of:

`E_WVISUALISER`	Visualiser
`E_WWORLDED`	World Editor
`E_WSHAPEED`	Shape Editor
`E_WCONSOLEED`	Layout Editor
`E_WSOUNDED`	Sound Editor
`E_WSPRITEED`	Image Editor
`E_WRESOURCEED`	Resource Editor
`E_WKEYED`	Key Editor

Example:

```
if(C_WhereAmI==E_WVISUALISER)
  /* We're visualizing */
```

C_WorldBuffer

Syntax: **`char **C_WorldBuffer;`**

Description: Points to the address of the world file buffer. For format of data in the buffer, refer to “Chapter 7: Data structures”.

See also: `C_WorldBufLength`

Example: `WldPtr=*C_WorldBuffer+sizeof(T_FILEHEADER);`

C_WorldBufLength

Syntax:	**`long *C_WorldBufLength;`**
Description:	Points to the length of the world file buffer, in bytes. For format of data in the buffer, refer to “Chapter 7: Data structures”.
See also:	`C_WorldBuffer`
Example:	`WldPtr+= *C_WorldBufLength-1;`

C_WorldFile

Syntax: **`char *C_WorldFile;`**

Description: Points to the name of the world file. This does not include the path.

See also: `C_WorldPath`

Example:

```
strcpy(FileName,C_WorldPath);
strcat(FileName,C_WorldFile);
```

C_WorldLen

Syntax:	**`long *C_WorldLen;`**
Description:	Points to the length of the data in the world file buffer, in bytes.
See also:	`C_WorldBuffer`, `C_WorldBufLength`
Example:	`WldPtr+= *C_WorldLen-1;`

C_WorldPath

Syntax: **`char *C_WorldPath;`**

Description: Points to the path for the world file. This does not include the file name.

See also: `C_WorldFile`

Example:

```
strcpy(FileName,C_WorldPath);
strcat(FileName,C_WorldFile);
```

C_WorldSym

Syntax: **`char **C_WorldSym;`**

Description: Points to the address of the start of symbol information in the world buffer.

See also: `C_WorldBuffer`

Example: `SymPtr=*C_WorldSym;`

C_Year

Syntax: **`char *C_Year;`**

Description: Points to the year portion of VRT's copy of the realtime clock. The year value starts at 0 in 1980 and continues from there.

See also: `C_Month`, `C_Day`, `C_Hour`, `C_Minute`, `C_Second`

Example: `Year=1980+*C_Year;`

C_ZPlane
C_ZPlaneScale

Syntax:
```
short *C_ZPlane;
short *C_ZPlaneScale;
```

Description: A measure of zoom for elevation views given by `(*C_ZPlane)<<(*C_ZPlaneScale)`.

Example: `dist= (*C_ZPlane)<<(*C_ZPlaneScale);`

Windows variables

The following variables are Windows specific types and are valid for WIN32 programming purposes after you have initialized the window.

C_hDC

Syntax: **`HDC* C_hDC;`**

Description: The device context for the graphics routines used to draw the world. You can use it with all Windows GDI calls to draw additional graphical objects.

C_hinstDLL

Syntax: **`HINSTANCE* C_hinstDLL;`**

Description: Contains the instance handle of the DLL.

C_hMainFrame

Syntax: **`HWND* C_hMainFrame;`**

Description: The handle of the main frame of Visualiser. You can use it to change the Menu structure or alter the appearance of the Non-client areas of the application.

C_hView

Syntax: **`HWND* C_hView;`**

Description: The handle of the view window in which the world is displayed. You can use it in WIN32 API calls.

C_hwndPlugin

Syntax: **`HWND* C_hwndPlugin;`**

Description: The pointer to the window handle of the Netscape plugin window.

C_WinOpenDialog

Syntax:	**`long C_WinOpenDialog;`**
Description:	Returns 0 if a Windows non-modal dialog box is being displayed, and non-zero otherwise.

Chapter 7 - Data structures

Introduction

VRT uses many different types of data—such as world data, graphics data, configuration data—which each have definite structures, represented as C structures and unions. These are presented as `typedefs` in APP_TYPE.H, and APP_API.H which is included by APP_TYPE.H

The data mainly consists of `typedefs` for the data structures, together with `#define`s for specific values within these structures.

Each different type of data is presented separately, with the structures and defines in alphabetical order.

Although there are many different types of data spread across the different file formats, they all have considerable similarities. The key concept is the chunk, a structure of data whose contents are dependant on the field called type. Each type of chunk holds different data, but they all look like this:

```
typedef struct
{
    unsigned short ChkType, Length;

    /* ..... other data ..... */
}    T_SOMESORTOFCHUNK;
```

`ChkType` defines what sort of chunk this is, and `Length` shows how many bytes of data there are in it, including `ChkType` and itself.

Chunks usually come in lists, terminated by a short value of `0xFFFF` after the last chunk. There is usually at most one of each type of chunk in each list, although there are exceptions.

For example, each object attribute (such as rotations, initial position, SCL) is held in its own chunk, and each object being a list of chunks terminated by `0xFFFF` as above. Each object may not have every type of chunk, just the ones defined for each attribute it actually possesses. Also, the order of attributes within an object is not important; it is the presence or absence of the chunk that determines its effect.

Objects must have a standard attributes chunk, which is defined as a mandatory chunk. Mandatory chunks are the only exception to the ordering rule; they must be first in any list of related chunks.

`ChkType` only determines what type of data is in a chunk in context. While chunk type 0 is a standard attribute chunk in the world file, it is a points chunk in the shape file and a general chunk in the configuration file. Most of the time it is invisible, but if an application moves data around between file types, the chunk types are recognized as invalid and may cause problems.

File header

All Superscape files, except backdrops and applications, start with a 256-byte file header, containing necessary information about the file. It has the following structure:

T_FILEHEADER

Structure:

```
typedef struct
   {
   char              Text[200];
   long              Spare[10];
   long              Symbols;
   char              Type[4];
   unsigned short    Machine;
   long              Revision;
   unsigned char     Version,SubVersion;
   } T_FILEHEADER;
```

Description: **`Text`** An identifying string which marks the file as a Superscape file. It must start off with the following sequence of characters exactly:

```
   “SuperScape (c) New Dimension Inter”
   “national Ltd.\0”
```

This is followed by a short description of the file’s contents, and then by `“\0\x1A”` (NUL EOF). This ensures that typing the file from the DOS prompt only displays the header text—`0x1A` is the DOS end-of-file marker.

`Spare` Reserved. Should be set to 0.

`Symbols` The offset (from the start of the file, in bytes), of symbol information—such as object names, SCL comments or variable names—in the file, or 0 if none.

`Type` A 4-character file type which can be one of the following:

`.VRT`	VRT file
`CNFG`	Configuration file
`DDRV`	Device driver file
`FONT`	Font file
`MESS`	Message file
`PALT`	Palette file
`PRNT`	Printer driver file
`RSRC`	Resource file (VRT 3-60 and before)
`RESC`	Resource file (VRT 4-00 and later)
`SHAP`	Shape file
`SOUN`	Sound file
`SPRT`	Image (sprite) file
`WRLD`	World file

`Machine` An identifier for the machine from which the file was last saved. Currently only value 0 is defined: IBM PC or compatible.

`Revision` The counter increases by one every time the file is saved.

`Version` and **`SubVersion`** The version of the VRT from which the file was last saved. For example, set Version to 5 and SubVersion to 50 for VRT version 5.50.

World data

The world data is simply a world file that has been loaded into memory. It is stored in a buffer pointed to by `C_WorldBuffer`, which is `C_WorldBufLength` bytes long. Since you may edit the world, there is additional space in the buffer over and above that actually required for the world data, whose length is held in `C_WorldLen`.

The first 256 bytes of the buffer contain a standard file header, as found on the start of all Superscape files (see above).

After this comes a list of objects. Each object consists of a list of world chunks, the first of which must be a standard chunk. After the last chunk is a short value `0xFFFF`, marking the end of the object. The objects are arranged as a tree structure, the links being part of the standard chunk.

Following the objects are a set of symbols. Each symbol list consists of a list of symbols chunks, terminated by `0xFFFF`. The first list is any global symbol information (including layer names). Then there is a symbols list for each object with symbols, terminated by an `0xFFFF`. The end of this list of lists is itself marked by another `0xFFFF`.

Buffer:	`File header`	
	`Standard chunk, object 1`	Object 1’s data
	`Rotations chunk`	
	`0xFFFF`	
	`Standard chunk, object 2`	Object 2’s data
	`Current colors chunk`	
	`SCL chunk`	
	`0xFFFF`	
Symbols:	`Layer name 1`	List of layer names
	`0xFFFF`	Empty global list
	`Name of object 1`	Object 1’s symbols
	`0xFFFF`	
	`Name of object 2`	Object 2’s symbols
	`SCL comments`	
	`0xFFFF`	
	`0xFFFF`	Empty-end symbols

The chunks in the world data are all members of the union `T_WORLDCHUNK`. Therefore, a pointer to this type can be used for any chunk and the relevant element read according to its chunk type. The `ChkType` fields for each type of chunk are all in the same place, and can always be read from the standard chunk element. The `T_WORLDCHUNK` union looks like this:

```
typedef union
{
    T_STANDARD    Std;
    T_COLOURS     Col;
    T_DEFCOLS     Def;
    T_ROTATIONS   Rot;
    T_DISTANCE    Dis;
    T_ANGVELS     Ang;
    T_SCL         SCL;
    T_ANIMATIONS  Ani;
    T_TRIGSCL     Loc;
    T_TRIGSCL     Glo;
    T_ANIMCOLS    Acl;
    T_ATTACHMENTS Att;
    T_SHOOTVEC    Sho;
    T_TEXTINFO    Tex;
    T_LIGHTSOURCE Lig;
    T_INITSIZE    Isz;
    T_BENDING     Ben;
    T_VIEWPOINT   Vpt;
    T_BUBBLE      Bub;
    T_COLLISION   Cln;
    T_INITPOS     Ips;
    T_DYNAMICS    Dyn;
    T_LITCOLS     Lit;
    T_DEFLITCOLS  Dlc;
    T_TEXTURES    Txr;       (Not used in VRT 5.50)
    T_SORTING     Sor;
    T_TRANSLATE   Spx;
    T_TRANSLATE   Snx;
    T_AUTOSOUND   Asn;
    T_ENTITY      Ent;
    T_ORIGINALCOL Ocl;
    T_PROPERTIES  Prp;
    T_TEXCOORDS   Txc;
    T_MATERIAL    Mat;
    T_PROJECTOR   Prj;
    T_HORIZON     Hrz;
    T_FOGVOLUME   Fog;
}   T_WORLDCHUNK;
```

For example, you can read the initial X position from the initial position chunk pointed to by a pointer `p` (which is of type `T_WORLDCHUNK *`) using:

```
XPos=p->Ips.IXPos;
```

`Ips` specifies an initial position chunk.

T_ANGVELS

Type: **E_CTANGVELS**

Union: **T_WORLDCHUNK.Ang**

Structure:

```
typedef struct
   {
   unsigned short   ChkType,Length;
   short            XAngV,YAngV,ZAngV,IXAngV,IYAngV,IZAngV;
   } T_ANGVELS;
```

Description: Specifies the angular velocity applied to the object (how much the rotation changes each frame).

ChkType The chunk type (E_CTANGVELS).

Length The length of the chunk, including ChkType and itself.

XAngV, **YAngV**, **ZAngV** The angular velocities around the x, y and z axes respectively. These are added to the x, y and z rotations of the object each frame unless the flag E_OFSTOPANGV has been set in the OFlags field of the standard chunk. The velocity is measured in brees per frame. If the object has no rotations, this velocity is ignored, as the object cannot rotate.

IXAngV, **IYAngV**, **IZAngV** The initial values for XAngV, YAngV and ZAngV respectively.

See also: T_ROTATIONS

T_ANIMATIONS
T_ANISPEC

Type: **E_CTANIMATIONS**

Union: **T_WORLDCHUNK.Ani**

Structure:

```
typedef struct
   {
   unsigned char     FCel,LCel;
   short             CCel, Phase, AniVel,IAniVel;
   unsigned char     AniMode, IAniMode, Tick, ITick;
   }                 T_ANISPEC;

   typedef struct
   {
   unsigned short    ChkType,Length;
   unsigned short    NAnims;
   T_ANISPEC         Anim[1];
   } T_ANIMATIONS;
```

Description: Specifies animation controllers for up to eight separate animated movements on this object. Each animated point is associated with one controller.

Each controller has the following form:

FCel The first (lower) limit for the displayed cel number.

LCel The last (upper) limit for the displayed cel number. Automatic animations step the cel number between these two limits in ways defined by the AniMode (see below).

CCel The current cel being displayed, times 256. This is effectively a fractional number.

Phase The initial value of CCel.

AniVel is added to CCel at each frame. This is a fractional number, allowing a velocity of less than one cel per frame.

IAniVel The initial animation velocity.

AniMode A number representing the action to be taken when the end of the animation is reached.

`E_AMSTOPPED` The animation is stopped.

`E_AMWRAP` The animation starts again at the first cel; it wraps around.

`E_AMBOUNCE` The animation velocity is negated so that the animation "bounces" from one end of its range to the other.

E_AMWRAPF The animation proceeds as for E_AMWRAP for one frame only, and is then set to E_AMSTOPPED. By setting the animation mode to E_AMWRAPF at each required step, an SCL program may advance easily at any required point to the next cel.

E_AMBOUNCEF The animation proceeds as for E_AMBOUNCE for one frame only.

E_AMWRAPC The animation proceeds as per E_AMWRAP for one complete cycle. This allows simple movements to be triggered simply. At the end of the cycle, the animation mode is set to E_AMSTOPPED.

E_AMBOUNCEC The animation proceeds for one full cycle as for E_AMBOUNCE and stops when it reaches the lower limit of its range.

E_AMBOUNCEC2 The animation proceeds for one half-cycle as for E_AMBOUNCE. When the animation reaches either end of its range, the animation mode is reset to E_AMSTOPPED.

E_AMPBOUNCE The animation proceeds as for E_AMBOUNCE, except that end frames are not repeated.

E_AMPBOUNCEF The animation proceeds for a single frame as for E_AMPBOUNCE and then set to E_AMSTOPPED.

E_AMPBOUNCEC The animation proceeds for one full cycle as for E_AMPBOUNCE and stops when it reaches the lower limit of its range.

E_AMPBOUNCEC2 The animation proceeds for one half-cycle as for E_AMPBOUNCE. When the animation reaches either end of its range, the animation mode is reset to E_AMSTOPPED.

IAnimode The initial value of AniMode.

Tick and **ITick** Reserved.

The whole chunk is structured as follows:

ChkType The chunk type (E_CTANIMATIONS).

Length The length of the chunk, including ChkType and itself.

NAnims The number of defined animation controllers, from 1 to 8.

Anim An array of up to 8 animation controllers, as defined above.

T_ANIMCOLS

Type: **E_CTANIMCOLS**

Union: **T_WORLDCHUNK.Acl**

Structure:

```
typedef struct
   {
   unsigned short   ChkType,Length;
   unsigned short   ColsCel,Cels;
   unsigned char    AnimCols[1];
   } T_ANIMCOLS;
```

Description: This chunk contains information about the coloring of an object, dependant on the state of animation controller 1.

ChkType The chunk type (E_CTANIMCOLS).

Length The length of the chunk, including ChkType and itself.

ColsCel The number of colors defined per cel.

Cels The total number of animation cels for which colors are defined.

AnimCols An array of color indices which are used to color the facets of the object. All the colors for the first cel are defined, then all the colors for the second cel, and so forth. The facet number is used to index into this array, such that facet n has color AnimCols[ColsCel*(Cel-1)+n-1]. This preserves the color chunk even if facets are deleted from the shape definition.

See also: T_COLOURS, T_LITCOLS.

T_ATTACHMENTS
T_ATTSPEC

Type: **E_CTATTACHMENTS**

Union: **T_WORLDCHUNK.Att**

Structure:

```
typedef struct
   {
   unsigned short    Facet,Object;
   unsigned char     Flag,Pad;
   } T_ATTSPEC;

   typedef struct
   {
   unsigned short    ChkType,Length;
   unsigned short    NAtts;
   T_ATTSPEC         Att[1];
   } T_ATTACHMENTS;
```

Description: Specifies any attachments between this object and facets on other objects. The attachments are used in sorting - if the facet that this object is attached to is not visible, this object is sorted behind the object containing the facet. If the facet is visible, this object is sorted in front. This gives a reliable way of sorting objects whose outer cubes must overlap. (For example, a dormer window on a pitched roof).

Attachments are automatically Z-buffered, unless Z buffering is switched off.

An object may have several attachments, each of which is specified as follows:

`Facet` The facet number in the object to which another object is attached.

`Object` The object number to attach to this facet.

`Flag` Used by the system during processing.

`Pad` A spare byte.

A complete attachments chunk is made up as follows:

`ChkType` The chunk type (`E_CTATTACHMENTS`).

`Length` The length of the chunk, including `ChkType` and itself.

`NAtts` The number of attachments specified.

`Att` An array of attachment specifiers, as defined above.

T_AUTOSOUND
T_AUTOSOUNDDEF

Type: **E_CTAUTOSOUND**

Union: **T_WORLDCHUNK.Asn**

Structure:

```
typedef struct
   {
   unsigned short    Mode, Sound, Pitch, Chan;
   long              Vol, Handle;
   } T_AUTOSOUNDDEF;

typedef struct
   {
   unsigned short    ChkType, Length;
   short             NumSounds;
   T_AUTOSOUNDDEF    Sound[1];
   } T_AUTOSOUND;
```

Description: Sounds can be played using a trigger specified in the autosounds chunk. Each object can have an unlimited number of autosounds, each of which can be triggered in a different way. Each autosound is specified in the `T_AUTOSOUNDDEF` structure:

`Mode` The trigger used to play the sound:

`E_ASNDTRIG` Mouse click plays sound.

`E_ASNDTOGGLE` Mouse click toggles sound on/off.

`E_ASNDINIT` Play sound at startup.

`E_ASNDLOOP` Loop the sound.

`E_ASNDCHILDTRIG` Play sound if any child is clicked.

`Sound` The sound number. This is taken from the sounds used chunk defined by a `T_TRANSLATE` structure.

`Pitch` The play pitch at which the sound is played.

`Chan` The channel on which the sound is played.

`Vol` The volume at which the sound is played.

`Handle` The handle for the sound to be played.

The entire chunk consists of several autosound definitions.

`ChkType` The chunk type (`E_CTAUTOSOUND`).

`Length` The length of the chunk, including `ChkType` and itself.

`NumSounds` The number of autosounds attached to the object.

`Sound` An array of autosound definitions, as described above.

T_BENDING
T_BENDSPEC

Type: **E_CTBENDING**

Union: **T_WORLDCHUNK.Ben**

Structure:

```
typedef struct
   {
   short            XBend,YBend,ZBend,IXBend,IYBend,IZBend;
   unsigned short   FBend,LBend,CBend;
   } T_BENDSPEC;

   typedef struct
   {
   unsigned short   ChkType,Length;
   unsigned short   NBends;
   T_BENDSPEC       Bend[1];
   } T_BENDING;
```

Description: Objects' shapes may be bent by rotating a set of points about another given point. Up to eight independent rotations may be active at any one time.

Each bend is specified as follows:

XBend, **YBend**, **ZBend** The x, y and z rotations (relative to the object itself) by which the nominated points are rotated.

IXBend, **IYBend**, **IZBend** The initial values for the above.

FBend The number of the first point to be rotated.

LBend The number of the last point to be rotated.

CBend The number of the point to use as the center of the rotation. This point must already have been processed (have a lower point number) than `FBend`. This, in turn must be lower than `LBend`. This imposes constraints as to the design of objects that can be bent.

The entire chunk consists of several bends:

ChkType The chunk type (`E_CTBENDING`).

Length The length of the chunk, including `ChkType` and itself.

NBends The number of bending specifiers.

Bend An array of bending specifiers, as described above.

Note that bends may be stacked on top of each other, although they may not overlap. For example:

```
Point       0123456789012345678901234567890
Bend
1: 9-22              +------------+
2:10-18               +-------+
3:25-30                              +----+
```

This is OK. Although bends 1 and 2 are stacked, they do so legally. In the example below, however, bend 1 continues to apply in the region marked with the exclamation marks (!!!):

```
Point       0123456789012345678901234567890
Bend
1: 9-18              +--------+!!!!
2:10-22               +-----------+
3:25-30                              +----+
```

The bend descriptions must be ordered in order of increasing first bent point (`FBend`).

T_BUBBLE

Type: **E_CTBUBBLE**

Union: **T_WORLDCHUNK.Bub**

Structure:

```
typedef struct
   {
   unsigned short    ChkType,Length;
   short             SprXOff,SprYOff;
   unsigned short    Sprite;
   short             TxtXOff,TxtYOff;
   char *            Text;
   unsigned short    InitMessage;
   unsigned char     BGColour,FGColour;
   } T_BUBBLE;
```

Description: Attaches a visible marker or speech bubble to an object on the screen, consisting of some text and a background image. It is sorted with the object so that it appears at the same depth into the screen as that object. It is not scaled at all.

ChkType The chunk type (E_CTBUBBLE).

Length The length of the chunk, including ChkType and itself.

SprXOff, **SprYOff** The x and y offsets of the hotspot of the background sprite from the centroid of the object on the screen. The centroid is the average of all the visible points of the object.

Sprite The number of the image reference to display, or -1 if no image is to be used.

TxtXOff, **TxtYOff** The x and y offsets of the anchor point of the text from the centroid of the object on the screen.

Text A pointer to a null-terminated ASCII string to display, or NULL if none.

InitMessage The number of a message (from the message file) to be used initially, or -1 if none.

BGColour, **FGColour** The background and foreground color of the text to use.

T_COLLISION
T_COLLSPEC

Type: **E_CTCOLLISION**

Union: **T_WORLDCHUNK.Cln**

Structure:

```
typedef struct
   {
   long              XSize,YSize,ZSize;
   long              XOff,YOff,ZOff;
   long              IXSize,IYSize,IZSize;
   long              IXOff,IYOff,IZOff;
   } T_COLLSPEC;

   typedef struct
   {
   unsigned short    ChkType,Length;
   unsigned short    NumColls;
   T_COLLSPEC        Collision[1];
   } T_COLLISION;
```

Description: Defines which parts of an object can collide with other objects. If this chunk is not present, then the outer cube of the object is used. This chunk allows you to redefine the collision areas to affect only small areas of the object.

Each collision area is defined as cuboid:

XSize, **YSize**, **ZSize** The x, y and z sizes of the collision cuboid.

XOff, **YOff**, **ZOff** The x, y and z offsets of the origin of the collision cuboid from the origin of the object.

IXSize, **IYSize**, **IZSize** The initial values for the size.

IXOff, **IYOff**, **IZOff** The initial values for the offsets.

The whole chunk is structured as follows:

ChkType The chunk type (`E_CTCOLLISION`).

Length The length of the chunk, including `ChkType` and itself.

NumColls The number of collision cuboids defined.

Collision An array of collision cuboids.

T_COLOURS

Type: **`E_CTCOLOURS`**

Union: **`T_WORLDCHUNK.Col`**

Structure:

```
typedef struct
   {
   unsigned short   ChkType,Length;
   unsigned char    Colour[1];
   } T_COLOURS;
```

Description: Contains information about the coloring of an object, in the absence of a dynamic lighting model.

`ChkType` The chunk type (`E_CTCOLOURS`).

`Length` The length of the chunk, including `ChkType` and itself.

`Colour` An array of color indices which are used to color the facets of the object. The facet number is used to index into this array, such that facet `n` has color `Colour[n-1]`. This preserves the color chunk even if facets are deleted from the shape definition. This chunk may only be changed at run time if a default colors chunk also exists, allowing a reset to a known state.

See also: `T_DEFCOLS`, `T_LITCOLS`

T_DEFCOLS

Type: **E_CTDEFCOLS**

Union: **T_WORLDCHUNK.Def**

Structure:

```
typedef struct
   {
   unsigned short   ChkType,Length;
   unsigned char    DefCol[1];
   }                T_DEFCOLS;
```

Description: Contains information about the initial coloring of an object, in the absence of a dynamic lighting model.

ChkType The chunk type (E_CTDEFCOLS).

Length The length of the chunk, including ChkType and itself.

DefCol An array of color indices which are used to color the facets of the object. The facet number is used to index into this array, such that facet n has color DefCol[n-1]. This preserves the color chunk even if facets are deleted from the shape definition. If there is a color chunk on this object, the default colors are copied into that chunk at startup. The color chunk may then be changed at runtime while retaining the ability to be reset to a known initial state.

See also: T_COLOURS, T_DEFLITCOLS.

T_DEFLITCOLS

Type: **E_CTDEFLITCOLS**

Union: **T_WORLDCHUNK.Dlc**

Structure:

```
typedef struct
   {
   unsigned short    ChkType,Length;
   unsigned char     DefLitCol[1];
   } T_DEFLITCOLS;
```

Description: This chunk contains information about the initial coloring of an object, in the presence of a dynamic lighting model.

ChkType The chunk type (E_CTDEFLITCOLS).

Length The length of the chunk, including ChkType and itself.

DefLitCol An array of color indices which are used to color the facets of the object. The facet number is used to index into this array, such that facet n has color DefLitCol[n-1]. This preserves the color chunk even if facets are deleted from the shape definition. If there is a lit colors chunk on this object, the default colors are copied into that chunk at startup. The lit color chunk may then be changed at run time while retaining the ability to be reset to a known initial state.

See also: T_DEFCOLS, T_DEFLITCOLS

T_DISTANCE

Type: **E_CTDISTANCE**

Union: **T_WORLDCHUNK.Dis**

Structure:

```
typedef struct
   {
   unsigned short    ChkType,Length;
   long              VisDist,InvDist;
   unsigned short    Replace;
   } T_DISTANCE;
```

Description: Specifies the range beyond which this object is not visible, and an object with which to replace it.

ChkType The chunk type (E_CTDISTANCE).

Length The length of the chunk, including ChkType and itself.

VisDist The distance from the viewpoint at which the object, having been out of visible range, becomes visible again. It should always be less than or equal to InvDist.

InvDist The distance at which a visible object becomes invisible.

Replace The number of the object which replaces this object when out of range (usually a less detailed version of the same object), or 0 if no replacement is required.

T_DYNAMICS

Type: **E_CTDYNAMICS**

Union: **T_WORLDCHUNK.Dyn**

T_WORLDCHUNK.ODy

Structure:

```
typedef struct
  {
    unsigned short  ChkType,Length;
    long            MType, IMType;
    short           Collided, CollCube;
    short           Flags;
    unsigned short  Coupled;
    short           Grav, IGrav;
    short           Climb, IClimb;
    T_LONGVECTOR    Drive, IDrive;
    T_LONGVECTOR    External, IExternal;
    T_LONGVECTOR    MaxForce, IMaxForce;
    T_VECTOR        GroundFric, IGroundFric;
    char            Spare1[24];
    T_VECTOR        Restitution,IRestitution;
    T_LONGVECTOR    Vel, IVel;
    T_LONGVECTOR    MaxVel, IMaxVel;
    char            Spare2[42];
    short           ObjIn,ObjOn,GVel;
    short           Spare3,Spare4,Spare5;
  } T_DYNAMICS;
```

Description: The dynamics chunk specifies a movement model.

Since there are many of them, any field beginning with “I” can be assumed to be an initial value, copied into its associated field at the start. A vector is a set of three numbers, either short or long, specifying a position or direction in 3D space.

The chunk is as follows:

ChkType The chunk type (E_CTDYNAMICS).

Length The length of the chunk, including ChkType and itself.

MType A set of flags characterizing the movement of this object. Those currently defined are:

E_VTSIMPLE Set if the simple movement routine is to be used. (This is the default - only the simple movement routine is currently implemented).

E_VTNOCHECK Set if no collision checks are active.

E_VTVIEWPOINT Set if the object is to act like a viewpoint—so that it does not retain its velocity from frame to frame.

E_VTFLYING Indicates whether this object is flying and consequently gravity has no effect on its motion.

E_VTPUSHABLE Set if this object can be pushed by another moving object.

E_VTCANTPUSH Set if the object cannot push other objects about (even if they are flagged as pushable). This is most useful for viewpoint objects.

Other flags are defined, but are for internal use only.

Collided The number of the object with which this object last collided.

CollCube The number of the collision cube on this object which last collided with something.

Flags Various flags for internal use only.

Coupled An object number, which specifies the rotations which affect the direction of the driving force on this object.

Grav A measure of the gravitational attraction which acts on this object. This is measured in units per frame per frame.

Climb A measure of the maximum height by which the object can climb in one frame.

Drive A vector specifying the driving force applied by this object to itself. It is defined as being "forward" when x and y are 0 and the z component of the vector is positive. This force is relative to the object's reference frame.

External A vector specifying the external force applied this frame by other objects. This is relative to the world reference frame.

MaxForce Specifies the maximum allowed driving force for this object.

GroundFric Specifies the amount of friction along the object's x, y and z axes accorded by contact with the ground. This is relative to the object's reference frame. 100% friction in a particular axis is represented by a value of 0x4000.

Spare1 Not used.

Restitution Specifies the amount by which the object rebounds when involved in a collision. 100% is represented by a value of 0x4000.

Vel The velocity of the object, relative to the world reference frame. This is automatically calculated by the program.

MaxVel A vector specifying the maximum velocity in each axis at which a collision can occur without setting the flag E_TRFALL in the object flags (see E_CTSTANDARD). This only currently applies to the y axis.

Spare2 Not used.

ObjIn, **ObjOn** The object numbers of the object that the moving object is inside and sitting on, respectively.

GVel The induced velocity due to gravity.

Spare3, **Spare4**, **Spare5** Not used.

T_ENTITY

Type: **E_CTENTITY**

Union: **T_WORLDCHUNK.Ent**

Structure:

```
typedef struct
     {
     unsigned short  ChkType, Length;
     long            EntityHandle;
     char            szName[64];
     } T_ENTITY;
```

Description: Contains information about an avatar in a Viscape multi user world.

ChkType The chunk type (`E_CTENTITY`).

Length The length of the chunk, including `ChkType` and itself.

EntityHandle The handle used to identify the avatar.

szName The name of the avatar.

T_FOGVOLUME

Type: **E_CTFOGVOLUME**

Union: **T_WORLDCHUNK.Fog**

Structure:

```
typedef struct
{
    unsigned short  ChkType,Length;
    long            Flags,IFlags;
    unsigned short  Object,IObject;
    long            Density,IDensity;
    unsigned char   RGBA[4],IRGBA[4];
} T_PROJECTOR;
```

Description: Contains the fog volume definition for this object:

ChkType The chunk type (`E_CTFOGVOLUME`).

Length The length of the chunk, including `ChkType` and itself.

Flags, **IFlags** The current and initial flags specifying projector properties:

`E_FOGINFINITE` Indicates that the fog volume has an infinite top plane.

`E_FOGVOID` Indicates that the fog volume is empty.

`E_FOGDISABLE` Indicates that the fog volume is disabled.

T_HORIZON

Type: **E_CTHORIZON**

Union: **T_WORLDCHUNK.Hrz**

Structure:

```
typedef struct
{
    unsigned short  ChkType,Length;
    long            Flags,IFlags;
    short           FullTex[6],IFullTex[6];
    short           HalfTex[5],IHalfTex[5];
    short           StripTex,
    unsigned char   RGBA[4],IRGBA[4];
} T_HORIZON;
```

Description: Contains the horizon definition:

ChkType The chunk type (E_CTHORIZON).

Length The length of the chunk, including ChkType and itself.

Flags, **IFlags** The current and initial flags specifying horizon properties:

E_HRZFOGHORZ Allow horizon to be fogged by global fog or fog volume on Root Object.

E_HRZFOGHORZ Mask for horizon strip repeat value.

FullTex, **IFullTex** The current and initial references for the fully textured horizon. This is translated by the Textures Used chunk, which is of type T_TRANSLATE. The references are for the Front, Back, Left, Right, Top, Bottom textures.

HalfTex, **IHalfTex** The current and initial references for the half textured horizon. This is translated by the Textures Used chunk, which is of type T_TRANSLATE. The references are for the Front, Back, Left, Right, Top textures.

StripTex, **IStripTex** The current and initial reference for the textured strip horizon. This is translated by the Textures Used chunk, which is of type T_TRANSLATE.

RGBA, **IRGBA** The current and initial color of the horizon filter which is used to modulate the underlying horizon. An array of 4 bytes specifying the red, green, blue values each 0 to 255. The 4th byte is not used.

T_INITPOS

Type: **E_CTINITPOS**

Union: **T_WORLDCHUNK.Ips**

Structure:

```
typedef struct
   {
   unsigned short    ChkType,Length;
   long              IXPos,IYPos,IZPos;
   } T_INITPOS;
```

Description: A movable object must be able to return to its initial position when the world is reset. This chunk stores that information:

ChkType The chunk type (E_CTINITPOS).

Length The length of the chunk, including ChkType and itself.

IXPos, **IYPos**, **IZPos** The initial x,y and z positions of the object. They are copied to XPos, YPos and ZPos in the standard chunk at initialization.

T_INITSIZE

Type: **E_CTINITSIZE**

Union: **T_WORLDCHUNK.Isz**

Structure:

```
typedef struct
   {
   unsigned short    ChkType,Length;
   long              IXSize,IYSize,IZSize;
   } T_INITPOS;
```

Description: A resizable object must be able to return to its initial size when the world is reset. This chunk stores that information:

ChkType The chunk type (E_CTINITSIZE).

Length The length of the chunk, including ChkType and itself.

IXSize, **IYSize**, **IZSize** The initial x,y and z sizes of the object. They are copied to XSize, YSize and ZSize in the standard chunk at initialization time.

T_LIGHTSOURCE

Type: **E_CTLIGHTSOURCE**

Union: **T_WORLDCHUNK.Lig**

Structure:

```
typedef struct
   {
   unsigned short   ChkType,Length;
   long             Bright,IBright;
   unsigned short   Flags;
   long             XVOff,YVOff,ZVOff,IXVOff,IYVOff,IZVOff;
   short            XRot,YRot,ZRot,IXRot,IYRot,IZRot;
   unsigned short   BeamWidth,IBeamWidth,
                    Dispersion,IDispersion;
   unsigned char    ColR,ColG,ColB,IColR,IColG,IColB;
   unsigned short   BeamEdge,IBeamEdge;
   } T_LIGHTSOURCE;
```

Description: Marks the object as being a source of light. When switched on, it is added to an "active lightsource" list, which may have up to `E_MAXLIGHTS` entries. This list is then used to calculate the displayed colors of the facets. The structure `T_LIGHT` is used internally in this list, which should not be altered by an application.

ChkType The chunk type (`E_CTLIGHTSOURCE`).

Length The length of the chunk, including `ChkType` and itself.

Bright The distance at which this light source by itself gives a 100% illumination value to a facet perpendicular to and in the center of its beam.

IBright The initial 100% brightness distance.

Flags A set of flags:

`E_LSON` Set if the light source is on.

`E_LSONDEF` The default state for the above.

`E_LSPARA` Set if the light source is a parallel source (like the sun). In this case, the angle and distance between the source and the world origin is taken, not between the source and the target object, unless `E_LSROTPARA` is also set (see below).

`E_LSOK` Used by the system.

`E_LSNONEGS` Set if facets facing away from this light source should not be calculated as having a negative contribution from it. In a one-light world, turning NoNegs on results in the unlit sides of objects being a single uniform color.

`E_LSROTPARA` Set if the parallel light direction is to be set by rotations rather than position. In this case, the light source is effectively placed at 10 000 units from the origin at such a position that the angle of incidence of the light follows the rotations of the light source.

`XVOff`, **`YVOff`**, **`ZVOff`** The x,y, and z offsets from the object's origin of the light source.

`IXVOff`, **`IYVOff`**, **`IZVOff`** The initial x, y and z offsets as above.

`XRot`, **`YRot`**, **`ZRot`** The x,y and z rotations of the beam axis relative to the lightsource object.

`IXRot`, **`IYRot`**, **`IZRot`** The initial rotations as above.

`BeamWidth` A measure of the width of the beam from the lightsource (0 = isotropic, 32767=very narrow).

`IBeamWidth` The initial beam width.

`Dispersion` A measure of the dispersion of the light with distance. It is 100 times the power factor (inverse square law: power = 2 therefore dispersion=200).

`IDispersion` The initial dispersion value.

`ColR`, **`ColG`**, **`ColB`** The R,G and B components of the color of the light source. At present, these are ignored so they should be set to 0xFF (white) to allow for future expansion.

`IColR`, **`IColG`**, **`IColB`** The initial values of the color components.

`BeamEdge` A value representing how sharply the edge of the beam is defined. High values mean sharp beam edges (0=broad edge, 1000=sharp edge, 30000=very sharp).

`IBeamEdge` The initial beam edge sharpness.

T_LITCOLS

Type: **E_CTLITCOLS**

Union: **T_WORLDCHUNK.Lit**

Structure:

```
typedef struct
   {
   unsigned short   ChkType,Length;
   unsigned char    LitCol[1];
   } T_COLOURS;
```

Description: Contains information about the coloring of an object, in the presence of a dynamic lighting model. It is used directly in a frame-by-frame model. In a one-pass lighting model, the color for each facet is calculated and stored in the current colors chunk. If no current colors chunk exists, it is ignored.

ChkType The chunk type (E_CTCOLOURS).

Length The length of the chunk, including ChkType and itself.

LitCol An array of color indices which are used to color the facets of the object. The facet number is used to index into this array, such that facet n has color LitCol[n-1]. This preserves the color chunk even if facets are deleted from the shape definition. This chunk may only be changed at run time if a default lit colors chunk also exists, allowing a reset to a known state.

See also: T_DEFLITCOLS, T_COLOURS

T_MATERIAL

Type: **E_CTMATERIAL**

Union: **T_WORLDCHUNK.Mat**

Structure:

```
typedef struct
{
    unsigned short   ChkType,Length;
    unsigned char    Shininess,IShininess
    unsigned char    Transparency,ITransparency
    long             Flags,IFlags;
} T_MATERIAL;
```

Description: Contains the material definition for this object:

ChkType The chunk type (E_CTMATERIAL).

Length The length of the chunk, including ChkType and itself.

Shininess, **IShininess** The current and initial shininess values as a power index (0=no specularity).

Transparency, **ITransparency** The current and initial transparency values (0=opaque, 255=fully transparent).

Flags, **IFlags** The current and initial flags specifying material properties:

E_MATCHILDREN - Indicates that this material will apply to all children unless they have their own material.

E_MATBLEND - Indicates that additive blending should be used with this material.

E_MATTEXMODULATE - Indicates that the facet colour should modulate any texture.

E_MATFADE - Indicates that facets should fade according to orientation with respect to the viewpoint.

E_MATFADEREVERSE - Indicates that if E_MATFADE is set, the mapping should be reversed.

E_MATNOPROJECT - Indicates that textures should not be projected onto objects using this material.

E_MATTRANSPMOD - Indicates that any stipple transparency will be modulated by the material transparency.

E_MATFLATLITCOL - Indicates that facets are not lit but still use lit colors.

E_MATIGNORELITCOL - Indicates that facets are not lit and use current colors (when combined with E_MATFLATLITCOL current colors are lit).

E_MATTWOTONE - Indicates that lit facets are rendered as two tone (if combined with both or neither of E_MATFLATLITCOL and E_MATIGNORELITCOL).

E_MATNOTEXLINEAR - Indicates that textures are not linearly filtered even if E_EXCFGTEXLINEAR is set (Direct3D only).

E_MATNOFOG - Indicates that facets are not affected by fog.

E_MATOUTLINEMAT - Indicates that outlines are made between different base colors.

E_MATOUTLINESIL - Indicates that a silhouette outline is drawn around object and children (in addition to their own material state).

E_MATOUTLINEWIDTH - Specifies outline width (0=thin, 7=thick).

E_MATOUTLINECOL - Specifies palette index for outline color (0-255, 0=no outline).

T_ORIGINALCOL

Type: **E_CTORIGINALCOL**

Union: **T_WORLDCHUNK.Ocl**

Structure:

```
typedef struct
   {
   unsigned short   ChkType, Length;
   unsigned short   ColsCel, Cels;
   unsigned char    AnimCols[1];
   } T_ORIGINALCOL;
```

Description: This structure is used to implement the Colors Palette in Superscape Do 3D™.

T_PROJECTOR

Type: **E_CTPROJECTOR**

Union: **T_WORLDCHUNK.Prj**

Structure:

```
typedef struct
{
    unsigned short   ChkType,Length;
    short            Image,IImage;
    long             Flags,IFlags;
    T_LONGVECTOR     BoundPos,IBoundPos;
    T_LONGVECTOR     BoundSize,IBoundSize;
    unsigned char    RGBA[4],IRGBA[4];
} T_PROJECTOR;
```

Description: Contains the projector definition for this object:

ChkType The chunk type (E_CTPROJECTOR).

Length The length of the chunk, including ChkType and itself.

Image, **IImage** The current and initial reference of the image to project. This is translated by the Textures Used chunk, which is of type T_TRANSLATE.

Flags, **IFlags** The current and initial flags specifying projector properties:

E_PROJLINEAR - Indicates a linear projection, perspective otherwise

E_PROJSINGLE - Indicates that a linear projection will not be repeated.

E_PROJREFLECT - Indicates that this is a reflector rather than projector.

E_PROJBOTHSIDES - Indicates that projector will project onto back facing facets (with respect to the projector).

E_PROJCHILDREN - Indicates that only parent and its children will be projected onto.

E_PROJNOBLEND - Indicates that the projected image will not be blended.

E_PROJDISABLE - Indicates that the projector is disabled.

E_PROJMIRROR - Indicates that this is a mirror projector.

E_PROJSHADOW - Indicates that this is a shadow projector.

E_PROJNOBOUNDS - Indicates that bounds values are ignored.

E_PROJNOUPDATE - Indicates that mirror/shadow image should not be updated.

E_PROJESCALEMASK - Environment map tweak value mask.

E_PROJESCALESHIFT - Environment map tweak value shift.

`E_PROJESCALESCALE` - Environment map tweak value scale.

`BoundPos`, **`IBoundPos`** The current and initial offset (relative to parent object) of the cuboid by which the projection is bounded.

`BoundSize`, **`IBoundSize`** The current and initial size of the cuboid by which the projection is bounded.

`RGBA`, **`IRGBA`** The current and initial colour of the projection. An array of 4 bytes specifying the red, green, blue and brightness values each 0 to 255.

T_PROPERTIES
T_PROPERTYDEF

Type: **E_CTPROPERTIES**

Union: **T_WORLDCHUNK.Prp**

Structure:

```
typedef struct
   {
   long              Value;
   long              Min;
   long              Max;
   } T_PROPERTY_LONG;

typedef struct
   {
   long              Offset;
   long              Length;
   long              Strings;
   } T_PROPERTY_STRING;

typedef struct
   {
   float             Value;
   float             Min;
   float             Max;
   } T_PROPERTY_FLOAT;

typedef struct
   {
   short             Value;
   } T_PROPERTY_BOOL;

typedef struct
   {
   unsigned short            Type;
   char                      Name[32];
   union {
         T_PROPERTY_LONG     Long;
         T_PROPERTY_STRING   String;
         T_PROPERTY_FLOAT    Float;
         T_PROPERTY_BOOL     Bool;
          } Value;
   } T_PROPERTYDEF;

typedef struct
   {
   unsigned short    ChkType, Length;
   unsigned short    NumProperties;
   unsigned short    Changed;
   T_PROPERTYDEF     Property[1];
   } T_PROPERTIES;
```

Description: Defines a properties attribute of type `E_CTPROPERTIES` for an object. Each attribute consists of a structure that contains one or more property definitions which can be of the following types:

`Long` A value of type `long:`

`Value` The current value assigned to the property.

`Min` The minimum value that the user can apply to the property.

`Max` The maximum value the user can apply to the property.

`String` A value of type `sting:`

`Offset` The offset from the start of the chunk at which the string is stored.

`Length` The maximum length of a string assigned to the property. This can be an unlimited number of characters.

`Strings` The offset from the start of the chunk at which a set of strings are stored. Each property can have more than one string that the user can select from. The strings are separated by '`\0`' with a '`\0\0`' terminator.

`Float` A value of type `float:`

`Value` The current value assigned to the property.

`Min` The minimum value that the user can apply to the property.

`Max` The maximum value the user can apply to the property.

`Bool` A value of type `boolean:`

`Value` The current boolean value assigned to the property.

Each property definition contains the following information:

`Type` The type of union:

0	`E_PROPERTY_LONG`
1	`E_PROPERTY_STRING`
2	`E_PROPERTY_FLOAT`
3	`E_PROPERTY_BOOL`

`Name` The name of the property definition, up to a maximum 32 characters.

`Value` The current value of the property.

Finally the properties attribute is constructed from the property definitions:

`ChkType` The chunk type (`E_PROPERTIES`).

`Length` The length of the chunk, including `ChkType` and itself.

`NumProperties` The number of property definitions assigned to the properties attribute.

`Changed` A flag that indicates when a value within the chunk has been changed. Set automatically if the value has been changed using the VRT interface, but must be set manually if changed dynamically using SCL.

`Property` An array of `T_PROPERTYDEF` structures.

T_ROTATIONS

Type: **`E_CTROTATIONS`**

Union: **`T_WORLDCHUNK.Rot`**

Structure:

```
typedef struct
   {
   unsigned short    ChkType,Length;
   short             XRot,YRot,ZRot,IXRot,IYRot,IZRot;
   unsigned short    Spare;
   long              XCentre,YCentre,ZCentre;
   } T_ROTATIONS;
```

Description: This chunk specifies the amount by which an object is rotated with reference to its parent, and about which center.

`ChkType` The chunk type (`E_CTROTATIONS`).

`Length` The length of the chunk, including `ChkType` and itself.

`XRot`, **`YRot`**, **`ZRot`** The angular rotations of the object about the x,y and z axes respectively. These angles are measured in brees (1/65536 of a circle).

`IXRot`, **`IYRot`**, **`IZRot`** The initial values for XRot,YRot and ZRot. They are copied there at initialization time, giving a known state at reset.

`Spare` Not used.

`XCentre`, **`YCentre`**, **`ZCentre`** The offset from the origin of the object of the rotation center point.

See also: `T_ANGVELS`

T_SCL

Type: **E_CTSCL**

Union: **T_WORLDCHUNK.SCL**

Structure:

```
typedef struct
   {
   unsigned short   ChkType,Length;
   unsigned char    SCL[1];
   } T_SCL;
```

Description: Specifies an SCL program to be executed on this object each frame.

ChkType The chunk type (E_CTSCL).

Length The length of the chunk, including ChkType and itself.

SCL A set of object code bytes for an SCL program. These are executed every frame unless the flag E_OFSTOPCONDS has been set in the OFlags field of the standard chunk. For a detailed description of the SCL language, see the "VRT User Guide: Chapter 18 - SCL" and the online "Reference Books".

See also: T_TRIGSCL

T_SHOOTSPEC
T_SHOOTVEC

Type:	**E_CTSHOOTVEC**
Union:	**T_WORLDCHUNK.Sho**

Structure:

```
typedef struct
   {
   unsigned short    Origin,Vector;
   } T_SHOOTSPEC;

   typedef struct
   {
   unsigned short    ChkType,Length;
   unsigned short    NShoot;
   T_SHOOTSPEC       Shoot[1];
   } T_SHOOTVEC;
```

Description: Specifies a set of vectors along which projectiles may be launched. The projectile has a velocity equal to the length of the vector, in the direction of the vector (relative to the object from which it was launched). The total velocity is the sum of this vector and the velocity of the launcher.

Each vector is specified as follows:

Origin The point number in the shape of this object of the origin of the vector.

Vector The point number in the shape of this object of the other end of the vector.

The whole chunk is:

ChkType The chunk type (`E_CTSHOOTVEC`).

Length The length of the chunk, including `ChkType` and itself.

NShoot The number of defined vectors.

Shoot An array of vectors, as defined above.

T_SORTING

Type: **`E_CTSORTING`**

Union: **`T_WORLDCHUNK.Sor`**

Structure:

```
typedef struct
   {
   unsigned short    ChkType,Length;
   long              XPos,YPos,ZPos,XSize,YSize,ZSize;
   } T_SORTING;
```

Description: Specifies an alternative to the outer cube to use in the standard sorting routine. For attenuated objects or ones with large “antennae”, a compact sorting cuboid near the center of the object may be preferable to a large outer cube containing the whole object.

`ChkType` The chunk type (`E_CTSORTING`).

`Length` The length of the chunk, including `ChkType` and itself.

`XPos`, **`YPos`**, **`ZPos`** The x, y and z offsets from the object’s origin of the sorting cuboid’s origin.

`XSize`, **`YSize`**, **`ZSize`** The x, y and z sizes of the sorting cuboid.

T_STANDARD

Type: **E_CTSTANDARD**

Union: **T_WORLDCHUNK.Std**

Structure:

```
typedef struct
   {
   unsigned short   ChkType,Length,TotLen,Number;
   long             Child,Sibling;
   void *           Parent;
   void **          List;
   unsigned short   MaxChunk;
   long             XSize,YSize,ZSize,XPos,YPos,ZPos,
                    DiagDis;
   unsigned short   Type,Layer;
   long             DFlags,OFlags;
   unsigned short   Trigger;
   }                T_STANDARD;
```

Description: This chunk contains the standard information that every object must have:

ChkType The chunk type (E_CTSTANDARD).

Length The length of the chunk, including ChkType and itself.

TotLen The total length of the whole object.

Number The (unique) number assigned to this object. The user does not, generally, use this number directly, since all the objects are named. The user refers to the object by name and the VRT deals with the numbering.

Child The offset from the start of this object to the start of its first child, or 0 if it has no children.

Sibling The offset from the start of this object to the start of its immediate sibling, or 0 if it has none.

Parent Contains the absolute address of this object's parent. This is filled in at startup.

List A pointer to the start of this object's chunk address list, which is an array of pointers to each of the object's chunks, indexed by chunk type. This is filled in at startup.

MaxChunk The index number of the largest chunk type in the object—the length of the chunk address list for this object. This is filled in at startup.

XSize, **YSize**, **ZSize** The size of the object's bounding cube in the x,y and z axes respectively. They may not be negative.

XPos, **YPos**, **ZPos** The x,y, and z positions of the origin of this object relative to that of its parent. These may be negative, but normally an object should be completely inside its parent.

`DiagDis` Not used.

`Type` Contains the index of the shape that this object is within the shape list. This number is not seen by the user since all the shapes are named. The user refers to shapes by name and lets the VRT assign the numbers.

`Layer` Contains the layer number which this object belongs to. This is used internally within the VRT.

`DFlags` Flags which reflect the data structure of the object in some way. Their masks and action if set are as follows:

`E_DFPRESORTED` Indicates that this object's children are already sorted and do not need to be sorted at run time.

`E_DFENTERABLE` Indicates that this object may be entered by a moving object.

`E_DFMOVABLE` Indicates that this object has a movable attribute and that it is sorted depending on its position in the world.

`E_DFREPLACEMENT` Indicates that this object is a replacement for a more detailed object.

`E_DFCOLOURDEFN` Indicates that the object's colors should be copied from the shape definition at reset time.

`E_DFDONTREJECT` Indicates that this object's children should not be trivially rejected if this object is not in view.

`E_DFROTSORT` Indicates that the object is rotated and should thus be handled differently by the sorting routine. This can be disabled if the object's rotated bounding cube is not markedly different from its unrotated size and position.

`E_DFROTCOLL` Indicates that an object's rotations should be included during the sorting calculation.

`E_DFDONTCHECK` Indicates that this object should not be included in the network checksum calculations.

`E_DFNOADOPT` Indicates that this object will not adopt other objects that are dragged or moved into it.

`OFlags` Flags reflecting the status of the object. The masks and actions are:

`E_OFINVISIBLE` Indicates that this object cannot be seen (is invisible). It may still move, rotate, and other attributes

`E_OFINVISDEF` Default state of above.

`E_OFKILLED` Indicates that the object may not move, or rotate.

`E_OFSTOPMOV` Indicates that this object may not move.

`E_OFAUTOANI` Indicates that the anmations on this object proceed automatically.

`E_OFAUTODEF` Default state of above.

`E_OFRANGE` Indicates that this object has been made invisible or replaced since it is out of viewing range.

`E_OFDECOUPLE` Indicates that the object's movement is decoupled from it rotations.

`E_OFDECOUPDEF` Default state of the above.

`E_OFSTOPCONDS` Indicates that the SCL on this object may not be executed automatically.

`E_OFSTOPANGV` Indicates that the angular velocities on this object are disabled.

`E_OFMSGWAITING` Indicates that there is a message waiting for this object in the message pipe.

`E_OFDEBUGSCL` Indicates that this object's SCL triggers the SCL debugger.

`E_OFDEBUGDEF` The default state of the above.

`E_OFLOCALCONT` Indicates that the locally triggered SCL on this object has not completed yet and should be triggered again next frame.

`E_OFGLOBALCONT` Indicates that the globally triggered SCL on this object has not completed yet and should be triggered again next frame.

`E_OFTANGIBLE` Indicates that this object should not be tested for collisions if visible, or that it <u>should</u> be tested if it is invisible.

`E_OFTANGDEF` Default state of `E_OFTANGIBLE`.

`E_OFNOPUSH` Indicates that this object will not push other objects, even if they are marked as being pushable.

`E_OFGLOBALCONT` The default state of `E_OFNOPUSH`.

The default flags can be extracted using the mask `E_OFDEFAULTS`.

`Triggers` Flags indicating the nature of up to 16 trigger events. Their masks and meanings are:

`E_TRHIT` Indicates that the object has hit something.

`E_TRFALLM` Indicates that the object has fallen too far to be unaffected.

`E_ACTIVATED` Indicates that the object has been activated by a left mouse click.

`E_ACTIVATEDR` Indicates that the object has been activated by a right mouse click.

T_TEXTINFO

Type: **`E_CTTEXTINFO`**

Union: **`T_WORLDCHUNK.Tex`**

Structure:

```
typedef struct
   {
   unsigned short    ChkType,Length;
   char              TextInfo[1];
   } T_TEXTINFO;
```

Description: Contains text relating to this object. This text may be retrieved and displayed by a small SCL program on the object, or just viewed in the editors. One suggested use is to include ordering information and part numbers in the object itself.

`ChkType` The chunk type (`E_CTTEXTINFO`).

`Length` The length of the chunk, including `ChkType` and itself.

`Text` A zero-terminated ASCII text string. It may contain any non-zero character, and be up to approximately 32 KB long.

T_TEXTURES
T_TEXSPEC

Type:	**`E_CTTEXTURES`**
Union:	**`T_WORLDCHUNK.Txr`**
Description:	This chunk is no longer used. See `T_TEXCOORDS` for details on texture chunks supported by VRT 5.50.

T_TEXCOORDS
T_TEXCOORDSPEC
T_TEXCOORD

Type: **E_CTTEXCOORDS**

Union: **T_WORLDCHUNK.Txc**

Structure:

```
typedef struct
   {
   float            tu, tv;
   } T_TEXCOORD

typedef struct
   {
   unsigned short   Facet, NumPoints;
   short            Texture, ITexture, TexScale, ITexScale;
   short            ScaleX, ScaleY, OffsetX, OffsetY;
   T_TEXCOORD       Coord[1];
   } T_TEXCOORDSPEC;

typedef struct
   {
   unsigned short   ChkType, Length;
   unsigned short   NTextures, Flags;
   T_TEXCOORDSPEC   Tex[1];
   } T_TEXCOORDS;
```

Description: Defines the texture coordinates of each point on a facet.

The texture coordinates for each point are stored in the `T_TEXCOORD` structure. Values between 0—1 indicate that a single texture or part of it is displayed in the facet. If the value is greater than 1, more than one instance of the texture is mapped onto the facet.

`tu` The ‘x’ coordinate of the point.

`tv` The ‘y’ coordinate of the point.

Each facet requires a coordinate point for each of its points.

`Facet` The facet number to which the texture is applied.

`NumPoints` The number of points used to define the facet. A `T_TEXCOORD` structure is required for each of these points.

`Texture`, **`ITexture`** The texture reference number of the texture (and initial texture) to apply to the facet. This is translated using the Textures used chunk, which is of type `T_TRANSLATE`.

`TexScale`, **`ITexScale`** The pixel size of the texture (and initial texture). This can be one of the following:

`E_TEXOFF` Textures off.

`E_TEX1X1` Display using a 1x1 grid.

`ScaleX`, **`ScaleY`** The X and Y scales. These are the number of copies of the texture to tile onto the facet in the X and Y directions respectively. These are specified in $^1/_{256}$ths.

`OffsetX`, **`OffsetY`** The amount to offset each of the texture coordinates. These are specified in $^1/_{256}$ths.

`Coord` An array of texture coordinates, as described above.

Finally, each set of texture coordinates are stored together in array.

`ChkType` The chunk type (`E_CTTEXCOORDS`).

`Length` The length of the chunk, including `ChkType` itself.

`NTextures` The number of texture coordinate entries.

`Flags` A set of flags that control how the texture is mapped onto the facet, and colored:

`E_TCLIGHT` The texture is lit using the color of the underlying facet.

`Tex` An array of `T_TEXCOORDSPEC` chunks.

T_TRANSLATE

Type: **E_CTSPRTRANS**
E_CTSNDTRANS

Union: **T_WORLDCHUNK.Spx**
T_WORLDCHUNK.Snx

Structure:

```
typedef struct
   {
   unsigned short   ChkType,Length;
   unsigned short   NumTrans;
   unsigned short   Translate[1];
   } T_TRANSLATE;
```

Description: Contains a translation table for images (textures) or sounds. These are the textures used and sounds used chunks.

ChkType The chunk type (`E_CTTEXTINFO`).

Length The length of the chunk, including `ChkType` and itself.

NumTrans The number of translation table entries.

Translate An array of translation values. The true sound or texture number is given by indexing into this array using the sound or texture reference number.

T_TRIGSCL

Type: **E_CTSCLLOCAL**
E_CTSCLGLOBAL

Union: **T_WORLDCHUNK.Loc**
T_WORLDCHUNK.Glo

Structure:

```
typedef struct
   {
   unsigned short    ChkType,Length;
   unsigned short    Triggers;
   unsigned char     SCL[1];
   } T_SCL;
```

Description: Specifies an SCL program to be executed on this object when a trigger is activated.

ChkType The chunk type (`E_CTSCLLOCAL` or `E_CTSCLGLOBAL`).

Length The length of the chunk, including `ChkType` and itself.

Triggers A set of bits corresponding to the local triggers (`Triggers` in the standard chunk on this object) or the global triggers (`C_Triggers`). If any of the triggers specified in this value are set, then the SCL is executed.

SCL A set of object code bytes for an SCL program. These are executed when triggered as above unless the flag `E_OFSTOPCONDS` has been set in the `OFlags` field of the standard chunk. For a detailed description of the SCL language, see the “VRT User Guide: Chapter 16 - SCL” and the VRT online “Reference Books.”

See also: `T_SCL`

T_VIEWPOINT

Type: **E_CTVIEWPOINT**

Union: **T_WORLDCHUNK.Vpt**

Structure:

```
typedef struct
   {
   unsigned short    KeyFrame;
   unsigned char     PFlag,PType;
   long              XVOff,YVOff,ZVOff;
   }                 T_POSCONT;

   typedef struct
   {
   unsigned short    KeyFrame,ObjAtt,X,Y,Z;
   short             Zoom;
   }                 T_ROTCONT;

   typedef struct
   {
   unsigned short    Length,ObjView,ObjCon,ObjMis,ObjFir,
                     ShootVec,Point;
   unsigned short    CurFrame,TotFrame,NumPosCont,
                     NumRotCont,OldZoom;
   unsigned char     VPLock,Type;
   T_POSCONT         PosCont[1];
   }                 T_VIEW;

   typedef struct
   {
   unsigned short    ChkType,Length;
   unsigned short    NumVPs;
   T_VIEW            View[1];
   } T_VIEWPOINT;
```

Description: Defines the relationship between an object and viewpoints attached to it. The first 20 defined viewpoints can be accessed through the standard functions in the keyboard table. Each viewpoint can move and rotate, which makes this chunk perhaps the most complex of all.

Each viewpoint consists of a set of positional and rotational control points which are interpolated at run time. The position and rotation cycle over a number of frames, with the control points being at key frames within this cycle. If only one position and rotation is defined, the viewpoint does not move.

A positional control point has the following structure:

Keyframe The number of the frame in the cycle that this point refers to.

`PFlag` Flags affecting the interpolation of position. At present these flags are not used.

`PType` The type of path that the position takes to the next key frame:

> `E_VPJUMP` The position jumps to the next position at the next key frame.
>
> `E_VPLINE` The position progresses smoothly along a straight line to the next key frame.
>
> `E_VPCURVE` The position is interpolated along a smooth curve going through the control points.

`XVOff`, **`YVOff`**, **`ZVOff`** The x,y and z offsets from the viewpoint object (relative to the object's frame of reference) of the viewpoint position at this key frame.

Rotational control points have the following structure:

`Keyframe` The number of the frame in the cycle that this point refers to.

`ObjAtt` The object number that the rotation is targeted at. The viewpoint points at the center of this target object, offset by the values in the X, Y, and Z fields. There are, however, three special values of this field which modify this meaning:

> `E_VPRELROT` The X,Y,Z fields specify the rotation of the viewpoint relative to the viewpoint object.
>
> `E_VPABSROT` The X,Y,Z fields specify the absolute rotation of the viewpoint, with respect to the world frame of reference.
>
> `E_VPPATH` The viewpoint points at the point on the path X frames forward of its current position. This gives a "look along the path" mode, with a certain amount of look-ahead.

`X`, **`Y`**, **`Z`** Offsets from the target object, although they may have modified meanings (see previous field description).

`Zoom` The zoom value prevailing at this point, or -1 for no change, or -2 to restore to the starting value.

Interpolation of values is dependant on the different types of `ObjAtt`. If they are both target types (`E_VPPATH` or a target object number), the interpolation takes place to follow a point on a straight line between the two target points. In all other cases, the interpolation goes between the absolute angles at the start and end.

Each separate viewpoint has the following structure:

`Length` The total length of this viewpoint description, including this length.

`ObjView` The object number to which the viewpoint is attached, or `0xFFFF` if this viewpoint cannot be selected.

`ObjCon` The object to be controlled when this viewpoint is selected.

`ObjMis` The missile object used by this viewpoint.

`ObjFir` The object from which the missile is launched. This and the previous field are used only for the default firing function.

`ShootVec` The vector number along which the missile is launched (see "Shooting" chunk).

`Point` The point number within the object to act as an origin (Not used - set to 0).

`CurFrame` The current frame number within the sequence.

`TotFrame` The total number of frames within the sequence.

`NumPosCont` The number of positional control points defined within the sequence.

`NumRotCon` The number of rotational control points defined within the sequence. Note that these two values need not be the same, and that control points need not even be at the same key frame numbers.

`OldZoom` The zoom value stored when this viewpoint was selected. It is the value used if a -2 value is placed in the Zoom field in a rotational control point.

`VPLock` A set of 3 flags showing which rotation axes are locked to the view object and which are free. They are:

Bit 2	X axis locked
Bit 1	Y axis locked
Bit 0	Z axis locked

`Type` The type of control that the proportional device exerts over this object.

`PosCont` An array of positional control points.

Following this but not explicitly stated in the structure because of its variable position is `RotCont`, an array of rotational control points.

The whole chunk thus has the following structure:

`ChkType` The chunk type (`E_CTVIEWPOINT`).

`Length` The length of the chunk, including `ChkType` and itself.

`NumVPs` The number of defined viewpoints.

`View` A list of viewpoints as defined above. This is not a simple array because of the variable length of viewpoints. The `Length` field of each one gives the offset to the next.

Shape data

The shape data, like the world data, is simply a shape file loaded into memory. The buffer in which it resides is at `C_ShapeBuffer`, and is `C_ShapeBufLength` bytes long. Of this, `C_ShapeLen` bytes are actually occupied by shape data.

The shape data consists of a standard file header, followed by a list of shapes. Each shape is, in turn, a list of shape chunks (which are detailed below) terminated by a short value of `0xFFFF` as usual. The end of the entire list is signified by a single short of value `0xFFFF`.

Shapes have three major components: points, lines and facets.

Points define positions within the shape for the construction of lines, and come in two types. Relative points, the first type, are specified in terms of the overall size of the outer cube of the object. These change position if the object is scaled. The proportions of the x, y and z axes to be added to make up the point are specified as 1/16384ths. Thus a relative point with position (8192,8192,8192) is at the exact center of the object.

Geometric points are constructed as a position along a straight line between two existing points. The position on the line is a fraction whose denominator is a power of two. These points are much faster to process than relative points, at around 11% of a basic rotation calculation in the worst case. If the numerator of the fraction is 1, then the processing time becomes almost negligible. They are not affected by bending, however.

The first eight points (numbers 0–7) are always the eight outer points of the surrounding cuboid. They are defined as follows:

Point 0 - Origin
Point 1 - Origin + Z size vector
Point 2 - Origin + Y size vector
Point 3 - Origin + Y size vector + Z size vector
Point 4 - Origin + X size vector
Point 5 - Origin + X size vector + Z size vector
Point 6 - Origin + X size vector + Y size vector
Point 7 - Origin + X size vector + Y size vector + Z size vector

Points are joined in pairs by lines, which are clipped if necessary and assembled into facets.

Facets are visible from one side only, and are made from lines. The visible side is the one around which the lines are defined in anti-clockwise order. Each defined line may be used in either direction, removing redundant line definitions.

The chunks in the shape data are all members of the union `T_SHAPECHUNK`. Thus, a pointer to this type can be used for any chunk and the relevant element read according to its chunk type. The `ChkType` fields for each different type of chunk are all in the same place, so this can always be read by reading the chunk type from the points chunk element. The `T_SHAPECHUNK` union looks like this:

```
  typedef union
  {
     T_POINTSCHK  Pnt;
     T_LINECHK    Lin;
     T_FACETCHK   Fac;
     T_COLOURS    Col;
     T_LITCOLS    Lit;
     T_TEXTINFO   Tex;
     T_SHAPESIZE  Siz;
     T_SCL        SCL;
     T_ANIMCOLS   Acl;
     T_TEXTURES   Txr;           (Not used in VRT 5.50)
     T_TRANSLATE  Spx;           (Not used in VRT 5.50)
     T_NORMALS    Nor;
  }  T_SHAPECHUNK;
```

For example, you can access the number of facets from a facet chunk pointed to by `p` (which is of type `T_SHAPECHUNK *`), using:

```
Facets=p->Fac.NumFacets;
```

`Fac` specifies which type of chunk you wish to examine.

T_ANIMCOLS

Type: **E_SCANIMCOLS**

Union: **T_SHAPECHUNK.Acl**

Structure:

```
typedef struct
   {
   unsigned short   ChkType,Length;
   unsigned short   ColsCel,Cels;
   unsigned char    AnimCols[1];
   } T_ANIMCOLS;
```

Description: This chunk contains information about the coloring of the shape, dependant on the current cel of animation controller 1.

ChkType The chunk type (E_SCANIMCOLS).

Length The length of the chunk, including ChkType and itself.

ColsCel The number of colors defined per cel.

Cels The total number of animation cels for which colors are defined.

AnimCols An array of color indices which are used to color the facets of the shape. All the colors for the first cel are defined, then all the colors for the second cel, and so forth. The facet number is used to index into this array, such that facet n has color AnimCols[ColsCel*(Cel-1)+n-1]. This preserves the color chunk even if facets are deleted from the shape definition.

See also: T_COLOURS, T_LITCOLS

T_COLOURS

Type: **E_SCCOLOURS**

Union: **T_SHAPECHUNK.Col**

Structure:

```
typedef struct
   {
   unsigned short    ChkType,Length;
   unsigned char     Colour[1];
   } T_COLOURS;
```

Description: Contains information about the coloring of an shape, in the absence of a dynamic lighting model.

ChkType The chunk type (E_SCCOLOURS).

Length The length of the chunk, including ChkType and itself.

Colour An array of color indices which are used to color the facets of the shape. The facet number is used to index into this array, such that facet n has color Colour[n-1]. This preserves the color chunk even if facets are deleted from the shape definition. This chunk is only used at run time if a colors chunk does not exist on an object using this shape.

See also: T_ANIMCOLS, T_LITCOLS

T_FACETCHK
T_FACET

Type: **E_SCFACETS**

Union: **T_SHAPECHUNK.Fac**

Structure:

```
typedef struct
   {
   unsigned char     NumLines,FacAtt;
   unsigned short    Number,Line[1];
   } T_FACET;

   typedef struct
   {
   unsigned short    ChkType,Length;
   unsigned short    NumFacets;
   T_FACET           Facet[1];
   } T_FACETCHK;
```

Description: The facets chunk defines the actual facets which make up the shape. These are the only visible parts of the shape. Each facet is defined as follows:

NumLines The number of lines that make up this facet—4 for a quadrilateral, 3 for a triangle, 2 for a line, 1 for a point.

FacAtt A set of attribute flags defined as follows:

E_FACNODIR No direction check is performed—the facet is always visible.

E_FACRAWCOL Use the raw color without applying the lighting model to this facet.

E_FACHOLE This facet is a hole in the next one to be drawn.

E_FACSPECIAL This facet is something special. The number of lines defines in what way:

4: Mesh The first and second line numbers define the number of facets in both directions on a rectangular grid, N and M. The following N*M facets are part of the mesh and are sorted against each other. The mesh need not be any particular shape, as long as it contains N*M facets. A dummy facet is used to store the information.

E_FACINSIDE This facet is an inside facet and is drawn before any children of this object.

E_FACASSOC This facet is associated with the previous one and derives its visibility from it.

`E_FACMGROUP` This facet is associated with the previous one for purposes of sorting within a mesh. It always stays with the other facets in its “mesh group”.

`E_FACASSCOL` This facet is associated with the previous one in terms of lighting. It is lit by the same amount as the previous one.

`Number` A unique number identifying this facet, which is used when looking up the color from the color chunk. When creating a facet, the first free facet number is used.

`Line` An array of lines which make up the facet. They must join up in an anti-clockwise direction as seen from the visible side of the facet. A defined line may be used in reverse by adding `0x8000` to its line number.

The whole facet chunk is constructed as follows:

`ChkType` The chunk type (`E_SCFACETS`).

`Length` The length of the chunk, including `ChkType` and itself.

`NumFacets` The number of facets defined.

`Facet` A list of facets, defined as above.

T_LINECHK
T_LINE

Type: **E_SCLINES**

Union: **T_SHAPECHUNK.Lin**

Structure:

```
typedef struct
   {
   unsigned short    Start,End;
   } T_LINE;

   typedef struct
   {
   unsigned short    ChkType,Length;
   unsigned short    NumLines;
   T_LINE            Line[1];
   } T_LINECHK;
```

Description: Defines the edges used to construct polygonal facets. Each defined line has the following structure:

Start The point number of the start of the line.

End The point number of the end of the line.

The complete chunk is composed as follows:

ChkType The chunk type (E_SCLINES).

Length The length of the chunk, including ChkType and itself.

NumLines The number of defined lines in the chunk.

Line An array of line specifiers, one for each defined line.

Users do not see lines at all in the editors.

T_LITCOLS

Type: **E_SCLITCOLS**

Union: **T_SHAPECHUNK.Lit**

Structure:

```
typedef struct
   {
   unsigned short   ChkType,Length;
   unsigned char    LitCol[1];
   } T_LITCOLS;
```

Description: Contains information about the coloring of a shape, in the presence of a dynamic lighting model.

ChkType The chunk type (E_SCLITCOLS).

Length The length of the chunk, including ChkType and itself.

LitCol An array of color indices which are used to color the facets of the shape. The facet number is used to index into this array, such that facet n has color LitCol[n-1]. This preserves the color chunk even if facets are deleted from the shape definition. This chunk is only used at run time if a lit colors chunk does not exist on an object using this shape and a dynamic lighting mode is active.

See also: T_ANIMCOLS, T_COLOURS

T_NORMALS

Type: **E_SCNORMALS**

Union: **T_SHAPECHUNK.Nor**

Structure:

```
typedef struct
   {
   unsigned short    ChkType,Length;
   unsigned short    NumNormals, NumCels;
   unsigned short    ShapeNum,Checksum;
   T_VECTOR          Normals[1];
   } T_NORMALS;
```

Description: Defines the normal vectors for each point in each facet:

ChkType The chunk type (E_SCNORMALS).

Length The length of the chunk including ChkType and itself.

NumNormals The number of normals per animation cel of the shape.

NumCels The number of animation cels in the shape.

ShapeNum Not used.

Checksum A checksum of the points, lines and facets chunks of the shape used to ensure that the normals are valid.

Normals An array of normal vectors which are normalized and multiplied by 32767.

Note: Very complex shapes may have more normals that will fit in one chunk. In this case, more than one normals chunk is used and all but the last one is full (contains E_MAXNORMSPERCHUNK normals). If multiple normals chunks are used they must be contiguous.

T_POINTSCHK
T_ABSPOINT
T_GENPOINT
T_GEOMPOINT

Type: **`E_SCPOINTS`**

Union: **`T_SHAPECHUNK.Pnt`**

Structure:

```
typedef struct
   {
   short             x,y,z;
   } T_ABSPOINT;

   typedef struct
   {
   short             Point1,Point2;
   unsigned char     Shift,Mult;
   } T_GEOMPOINT;

   typedef union
   {
   T_ABSPOINT        Rel;
   T_GEOMPOINT       Geom;
   } T_GENPOINT;

   typedef struct
   {
   unsigned short    ChkType,Length;
   unsigned short    NumPoints,NumCels;
   } T_POINTSCHK;
```

Description: Defines the positions of the basic points that make up the vertices of the object.

Relative points have the following structure:

`x`, **`y`**, **`z`** The x, y and z coordinates of the point offset from the origin of the outer cube. They specify the fractions (in 16384ths) of the x, y and z axes to add to the origin to get the actual point position.

Geometric points are defined by:

`Point1`, **`Point2`** The point numbers (which must already have been defined) of the points between which the geometric point is constructed.

`Shift`, **`Mult`** Specify the fractional position along that line where the point is to be placed. The fraction is `Mult`/ ($2^{\texttt{Shift}}$).

These point types are bundled together in a union as a “general point”, which is six bytes long. A complete point definition is implicit and consists of a `Flags` word followed by an array of general points. The `Flags` word is defined as follows:

`E_PNTDYNAMIC` This point is a movable (dynamic) point. The array of general points contains as many points as there are cels in the shape definition (see below). If not, then only a single general point definition follows (a static point).

`E_PNTANIMMASK` Specifies a mask of 3 bits showing which animation controller this point is controlled by.

`E_PNTTYPEMASK` Specifies a mask of two bits containing the point type:

`E_PNTRELTYPE` Shows that the point is a relative point.

`E_PNTGEOMTYPE` Shows that the point is a geometric point.

These point types apply to all of the point positions following in a dynamic point - you cannot have a “mix and match” point that is geometric during one part of an animation and relative at other times.

The entire points chunk is set out as follows:

`ChkType` The chunk type (`E_SCPOINTS`).

`Length` The length of the chunk, including `ChkType` and itself.

`NumPoints` The number of defined points. This does not include the outer cube points, which are already generated by the system. Therefore, the first defined point is point 8.

`NumCels` The number of animation cels specified for the longest animation on the shape. Any dynamic points have this number of defined positions.

Following this, although not explicitly defined, is a list of complete point definitions, each of which consists of a Flags word and one or more point positions, as explained above.

T_SCL

Type: **E_CTSCL**

Union: **T_SHAPECHUNK.SCL**

Structure:

```
typedef struct
   {
   unsigned short    ChkType,Length;
   unsigned char     SCL[1];
   } T_SCL;
```

Description: This chunk specifies an SCL program to be executed to reconstruct this shape.

ChkType The chunk type (E_CTSCL).

Length The length of the chunk, including ChkType and itself.

SCL A set of object code bytes for an SCL program. These are executed when the Remake Shape function is selected in the Shape Editor. For a detailed description of the SCL language, see the “VRT User Guide: Chapter 16 - SCL” and the VRT online “Reference Books.”

T_SHAPESIZE

Type: **E_SCSIZE**

Union: **T_SHAPECHUNK.Siz**

Structure:

```
typedef struct
   {
   unsigned short    ChkType,Length;
   long              XSize,YSize,ZSize;
   } T_SHAPESIZE;
```

Description: This chunk contains a default size for this shape. It is used in the editors to decide what size to display a shape when editing it.

ChkType The chunk type (E_SCSIZE).

Length The length of the chunk, including ChkType and itself.

XSize, **YSize**, **ZSize** The x, y and z sizes of the bounding cube of this shape.

T_TEXTINFO

Type: **E_SCTEXT**

Union: **T_SHAPECHUNK.Tex**

Structure:

```
typedef struct
   {
   unsigned short    ChkType,Length;
   char              TextInfo[1];
   } T_TEXTINFO;
```

Description: This chunk contains some text relating to this shape. This text may be viewed in the editors. One suggested use is to include ordering information and part numbers in the shape itself.

`ChkType` The chunk type (`E_SCTEXT`).

`Length` The length of the chunk, including `ChkType` and itself.

`TextInfo` A zero-terminated ASCII text string. It may contain any non-zero character, and be up to approximately 32 KB long.

T_TEXTURES
T_TEXSPEC

Type: **E_SCTEXTURES**

Union: **T_SHAPECHUNK.Txr**

This structure is not used in VRT 5.50.

Note: If you import an object into VRT that has a shape with a textures chunk, the chunk is automatically converted to a texture coordinates chunk for the object. See T_TEXCOORDS for further details. If you import a shape with a texture chunk as a shape file, the shape will be displayed with the texture, but you will not be able to edit the texture.

T_TRANSLATE

Type: **E_SCSPRTRANS**

Union: **T_SHAPECHUNK.Spx**

This structure is not used in VRT 5.50. It is included for compatibility with earlier versions of VRT.

Symbol data

The symbol table stores editor-specific data that makes the user's life easier, but which plays no part in running of the world. It includes shape and object names, SCL comments and variable names. The symbol table format is common to all files, although certain chunks only make sense when attached to particular file types. The symbols are pointed to by the Symbols field in the file header, or if there are none, that field is zero.

The format of a symbol chunk is basically two lists of chunks. The first list is a list of global information which does not apply specifically to individual items. This is a list of chunks terminated by a `0xFFFF` as usual. Following immediately on from this is a set of chunk lists for each item, each ending with a `0xFFFF`. The final entry consists only of a single `0xFFFF`, marking the end of the list.

When loaded into memory, the symbols for the World and Shape files are pointed to by `C_WorldSym` and `C_ShapeSym` respectively.

The chunks in the symbol data are all members of the union `T_OBJSYM`. Thus, a pointer to this type can be used for any chunk and the relevant element read according to its chunk type. The `ChkType` fields for each different type of chunk are all in the same place, and can be read from the `ObjName` chunk element. The `T_OBJSYM` union looks like this:

```
typedef union
 {
   T_SYMNAME           ObjName;
   T_SYMNAME           ShpName;
   T_SYMVARNAMES       VarNames;
   T_SYMCOMMENTS       Comments;
   T_SYMLAYERNAMES     LayerNames;
   T_SYMSHAPESIZE      ShpSize;
   T_SYMNAME           SndName;
   T_SYMNAME           SprName;
   T_SYMVARNAMES       LocVars;
   T_SYMCOMMENTS       LocComm;
   T_SYMVARNAMES       GloVars;
   T_SYMCOMMENTS       GloComm;
 } T_OBJSYM;
```

T_SYMCOMMENT
T_SYMCOMMENTS

Type: **E_SYMCOMMENTS**
E_SYMGLOCOMM
E_SYMLOCCOMM

Union: **T_OBJSYM.Comments**
T_OBJSYM.GloComm
T_OBJSYM.LocComm

Structure:

```
typedef struct
   {
   short            Offset;
   char             Text[1];
   } T_SYMCOMMENT;

   typedef struct
   {
   short            ChkType,Length;
   T_SYMCOMMENT     Comment[1];
   } T_SYMCOMMENTS;
```

Description: Stores comments from an SCL program (normal, globally or locally triggered) attached to an object. Each comment has the following structure:

`Offset` The offset, in bytes, from the start of the SCL object code where this comment is met, or `0x7FFF` if there is no following text string.

`Text` The actual text of the comment, terminated by a zero.

The entire chunk consists of the following:

`ChkType` The chunk type (`E_SYMCOMMENTS`, `E_SYMGLOCOMM` or `E_SYMLOCCOMM`).

`Length` The length of the chunk, including `ChkType` and itself.

`Comment` A list of comments as defined above, terminated by an offset value of `0x7FFF`.

T_SYMNAME

Type: **E_SYMOBJNAME**
E_SYMSHPNAME
E_SYMSNDNAME
E_SYMSPRNAME

Union: **T_OBJSYM.ObjName**
T_OBJSYM.ShpName
T_OBJSYM.SndName
T_OBJSYM.SprName

Structure:

```
typedef struct
   {
   short            ChkType,Length;
   short            Number;
   char             Name[32];
   } T_SYMNAME;
```

Description: Defines the name and number of an object, shape, sound or image (sprite). It must be the first chunk of a list of symbol chunks for that item, since it defines the number to which they are attached. It is defined as follows:

ChkType The chunk type (E_SYMOBJNAME, E_SYMSHPNAME, E_SYMSNDNAME or E_SYMSPRNAME).

Length The length of the chunk, including ChkType and itself.

Number The item number for which this is the name.

Name A zero terminated ASCII string defining the name of the item.

T_SYMSHAPESIZE

Type: **E_SYMSHAPESIZE**

Union: **T_OBJSYM.ShpSize**

Structure:

```
typedef struct
   {
   unsigned short    ChkType,Length;
   long              XSize, YSize, ZSize;
   } T_SYMVARNAMES;
```

Description: Defines the size of a shape’s bounding cube:

ChkType The chunk type (`E_SYMSHAPESIZE`).

Length The length of the chunk, `ChkType` and itself.

XSize, YSize, ZSize The size of the shapes bounding cube.

T_SYMLAYERNAMES
T_SYMLYRSPEC

Type: **E_SYMLAYERNAMES**
E_SYMLYRSPEC

Union: **T_OBJSYM.LayerNames**

Structure:

```
typedef struct
   {
   unsigned char     Number;
   char              Name[34];
   } T_SYMLYRSPEC;

   typedef struct
   {
   unsigned short    ChkType,Length;
   short             NumLayers;
   T_SYMLYRSPEC      Layer[1];
   } T_SYMLAYERNAMES;
```

Description: Defines the names of the layers used in the world. It should be the only chunk in the global symbols list. It is defined as follows:

Number The number of the layer, 0 to 255 in each world.

Name An ASCII string defining the number of the layer, up to 8 characters long (plus the zero termination character).

The whole chunk is made up as follows:

ChkType The chunk type (`E_SYMLAYERNAMES`, `E_SYMLYRSPEC`).

Length The length of the chunk, `ChkType` and itself.

NumLayers The number of defined layers.

Layer An array of layer number specifiers, as above.

T_SYMVARSPEC
T_SYMVARNAMES

Type: **E_SYMVARNAMES**
E_SYMGLOVARS
E_SYMLOCVARS

Union: **T_OBJSYM.VarNames**
T_OBJSYM.GloVars
T_OBJSYM.LocVars

Structure:

```
typedef struct
   {
   unsigned char     Number;
   char              Name[9];
   } T_SYMVARSPEC;

   typedef struct
   {
   short             ChkType,Length;
   short             NumVars;
   T_SYMVARSPEC      Var[1];
   } T_SYMVARNAMES;
```

Description: Defines the names of any variable defined in an SCL program (normal, globally or locally triggered) attached to the object. Each variable is defined as:

Number The number of the variable, 0 to 255. This imposes a limit of 256 named variables per object.

Name An ASCII string defining the name of the variable, up to 8 characters long (plus the zero termination character).

The whole chunk is made up as follows:

ChkType The chunk type (E_SYMVARNAMES, E_SYMGLOVARS, E_SYMLOCVARS).

Length The length of the chunk, ChkType and itself.

NumVars The number of defined variables.

Var An array of variable name specifiers, as above.

Palette data

The palette data is simply a palette file loaded into memory. The buffer into which it is loaded is at `C_PalBuffer`, which has length `C_PalBufLength`, of which `C_PalLen` bytes are actually occupied by palette data.

The Palette file consists of a standard file header, followed by three types of information defining the colors to use in the world, as follows:

T_PALETTE

Type: **`E_PCPALETTE`**

Structure:

```
struct
   {
   unsigned short    ChkType,Length;
   unsigned char     Palette[0x300];
   } T_PALETTE;
```

Description: Contains information about the actual hardware palette to use, specifying the possible colors for any one pixel on the screen.

`ChkType` The chunk type (`E_PCPALETTE`)

`Length` The length of the chunk, including `ChkType` and itself.

`Palette` An array of 256 triplets of R,G,B palette information. These are normalized so that a value of `0xFF` is the maximum intensity for that color component, and 0 is the minimum, regardless of the actual display device used.

The first sixteen colors are defined as system colors and should not be changed by the user since they are used for system resources such as dialog boxes and text messages. Only colors 0 (transparent) and 1 (black) should be used in the world.

The first sixteen colors are defined as follows:

No	Name	RGB values
0	E_COLTRANSPARENT	0x00 0x00 0x00
1	E_COLBLACK	0x00 0x00 0x00
2	E_COLBLUE	0x00 0x00 0xA7
3	E_COLRED	0xAB 0x00 0x00
4	E_COLGREEN	0x00 0x7F 0x00
5	E_COLMAGENTA	0xAB 0x00 0xAB
6	E_COLYELLOW	0xA3 0x00 0xA7

7	E_COLGREY	0xAA	0xAA	0xAA
8	E_COLDKGREY	0x55	0x55	0x55
9	E_COLLTBLUE	0x00	0xBB	0xFF
10	E_COLBRBLUE	0x00	0x00	0xFF
11	E_COLBRRED	0xFF	0x00	0x00
12	E_COLBRGREEN	0x00	0xFF	0x00
13	E_COLLTGREY	0xCC	0xCC	0xCC
14	E_COLBRYELLOW	0xFF	0xFF	0x00
15	E_COLWHITE	0xFF	0xFF	0xFF

(Color 0 is transparent)

Dialog boxes, messages, and system resources should only use these colors, since none of the other colors can be guaranteed to be visible.

T_RANGES

Type: **E_PCRANGES**

Structure:

```
struct
   {
   unsigned short    ChkType,Length;
   unsigned char     Ranges[0x100];
   } T_RANGES;
```

Description: Contains information about the grouping of colors into ranges.

ChkType The chunk type (E_PCRANGES)

Length The length of the chunk, including ChkType and itself.

Ranges An array of 256 range lengths for lighting. The number stored is the length of the range left. Ranges are used in the lighting model to decide what color a facet should be. If lighting is not active in the world, the color range can be used to set the brightness level of a texture according to the position of the facets color in the range using the E_TCLIGHT flag in the T_TEXCOORDS chunk. A range should consist of a smooth progression of lighter to darker colors.

T_STIPPLES

Type: **`E_PCSTIPPLES`**

Structure:

```
struct
   {
   unsigned short    ChkType,Length;
   unsigned char     Stipples[0x800];
   } T_STIPPLES;
```

Description: Contains information about the pixel colors used when coloring facets.

`ChkType` The chunk type (`E_PCSTIPPLES`)

`Length` The length of the chunk, including `ChkType` and itself.

`Stipples` An array of octets of data. The first four bytes of an octet give the palette colors to use in the following positions in the stipple:

```
0 1

2 3
```

The second four give the alpha (transparency) values to use in these positions. These are again normalized so that `0` is completely opaque, and `0xFF` is completely transparent.

Stipple 0 should be completely transparent, and stipples 1–15 should be opaque solid versions of the 15 system colors. Currently only `0x00` and `0xFF` are valid values for the transparency information.

Sound data

The sound data is a sound file loaded into memory. It resides in a buffer at `C_SoundBuffer`, and is `C_SoundBufLength` bytes long. Of this, `C_SoundLen` bytes are actually occupied by sound data. The first sound in the list is sound 1, then sound 2 and so on.

The sound data consists of a standard file header, followed by a list of sounds.

Following the sounds are a set of symbols. Each symbol list consists of a list of symbols chunks, terminated by `0xFFFF`. The first list is any global symbol information, then there is a symbols list for each sound with symbols, terminated by an `0xFFFF`. The end of this list of lists is itself marked by another `0xFFFF`.

The end of the entire list is signified by a single short of value `0xFFFF`.

T_SOUNDREC

Type: **`E_STSAM8`**
`E_STSAM16`

Structure:

```
struct
   {
   unsigned short    Type;
   long              Length;
   unsigned char     Pitch, Spare;
   long              Flags;
   unsigned char     Data[1]
   }                 T_SOUNDREC;
```

Description: Contains information about the sounds.

`Type` The chunk type:

`E_STSAM8` 8-bit PCM

`E_STSAM16` 16-bit PCM

`Length` The length of the entire chunk, in bytes.

`Pitch` The sample pitch as a MIDI note number.

`Spare` Not used.

`Flags` Set the channel on which the sound is output:

`E_SFRIGHT` Right channel

`E_SFLEFT` Left channel

`Data` The start of the actual sound data. These are 8-bit unsigned samples, or

16-bit signed values -32768 to +32767.

Message data

The message data is simply a message file loaded into memory. The buffer in which the system messages reside is at `C_SysMessBuffer`, and is `C_SysMessBufLength` bytes long. Of this, `C_SysMessLen` bytes are actually occupied by message data.

The user messages reside at `C_MessBuffer`, and are `C_MessBufLength` bytes long. Of these, `C_MessLen` bytes are actually occupied by message data.

The message data consists of a standard file header, followed by a list of messages. The end of the entire list is signified by a single short of value `0xFFFF`.

T_MESSAGE

Structure:

```
struct
{
unsigned short    MsgNum;
unsigned short    Length;
char              Text[];
} T_MESSAGE;
```

Description: Contains information about the messages.

`MsgNum` The number of the message, from 0.

`Length` The length of the entire chunk, in bytes.

`Text` The start of the actual text data. The message as a whole is a zero-terminated ASCII string.

Font data

The font data is simply a font file loaded into memory. The buffer in which the font information reside is at `C_FontBuffer`, and is `C_FontBufLength` bytes long. Of this, `C_FontLen` bytes are actually occupied by font data.

The font data consists of a standard file header, followed by a list of fonts. The end of the entire list is signified by a single short of value `0xFFFF`.

T_FONTINFO

Structure:

```
struct
{
unsigned short   FontID;
unsigned short   Length;
unsigned short   Width;
unsigned short   Height;
unsigned char    FontData[];
} T_FONTINFO;
```

Description: Contains information about the font.

`FontID` The ID number of the font, from 0.

`Length` The length of the entire chunk, in bytes.

`Width`, **`Height`** The size of a single character in the font. All fonts are monospaced.

`FontData` The actual data defining the font. This consists of a series of bytes into which each character is formatted. A set bit indicates foreground color, a reset bit indicates background color.

For example the letter ‘A’ in 6x6 font:

```
.###..     01110000      0x70
#...#.     10001000      0x88
#####.     11111000      0xF8
#...#.     10001000      0x88
#...#.     10001000      0x88
......     00000000      0x00
```

The significant data are padded to the right so that they occupy the most significant bits of each byte. For widths of more than 8 bits, the line of data is padded to a multiple of 8 bits on the right, and split into bytes in sequence.

For example the letter ‘A’ in a 10x10 font:

```
...###....    0001110000000000    0x1c 0x00
..##.##...    0011011000000000    0x36 0x00
.##...##..    0110001100000000    0x63 0x00
##.....##.    1100000110000000    0xC1 0x80
#########.    1111111110000000    0xFF 0x80
##.....##.    1100000110000000    0xC1 0x80
##.....##.    1100000110000000    0xC1 0x80
##.....##.    1100000110000000    0xC1 0x80
##.....##.    1100000110000000    0xC1 0x80
..........    0000000000000000    0x00 0x00
```

Image data

The image (sprite) data is simply an image file loaded into memory. The buffer in which the system images reside is at C_SpriteBuffer, and is C_SpriteBufLength bytes long. Of this, C_SpriteLen bytes are actually occupied by image data. The user sprites use the corresponding C_UserSprBuffer, C_UserSprBufLength, and C_UserSprLen variables.

The image file consists of a standard file header, followed by a single short value containing the number of images. Following this is an array of T_SPRITEINFO structures, one for each defined image. Immediately following this is the image data itself.

T_GRSPRSINGLE

Structure:

```
typedef struct
{
    short     Sprite;
    short     Flags;
} T_GRSPRSINGLE;
```

Description: Passes data to GrSetOneSprite.

Sprite Number of sprite to update.

Flags A set of flags:

E_SPRUSER - Set if sprite is a user sprite, reset if a system sprite.

T_SPRITEINFO

Structure:

```
struct
{
unsigned short    Width;
unsigned short    Height;
long              Offset;
} T_SPRITEINFO;
```

Description: This chunk contains information about the images.

`Width`

Bits 0–13 Define the width of the image in pixels. This sub-field can be isolated by anding the `Width` value with the mask `E_SPRMASK`.

Bit 14 When set, indicates that the image has hotspot data in the form `T_POINTREC2D` immediately following the image data.

`E_SPRHOTSPOT`	Hotspot

Bit 15 When set, indicates that the screen area under the image will not be saved when the image is drawn.

`E_SPRNOSAVE`	No screen save

`Height`

Bits 0–13 Define the height of the image in pixels. This subfield can be isolated by adding the `Height` value with the mask `E_SPRMASK`.

Bit 14 Reserved

Bit 15 When set, indicates that the image has palette data in the form `unsigned char [0x300]` (see `T_PALETTE` for further information), immediately following the image hot spot if there is one or the image data if not.

`E_SPRPALETTE`	Palette

`Offset` The offset from the start of the image file at which the image data is stored. The image data is one byte per pixel, consisting of `Height` rows of `Width` columns each.

When a sprite file is loaded, `Offset` is converted from an offset from the start of the file, to a pointer to the sprite data.

Printer driver data

The printer driver data is simply a printer driver file loaded into memory. The buffer in which the driver resides is at `C_PrinterBuffer`, and its length is `C_PrinterBufLength` bytes.

The printer file consists of a standard file header, followed by a `T_PRINTER` structure defining the printer. Immediately following this is miscellaneous data pointed to by the `T_PRINTER` structure.

T_PRINTER

Structure:

```
struct
{
char                Name[20];
unsigned short      width,height;
unsigned short      pagewidth, pageheight;
unsigned short      dpix,dpiy;
unsigned short      pinsx,pinsy;
char                *Spare1,*Spare2;
long                Code,Init,LF,LData,Exit;
char                LOffset,LSize,LFlags;
}                   T_PRINTER;
```

Description: Contains information about the printer.

`Name` The name of the printer as displayed on the pick list.

`width`, **`height`** Filled in by the VRT and define the width and height of the screen, in pixels.

`pagewidth`, **`pageheight`** The size of the page to print, in $^1/_{16}$ths of an inch.

`dpix`, **`dpiy`** The resolutions of the printer (in dots per inch) across the page and down the page respectively.

`pinsx`, **`pinsy`** The number of ‘pins’ on the print head across and down respectively.

Since laser printers use a byte-wide method of printing, this gives pinsx=8, pinsy=1. A 24 pin dot matrix printer would give pinsx=1, pinsy=24.

`Spare1`, **`Spare2`** Not used.

`Code` An offset (measured from the start of the file) to some user executable code. If non-zero, this code is called instead of the inbuilt generic printer driver routine. It is provided for driving exotic printers. The code must be completely (intrinsically) relocatable.

`Init` An offset to the start of the initialization string for the printer. The first byte of the string is a length byte, followed by the data to actually send.

`LF` An offset to the start of the linefeed string for the printer. The first byte of the string is a length byte, followed by the data to actually send.

`LData` An offset to the start of the data header string for the printer. This is normally output at the beginning of each line of data sent to the printer. The first byte of the string is a length byte, followed by the data to actually send.

`Exit` An offset to the start of the exit string for the printer. The first byte of the string is a length byte, followed by the data to actually send.

`LOffset` An offset (measured from the length byte of the data header string) of where the length of data is to be put.

`LSize` The number of bytes of length data to be included into the data header string.

`LFlags` A set of flags as follows:

> `E_TOFILE` Print is redirected to a file. The name of the file is derived from the VRT file name.
>
> `E_TIFF` Send data in TIFF run-length encoded format.
>
> `E_ASCII` Data length is in ASCII decimal, not binary.

Configuration data

The configuration file consists of a standard file header, followed by a list of chunks defining various information about the configuration of the system.

There are, in fact, two configuration files. The first is the default configuration, or preferences file. This is called VRT.CFG and is found in the default VRT directory. The second is the actual configuration file, which resides in the VRT file and takes precedence over the preferences file in cases of conflict.

The preferences file should contain things which are common to all the worlds on the system, such as keyboard layouts, device driver and system resource filenames.

The configuration file should contain specific information, such as the filenames of the world and shape files to use in the world, any changes to the default key maps.

Each chunk consists of a short ChunkType field, containing the chunk type, and a Length field, which is the length of the whole chunk. The list is terminated by an `0xFFFF`, as usual.

The chunk definitions shown below do not include the `ChunkType` and `Length` fields.

Unless specifically mentioned, only one of each type of chunk may be defined in a configuration file.

The configuration file is loaded into memory at `C_ConfigBuffer`, which has length `C_ConfigBufLen`, of which `C_ConfigLen` bytes are actually occupied.

The preferences file is loaded into memory at `C_PrefsBuffer`, which has length `C_PrefsBufLen`, of which `C_PrefsLen` bytes are actually occupied.

T_CONSOLE

Type: **E_CCCONSOLE**

Structure:

```
typedef struct
   {
   unsigned short    XPos,YPos,Width,Height,Type,Index;
   T_LONGORPTR       PrevInsVal,PrevInsVal2;
   } T_INSTRUMENT;

   typedef struct
   {
   unsigned short    WindWidth,WindHeight,WindXMid,WindYMid;
   unsigned short    WindXMin,WindXMax,WindYMin,WindYMax;
   T_POINTREC2D      Corners[8];
   short             XScale,YScale,ZScale;
   long              Vx,Vy,Vz,VxOff,VyOff,VzOff;
   short             Vrx,Vry,Vrz,VrxOff,VryOff,VrzOff;
   unsigned char     VPLock;
   short             Flags;
   short             BackDrop,IconTable,FuncTable,StartVP,
                     GRDevNumber,CurrentVP,NextConsole;
   unsigned short    NumInstr;
   unsigned short    IWindWidth,IWindHeight,
                     IWindXMid,IWindYMid;
   short             RTBackDrop;
   }                 T_CONSOLE;
```

Description: This chunk defines the appearance and position of a window onto the world, and the instruments associated with it. A complete console consists of several of these chunks. The first window in each console must have its `E_CONSMASTER` flag set (see Flags, below). Up to `E_MAXCONS` console chunks may be defined.

Each instrument is defined as follows:

XPos, **YPos** The position, in pixels, of the top left of a rectangle defining the extent of the instrument.

Width, **Height** The width and height of the rectangle defining the extent of the instrument. If the instrument is text-based, this is in characters, otherwise it is in pixels.

Type The type of instrument to display:

0:	Decimal number
1:	Hex number
2:	Binary number
3:	Text string
4:	Image

`5`:	Horizontal Bar Left to Right
`6`:	Horizontal Bar Right to Left
`7`:	Vertical Bar Bottom to Top
`8`:	Vertical Bar Top to Bottom
`9`:	Dial
`10`:	User function

`Index` An index into an array of instrument values which are updated by SCL programs.

`PrevInsVal`, **`PrevInsVal2`** The previous values of the primary and secondary instrument values. The instruments are only redrawn if these change.

The window as a whole is defined as follows:

`WindWidth`, **`WindHeight`** Half the width and height of the window onto the world, in pixels.

`WindXMid`, **`WindYMid`** The x and y coordinates of the center of the window.

`WindXMin`, **`WindXMax`**, **`WindYMin`**, **`WindYMax`** The minimum and maximum x and y values for the window, which are filled in at startup.

`Corners` An array of points marking the corners of the window, used in clipping. It too is filled in at startup.

`XScale`, **`YScale`**, **`ZScale`** The x,y and z scaling factors for the viewing transformation. XScale and YScale can be worked out from the window size at startup, if required.

`Vx`, **`Vy`**, **`Vz`** The current position in the world of the viewpoint, calculated from the current position of the object to which the viewpoint is attached, and the offsets `VxOff`, `VyOff`, `VzOff`.

`VxOff`, **`VyOff`**, **`VzOff`** The current offset from the viewpoint object of the viewpoint.

`Vrx`, **`Vry`**, **`Vrz`** The current absolute rotation of the viewpoint, calculated from the current rotation of the object to which the viewpoint is attached and the rotation offsets `VrxOff`, `VryOff`, `VrzOff`.

`VrxOff`, **`VryOff`**, **`VrzOff`** The current rotation offset of the viewpoint from the viewpoint object.

`VPLock` A set of 3 flags showing which rotation axes are locked to the view object and which are free. They are:

Bit 2	X axis locked
Bit 1	Y axis locked
Bit 0	Z axis locked

`Flags` A set of flags as follows:

`E_CONSFULLSCR` If set, the position and size of the window are ignored and it is set to full screen size.

`E_CONSAUTOSCALE` If set, the X and Y scale of the window are calculated at startup to give a square aspect ratio.

`E_CONSDISABLED` If set, the window is not displayed.

`E_CONSDEFDISABL` The default state of the above flag.

`E_CONSOUTLINE` If set, the window is drawn with a single pixel black outline.

`E_CONSMASTER` If set, this window is the first window on a whole new screen layout. Any windows immediately following with this bit unset are drawn with it on the same screen. The Backdrop, IconTable, and FuncTable fields only apply to a window with this flag set.

`E_CONSRELBOTTOM` Bottom edge of the window is relative to the bottom of the screen.

`E_CONSRELLEFT` Left side of the window is relative to the left edge of the screen.

`E_CONSRELRIGHT` Right edge of the window is relative to the right side of the screen.

`E_CONSRELTOP` Top edge of the window is relative to the top of the screen.

`E_CONSTEXTURE` If set, redirects the drawing of the window into a user image. The image number used is specified as the contents of the IconTable field +1.

`BackDrop` The number of the defined foredrop to go in front of the console. Only makes sense for the master window (`E_CONSMASTER` set in Flags).

`IconTable` The number of the icon table to overlay over the console defining the active areas of the screen. Only makes sense for the master window (`E_CONSMASTER` set in Flags).

`FuncTable` The number of the keyboard function table to use for this console. Only makes sense for the master window (`E_CONSMASTER` set in Flags).

`StartVP` The initial viewpoint number to use in this window.

`GRDevNumber` The graphics device number to use to display this window (default to 0).

`CurrentVP` The viewpoint number currently being displayed in this window.

`NextConsole` Allows several windows onto the world to be defined. This is the number of the next console to display over this one, or -1 if none. Beware of closed loops.

`NumInstr` The number of instruments defined for this window.

`IWindWidth`, **`IWindHeight`**, **`IWindXMid`**, **`IWindYMid`** The initial values for the size and position of the window onto the world.

`RTBackdrop` The number of the backdrop to display within the window, or -1 if none.

Immediately following this, but not explicitly defined, is an array of instrument definitions, as above.

T_EDITCONFIG

Type: **E_CCEDITCONFIG**

Structure:

```
typedef struct
   {
   unsigned char    RelPtCol,GeomPtCol,BoundPtCol,
                    BoundCol,XAxisCol,YAxisCol,
                    ZAxisCol,BackgndCol,CollCubeCol,
                    SortCubeCol;
   short            GeomShift,MovStep;
   long             ShpOpts;
   char             UpdateFlags,DecompileNum;
   unsigned char    NextCol,SelPntCol,SelFacCol,
                    FacNumCol,PntNumCol,OXYZCol,
                    ConstCol,SelectCol,InvisCol;
   short            EditorView;
   char             Spare[91];
   } T_EDITCONFIG;
```

Description: This chunk specifies preferences information for the editors. It is usually only found in the preferences file.

RelPtCol, **GeomPtCol**, **BoundPtCol** The colors to use for relative, geometric and bounding cube points respectively in the Shape Editor.

BoundCol The color to use to display the bounding cube.

XAxisCol, **YAxisCol**, **ZAxisCol** The colors used to display the axes of the bounding cube in the Shape Editor.

BackgndCol The color to use to clear the screen in the Shape Editor.

CollCubeCol, **SortCubeCol** The colors to use to display the collision and sorting cuboids.

GeomShift, **MoveStep** The amounts by which to move geometric and relative points in the Shape Editor (geometrics move by $1/2^{geomshift}$).

UpdateFlags The display options flags used in the World Editor.

DecompileNum Non-zero if SCL is to be decompiled using object numbers instead of names, or zero if names are to be used where possible.

NextCol The color to use when creating the next facet in the Shape Editor.

SelPntCol The color to use to display selected points in the Shape Editor.

SelFacCol The color to use to outline selected facets in the Shape Editor.

`FacNumCol` The color to use for facet numbers in the Shape Editor.

`PntNumCol` The color to use for point numbers in the Shape Editor.

`OXYZCol` The color to use to display the axis labels (O, X, Y and Z) in the Shape Editor.

`ConstCol` The color used to display construction lines for geometric points in the Shape Editor.

`SelectCol`, **`InvisCol`** The colors used to display the selection and invisible object cuboids.

`EditorView` The current active view in the World and Shape Editors. It can take one of the following values:

`E_VMPLAN`	Plan view
`E_VMNORTH`	North elevation
`E_VMSOUTH`	South elevation
`E_VMEAST`	East elevation
`E_VMWEST`	West elevation
`E_VMPERSP`	Perspective view
`E_VMUNDERSIDE`	Underside view

`Spare` 91 bytes of spare space, reserved for future expansion, and should be set to 0.

T_EXTRACONFIG

Type: **E_CCEXTRACONFIG**

Structure:

```
typedef struct
   {
   long              Flags;
   float             CosSmoothAngle,PlaneError,InsideError;
   long              ZBias,NearClip,FarClip;
   long              NumLights;
   T_LIGHTDEF        *LightList;
   long              FogStart,IFogStart;
   long              FogEnd,IFogEnd;
   long              FogDensity,IFogDensity;
   unsigned char     FogR,FogG,FogB,FogA;
   unsigned char     IFogR,IFogG,IFogB,IFogA;
   unsigned char     AmbientR,AmbientG,AmbientB,AmbientA;
   unsigned char     IAmbientR,IAmbientG,IAmbientB,
                     IAmbientA;
   short             SelectedObj;
   char              Spare[20];
   } T_EXTRACONFIG;
```

Description: Specifies configuration for the Superscape graphics device and 3D graphics cards with Direct3D support.

Flags Flags defining advanced rendering states:

E_EXCFGZBUFAUTO Automatic, will Z buffer intersecting objects, meshed objects and attached objects.

E_EXCFGZBUFFULL Full Z buffering with presort (sometimes required with transparent objects).

E_EXCFGZBUFNOSORT Full Z buffering without presort.

E_EXCFGZBUFOFF No Z buffering.

E_EXCFGLITCOLGUESS Renderer will choose color to light facet based on the range the actual facet color is in. Only applies if object has no lit colors and E_EXCFGLITCOLONLY is not set.

E_EXCFGLITCOLONLY Renderer will only light objects with lit colors.

E_EXCFGNODITHER Dithering is off.

E_EXCFGNOTEXPERSP Texture perspective correction is off.

E_EXCFGTEXLINEAR Texture filtering is set to linear.

E_EXCCFGFOGOFF Fog is off.

`E_EXCFGFOGLINEAR` Fog is on.

`E_EXCFGFOGEXP` Not used.

`E_EXCFGFOGEXP2` Not used.

`CosSmoothAngle` The cosine of the maximum angle between facets for smooth-shading to be applied. Used when generating normals.

`PlaneError,InsideError,ZBias` Reserved for future enhancements.

`NearClip,FarClip` Near and far clipping planes. Only objects between NearClip and FarClip units from the viewpoint will be displayed. The closer the two values, the more accurate Z buffering will be. NearClip must be greater than 0.

`NumLights` Used internally.

`LightList` Used internally.

`FogStart`, **`IFogStart`** The current and initial distance at which fog starts from the current viewpoint in Superscape units. Fog uses a linear calculation.

`FogEnd`, **`IFogEnd`** The current and initial distance at which fog ends from the current viewpoint in Superscape units. At this points objects completely disappear into the.

`FogDensity`, **`IFogDensity`** Not used.

`FogR`, **`FogG`**, **`FogB`**, **`FogA`**, **`IFogR`**, **`IFogG`**, **`IFogB`**, **`IFogA`** The current and initial RGB values for fog.

`AmbientR`, **`AmbientG`**, **`AmbientB`**, **`AmbientA`**, **`IAmbientR`**, **`IAmbientG`**, **`IAmbientB`**, **`IAmbientA`** The current and initial RGB values for ambinet light.

`SelectedObj` Internal use only.

`Spare[20]` Not used.

T_FILENAME

Type:

```
E_CCADDDEVICES
E_CCAPPNAME
E_CCAPPNAMEDOS
E_CCAPPNAMEWIN
E_CCBACKDROP
E_CCDEVICE
E_CCFONTFILE
E_CCMESSAGE
E_CCPALETTE
E_CCRESOURCE
E_CCSHAPE
E_CCSNDNAME
E_CCSPRITES
E_CCSYSMESSAGE
E_CCUSERRESOURCE
E_CCUSERSPRITES
E_CCWORLD
```

Structure:

```
struct
   {
   char              Name[E_MAXPATHLEN];
   } T_FILENAME;
```

Description: Define the names of the files to use in the world, respectively:

Additional device drivers
SDK Applications
Backdrop
Device drivers
Font
Messages
Palette
System resources
Shapes
Sounds
System images (sprites)
System messages
User resource
User images (sprites)
World files.

`Name` Contains the zero terminated name of the file to load.

T_GENERAL

Type: **E_CCGENERAL**

Structure:

```
typedef struct
{
   unsigned short   TimerA, TimerB, TimerC, TimerD;
   unsigned char    Ambient;
   unsigned short   KDel,KRep;
   short            Detail;
   unsigned char    Resolution;
   unsigned char    BufClr;
   long             DrawSize;
   unsigned char    BufClrCol;
   char             TextureScale;
   long             Flags;
   unsigned short   StepSize;
   short            AngleSize;
   short            Spare2;
   long             UndoSize;
   short            BackDrop;
   short            StepSize2;
   unsigned short   AbsTime;
   unsigned short   FPSLimit;
   unsigned short   PointSize;
   unsigned short   TimeScale;
   unsigned char    Spare[6];
} T_GENERAL;
```

Description: Defines general information about the system set up.

TimerA, **TimerB**, **TimerC**, **TimerD** The delays (in milliseconds) between subsequent triggers by the SCL programmable timers, or 0 if not used.

Ambient The amount of ambient light present in the environment, from 0 (none) to 255 (100% lighting from ambient light alone).

KDel, **KRep** The delay and repeat times for the keyboard, measured in 1/100s intervals.

Detail The detail level setting, which alters the distancing algorithm to display or hide varying levels of detail.

Resolution The initial resolution to use when displaying the world. 0 is the default resolution.

BufClr A flag indicating the current screen buffer clear state:

0 Clear screen, draw horizon.

1	Do not clear screen between frames.
2	Clear screen to solid color.
3	Do not clear screen between frames.
4	Draw backdrop behind screen, if present.
5	Do not clear screen between frames.

Other values are undefined.

`DrawSize` The amount of memory to allocate for the drawing information buffer.

`BufClrCol` The color to clear the screen with when `BufClr` is 2 (see above).

`TextureScale` The default resolution for the textures. This may be:

`E_TEXNONE`	No textures
`E_TEX4X4`	}
`E_TEX3X3`	} Not used. See `E_TEX1X1`.
`E_TEX2X2`	}
`E_TEX1X1`	Textures displayed using 1x1 pixels.

`Flags` A longword reserved for various flags. When set the following conditions apply:

`E_MMABSTIME` The world should be run in 'absolute time', a certain number of events per second, as opposed to a frame-by-frame basis.

`E_MMCROSSHAIR` Center crosshair is on.

`E_MMINTERLACE` The top and bottom halves of the display are interlaced together.

`E_MMLIMITFPS` The frame rate limit (as defined by `FPSLimit`) is applied.

`E_MMMOUSEMOVE` Mouse movement is set (and cannot be set by the user).

`E_MMNOSHUTUP` Sounds are not stopped during a palette change.

`E_MMOUTERCUBE`, `E_MMSORTCUBE`, `E_MMCOLLCUBE`, `E_MMGROUP` Outer (bounding) cubes, sorting cuboids, collision cuboids and groups are displayed.

`E_MMPAUSERR` Pauses on SCL run-time errors.

`E_MMSHADEONCE` The world is lit once at startup time, and the resultant colors are assigned to the current colors.

`E_MMSHADEVERY` The world is lit every frame.

`E_MMSHOWSEL` Selected objects are displayed.

`E_MMSHOWINV` Invisible objects are displayed.

`E_MMTIMINGS` The Timings dialog box is displayed.

`E_MMTRANSPFAC` Transparent facets are processed and can be clicked on.

`E_MMUSECROSS` The center crosshair is used for activation instead of the mouse.

`E_MMWORLDAXES` World axes are displayed.

`E_MMZOOMHOR` The horizon bands expand when zooming in.

`StepSize` The low 16 bits of the step size by which to move an object or viewpoint when using the keyboard. The top 16 bits are in `StepSize2`.

`AngleSize` The size of the angle by which to turn an object or viewpoint using the keyboard.

`Spare2` Not used.

`UndoSize` Specifies the size, in bytes, of the Undo buffer.

`Backdrop` Specifies the number of the backdrop to display behind the main menu, or -1 if none.

`StepSize2` The top 16 bits of the step size by which to move an object or viewpoint when using the keyboard. The low 16 bits are in `StepSize`.

`AbsTime` The time, in milliseconds, between subsequent world updates when the `AbsTime` flag is set (see `Flags`, above). If this is set to less time than it takes to update the world, the results are unpredictable, and may cause the computer to lock up.

`FPSLimit` Sets an upper limit to the speed at which frames are displayed. This is the minimum time, in milliseconds, between subsequent display frames.

`PointSize` The number of bytes taken for a point representation in the drawing list. At present this is just set to 4.

`TimeScale` The rate at which the time and monotronic time (SCL command `vrtime`) pass multiplied by 256. For example, 512 equals twice as fast as the real world, and 128 equals half as fast.

`Spare` 6 bytes of spare memory, and should be set to 0. They are reserved for future expansion.

T_ICON
T_ICONTABLE

Type: **E_CCICONTABLE**

Structure:

```
typedef struct
   {
   unsigned short    XPos,YPos,Width,Height;
   long              Function[4];
   } T_ICON;

   typedef struct
   {
   unsigned short    NumIcons;
   T_ICON            Icons[1];
   } T_ICONTABLE;
```

Description: Defines a set of icons on the screen. They are not drawn by this chunk, it only defines a rectangle on the screen which performs some function.

This chunk may occur more than once in a configuration file. Where a number for this chunk is required, the first one defined is number 0, the next number 1, and so on.

Each icon has the following structure:

XPos, **YPos** The x and y position of the top left corner of the active rectangle.

Width, **Height** The width and height of the active rectangle.

Function An array of four functions to perform if the mouse pointer is within the rectangle with, respectively, no button pressed, the left button pressed, the right button pressed, or both buttons pressed. A value of 0 indicates ‘no action’. Function codes are listed in the VRT online “Reference Books.”

The complete icon list is:

NumIcons The number of icons defined.

Icons An array of icons as defined above.

T_INITINSVAL

Type: **`E_CCINSVALS`**

Structure:

```
typedef struct
   {
   short              InsNumber;
   T_LONGORPTR        InsVal,InsVal2;
   } T_INITINSVAL;
```

Description: Specifies the initial values of instruments. For each instrument:

`InsNumber` The index number of the instrument to initialize, or -1 to mark the end of the list.

`InsVal`, **`InsVal2`** The initial values of the primary and secondary instrument values.

T_KEYENTRY
T_KEYTABLE

Type: **E_CCFUNCTION**

E_CCADDFUNCTION

Structure:

```
typedef struct
   {
   short               KeyNum;
   long                Function;
   } T_KEYENTRY;

   typedef struct
   {
   short               NumKeys;
   T_KEYENTRY          KeyDef[1];
   } T_KEYTABLE;
```

Description: Defines which functions are assigned to which keys on the keyboard. Any keys which are not defined have a null function assigned to them. The chunk type `E_CCADDFUNCTION` indicates an additional key list (used in the configuration file to add custom keys onto the default keymaps in the preferences file).

This chunk may occur more than once in a config file. Where a number for this chunk is required, the first one defined is number 0, the next number 1, and so on.

Each defined key has the following structure:

`KeyNum` The key number on the keyboard, plus 0 for unshifted, `0x100` for SHIFT, `0x200` for CONTROL, or `0x300` for ALT. See the VRT online “Reference Books.”

`Function` A function number to execute when the key is pressed. Function codes are defined in the VRT online “Reference Books.”

The complete chunk is:

`NumKeys` The number of defined keys.

`KeyDef` An array of key definitions, as above.

T_KEYMAP

Type: **E_CCKEYTABLE**

Structure:

```
typedef struct
   {
   unsigned char    Key[256];
   unsigned char    Shift[256];
   } T_KEYMAP;
```

Description: Defines which characters are placed on which keys. The keys are defined in order of key number (see the VRT online “Reference Books”). It is included so that international keyboards may be set up.

The complete chunk is:

Key An array of characters for each key, as above.

Shift A corresponding array for each shifted key.

T_NETPARAMS

Type: **E_CCNETPARAMS**

Structure:

```
typedef strct
   {
     short          NumUsers, UserNum;
     long           LongTimeOut,ShortTimeOut,CheckRate;
   } T_NETPARAMS;
```

Description: Defines the parameters for the network device.

NumUsers The number of users on the network.

UserNum The user number. The first user is 0. `UserNum` ranges from 0 to `NumUsers` - 1.

LongTimeOut The timeout at the start of the network in milliseconds.

ShortTimeOut The timeout while the network is running in milliseconds.

CheckRate The number of frames between checksums.

T_PROPMULTI
T_PROPSETUP

Type: **E_CCMULTIDEVS**

Structure:

```
typedef struct
   {
   short             PropId;
   char              Enabled;
   short             NumBytes;
   char              Data[1];
   } T_PROPSETUP;

typedef struct
   {
   unsigned short    DevId,Length,NumSetups;
   T_PROPSETUP       Setup[1];
   } T_PROPMULTI;
```

Description: Defines the state of all devices (the `PROP` part of the name is kept for historic reasons). Each device setup contains the following information:

PropId The ID number of the device driver.

Enabled If the device is enabled, this flag is set to 1, if not it is set to 0.

NumBytes The number of bytes of setup data to send to the device.

Data The actual data to send when setting the device up.

Each device type has a `T_PROPMULTI` structure as follows:

DevId The device type (such as `E_DEVGRAPHICS`).

Length The total number of bytes in the `T_PROPMULTI` structure.

NumSetups The number of setup records, one for each available device.

Setup An array of setup structures, one for each available device.

The list of `T_PROPMULTI` structures is finally terminated by a short value of `0xFFFF`.

T_PROPTABLE

Type: **E_CCPROPTABLE**
E_CCADDPROPTABLE

Structure:

```
typedef struct
   {
   char             DialName[16];
   char             ControlName[16];
   short            Spare;
   char             Config[46],HWConfig[44];
   long             * MonoTime;
   long             ButtFunc[32];
   long             AxisFunc[5];
   }T_PROPTABLE;
```

Description: Defines the action of a proportional device like the Spacemouse. Type `E_CCADDPROPTABLE` defines additional custom controls to be added to the table defined in the preferences file. In this case, a value of 0 for a function indicates that the original function be left undisturbed.

DialName A string with the dialog name to use for editing configuration parameters. Under Windows, this dialog is ignored so set `DialName` to an empty string.

ControlName A string describing the type of control the proportional device gives.

Spare Not used.

Config An array of hardware configuration data. For each set of six axes there are, 6 sensitivity values (`unsigned char`), 6 deadzone values (`unsigned char`), and a maximum number of significant axes (`unsigned char`) as setup in the Proportional Control Setup dialog box.

HWConfig An array of hardware configuration data.

MonoTime A pointer to the monotonic (millisecond) timer and must be filled in before passing this table to the device driver.

ButtFunc An array of functions, one for each of up to 32 buttons on the device.

AxisFunc An array of functions, ten for each axis. For each axis, function 0 is the one to execute when the axis value is positive, and function 5 is the one to execute when the axis value is negative.

T_USERFUNC

Type: **E_CCUSER**

Structure:

```
struct
   {
   unsigned short    FuncNum;
   unsigned char     SCLProg[];
   } T_USERFUNC;
```

Description: Defines a user function, written in SCL:

FuncNum The number of the user function, from `0x8000` to `0x807F`. The full function number when using this function in a keyboard table, for example, is `0x46008000` to `0x4600807F`.

SCLProg An array of SCL object code bytes. The SCL program defined by these is executed whenever the defined function number is accessed. You may use this function anywhere, just like a normal function.

This chunk may occur more than once in a file, however, each function must have a unique number.

Resource data

Resources are chunked in a similar way to the world or shape data. Each dialog consists of a root item, which is always numbered 0, followed by a list of items within it.

Each item is represented by a chunk which contains all of the information required to draw and interact with that item. Chunks always start with the same information, represented by the structure `T_DIALCHKHEADER`. This contains the type of the chunk, its length, and information about its relatives and basic appearance. Each type of chunk additionally has information specific to the type of item it represents.

A dialog box is presented as a list of chunks (the first of which is the root item) terminated by the terminator `0xFFFF`. The dialog list as a whole is terminated by another `0xFFFF` terminator. Each dialog's root item contains a field specifying the total length of all chunks in that dialog box, including the terminator. This allows the list of dialog boxes to be traversed easily.

Each changeable value in an item is represented by a `T_DIALVALUE` structure, which can store actual values for immediate use, or point to values stored in the buffer of data passed to the dialog box routine.
The chunks in the resource data are all members of the union `T_DIALCHUNK`. Therefore, a pointer to this type can be used for any chunk and the relevant element read according to its chunk type. The `ChkType` fields for each type of chunk are all in the same place, and can always be read from the resource chunk element. The `T_DIALCHUNK` union looks like this:

```
typedef union
{
      T_DIALCHKHEADER       Header;
      T_DIALBOX             Box;
      T_DIALCASCADE         Cascade;
      T_DIALCHECKBOX        CheckBox;
      T_DIALCHECKMENU       CheckMenu;
      T_DIALCOLCHART        ColChart;
      T_DIALDROPDOWN        DropDown;
      T_DIALDUMMY           Dummy;
      T_DIALENUM            Enumerated;
      T_DIALHORZSLIDER      HorzSlider;
      T_DIALMENUENTRY       MenuEntry;
      T_DIALNUMERIC         Numeric;
      T_DIALOBJECT          Object;
      T_DIALOBJTREE         ObjTree;
      T_DIALPANE            Pane;
      T_DIALRADIO           Radio;
      T_DIALRADIOMENU       RadioMenu;
      T_DIALROOT            Root;
      T_DIALSCROLLBOX       ScrollBox;
      T_DIALSHAPE           Shape;
      T_DIALTAB             Tab;
      T_DIALTEXT            Label;
      T_DIALTEXTLINE        TextLine;
      T_DIALTEXTPAGE        TextPage;
      T_DIALTOGGLE          Toggle;
      T_DIALVERTSLIDER      VertSlider;
}     T_DIALCHUNK;
```

T_DIALVALUE

Structure:

```
typedef struct
   {
   unsigned short    Flags;
   T_LONGORPTR       Value;
   } T_DIALVALUE;
```

Description: **`Flags`** A set of bits which define how the `Value` field is to be interpreted.

Bits 0–4 define the type of value that is stored:

`E_TDVINTEGER` Integer

`E_TDVFLOAT` Floating point value

`E_TDVSTRING` String

`E_TDVFP`*`n`* Fixed point value with *n* bits of fractional data. *n* can be 1–16.

This value type subfield can be isolated by adding the `flags` value with mask `E_TDVVT`.

Bit 5, when set, specifies that the value is unsigned.

`E_TDVUNSIGNED` Unsigned

Bits 6–11 are reserved.

Bits 12–14 are the addressing mode used to retrieve the value.

`E_TDVNUL` Null. `Value` is ignored, and a zero or `NULL` pointer is returned.

`E_TDVIMM` Immediate. `Value` stores the actual returned value, or in the case of a string, a byte offset of the start of the string from the start of the chunk of which this `T_DIALVALUE` is a part. The string data is stored immediately after the end of the chunk proper, and is included in its length.

`E_TDVBUF` The value is in `C_EditBuffer`. Like the previous release, its precise position in the buffer is represented by an offset, a shift and a width. `Value` is treated as an integer with the following bit composition:

Bits 0–23 The offset of the value within `C_EditBuffer`, in bytes.

Bits 24–26 The shift of the value within that field, in bits.

Bits 27–31 The width of the value, in bits, less one (for example, a value of 1 in this subfield indicates a width of two bits).

`E_TDVPTR` A pointer to the value is in `C_EditBuffer`. `Value` is treated as an integer with the following bit composition:

Bits 0–23 The offset of the pointer to the value within C_EditBuffer, in bytes. The pointer is always 32 bits in width, and should not be shifted.

Bits 24–26 The shift of the value relative to the pointer address, in bits.

Bits 27–31 The width of the value, in bits, less one (a value of 1 in this subfield indicates a width of two bits).

The addressing mode sub-field can be isolated by adding the Flags value with E_TDVAM.

A value can be marked as read-only by setting the topmost bit, E_TDVREADONLY.

T_DIALVALUEs are most often read and written using a set of routines that refer to them by number. These indices all begin E_IF, and are presented with each dialog chunk.

T_DIALCHKHEADER

Structure:

```
typedef struct
   {
   unsigned short    ChkType;
   unsigned short    Length;
   short             ItemID;
   short             Child;
   short             Sibling;
   short             NextTab;
   short             PrevTab;
   unsigned short    Update;
   unsigned short    FocusKey;
   T_DIALVALUE       XPos;
   T_DIALVALUE       YPos;
   T_DIALVALUE       XSize;
   T_DIALVALUE       YSize;
   T_DIALVALUE       Flags;
   T_DIALVALUE       FGCol;
   T_DIALVALUE       BGCol;
   T_DIALVALUE       Return;
   T_DIALVALUE       Title;
   T_DIALVALUE       Font;
   }T_DIALCHKHEADER;
```

Description:

ChkType The type of the chunk. Chunk type values are given in the individual chunk structure definitions.

Length The total length of the chunk, including `ChkType` and itself.

ItemID A unique number identifying the item within this dialog box. Root items are always item 0. ItemIDs, like object numbers, must be unique but need not be consecutive (they should be positive as -1 is used as a ‘no item’ number).

Child The item ID of this item’s first child, or -1 if none.

Sibling The item ID of this item’s next sibling, or -1 if none.

NextTab, **PrevTab** The item IDs of the next and previous items in the tab list, or both -1 if the item is not in the tab list. Item 0 (the root item) has its `NextTab` field set to the ID of the first item in the tab list.

Update The item ID of the item to redraw when the value of this item has changed. Usually, this is the item’s own ID, but for radio buttons and tab items this may refer to the item’s parent. In this case, since all the redrawn item’s descendants are also redrawn, all the radio button’s siblings are also redrawn.

FocusKey The ASCII code of the key which, when pressed with ALT, moves the focus to this item. No other item may use the same focus key unless they cannot both be visible at once. This is to allow the same focus key to be used for items in different panes.

The following items are all `T_DIALVALUE`s and can contain immediate values or values read from the `C_EditBuffer` buffer.

`XPos`, **`YPos`** Specify the position of the top left of the rectangle enclosing the item. If zero or negative, they specify a position relative to the right or bottom edge of the parent rectangle (a position of -56, -17 represents a position 56 pixels in from the right, and 17 pixels up from the bottom of this item's parent). Indices `E_IFXPOS`, `E_IFYPOS`.

`XSize`, **`YSize`** Specify the width and height of the rectangle enclosing the item. If 0 or negative, they specify the position of the right or bottom edge of the item relative to the right or bottom edge of its parent (a size of -4, -24 represents a rectangle whose right edge is four pixels in from the right, and whose bottom edge is 24 pixels up from the bottom of its parent). Indices `E_IFXSIZE`, `E_IFYSIZE`.

`Flags` An integer, interpreted as a set of bits as follows (index `E_IFFLAGS`):

`E_MFINVISIBLE` Item is invisible.

`E_MFROOT` Item is a root item (must be set on root items).

`E_MFMENU` Item selects like a menu entry.

`E_MFGOSUB` Item pops up another dialog instead of returning.

`E_MFBACKGROUND` Background color is a palette reference, not a stipple.

`E_MFFOREGROUND` Foreground color is a palette reference, not a stipple.

`E_MFJUSTB` Justify text to the bottom of the enclosing rectangle.

`E_MFJUSTT` Justify text to the top of the enclosing rectangle.

`E_MFJUSTL` Justify text to the left of the enclosing rectangle.

`E_MFJUSTR` Justify text to the right of the enclosing rectangle.

`E_MFDOUBLECLICK` Double clicking on this item activates default button.

`E_MFINDENTED` Item is drawn indented.

`E_MFSAVEDBG` Not used.

`E_MFHANDLE` Dialog can be dragged by this item.

`E_MFSELECTABLE` Item can be selected (and edited) by user.

`E_MFDISABLED` Item is disabled (grayed out).

`E_MFBORDER1`, `E_MFBORDER2` Item is drawn with a border that is specified by the 2 bit value.

`E_MFTOUCHEXIT` If `E_MFEXIT` is not set, exits immediately, before item's value is changed in any way. If `E_MFEXIT` is set, exits immediately after item's value is changed, but before mouse button is released.

`E_MFTOUCHSELECT` Item is selected by mouse pointer going over it.

`E_MFINVISCHILD` Children of this item are invisible.

`E_MFSELECTED` Item has been selected by the user.

`E_MFEXIT` Item causes an exit when selected. See also `E_MFTOUCHEXIT`.

`E_MFCLEARSEL` On entry, this item's selected flag is cleared. Cursor and block positions are also reset in editable text page items.

`E_MFDEFAULT` Item is the default button (usually OK).

`E_MFNOHIGH` Item is not highlighted when selected.

`E_MFGOTO` Item jumps to another dialog rather than exiting.

`E_MFNOREDRAW` Do not redraw screen when exiting due to a click on this item.

`FGCol`, **`BGCol`** The foreground and background color in which the item is drawn. Indices are `E_IFBGCOL`, `E_IFFGCOL`.

`Return` The return value from this item if it is flagged as causing an exit. Root items have an exit code which is returned when ESC is pressed (usually an immediate value of `0x58000002`, 'Cancel'). Items which are flagged with `E_MFGOTO` or `E_MFGOSUB` flags have a return value which is a string, containing the name of the destination dialog box. Index in either case is `E_IFRETURN`.

`Title` The title of the item, index `E_IFTITLE`. Some items ignore the title field. If a `null` type, no title is drawn. In a text item or a box, if the title is an integer instead of a string, it is assumed to be an image number. Positive numbers are user images, negative numbers are system images.

`Font` The to use for any text drawn by the item. This is usually 10, the 6x9 font. Index is `E_IFFONT`.

T_DIALBOX

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER   Header;
   T_DIALVALUE       Border;
   } T_DIALBOX;
```

Description: Defines the data required for a box.

Header Contains all the standard information including the size of the box and its title if any. The chunk type is set to E_DIBOX.

Border The width of the border in pixels. If the item is outlined or indented, this is ignored and a single pixel border is drawn. Its index is E_IFBORDER.

T_DIALCASCADE

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER   Header;
   } T_DIALCASCADE;
```

Description: Defines the data required for a menu item with a cascade.

Header Contains all the required information including the title of the menu entry, and the name of the menu to open as a cascade (stored in its Return value field). The chunk type is set to E_DICASCADE.

T_DIALCHECKBOX

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER   Header;
   T_DIALVALUE       Value;
   T_DIALVALUE       Select;
   T_DIALVALUE       Deselect;
   } T_DIALCHECKBOX;
```

Description: Defines the data required for a check box.

Header Contains all the standard information including the size of the box and its title if any. The chunk type is set to E_DICHECKBOX.

Value The displayed value, usually specifying a value in C_EditBuffer. Its index is E_IFVALUE.

Select The value that is assigned to Value when the check box is selected. Its index is E_IFSELECT.

Deselect The value assigned to Value when the check box is deselected. Its index is E_IFDESELECT.

T_DIALCHECKMENU

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER   Header;
   T_DIALVALUE       Value;
   T_DIALVALUE       Select;
   T_DIALVALUE       Deselect;
   }T_DIALCHECKMENU;
```

Description: Defines the data required for a menu entry with a check.

Header Contains all the standard information including the size of the menu entry and its title. The chunk type is set to E_DICHECKMENU.

Value A value representing the selection state of the item, usually specifying a value in C_EditBuffer. Its index is E_IFVALUE.

Select The value that is assigned to Value when the menu entry is selected. Its index is E_IFSELECT.

Deselect The value assigned to Value when the menu entry is deselected. Its index is E_IFDESELECT.

T_DIALCOLCHART

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER   Header;
   T_DIALVALUE       StartCol;
   T_DIALVALUE       Cols;
   T_DIALVALUE       Rows;
   T_DIALVALUE       EndCol;
   }T_DIALCOLCHART;
```

Description: Defines the data required for a color chart.

`Header` Contains all the standard information including the size of the chart. Color charts do not have a title. The chunk type is set to `E_DICOLCHART`.

`StartCol` The index of the first color selected, usually specifying a value in `C_EditBuffer`. Its index is `E_IFSTARTCOL`.

`Cols`, **`Rows`** The number of columns and rows of colors displayed, respectively. Their indices are `E_IFCOLS` and `E_IFROWS`.

`EndCol` The index of the last color selected. If this is set to an immediate value of -1, dragging to select a range of colors is disallowed, and the color selected is stored in the value `StartCol`. If this specifies a value in `C_EditBuffer`, dragging to select ranges of colors is allowed, and the end color index is stored here. Its index is `E_IFENDCOL`.

T_DIALDROPDOWN

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER   Header;
   } T_DIALDROPDOWN;
```

Description: Defines the data required for a menu item with a drop down menu below it.

`Header` Contains all the required information including the title of the menu entry, and the name of the menu to open as a drop down (stored in its `Return` value field). The chunk type is set to `E_DIDROPDOWN`.

T_DIALDUMMY

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER   Header;
   T_DIALVALUE       v11;
   T_DIALVALUE       v12;
   T_DIALVALUE       v13;
   T_DIALVALUE       v14;
   T_DIALVALUE       v15;
   T_DIALVALUE       v16;
   T_DIALVALUE       v17;
   T_DIALVALUE       v18;
   T_DIALVALUE       v19;
   T_DIALVALUE       v20;
   }T_DIALDUMMY;
```

Description: Defines a way of generically accessing any value in any type of chunk.

Header Contains all the standard information. The chunk type can be set to the same value as any other dialog chunk type.

v11–v20 Generic labels for the T_DIALVALUE entries. Their indices are the values 11–20.

T_DIALENUM

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER   Header;
   T_DIALVALUE       Index;
   T_DIALVALUE       Min;
   T_DIALVALUE       Max;
   T_DIALVALUE       Increment;
   } T_DIALENUM;
```

Description: Defines the data required for an enumerated button.

`Header` Contains all the required information including the title of the button. The title consists of several lines of text. Which line is displayed is determined by the other fields of the chunk. The chunk type is set to `E_DIENUMERATED`.

`Index` The value to be interpreted, and usually specifies a value in `C_EditBuffer`. Its index is `E_IFINDEX`.

`Min` The value associated with the first line of the title. Its index is `E_IFMIN`.

`Max` The value associated with the last line of the title. Its index is `E_IFMAX`.

`Increment` The difference in value between successive lines in the title. Its index is `E_IFINC`.

T_DIALHORZSLIDER

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER   Header;
   T_DIALVALUE       Offset;
   T_DIALVALUE       Min;
   T_DIALVALUE       Max;
   T_DIALVALUE       LineInc;
   T_DIALVALUE       Width;
   T_DIALVALUE       MinWidth;
   T_DIALVALUE       PageInc;
   } T_DIALHORZSLIDER;
```

Description: Defines the data required for a horizontal slider bar.

`Header` Contains all the standard information including the size of the bar. Sliders do not have a title. The chunk type is set to `E_DIHORZSLIDER`.

`Offset` The value to display. This is in arbitrary units. Its index is `E_IFOFFSET`.

`Min`, **`Max`** The values which are represented when the slider is at the extreme ends of its travel (left and right, respectively). If `Min` is greater than `Max`, the direction of travel is reversed. These are measured in the same units as `Offset`. Their indices are `E_IFMIN` and `E_IFMAX`.

`LineInc` The minimum step in which the slider can move when dragged, measured in the same units as `Offset`. Its index is `E_IFLINEINC`.

`Width` The apparent width of the slider itself, measured in the same units as `Offset`. Its index is `E_IFWIDTH`.

`MinWidth` The minimum width, in pixels, that the slider can be drawn. This is to prevent it disappearing entirely when `Width` is small compared to `Min` and `Max`. Its index is `E_IFMINWIDTH`.

`PageInc` The distance the slider moves when the mouse is clicked to its left or right. This is measured in the same units as `Offset`. Its index is `E_IFPAGEINC`.

T_DIALLISTBOX

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER   Header;
   T_DIALVALUE       Index;
   T_DIALVALUE       Min;
   T_DIALVALUE       Max;
   } T_DIALLISTBOX;
```

Description: Defines the data required for a list box.

Header Contains all the required information including the title of the list box. The title consists of several lines of text. Which line is displayed is determined by the other fields of the chunk. The chunk type is set to E_DILISTBOX.

Index The value to be interpreted, and usually specifies a value in C_EditBuffer. Its index is E_IFINDEX.

Min The value associated with the first line of the title. Its index is E_IFMIN.

Max The value associated with the last line of the title. Its index is E_IFMAX.

T_DIALMENUENTRY

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER   Header;
   } T_DIALMENUENTRY;
```

Description: Defines the data required for a menu item.

Header Contains all the required information including the title of the menu entry, and the code which it returns when selected (stored in its Return value field). The chunk type is set to E_DIMENUENTRY.

T_DIALNUMERIC

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER   Header;
   T_DIALVALUE       Value;
   T_DIALVALUE       Min;
   T_DIALVALUE       Max;
   T_DIALVALUE       Base;
   } T_DIALNUMERIC;
```

Description: Defines the data for an editable number.

`Header` Contains all the required information including the title of the item. This should include '_' characters to indicate where in the title the editable number is to be displayed. The chunk type is set to `E_DINUMERIC`.

`Value` Specifies the numeric value to display and edit, usually as a buffered value in `C_EditBuffer`. It can be an integer, fixed point value, or float. Its index is `E_IFVALUE`.

`Min`, **`Max`** The minimum and maximum allowable values for the item. Their indices are `E_IFMIN` and `E_IFMAX`.

`Base` Defines how the data is presented to the user. Its index is `E_IFBASE`. Bits 8–11 indicate the basic format of the displayed number and can be one of:

`E_NUMERICDECIMAL` Decimal integer

`E_NUMERICFLOAT` Floating point number, (1234.567)

`E_NUMERICSCIENTIFIC` Scientific format number, (1.2345e+3)

`E_NUMERICHEXADECIMAL` Hexadecimal integer

`E_NUMERICANGLE` Angle. Integer values are converted from brees to degrees, and back.

For the floating point and scientific types, bits 20–23 represent the number of digits to display after the decimal point, from 0–15.

T_DIALOBJECT

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER   Header;
   T_DIALVALUE       Object;
   T_DIALVALUE       Min;
   T_DIALVALUE       Max;
   } T_DIALOBJECT;
```

Description: Defines the data required for an editable object name.

`Header` Contains all the required information including the title of the item. This should include '_' characters to indicate where in the title the editable name is to be displayed. The chunk type is set to `E_DIOBJECT`.

`Object` Specifies the object number to display and edit, usually as a buffered value in `C_EditBuffer`. This is an integer representing the object number, which is converted to a name for editing purposes. Its index is `E_IFOBJECT`.

`Min`, **`Max`** The minimum and maximum allowable values for the item. Their indices are `E_IFMIN` and `E_IFMAX`.

T_DIALOBJTREE

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER  Header;
   T_DIALVALUE      ObjTree;
   T_DIALVALUE      Elevation;
   T_DIALVALUE      Azimuth;
   T_DIALVALUE      Distance;
   }T_DIALOBJTREE;
```

Description: Defines the data required for an object tree. This produces a view onto the object tree as a complete world. Usually, only one object is viewed at a time, and the root item is wrapped to just contain that object.

Header Contains all the standard information including the size of the item. Object trees do not have a title. The chunk type is set to E_DIOBJTREE.

ObjTree Specifies a pointer which points to the root item of the world to be drawn. Its index is E_IFOBJTREE.

Elevation Specifies the angle of elevation (above or below) from which the world is to be viewed. Its index is E_IFELEVATION.

Azimuth Specifies the horizontal angle from which the world is to be viewed. Its index is E_IFAZIMUTH.

Distance Specifies the distance of the viewpoint from the center of the world. Its index is E_IFDISTANCE.

T_DIALPANE

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER   Header;
   T_DIALVALUE       Value;
   T_DIALVALUE       Select;
   } T_DIALPANE;
```

Description: Defines the data required for a pane. Panes are similar to boxes, in that they are used as containers for other items. They may be visible or invisible depending on the value placed in the buffer.

Header Contains all the standard information including the size of the box and its title if any. The chunk type is set to `E_DIPANE`.

Value The displayed value, usually specifying a value in `C_EditBuffer`. If this matches `Select`, the pane and its children become visible. If not, they become invisible. Its index is `E_IFVALUE`.

Select The value of `Value` which causes the pane to become visible. Its index is `E_IFSELECT`.

T_DIALRADIO

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER   Header;
   T_DIALVALUE       Value;
   T_DIALVALUE       Select;
   }T_DIALRADIO;
```

Description: Defines the data required for a radio button.

Header Contains all the standard information including the size of the item and its title if any. The chunk type is set to `E_DIRADIO`.

Value The displayed value, usually specifying a value in `C_EditBuffer`. If it matches `Select`, the radio button is drawn selected, otherwise it is drawn unselected. Its index is `E_IFVALUE`.

Select The value that is assigned to `Value` when the radio button is selected. Its index is `E_IFSELECT`.

T_DIALRADIOMENU

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER   Header;
   T_DIALVALUE       Value;
   T_DIALVALUE       Select;
   } T_DIALRADIOMENU;
```

Description: Defines the data required for a radio button.

`Header` Contains all the standard information including the size of the item and its title. The chunk type is set to `E_DIRADIOMENU`.

`Value` The displayed value, usually specifying a value in `C_EditBuffer`. If it matches `Select`, the menu entry is drawn selected, otherwise it is drawn unselected. Its index is `E_IFVALUE`.

`Select` The value that is assigned to `Value` when the menu entry is selected. Its index is `E_IFSELECT`.

T_DIALROOT

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER   Header;
   T_DIALVALUE       Length;
   T_DIALVALUE       RootCtrl;
   } T_DIALROOT;
```

Description: Defines the data required for a dialog box root item.

`Header` Contains all the standard information including the size of the item. Its title specifies the name of the dialog box. The chunk type is set to `E_DIMENU` for a drop-down menu, `E_DITOOL` for a toolbox or menu bar, and `E_DICONTROL` for a dialog box.

`Length` The length of the entire dialog box, including the chunk list terminator. It must be an immediate integer, and is not editable by the user. Its index is `E_IFLENGTH`.

`RootCtrl` A set of flags that control various aspects of the dialog box's behavior:

`E_MRRESIDENT` Dialog is a resident box.

`E_MRNOSAVEBG` Dialog does not save or restore its background.

`E_MRREFRESH` Dialog is redrawn every frame (resident dialogs only).

`E_MRBRINGTOTOP` Dialog is brought to the front when selected.

`E_MRRESAVE` Dialog resaves its background every frame (resident dialogs only).

Its index is `E_IFROOTCTRL`.

T_DIALSCROLLBOX

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER   Header;
   T_DIALVALUE       Offset;
   T_DIALVALUE       Min;
   T_DIALVALUE       Max;
   T_DIALVALUE       LineInc;
   T_DIALVALUE       Width;
   T_DIALVALUE       MinWidth;
   T_DIALVALUE       PageInc;
   T_DIALVALUE       Children;
   } T_DIALSCROLLBOX;
```

Description: This structure defines the data required for a scroll box

Header Contains all the standard information including the size of the box. Scroll boxes to not have a title. The chunk type is set to E_DISCROLLBOX.

Offset Specifies the value which is added to any T_DIALVALUE offsets in all the scroll box's children. This is usually a value in C_EditBuffer. Its index is E_IFOFFSET.

Min, **Max** Values representing the minimum and maximum allowed values for Offset. Their indices are E_IFMIN and E_IFMAX.

LineInc The minimum offset step by which the slider can move when dragged, or when the up and down arrows are clicked. It is usually the same as the amount of memory occupied by one data element in the scroll list. Its index is E_IFLINEINC.

Width The apparent width of the slider itself. This is usually set to the value of LineInc multiplied by the number of visible scroll list items (the total size of all the visible scroll list items). Its index is E_IFWIDTH.

MinWidth The minimum width, in pixels, that the slider can be drawn. This is to prevent it disappearing entirely when Width is small compared to Min and Max. Its index is E_IFMINWIDTH.

PageInc The amount that Offset changes when the mouse is clicked above or below the slider. This is usually set to the value of LineInc multiplied by the number of visible scroll items less one. This ensures that when paging down the list, the last item on the previous page is still displayed as the first item on the new page. Its index is E_IFPAGEINC.

Children The number of children that are not visually affected by the scrolling process. Their offsets are still modified, but they are not included in the process which makes items visible or invisible. Its index is E_IFCHILDREN.

T_DIALSHAPE

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER   Header;
   T_DIALVALUE       Shape;
   T_DIALVALUE       Min;
   T_DIALVALUE       Max;
   } T_DIALSHAPE;
```

Description: Defines the data required for an editable shape name.

`Header` Contains all the required information including the title of the item. This should include '_' characters to indicate where in the title the editable name is to be displayed. The chunk type is set to `E_DISHAPE`.

`Shape` Specifies the object number to display and edit, usually as a buffered value in `C_EditBuffer`. This is an integer representing the object number, which is converted to a name for editing purposes. Its index is `E_IFSHAPE`.

`Min`, **`Max`** The minimum and maximum allowable values for the item. Their indices are `E_IFMIN` and `E_IFMAX`.

T_DIALTAB

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER   Header;
   T_DIALVALUE       Value;
   T_DIALVALUE       Select;
   } T_DIALTAB;
```

Description: Defines the data required for a tab, as used in conjunction with panes.

`Header` Contains all the standard information including the size of the item and its title. The chunk type is set to `E_DITAB`.

`Value` The displayed value, usually specifying a value in `C_EditBuffer`. If it matches `Select`, the tab is drawn selected, otherwise it is drawn unselected. Its index is `E_IFVALUE`.

`Select` The value that is assigned to `Value` when the tab is selected. Its index is `E_IFSELECT`.

T_DIALTEXT

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER   Header;
   } T_DIALTEXT;
```

Description: Defines the data required for a non-editable text item.

`Header` Contains all the required information including the size and title of the text. If the title is set to an integer value instead of a string, it is taken as an image number. Positive numbers are user images, negative numbers are system images. The chunk type is set to `E_DITEXT`.

T_DIALTEXTLINE

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER   Header;
   T_DIALVALUE       String;
   T_DIALVALUE       Validate;
   T_DIALVALUE       MaxLen;
   } T_DIALTEXTLINE;
```

Description: Defines the data required for an editable text line.

`Header` Contains all the standard information including the title of the item. This should include ‘_’ characters to indicate where in the title the editable string is to be displayed. The chunk type is set to `E_DITEXTLINE`.

`String` Defines the string to edit, and is usually specified as a string in `C_EditBuffer`. Its index is `E_IFSTRING`.

`Validate` Not used. Set to a `null` type. Its index is `E_IFVALID`.

`MaxLen` The maximum allowed length, in characters, of the editable string. Note that this does not include the zero terminator character. Its index is `E_IFMAXLEN`.

T_DIALTEXTPAGE

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER   Header;
   T_DIALVALUE       String;
   T_DIALVALUE       Validate;
   T_DIALVALUE       MaxLen;
   T_DIALVALUE       MaxWidth;
   T_DIALVALUE       XOffset;
   T_DIALVALUE       YOffset;
   T_DIALVALUE       Cursor;
   T_DIALVALUE       BlockStart;
   T_DIALVALUE       BlockEnd;
   } T_DIALTEXTPAGE;
```

Description: Defines the data required for an editable text page.

Header Contains all the standard information including the size of the item. Text pages do not have a title. The chunk type is set to E_DITEXTPAGE.

String Defines the string to edit, and is usually specified as a string in C_EditBuffer. Its index is E_IFSTRING.

Validate Not used. Set to a null type. Its index is E_IFVALID.

MaxLen The maximum allowed length, in characters, of the editable string. Note that this does not include the zero terminator character. Its index is E_IFMAXLEN.

MaxWidth The maximum allowed length, in characters, of any single line in the editable string. Its index is E_IFMAXWIDTH.

XOffset, **YOffset** The position of the first character displayed within the string. These are set automatically by the text page interpreting the cursor position, but they may be interrogated. Their indices are E_IFXOFFSET and E_IFYOFFSET.

Cursor The current position of the cursor, measured as the number of characters from the start of the string. If the E_MFCLEARSEL flag is set on this item, the cursor position and block positions are reset on entry to the dialog box in which it resides. If this is not set, you must set up Cursor, BlockStart and BlockEnd before bringing up the dialog box. It index is E_IFCURSOR.

BlockStart, **BlockEnd** The positions of the ends of the currently marked block, measured in the same way as Cursor. If no block is marked, these should both be set to -1. Their indices are E_IFBLOCKSTART and E_IFBLOCKEND.

T_DIALTOGGLE

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER   Header;
   T_DIALVALUE       Value;
   T_DIALVALUE       Select;
   T_DIALVALUE       Deselect;
   } T_DIALTOGGLE;
```

Description: Defines the data required for a toggle button.

Header Contains all the standard information including the size of the button and its title if any. The chunk type is set to `E_DITOGGLE`.

Value The displayed value, usually specifying a value in `C_EditBuffer`. Its index is `E_IFVALUE`.

Select The value that is assigned to `Value` when the button is selected. Its index is `E_IFSELECT`.

Deselect The value assigned to `Value` when the button is deselected. Its index is `E_IFDESELECT`.

T_DIALVERTSLIDER

Structure:

```
typedef struct
   {
   T_DIALCHKHEADER   Header;
   T_DIALVALUE       Offset;
   T_DIALVALUE       Min;
   T_DIALVALUE       Max;
   T_DIALVALUE       LineInc;
   T_DIALVALUE       Width;
   T_DIALVALUE       MinWidth;
   T_DIALVALUE       PageInc;
   } T_DIALHORZSLIDER;
```

Description: Defines the data required for a vertical slider bar.

`Header` Contains all the standard information including the size of the bar. Sliders do not have a title. The chunk type is set to `E_DIVERTSLIDER`.

`Offset` The value to display. This is in arbitrary units. Its index is `E_IFOFFSET`.

`Min`, **`Max`** The values which are represented when the slider is at the extreme ends of its travel (bottom and top, respectively). If `Min` is greater than `Max`, the direction of travel is reversed. These are measured in the same units as `Offset`. Their indices are `E_IFMIN` and `E_IFMAX`.

`LineInc` The minimum step in which the slider can move when dragged, measured in the same units as `Offset`. Its index is `E_IFLINEINC`.

`Width` The apparent width of the slider itself, measured in the same units as `Offset`. Its index is `E_IFWIDTH`.

`MinWidth` The minimum width, in pixels, that the slider can be drawn. This is to prevent it disappearing entirely when `Width` is small compared to `Min` and `Max`. Its index is `E_IFMINWIDTH`.

`PageInc` The distance the slider moves when the mouse is clicked above or below it. This is measured in the same units as `Offset`. Its index is `E_IFPAGEINC`.

Drawing list data

The drawing list is generated by VRT during processing. At its simplest, it is a tree structure containing 2D polygonal representations of the objects in the world, as seen from the current viewpoint.

Each node in the tree consists of a list of chunks, terminated as usual by an `0xFFFF`. The first chunk in each node must be of type `E_DCOBJECT`.

The chunks in the drawing tree are all members of the union `T_DRAWCHUNK`. Thus, a pointer to this type can be used for any chunk and the relevant element read according to its chunk type. The `ChkType` fields for each different type of chunk are all in the same place, so this can always be read by reading the chunk type from the object chunk element.

The `T_DRAWCHUNK` union looks like this:

```
typedef union
{
  T_DCOBJECT          Obj;
  T_DCFACET           Fac;
  T_DCTEXT            Txt;
  T_DCSPRITE          Spr;
  T_DCATTACH          Att;
  T_DCPALFACET        PFc;
  T_DCRECT            Rec;
  T_DCPALRECT         PRc;
} T_DRAWCHUNK;
```

As an example, reading the facet number from the facet chunk pointed to by a pointer `p` (which is of type `T_DRAWCHUNK *`) would be done using:

```
FacNum=p->Fac.FacNum;
```

The `Fac` part specifies that what you are looking at is a facet chunk.

The structures used in the drawing list are as follows:

T_DCATTACH

Type: **E_DCATTACH**

Union: **T_DRAWCHUNK.Att**

Structure:

```
typedef struct
   {
   unsigned short    ChkType,Length;
   unsigned short    Attach[1];
   } T_DCATTACH;
```

Description: Defines attachments between objects, used for sorting purposes.

ChkType The chunk type (E_DCATTACH).

Length The length of the chunk, including ChkType and itself.

Attach An array of object numbers. If the top bit is reset, the nominated object must be sorted in front of this one; if set it must be sorted behind.

T_DCFACET
T_DRAWFACET

Type: **E_DCFACET**

Union: **T_DRAWCHUNK.Fac**

Structure:

```
typedef struct
   {
   unsigned short    FacNum;
   unsigned char     FacAtt;
   unsigned char     Colour;
   unsigned short    NumPoints;
   } T_DRAWFACET;


typedef struct
   {
   unsigned short    ChkType,Length;
   unsigned short    NumFacets;
   T_DRAWFACET       Facet[1];
   } T_DCFACET;
```

Description: Define a chunk containing a number of polygons (facets) to be drawn.

FacNum The facet number copied from the shape definition.

FacAtt The facet attributes byte, again copied from the shape definition.

Colour The color in which to draw the facet.

NumPoints The number of following point records, defining the vertices of this polygon.

Following this, but not explicitly defined, is a list of `T_GRPOINTREC` records defining the position of each vertex of the polygon.

The entire chunk is defined as follows:

ChkType The chunk type (`E_DCFACET`).

Length The length of the chunk, including `ChkType` and itself.

NumFacets The number of facets in the chunk.

Facet A list of facet descriptions, as defined above.

See also: `T_DCPALFACET`

T_DCOBJECT

Type: **E_DCOBJECT**

Union: **T_DRAWCHUNK.Obj**

Structure:

```
typedef struct
   {
   unsigned short    ChkType,Length;
   unsigned short    ObjNum;
   void              *ObjAdd;
   void              *Sibling,*PrevSib,*Child;
   unsigned short    Flags;
   } T_DCOBJECT;
```

Description: Defines the start of a list of drawing list chunks for a particular object. In drawing lists which do not refer to objects themselves, such as those generated by the dialog box routine, this contains dummy information.

ChkType The chunk type (E_DCOBJECT).

Length The length of the chunk, including ChkType and itself.

ObjNum The number of the object to which the following chunks refer.

ObjAdd A pointer to the object's standard chunk in the world information, used for sorting the drawing tree.

Sibling, **PrevSib**, **Child** Pointers to the next and previous siblings of this chunk, and its children if any. A NULL value in these fields indicates that the nominated chunk does not exist.

Flags A set of flags:

E_DCOBJZREQ

E_DCOBJTRANSP

E_3DOBJECT

E_3DOBJMASK

T_DCPALFACET

Type: **E_DCPALFACET**

Union: **T_DRAWCHUNK.Fac**

Structure:

```
typedef struct
   {
   unsigned short    ChkType,Length;
   unsigned short    NumFacets;
   T_DRAWFACET       PalFacet[1];
   } T_DCPALFACET;
```

Description: Defines a chunk containing a number of polygons (facets) to be drawn using palette colors directly rather than using the stipple table.

ChkType The chunk type (E_DCPALFACET).

Length The length of the chunk, including ChkType and itself.

NumFacets The number of facets in the chunk.

PalFacet A list of facet descriptions, defined by T_DRAWFACET.

See also: T_DCFACET

T_DCPALRECT

Type: **E_DCPALRECT**

Union: **T_DRAWCHUNK.PRc**

Structure:

```
typedef struct
   {
   unsigned short    ChkType,Length;
   unsigned short    NumRects;
   T_DCRECTANGLE     PalRect[1];
   } T_DCPALRECT;
```

Description: Defines a list of rectangles to be drawn on the screen using the palette colors directly rather than using the stipple table.

ChkType The chunk type (E_DCPALRECT).

Length The length of the chunk, including ChkType and itself.

NumRects The number of rectangles to be drawn.

PalRect A list of rectangles defined by T_DCRECTANGLE.

See also: T_DCRECT

T_DCRECT
T_DCRECTANGLE

Type: **E_DCRECT**

Union: **T_DRAWCHUNK.Rec**

Structure:

```
typedef struct
   {
   short              XStart,YStart;
   short              XEnd,YEnd;
   unsigned char      Colour,Spare;
   } T_DCRECTANGLE;

   typedef struct
   {
   unsigned short     ChkType,Length;
   unsigned short     NumRects;
   T_DCRECTANGLE      Rect[1];
   } T_DCRECT;
```

Description: Defines a list of rectangles to be drawn on the screen.

XStart, **YStart** The coordinates of the top left corner of the rectangle.

XEnd, **YEnd** The coordinates of the bottom right corner of the rectangle.

Colour The color in which to draw the rectangle.

Spare Not used.

The entire chunk is defined as follows:

ChkType The chunk type (E_DCRECT).

Length The length of the chunk, including ChkType and itself.

NumRects The number of rectangles to be drawn.

Rect A list of rectangles as defined above.

See also: T_DCPALRECT

T_DCSPRITE

Type: **E_DCSPRITE**

Union: **T_DRAWCHUNK.Spr**

Structure:

```
typedef struct
   {
   unsigned short    ChkType,Length;
   T_GRSPRITE        Sprite;
   } T_DCSPRITE;
```

Description: Defines an image (sprite) to be drawn onto the screen.

ChkType The chunk type (E_DCSPRITE).

Length The length of the chunk, including ChkType and itself.

Sprite The image (sprite) description to be drawn.

See also: T_GRSPRITE

T_DCTEXT

Type: **E_DCTEXT**

Union: **T_DRAWCHUNK.Txt**

Structure:

```
typedef struct
   {
   unsigned short    ChkType,Length;
   T_GRTEXT          Text;
   char              String[1];
   } T_DCTEXT;
```

Description: Describes text to be drawn on the screen.

ChkType The chunk type (E_DCTEXT).

Length The length of the chunk including ChkType and itself.

Text A text descriptor defining the appearance and position of the text on screen.

String The zero terminated string to be drawn on the screen.

See also: T_GRTEXT

T_DRAWLIST

Structure:

```
typedef struct
   {
   char *            DrawList;
   char *            DrawStart;
   long              Length;
   short             NumFSF;
   short             Flags;
   } T_DRAWLIST;
```

Description: The structure passed to the graphics device driver function `GrDrawSorted`.

`DrawList` Points to the drawing list buffer.

`DrawStart` Points to the root node in the drawing list.

`Length` The length of the whole drawing list, in bytes.

`NumFSF` The number of full screen facets detected in the drawing list (there is no point drawing anything behind an opaque facet which occupies the whole screen).

`Flags` Not used.

T_DRAWTEXTURE

Structure:

```
typedef struct
   {
   unsigned short    TexWidth, TexHeight;
   char *            TexAd;
   unsiged short     ScaleX, ScaleY, OffsetX, OffsetY,
                     TexScale;
   T_VECTOR          Points[4];
   } T_DRAWTEXTURE;
```

Description: This structure is not used by VRT 5.50 or later. It is included for compatibility with earlier versions of VRT.

The structure immediately follows a facet definition (`T_DRAWFACET`) if the facet has a texture.

`TexWidth`, **`TexHeight`** The size of the texture definition, in pixels.

`TexAd` The address of the texture data.

`ScaleX`, **`ScaleY`** The number of complete textures to fit on the facet in the X and Y directions, measured in $^1/_{256}$ths.

`OffsetX`, **`OffsetY`** The X and Y offsets of the origin of the facet from the origin of the texture, measured in $^1/_{256}$ths.

`TexScale` A measure of the coarseness of the drawn texture, and can be one of:

`E_TEX4X4`	Draw using 4x4 pixel grid.
`E_TEX3X3`	Draw using 3x3 pixel grid.
`E_TEX2X2`	Draw using 2x2 pixel grid.
`E_TEX1X1`	Draw using 1x1 pixel grid.

`Points` A set of 3D points defining a rectangle around the texture. These are used to calculate the effect of perspective on the texture.

Device driver data

The device drivers have their own data structures for passing arguments and returning results. Some (especially the graphics device) have large data areas defined, such as the drawing buffer where pictures are constructed prior to being sent to the graphics board. These are described as they are encountered.

In general, there are `E_MAXDEVTYPES` different device types, with a maximum of `E_MAXPERDEV` device drivers loaded for each type of device.

The data structures used are as follows:

T_DEVINSTALL

Structure:

```
typedef struct
   {
   T_STDFUNCPTR      **BaseAddress;
   unsigned char     *NumDevActive;
   unsigned char     *DevActive;
   void              *Config;
   } T_DEVINSTALL;
```

Description: This structure contains the installation information common to each device.

`BaseAddress` A pointer to the array of device driver base addresses. This allows the device driver to call other devices if necessary.

`NumDevActive`, **`DevActive`** Pointers to the active device lists, `C_NumDevsActive` and `C_DeviceActive`.

`Config` A pointer to the device specific install information (of type `T_??INSTALL`).

T_GRBACKDROP

Structure:

```
typedef struct
   {
   unsigned char     *Data;
   long              Length;
   short             Flags;
   unsigned short    XPos,YPos,XSize,YSize,XStart,YStart;
   } T_GRBACKDROP;
```

Description: Describes a backdrop to be drawn, as passed to `GrDrawBackdrop`.

`Data` A pointer to the loaded backdrop file.

`Length` The length of the backdrop data.

`Flags` A set of flags, as follows:

- `E_BDFORE` If set, draw on foreground screen.
- `E_BDBACK` If set, draw on background screen.
- `E_BDNEW` Should be set if the previously drawn backdrop is not this one.

`XPos`, **`YPos`** The position on the screen of the top left of the backdrop.

`XSize`, **`YSize`** The size of the portion of the backdrop to draw.

`XStart`, **`YStart`** The position within the backdrop to start drawing (set to 0,0).

T_GRINSTALL

Structure:

```
typedef struct
   {
   short              FirstCol,PalSize;
   unsigned char      *Palette;
   unsigned char      *Stipples;
   short              ModeFlags;
   void               Flags;
   unsigned char      *FontTable;
   void *             (*ReAllocate)(long);
   long               DrawSize;
   char               *Environment;
   char               VideoMode;
   void *             hViewWnd;
   char *             DrawBuffer;
   char *             SpriteBuffer;
   short              ScreenWidth;
   short              ScreenHeight;
   } T_GRINSTALL;
```

Description: A pointer to a structure of this type is passed to `GrInstall` using a structure of type `T_DEVINSTALL`.

`FirstCol`, **`PalSize`** The first color and number of colors to set using the palette data. These are usually set to 0 and 256 respectively.

`Palette` A pointer to the palette data to display, as contained in the palette file.

`Stipples` A pointer to the stipple table, as contained in the palette file.

`Flags` A set of flags that specify options at startup. At present the only flag defined is:

`E_GRIFNOSHUTUP` If set, sound is not stopped when palettes are changed.

`FontTable` A pointer to the font data, defined in the system font file.

`Reallocate` A pointer to C's routine `realloc`, used for allocating memory for various buffers needed by the device.

`Drawsize` The size of the required drawing buffer, in bytes.

`Environment` Not used.

`VideoMode` Not used.

`hViewWnd` The Windows handle of the main view window.

`DrawBuffer` The address of the bitmap into which the view is drawn.

`SpriteBuffer` Usually set to point to the same area of memory as DrawBuffer, but can be rerouted for drawing into a texture.

`ScreenWidth`, **`ScreenHeight`** The width and height of the actual view portion of window.

T_GRINSTALLRET

Structure:

```
typedef struct
   {
   short              Error,Id;
   char               Name[16];
   char               DialName[16];
   }  T_GRINSTALLRET;
```

Description: A pointer to this structure is returned by `GrInstall`.

`Error` Non-0 if the graphics device cannot be installed.

`Id` The ID number for this device. Current values are:

`E_GRIDSVGA`:	SVGA
`E_GRIDD3D`:	Superscape graphics device and Direct3D

`Name` The name of the device.

`DialName` The name of the dialog box used to edit the setup data for this device (if any).

T_GRLINE

Structure:

```
typedef struct
   {
   short            XStart,YStart;
   long             ZStart;
   short            XEnd,YEnd;
   long             ZEnd;
   unsigned char    Colour,Illumination;
   } T_GRLINE;
```

Description: Defines a straight line on the screen, as passed to `GrDrawLine`.

`XStart`, **`YStart`** The position of one end of the line on the screen.

`ZStart` Reserved - set it to 0.

`XEnd`, **`YEnd`** The position of the other end of the line.

`ZEnd` Reserved - set it to 0.

`Colour` The color in which to draw the line.

`Illumination` Reserved—set it to 255.

T_GROPTIONS

Structure:

```
typedef struct
   {
   short               PointSize,Flags;
   unsigned char       Resolution;
   short               Hres,VRes;
   void *              DrawAddress;
   void *              ExtraConfig;
   } T_GROPTIONS;
```

Description: Defines various options as to how the graphics device driver operates, as passed to `GrSetup`.

`PointSize` The number of bytes used by a point record. At present, this is set at 4.

`Flags` A set of flags as follows:

`E_PALETTE` If set, draws polygons using the palette directly, rather than the stipple table.

`E_TEXTURE` If set, the screen is to be drawn into a texture rather than on to the real screen.

`E_INTERLACE` Interlace the top and bottom halves of the screen together on alternate scan lines. This is used by some headsets to generate stereo views.

`E_EXTRACFG` Not currently used.

`E_GRMOVE` If set, all data related to the window position is updated if it is moved.

`Resolution` The resolution index in which the card is to operate (`0`=default, `0xFF`=no change).

`Hres`, **`VRes`** The horizontal and vertical resolution of the texture if the screen is to be drawn into a texture.

`DrawAddress` The address of the data in a texture in which to draw the picture.

`ExtraConfig` The address of the `Data` field of the `T_PROPSETUP` structure for the graphics device.

T_GRPALETTE

Structure:

```
typedef struct
   {
   short              FirstCol,PalSize;
   unsigned char      *Palette;
   } T_GRPALETTE;
```

Description: Describes a palette, as passed to `GrSetPalette`.

`FirstCol`, **`PalSize`** The first color and number of colors to set, respectively.

`Palette` A pointer to the actual palette data, organized as sets of three bytes R,G,B.

T_GRPICKOBJ

Structure:

```
typedef struct
   {
   char *             DrawList;
   short              x, y;
   unsigned short     Exclude;
   } T_GRPOINTREC;
```

Description: Identifies an object and its position for graphics devices that can pick facets from the screen—the `E_ABILPICK` flag must be set in `T_GRSETUP`.

`DrawList` Pointer to the drawing list you want to pick a facet from.

`x`, **`y`** The position of the facet on screen that you want to pick.

`Exclude` Not currently used. Set to `FFFFFFFF`.

T_GRPOINTREC

Structure:

```
typedef struct
   {
   short            x,y;
   unsigned char    Illum,Pad;
   long             z;
   } T_GRPOINTREC;
```

Description: Describes a point on the screen. Only the first `PointSize` bytes are used (see `T_GROPTIONS`), at present this is 4. Therefore, only the following fields should be used:

x, **y** The coordinates of the point on the screen.

T_GRPOLYGON

Structure:

```
typedef struct
   {
   unsigned short   FacNum;
   unsigned char    Flags,Colour;
   short            NumPoints;
   T_GRPOINTREC     Points[E_GRMAXPOLYSIZE];
   } T_GRPOLYGON;
```

Description: Describes a convex polygon to be drawn on the screen, as passed to `GrDrawPolygon`. The Superscape graphics device then takes the data and smooth-shades the polygons if lighting is active in the world.

FacNum The facet number, copied from the 3D facet being drawn.

Flags Copied from the facet attributes of the facet being drawn.

Colour The color of the polygon.

NumPoints The number of points in the polygon.

Points An array of point positions defining the positions of the vertices of the polygon.

T_PROCINFO

Structure:

```
typedef struct
   {
   T_WORLDCHUNK *    Tree;
   short             ObjView;
   T_LINE **         LineAd;
   T_FACET **        FacetAd;
   void **           DrawAd;
   short             Plane, ZPlane, ZPlaneScale;
   T_NORMALS **      NormAd;
   long              XDisOpts;
   unsigned char *   PointsDisplay;
   short *           PointsSelected;
   } T_GRPROCINFO;
```

Description: If the `E_ABILTRANSOBJ` flag is set in `T_GRSETUP`, this chunk is used to define the data and how it is processed.

`Tree` The object tree to be processed.

`ObjView` The object to which the current viewpoint is attached.

`LineAd` Pointer to an array that stores the line chunk addresess for shapes.

`FacetAd` Pointer to an array that stores the facet chunk addresses for a shape.

`DrawAd` Pointer to the address of the drawing list.

`Plane` Sets the elevation view as follows, unless `ZPlane` is 0 when Plane is ignored:

0: Plan
1: North
2: South
3: East
4: West
5: Underside

`ZPlane` 0 if projection is set to perspective. Otherwise combined with `ZPlaneScale`.

`ZPlaneScale` Combined with `ZPlane` (if non-zero) to give a zoom factor (`C_ZPlane<<C_ZPlaneScale`).

`NormAd` Pointer to an array that stores the normal chunk address for shapes.

`XDisOpts` Sets the following display options in the Shape Editor:

`E_XDOPOINTS` Point numbers are displayed.

`E_XDOPCONST` Construction lines are displayed between geometric anchor points.

`E_XDOPPNTNUM` Point numbers are displayed.

`E_XDOPCUBE` Bounding cube edges are displayed.

`E_XDOPSHOWBP` Bounding cube points are displayed.

`E_XDOPALLCON` All construction lines between geometric anchor points are displayed.

`E_XDOPFACNUM` Facet numbers are displayed.

`E_XDOPNODEPTH` Each point's color varies according to its distance from the viewpoint.

`E_XDOPMARKER` The point marker shape as follows:

0x0000 small x

0x0100 small .

0x0200 small +

0x0400 large x

0x0500 large .

0x0600 large +

`E_XDOPNOIMPORT` An object's children are displayed. This lets you to import an object and its children from the World Editor. Although you cannot edit the children in the Shape Editor, they may be useful in guiding shape design.

`E_XDOPNEWFACET` Decides how the next new facet is colored. Set as follows:

0x00000 Use next colour

0x20000 Alternate colours

0x40000 Light facet

`E_XDOPSHADESHAPE` The lighting calculation is made every frame so that all shapes in the Shape Editor are lit.

`PointsDisplay` List of points to display.

`PointsSelected` List of selected points.

T_GRRECT

Structure:

```
typedef struct
   {
   short             XPos,YPos,Width,Height;
   } T_GRRECT;
```

Description: Describes a rectangular area on the screen, as returned by `GrDrawText`.

`XPos`, **`YPos`** The position of the top left corner of the rectangle.

`Width`, **`Height`** The width and height of the rectangle.

T_GRSCAN

Structure:

```
typedef struct
   {
   short             YStart,Height;
   unsigned char     Colour,Illumination;
   long              *ZBuffer;
   short             ScanBuf[E_GRMAXSCANHEIGHT]
                     [E_GRMAXSCANDEPTH*2+1];
   } T_GRSCAN;
```

Description: Describes a scan buffer—a set of horizontal lines which can be used to draw polygons (using `GrDrawScan`) if the device does not support this itself.

`YStart`, **`Height`** The Y coordinate of the first line in the scan buffer and the number of scan lines to draw, respectively.

`Colour` The color in which to draw the lines.

`Illumination` Reserved - set to 255.

`ZBuffer` Reserved - set it to NULL.

`ScanBuf` An array of X coordinates for each line on the screen. Each line in the array is made up of the number of X coordinates, then each X coordinate in turn. As an example, if a line in the array had the following values:

```
    4,         100,       120,       200,       220
```

the device would interpret this as two horizontal lines between x=100 and x=120, and between x=200 and x=220. Complex shapes can be built up in this way.

T_GRSETUP

Structure:

```
typedef struct
   {
   short              Width,Height;
   long               NumCols;
   long               Abilities;
   short              BitsPerPixel;
   char               NumResolutions;
   unsigned char *    ScreenBuffer;
   void *             DriverInfo;
   short              NumConfigs, currentConfig;
   long *             ConfigFlags;
   char **            ConfigNames;
   } T_GRSETUP;
```

Description: A pointer to a structure of this type is returned from `GrSetup`.

`Width`, **`Height`** The width and height of the screen.

`NumCols` The number of available colors on screen at one time.

`Abilities` A set of flags. If set the following conditions apply to the device:

`E_ABILDITHER` Can dither colors.

`E_ABILFOGAVAIL` Can display fog effects

`E_ABILHOLES` Can draw holes in polygons.

`E_ABILHORIZON` Will draw the horizon facets.

`E_ABILMOVEMESS` Needs to be notified of window move messages.

`E_ABILPICK` Will handle picking facets from the screen.

`E_ABILPOLY` Can draw polygons.

`E_ABILSCRSWAP` Can swap foreground and background screens.

`E_ABILSMOOTH` Can smooth-shade facets with normals when lighting is active.

`E_ABILSORTED` Can draw a sorted drawing list.

`E_ABILTEXLINEAR` Can draw textures with linear interpolation.

`E_ABILTEXPERSP` Can draw perspective correct textures.

`E_ABILTRANSOBJ` Can transform a 3D model into a drawing list.

`E_ABILUNSORTED` Can draw an unsorted drawing list.

`E_ABILZBUF` Is Z-buffering; do not sort.

`E_ABILZAVAIL` Can Z buffer.

`E_ABILFOG` Can set fog.

`BitsPerPixel` The number of bits used to represent a single pixel on the screen.

`NumResolutions` The number of different resolutions that the device supports.

`ScreenBuffer` A pointer to the area of memory used to draw the screen.

`DriverInfo` Pointer to device specific information.

`NumConfigs` Number of available graphics device configurations.

`CurrentConfig` The current device configuration.

`ConfigFlags` A pointer to an array of flags, one entry for each available configuration:

`E_GRCOPYMASK` Specifies a mask for the screen copy flags.

`E_GRCOPYUNKNOWN` Unknown method used to copy back buffer to the screen.

`E_GRCOPYWING` WinG used to copy back buffer to the screen.

`E_GRCOPYDIRECTDRAW` DirectDraw used to copy back buffer to the screen.

`E_GRDRAWMASK` Specifies a mask for the screen draw flags.

`E_GRDRAWUNKNOWN` Unknown method used to draw into the back buffer.

`E_GRDRAWSUPERSCAPE` Superscape code used to draw into the back buffer.

`E_GRDRAWDIRECT3D` Direct3D used to draw into the back buffer.

`E_GRDRAWOPENGL` OpenGL used to draw into the back buffer.

`E_GRPROCMASK` Specifies a mask for the processing flags.

`E_GRPROCSUPERSCAPE` Superscape code used to transform 3D data.

`E_GRPROCDIRECT3D` Direct3D used to transform 3D data.

`E_GRPROCOPENGL` OpenGL used to transform 3D data.

`E_GRPROCHARDWARE` 3D graphics card used to transform 3D data.

`ConfigNames` Pointer to an array of graphics device names, as displayed on the Display Modes menu.

T_GRSPRINST
T_GRSPRINSTALL

Structure:

```
typedef struct
   {
   short            Width,Height;
   char             *SpriteData;
   } T_GRSPRINST;

   typedef struct
   {
   short            NumSprites;
   T_GRSPRINST      *SpriteTable;
   short            Flags;
   } T_GRSPRINSTALL;
```

Description: Passes sprite data to `GrSetSprites`.

`Width`, **`Height`** Mask with E_SPRMASK to get the width and height in pixels of the sprite data. The top 2 bits of `Width` and `Height` are used for Flag:

`E_SPRNOSAVE` Bit 15 of `Width`. If set the background behind the sprite is not saved if it moves.

`E_SPRHOTSPOT` Bit 14 of `Width`. If set the sprite has a hot spot.

`E_SPRPALETTE` Bit 15 of `Height`. If set the sprite uses its own palette.

Bit 14 of `Height` is not used.

`SpriteData` A pointer to a 1 byte per pixel rectangular pixel map.

`NumSprites` The number of sprites to install.

`SpriteTable` A pointer to an array of NumSprites sprite descriptions as defined above.

`Flags` A set of flags:

`E_SPRUSER` Set if these sprites are user sprites, reset if they are system sprites.

T_GRSPRITE

Structure:

```
typedef struct
   {
   short            XPos,YPos,Width,Height,Hotx,Hoty,Flags;
   char             *SaveBuffer;
   char             *SpriteData;
   } T_GRSPRITE;
```

Description: Defines the position and appearance of a sprite on the screen, as passed to `GrDrawSprite`.

`XPos`, **`YPos`** The position of the 'hotspot' of the sprite on the screen.

`Width`, **`Height`** The width and height of the sprite, respectively.

`Hotx`, **`Hoty`** The offsets of the hotspot from the top left corner of the sprite.

`Flags` Defines on which screens the sprite is displayed:

`E_SPRFORE`	Drawn on the foreground screen.
`E_SPRBACK`	Drawn on the background screen.

`SaveBuffer` A pointer to an area of memory in which to save the screen data behind the sprite.

`SpriteData` A pointer to the actual sprite data.

T_GRSPRITERET
T_GRSPRRETURN

Structure:

```
typedef struct
   {
   long              SaveSize;
   char              *SpriteData;
   } T_GRSPRRETURN;

   typedef struct
   {
   short             Error;
   T_GRSPRRETURN     *SpriteTable;
   } T_GRSPRITERET;
```

Description: These structures are returned by `GrSetSprites`.

`SaveSize` The size of the buffer (in bytes) required to save the background behind this sprite.

`SpriteData` A pointer to the converted sprite data. This may be an internal address on the graphics board.

`Error` Non-zero if the sprites were not installed correctly.

`SpriteTable` A pointer to an array of sprite descriptions as defined above.

T_GRSTIPPLES

Structure:

```
typedef struct
   {
   short             FirstCol,StipSize;
   unsigned char     *Stipples;
   } T_GRSTIPPLES;
```

Description: Describes a set of stipples, as passed to `GrSetStipples`.

`FirstCol`, **`StipSize`** The first stipple and number of stipples to set, respectively.

`Stipples` A pointer to the actual stipple data, organized as octets of bytes. The first four are the palette values for each pixel in the stipple, the second four are the alpha (transparency) values: `0x00` opaque, `0xFF` completely transparent.

T_GRTEXT

Structure:

```
typedef struct
   {
   short              XPos,YPos,Align;
   short              FontID,MaxLen;
   unsigned char      BGColour,FGColour;
   char               *String;
   unsigned char      Flags;
   } T_GRTEXT;
```

Description: Describes the position and appearance of text on the screen, as passed to `GrDrawText`.

`XPos`, **`YPos`** The position of the ‘hot spot’ of the text on the screen.

`Align` The position of the ‘hot spot’ relative to the position of the text, and can take the following values:

0 - Top left	1 - Top center	2 - Top right
4 - Center left	5 - Center	6 - Center right
8 - Bottom left	9 - Bottom center	10 - Bottom right

`FontID` The index of the font in which to display the text.

`MaxLen` The maximum number of characters to display, or a -1 if all characters are to be displayed.

`BGColour`, **`FGColour`** The background and foreground colors for the text respectively.

`String` Points to a null-terminated string to display. The carriage return character `\r` is interpreted as a line break. All other characters are printed “as is”.

`Flags` A set of flags defining where the text appears:

`E_SPRFORE`	Drawn on the foreground screen.
`E_SPRBACK`	Drawn on the background screen.

T_KBINSTALLRET

Structure:

```
typedef struct
  {
  short            Error,Id;
  char             Name[16];
  char             DialName[16];
  volatile char    *KeyMap;
  volatile short   Head,Tail;
  char             *AscTable;
  volatile char    ShiftState,LockState;
  } T_KBINSTALLRET;
```

Description: A pointer to a structure of this type is returned by `KbInstall`.

`Error` Non-zero if the Install routine failed for some reason.

`Id` The device ID number. This can be one of the following:

`E_KBIDSTANDARD`	Standard IBM keyboard.

`Name` The name of the device.

`DialName` The name of the dialog box used to edit this device's setup data (if any).

`KeyMap` A pointer to a matrix of keystates. Each key is assigned a byte made up of the following bits:

`E_KBKEYPRESS`	Key currently pressed.
`E_KBKEYDEPRESS`	Key pressed since bit last reset.
`E_KBKEYRELEASE`	Key released since bit last reset.
`E_KBKEYREAD`	Key has been read at least once.
`E_KBKEYSHIFT`	SHIFT was active when depressed.
`E_KBKEYCTRL`	CTRL was active when depressed.
`E_KBKEYALT`	ALT was active when depressed.

`Head`, **`Tail`** Head and tail indices into a circular buffer of the last key presses, used by `KbReadKey` and `KbKeyReady`.

`AscTable` Points to the keycode to ASCII translation tables. Each has 256 entries; the first is for unshifted keys, the second for shifted keys.

`ShiftState` Indicates the current state of the SHIFT keys:

`E_KBLSHIFT` Left SHIFT key pressed.

`E_KBRSHIFT` Right SHIFT key pressed.

`E_KBCTRL` CTRL key pressed.

`E_KBALT` ALT key pressed.

`E_KBAUX1` Reserved.

`E_KBAUX2` Reserved.

`E_KBAUX3` Reserved.

`E_KBAUX4` Reserved.

`LockState` Indicates the current state of the keyboard locks and PRINT SCREEN key:

`E_KBCAPSLOCK` CAPS LOCK active.

`E_KBNUMLOCK` NUM LOCK active.

`E_KBSCROLLLOCK` SCROLL LOCK active.

`E_KBPRINTSCREEN` PRINT SCREEN pressed.

T_KBKEYREC

Structure:

```
typedef struct
   {
   char              Ascii,KeyNum,ShiftState;
   } T_KBKEYREC;
```

Description: A pointer to a structure of this type is returned by `KbReadKey`. It describes the next pending keypress.

`Ascii` The extended ASCII code for the key.

`KeyNum` The keyboard scan code that generated the keypress.

`ShiftState` A copy of the shift state (as described above) as it was when the key was pressed.

T_KBSETUP

Structure:

```
typedef struct
   {
   short             Delay,Repeat,BaseRate;
   } T_KBSETUP;
```

Description: A pointer to a structure of this type is passed to `KbSetup`. It defines the delay and repeat of the auto-repeat feature.

Delay, **Repeat** The number of ‘ticks’ before a key starts to repeat, and then between subsequent repeats.

BaseRate The number of milliseconds per ‘tick’. If possible, this should be chosen so that `Repeat` is 1 tick, thus minimizing the interrupt load on the system.

T_MSINSTALLRET

Structure:

```
typedef struct
   {
   short             Error,Id;
   char              Name[16];
   char              DialName[16];
   } T_MSINSTALLRET;
```

Description: The mouse is controlled by Windows. This structure is included for compatibility with earlier versions of the SDK.

A pointer to a structure of this type is returned by `MsInstall`.

Error Non-zero if the Install routine failed for some reason.

Id The device ID number. This can be one of the following:

`E_MSIDMICROSOFT`	Microsoft mouse.

Name The name of the device.

DialName The name of the dialog box used to edit this device’s setup data (if any).

T_MSPOS

Structure:

```
typedef struct
   {
   volatile short              XPos,YPos,ButtonState;
   volatile unsigned char     PosFlag,ButtonFlag;
   T_TIMENODE                 *MouseTime;
   } T_MSPOS;
```

Description: The mouse is controlled by Windows. This structure is included for compatibility with earlier versions of the SDK.

A pointer to this structure is returned by `MsSetup`. It describes the current mouse position.

`XPos`, **`YPos`** The current X and Y positions of the mouse pointer.

`ButtonState` Reflects the current state of the mouse buttons: bit 0 is set if the left button is pressed, bit 1 if the right button is pressed, and bit 2 if the center mouse button pressed. Other bits are reserved.

`PosFlag`, **`ButtonFlag`** Set to 1 if the position or button state changes. These can be reset to 0 by the application and used to detect if the mouse status has changed.

T_MSSETUP

Structure:

```
typedef struct
   {
   short              XMin,YMin,XMax,YMax;
   short              XStart,YStart;
   } T_MSSETUP;
```

Description: The mouse is controlled by Windows. This structure is included for compatibility with earlier versions of the SDK.

A pointer to a structure of this type is passed to `MsSetup`. It defines the screen size and start position for the mouse.

`XMin`, **`YMin`**, **`XMax`**, **`YMax`** The minimum and maximum X and Y values for the rectangle within which the mouse must stay. Usually, minima are both 0, and the maxima are the width and height of the screen.

`XStart`, **`YStart`** The start coordinates for the mouse pointer.

T_NTINPUT

Structure:

```
typedef struct
   {
   short            User;
   short            Length;
   char             *Buffer;
   } T_NTINPUT;
```

Description: Describes function codes to be transmitted to other machines on the network, if fitted. It is passed to `NtNetwork`.

`User` The machine's user number.

`Length` The length (in bytes) of the data in the function code buffer.

`Buffer` A pointer to the buffer containing the function codes to be transmitted.

T_NTINSTALL

Structure:

```
typedef struct
   {
   short             UserNum, NumUsers;
   long              LongTimeOut, ShortTimeOut;
   long *            MonoTime;
   void **           WorldAddAdd;
   long              CheckRate;
   void              (*PrintTime)(short);
   void              (*PrintErr)(short,long);
   } T_NTINSTALL;
```

Description: Describes the installation parameters for the network, if fitted. It is passed to `NtInstall` using a `T_DEVINSTALL` structure.

`UserNum` The machine's user number.

`NumUsers` The number of users on the network.

`LongTimeOut` The initial startup delay in milliseconds (usually 30 000).

`ShortTimeOut` The maximum delay between successive frames, in milliseconds (usually 1500).

`MonoTime` Points to the monotonic timer (which is updated under interrupt). This gives the basis for deciding on timeouts.

`WorldAddAdd` The address of the pointer to the world buffer. This is passed to the network device to allow the checksum operation.

`CheckRate` The number of frames between subsequent checksum operations (usually 100).

`PrintTime` The address of a routine to print the time remaining during the startup of the network it is passed the time remaining in milliseconds.

`PrintErr` The address of a routine to print network checksum errors. It is passed the user number and magnitude of the error.

T_NTINSTALLRET

Structure:

```
typedef struct
   {
   short             Error,Id;
   char              Name[16];
   char              DialName[16];
   } T_NTINSTALLRET;
```

Description: A pointer to a structure of this type is returned by `NtInstall`.

`Error` Non-zero if the Install routine failed for some reason.

`Id` The device ID number. This can be one of the following:

`E_NTIDCLARKSON`	Clarkson drivers
`E_NTIDNETBIOS`	NetBios

`Name` The name of the device.

`DialName` The name of the dialog box used to edit this device's setup data (if any).

T_NTOUTPUT

Structure:

```
typedef struct
   {
   short             NumUsers;
   T_NTINPUT         Buffers[1];
   } T_NTOUTPUT;
```

Description: Describes the functions received from every machine on the network, as returned by `NtNetwork`.

`NumUsers` The number of users on the network.

`Buffers` An array of function buffer descriptions, as defined above, one for each machine on the network.

T_NTSETUP

Structure:

```
typedef struct
   {
   short             Length;
   long              Seed;
   } T_NTSETUP;
```

Description: Describes the functions received from every machine on the network, as returned by `NtNetwork`.

`Length` The length of the data passed to setup.

`Seed` The seed value for the random number generator.

T_PCXHEADER

Structure:

```
typedef struct
  {
  unsigned char     Manufacturer, Version,Encoding,
                    BitsPerPixel;
  unsigned short    XMin, YMin, XMax, YMax, HRes, VRes;
  unsigned char     ColourMap[48];
  unsigned char     Reserved, NPlanes;
  unsigned short    BytesPerLine, PaletteInfo,
                    HScreenSize, VScreenSize;
  unsigned char     Filler[54];
  } T_PCXHEADER;
```

Description: Defines the fields in the header of a PCX format file.

`Manufacturer` A constant reflecting the source of the PCX file (should be `0x0A`).

`Version` One of the following, showing which exact format is being used (should be `0x05`):

0	Version 2.5 of PC Paintbrush
2	Version 2.8 with palette information
3	Version 2.8 without palette information
4	PC Paintbrush for Windows
5	Version 3.0 and above of PC Paintbrush/+, including Publisher’s Paintbrush

`Encoding` Always 1, for run length encoding.

`BitsPerPixel` The number of bits used to represent a single pixel, and should be 8.

`XMin`, **`YMin`**, **`XMax`**, **`YMax`** The size and position of the image. These are used to calculate the actual size of the image to display.

`HRes`, **`VRes`** The horizontal and vertical resolution of the picture, in dots per inch (usually ignored).

`ColourMap` A 16 color palette. This is ignored for a 256 color image. In this case the palette is stored as the last 768 bytes of the file, preceded by a single `0x0C` byte.

`Reserved` Should always be 1.

`NPlanes` The number of separate planes of information making up the image.

For 256 color images this is 1.

`BytesPerLine` The amount of memory to allocate for a scan line buffer, and is always even.

`PaletteInfo` Not used.

`HScreenSize`, **`VScreenSize`** The size of the screen in pixels.

`Filler` Fills out the header to 128 bytes. All these values should be 0.

T_PRAXES

Structure:

```
typedef strct
   {
   unsigned char    Shift[6];
   unsigned char    Deadzone [6];
   unsigned char    MaxAxes;
   } T-PRAXES;
```

Description: A structure of this type is passed to proportional devices as part of their device specific setup data (`T_PRINSTALL.Sense`).

`Shift` An array of sensitivity values as set in the Proportional Control Setup dialog box, one for each axis.

`DeadZone` An array of deadzone values as set in the Proportional Control Setup dialog box, one for each axis.

The six axes are:

0	X translation
1	Y translation
2	Z translation
3	X rotation
4	Y rotation
5	Z rotation

`MaxAxes` The number of axes that are allowed to return non-zero values at any one time.

T_PRINSTALL

Structure:

```
typedef struct
   {
   char              Spare1 [16];
   char              CtrlName [16];
   char              Spare2 [2];
   T_PRAXES          Sense [3];
   unsigned char     Spare3 [7]
   } T_PRINSTALL;
```

Description: A structure of this type is passed to the proportional device during installation.

Spare1, **Spare2**, **Spare3** Not used.

CtrlName The name of the control type in use.

Sense An array of up to 3 `T_PRAXES` structures which are used to set the sensitivity and deadzones for each device attached to the driver.

T_PRINSTALLRET

Structure:

```
typedef struct
   {
   short             Error,Id;
   char              Name[16];
   char              DialName[16];
   } T_PRINSTALLRET;
```

Description: A pointer to a structure of this type is returned from `PrInstall`.

`Error` Non-zero if the device did not install properly.

`Id` A unique identification number for this device. It can be one of the following:

`E_PRIDSBALL:`	Spaceball
`E_PRIDJOYSTICK:`	Joystick
`E_PRID2STICK:`	2nd joystick
`E_PRIDFLOB:`	Ascension Flock of Birds tracker
`E_PRIDFASTRAK:`	Polhemus FASTRAK tracker
`E_PRIDSMOUSE:`	Spacemouse
`E_PRIDMOUSE:`	Proportional mouse control
`E_PRID6MOUSE:`	Logitech 3D Mouse

`Name` The name of this device.

`DialName` The name of the dialog box to use to edit the configuration data of this device (if any).

T_PRPOS

Structure:

```
typedef struct
   {
   long              Buttons;
   short             Axis[];
   } T_PRPOS;
```

Description: Describes the position of the proportional device, as returned by `PrGetPos`.

`Buttons` A set of bits reflecting the state of up to 32 buttons on the device. Button 0 is the least significant bit, up to button 31 as the most significant.

`Axis` An array of proportional values, one for each axis on the device, reflecting the position of each axis.

T_PRSETUP

Structure:

```
typedef struct
   {
   short             NumAxes;
   short             NumButtons;
   } T_PRSETUP;
```

Description: Describes the proportional device, as returned by `PrSetup`.

`NumAxes` The number of axes supported by the device.

`NumButtons` The number of buttons supported by the device.

T_SDINSTALLRET

Structure:

```
typedef struct
   {
   short              Error,Id;
   char               Name[16];
   char               DialName[16];
   } T_SDINSTALLRET;
```

Description: A pointer to a structure of this type is returned by `SdInstall`.

`Error` Non-zero if an error occurred during installation.

`Id` A number reflecting the type of sound device:

`E_SDIDMIDI`	MIDI card.
`E_SDIDADLIBGOLD`	AdLib Gold card.
`E_SDIDSOUNDBL`	Sound Blaster card.
`E_SDIDSOUNDBL16`	SoundBlaster 16 card.

`Name` The name of the device.

`DialName` The name of the dialog box used to edit the device’s setup data (if any).

T_SOUNDMODIFY

Structure:

```
typedef struct
{
long                        Handle;
unsigned short              Command;
long                        Value;
} T_SOUNDMODIFY;
```

Description: Specifies the data required to modify or query an aspect of a sound as it is playing or being recorded.

`Handle` The handle of the sound to modify or query (as supplied from `SdPlaySound` or `SdRecord`).

`Command` Specifies which aspect of the sound to modify or query. It may have one of the following values:

- `E_SMODLENGTH` Sets the number of frames (samples) to play.
- `E_SMODPITCH` Modifies the pitch of the sound (play only).
- `E_SMODPAN` Modifies the pan position of the sound (play only).
- `E_SMODVOL` Modifies the volume of the sound (play only).
- `E_SMODLOOP` Modifies the looping state of a sound (play only).
- `E_SMODQUERY` When added to the above values, the playing sound is not modified.

`Value` The value which is placed in the sound. If `E_SMODQUERY` is specified, this is ignored.

T_SOUNDRECORD

Structure:

```
typedef struct
  {
  short           Type:
  long            Length;
  char            *Buffer;
  } T_SOUNDRECORD;
```

Description: A structure of this type is passed to the `SdRecord` function.

`Type` The type of sound sample required. It can be:

`E_STSAM8`	8 bits
`E_STSAM16`	16 bits

`Length` The number of samples to record.

`Buffer` A pointer to an area of memory used to store the recorded samples. Remember that 16 bit samples are 2 bytes in size so that the required length of this buffer would be `Length` multipied by 2.

T_SRCALLBACK

Structure:

```
typedef struct
   {
   unsigned char     RChar;
   unsigned char     Status;
   } T_SRCALLBACK;
```

Description: A pointer to a structure of this type is passed to the serial driver's callback routine.

RChar The received character (if valid).

Status The reason that the callback function was invoked, and is one of the following:

E_SRSMODEM	Modem status error.
E_SRSEMPTY	Transmitter buffer empty error.
E_SRSRECVD	Character received (RChar valid).
E_SRSLINE	Line status error.

Using callback functions from the serial device driver may cause a lockup if the callback uses certain machine registers—mainly EBX. You can overcome this problem by instructing the compiler to save any registers it uses during the callback routine. In Watcom C you can use the compiler directive #pragma to specify exactly what is required to call a particular function, and what registers the function can use. For example, add a #pragma directive just before the callback function prototype as follows:

```
#pragma aux CallBackFunction modify [] ;
void CAllBackFunction(T_SRCALLBACK *p);
    /*     other code here          */
void CallBackFunction(T_SRCALLBACK *p)
{
    /*      callback routine here   */
}
```

If necessary, the name of the callback function should be modified in both the #pragma and the function prototype, as well as in the function definition itself. The spaces in the #pragma definition are significant—do not remove them.

T_SRDATA

Structure:

```
typedef struct
   {
   short             PortNumber;
   short             NumBytes;
   char              *Buffer;
   } T_SRDATA;
```

Description: A pointer to a structure of this type is passed to the serial drivers transmit and receive routines.

PortNumber The number of the serial port to use (0=COM1, 1=COM2, and so on).

NumBytes The number of bytes to transmit or receive.

Buffer The address of a buffer in which the data is stored.

T_SRINSTALLRET

Structure:

```
typedef struct
   {
   short             Error,Id;
   char              Name[16];
   char              DialName[16];
   } T_SRINSTALLRET;
```

Description: A pointer to a structure of this type is returned from `SrInstall`.

Error Non-zero if the device did not install properly.

Id A unique identification number for this device. It can be one of the following:

E_SRIDDEFAULT	Default serial device

Name The name of this device.

DialName The name of the dialog box to use to edit the configuration data of this device.

T_SRPORTRET

Structure:

```
typedef struct
   {
   short              Error;
   short              PortAddress;
   char               IRQNumber;
   char               IntNumber;
   char               IRQMask;
   } T_SRPORTRET;
```

Description: A pointer to a structure of this type is returned from the serial device get port info routine.

Error Non-zero if the port was not in use.

PortAddress The base I/O port address for the serial port.

IRQNumber The IRQ line on which the interrupt appears from this port.

IntNumber The interrupt vector number called by the interrupt.

IRQMask The interrupt enable mask required by the interrupt from this port.

T_SRRELPORT

Structure:

```
typedef struct
   {
   short              PortNumber;
   } T_SRRELPORT;
```

Description: A pointer to a structure of this type is passed to the serial device release port routine.

PortNumber The number of the serial port to use (0=COM1, 1=COM2, and so on).

T_SRRETURN

Structure:

```
typedef struct
   {
   short             Error;
   } T_SRRETURN;
```

Description: A pointer to a structure of this type is returned from most serial device routines.

`Error` An error code; 0 for OK, or a negative error number.

T_SRSETCALLBACK

Structure:

```
typedef struct
   {
   short             PortNumber;
   void              *Routine;
   } T_SRSETCALLBACK;
```

Description: A pointer to a structure of this type is passed to the serial device `SetCallBack` routine.

`PortNumber` The number of the serial port to use (`0`=COM1, `1`=COM2, and so on).

`Routine` The address of a routine to call every time a byte is received from that port.

T_SRSETPORT

Structure:

```
typedef struct
   {
   short              PortNumber;
   short              BaudRate;
   char               Control;
   } T_SRSETPORT;
```

Description: A pointer to a structure of this type is passed to the serial device set port routine.

PortNumber The number of the serial port to use (0=COM1, 1=COM2, and so on).

BaudRate The baud rate to use (up to 9600).

Control A value to place into the line control register for this port. This is made up from combinations of the following values:

E_SRLBITS5	Use 5 bits per character
E_SRLBITS6	Use 6 bits per character
E_SRLBITS7	Use 7 bits per character
E_SRLBITS8	Use 8 bits per character
E_SRLSTOP1	Use 1 stop bit
E_SRLSTOP15	Use $1^1/_2$ stop bits
E_SRLPARITYN	No parity checking
E_SRLPARITYE	Even parity
E_SRLPARITYO	Odd parity
E_SRLPARITYS	“Stick” parity
E_SRLSBRK	Send “break” signal

T_TMINSTALLRET

Structure:

```
typedef struct
   {
   short            Error,Id;
   char             Name[16];
   char             DialName[16];
   } T_TMINSTALLRET;
```

Description: A pointer to a structure of this type is returned by `TmInstall`.

`Error` Non-zero if an error occurred during installation.

`Id` A number reflecting the type of timer device:

`E_TMIDSTANDARD:`	Standard timer device

`Name` The name of the device.

`DialName` The name of the dialog box used to edit the device's setup data (if any).

T_TIMENODE

Structure:

```
typedef struct
   {
   long             Rate;
   volatile long    Counter;
   void             *NextNode;
   void __vrtcall   (*Function)(void);
   } T_TIMENODE;
```

Description: The timer node structure is used to keep track of events to be called by the timer device at regular intervals. It is passed to `TmAddList` and `TmRemList`.

`Rate` The number of milliseconds between each call to the event routine.

`Counter` The number of milliseconds left until the next call to the event routine, and is set up by the timer device.

`NextNode` A pointer to the next node in the list, or `NULL` if none, and is set up by the timer device.

`Function` A pointer to a routine to be called by the timer device when the timer event happens. This acts like a normal C routine, but care must be taken to ensure that it does not take too much time. If this is still executing when the next timer event occurs, interrupts will overrun and the machine will probably hang.

Data converter

The main element of the data converter is the `T_ELEMENT` structure which stores information about the current shape being processed. It is also used when an object is created in the world.

Elements are generated by filling in a list of facet definitions for the element in `CnvFacet` and then calling `Conv_Element()`, which performs all the requested functions on the facet definitions. The facets used within the intermediate format have a maximum of four points.

An element may have children, which may in turn have children, in the same way as the tree structure used in the Superscape data format. A list of children is stored on the element using `T_Child`. When an instance of the element is made in the world, the converter scans the child list and creates any children as necessary.

All entities within the intermediate format are associated with a layer using `CnvLayer`. Layers must be generated before any elements are created, as the converter uses them to get color information about certain entities. Therefore, it is best to generate the layer table during pass 1 or at the beginning of pass 2.

T_ELEMENT

Structure:

```
typedef struct
  {
  char                Name[256];
  long                Colour;
  short               SubObCnt;
  T_CNVSUBOBJECT      *SubObjects;
  double              OriginX;
  double              OriginY;
  double              OriginZ;
  double              Factor_X;
  double              Factor_Y;
  double              Factor_Z;
  double              Max_X;
  double              Max_Y;
  double              Max_Z;
  double              Min_X;
  double              Min_Y;
  double              Min_Z;
  double              Insert_X,Insert_Y,Insert_Z;
  double              Scale_X,Scale_Y,Scale_Z;
  double              Rotation_X,Rotation_Y,Rotation_Z;
  short               Flags;
  char                Layer[256];
  short               NumChildren;
  struct              T_CHILD *Children;
  char                UserData[128];
  long                NumFacets;
  short               *FacetCols;
  } T_ELEMENT;
```

Description: Stores information about the shape being processed.

Name The name of the shape, this is used when creating a shape and also when creating an instance of the shape in the world.

Colour The base color of the object as an index into the current palette.

SubObCnt The number of subobjects this element is made up of. This is for system use only and is filled in by the function `Conv_Element()`. Subobjects are used to store information about `T_ELEMENT` structures made up of many shapes, such as a `T_ELEMENT` which has been cut with the Cookie Cutter.

SubObjects A pointer to the actual `T_SUBOBJECT` definitions for this element.

`OriginX/Y/Z` The placement offset of the element. Used when creating an instance of this element in the world, this offset value is at the specified insert position.

`Factor_X/Y/Z` The size factor of the element. This is for system use only. It is used by the internal converter functions to make sure that any slack space in a shape is filled during shape creation, in a similar manner to the Wrap function in the Shape Editor.

`Max_X/Y/Z` The maximum point coordinate in this element. Filled in by the VRT based on the facet list and any children.

`Min_X/Y/Z` The minimum point coordinate in this element. Filled in by the VRT based on the facet list and any children.

`Insert_X/Y/Z` The insert position for creating an instance of the element. Filled in by the converter module, prior to calling `Conv_MakeWorldObject()`.

`Scale_X/Y/Z` Relative scale for creating an instance of the element. Filled in by the converter module prior to calling `Conv_MakeWorldObject()`.

`Rotation_X/Y/Z` Rotation values for creating an instance of the element. Filled in by the converter module prior to calling `Conv_MakeWorldObject()`.

`Flags` Flags for this element. Valid values are:

> `E_CNVCUBES` Do not convert shapes, but represent each converted entity as a cube.
>
> `E_CNVCUT` Cut with the cookie cutter
>
> `E_CNVLGHT` Light source
>
> `E_CNVINVALID` Invalid element which should not be processed
>
> `E_CNVSELECTED` Process during Pass 2
>
> `E_CNVDIRCHECK` Perform facet direction calculations
>
> `E_CNVASSOCS` Perform facet association calculations

`Layer` Layer name for the current element. Used during coloring to get the required color if the color method for the facets is BYLAYER.

`NumChildren` The number of child elements in the current element.

`Children` Pointer to the list of child elements for this element. These values are used to specify nested elements, and must be filled in with valid values by the converter module.

`UserData` Any user specific data associated with the element.

`NumFacets` Total number of facets in all subobjects in this element. Filled in by the system during `Conv_Element()`.

FacetCols Used by the system to store facet colors for the shape and lighting values for each facet. Once the facet list for an element has been converted into a shape it is then not easy to get lighting values for the individual facets as the facet list is destroyed. The converter must not modify this value.

See also: T_CNVFACET, T_CNVPOINT, T_CHILD, T_CNVLAYER

T_CHILD

Structure:

```
typedef struct
  {
  struct          T_ELEMENT *ChildElement;
  double          Child_X, Child_Y, Child_Z;
  double          Child_Scale_X;
  double          Child_Scale_Y;
  double          Child_Scale_Z;
  double          Child_Rot_X, Child_Rot_Y, Child_Rot_Z;
  char            UserData[128];
  } T_CHILD;
```

Description: Lists the children stored on the element.

ChildElement A pointer to the T_ELEMENT structure which is a child.

Child_X/Y/Z 3-D position of the child relative to the origin of the parent T_ELEMENT.

Child_Scale_X/Y/Z The relative scale of the child T_ELEMENT.

Child_Rot_X/Y/Z The 3-D rotation of the child T_ELEMENT.

UserData Any user specific data associated with the element.

See also: T_ELEMENT, T_CNVFACET, T_CNVPOINT, T_CNVLAYER

T_CNVFACET

Structure:

```
typedef struct
   {
   char              NumPoints;
   short             Special;
   long              Colour;
   T_CNVPOINT        Points[1];
   } T_CNVFACET;
```

Description: Defines a list of facets for the element.

NumPoints The number of valid points.

Special Not used.

Colour The color of the facet.

Points The 3-D position of the points making up the facet.

See also: T_ELEMENT, T_CNVPOINT, T_CHILD, T_CNVLAYER

T_CNVLAYER

Structure:

```
typedef struct
   {
   char              Name[32];
   long              Flags;
   short             Colour;
   short             LineType;
   } T_CNVLAYER;
```

Description: Defines a layer and its color.

Name Layer name up to 32 characters.

Flags Not used.

Colour Layer color.

LineType Not used.

See also: T_ELEMENT, T_CNVFACET, T_CNVPOINT, T_CHILD

T_CNVPOINT

Structure:

```
typedef struct
   {
   short            PointNo;
   double           X;
   double           Y;
   double           Z;
   } T_CNVPOINT;
```

Description: **PointNo** The point number used in the Superscape representation of this shape. It is filled in by `Conv_Element()` during conversion.

X/**Y**/**Z** 3-D position of the point relative to the `T_ELEMENT` origin.

See also: `T_ELEMENT`, `T_CNVFACET`, `T_CHILD`, `T_CNVLAYER`

Miscellaneous data

In addition to the main data areas described above, there are various data structures and defines used by VRT for many different purposes. These are detailed on the following pages.

T_APISHARE

Structure:

```
typedef struct
   {
   char             *Name;
   char             *Description;
   void             * __vrtcall (*CallBack)
                         (long Reason, void *Data);
   } T_APISHARE;
```

Description: The shared information data structure that allows you to communicate between application modules that are loaded at the same time. Data is passed to `C_RegisterShare`, and returned by `C_GetShare`.

`Name` A pointer to the name by which the application wishes to be known. This should be unique. To ensure this, attempt to find another application with the same name (using `GetShare`) before registering. If the search succeeds, then a module with this name already exists. You can then take appropriate action (usually returning with an error is sufficient).

`Description` An optional description of the purpose of the application module. If not required, set this to `NULL`.

`CallBack` The address of a function which can be called by other application modules. It takes two parameters: a reason why it was called (so it knows what action to perform), and a pointer to some data. It returns a pointer to some result data, or `NULL` if the reason was unknown. It is good practise to always return a non-`NULL` pointer from a successful completion, even if it is only a pointer to a zero error value. `CallBack` may be `NULL` if the only reason for registration is to avoid multiple instances of this application being loaded at once. For this reason, always check that this field is non-`NULL` before attempting to call it.

T_COMPILEREC

Structure:

```
typedef struct
   {
   char              Name[10];
   short             OpCode;
   char              InOut,Action;
   unsigned short    Type;
   unsigned char     RetType;
   unsigned char     ArgType[10];
   } T_COMPILEREC;
```

Description: Describes the name of an SCL instruction, and information for the compiler about how to compile and decompile it. This is only ever used as an argument to `RegisterSCL`.

`Name` The name of the instruction, up to 8 characters long. If less than 8 characters, it is padded with \0 characters.

`OpCode` The preferred object code representation of the instruction (`0x400` to `0x4FF`), or `0` if no preference. If a function is already registered using the preferred `opcode` or the `opcode` is invalid, `RegisterSCL` will allocate a different `opcode` number. This can be checked against the preferred code to see of the registration succeeded.

`InOut` Describes how many inputs and outputs there are from this function. The top 4 bits are the number of input values, the bottom 4 bits the number of outputs (0 or 1). Thus a function with three inputs and one output would have an `InOut` value of `0x31`.

`Action` Sets up special compile-time actions, and is not meaningful in the API. It must be set to 0.

`Type` Indicates the type of this instruction. This can be:

`E_CONSTANT` A pointer to a constant value.

`E_POINTER` A pointer that is directly used as such by SCL.

`E_PROCEDURE` A procedure which returns a value direct.

`E_VARIABLE` A variable which returns the address of a variable, and is treated as one).

`RetType` The type of value returned by this instruction (if any).

`ArgType` An array of types of each argument taken by this instruction, in order. The available types are:

`E_SSINTEGER` Integer, float or fixed value.

`E_SSOBJNUM` Object number.

`E_SSPOINTER` Constant pointer.

`E_SSPOINTER+E_SSWRITE` Writable pointer (variable address).

`E_SSRPOINTER` Explicit pointer.

T_CONMSG

Structure:

```
typedef struct
   {
   long               Message;
   unsigned short     Sender, Destination;
   void *             Next;
   } T_CONMSG;
```

Description: Describes the buffer for SCL inter-object messages.

`Message` The message to send to the destination object.

`Sender` The object that is sending the message.

`Destination` The object that is receiving the message

`Next` A pointer to the next message in the buffer.

T_FUNCPTR

Structure: **`typedef void *(*T_FUNCPTR)(void *);`**

Description: Defines a pointer to a C function which takes and returns a pointer.

T_STDFUNCPTR

Structure: **`typedef void* __vrtcall(*T_STDFUNCPTR)(void*);`**

Description: Defines a pointer to a C function which takes and returns a pointer. This is used in the device drivers, since all the device driver functions are of this type.

T_SCLFUNCPTR

Structure: **`typedef void (*T_SCLFUNCPTR)(void);`**

Description: Defines a pointer to an SCL function either as registered by the SDK application, or to an internal SCL function in `C_SCL`.

T_FUNCNAME

Structure:

```
typedef struct
   {
   long              Function;
   short             Name;
   short             Flags;
   } T_FUNCNAME;
```

Description: Used in the table of function names.

`Function` The actual function number.

`Name` The system message number associated with the function.

`Flags` A set of flags, as follows:

`E_FNCV` Set if the function is valid in Visualiser and Viscape.

`E_FNCW` Set if the function is valid in the World Editor.

`E_FNCS` Set if the function is valid in the Shape Editor.

`E_FNCL` Set if the function is valid in the Layout Editor.

`E_FNCN` Set if the function is valid in the Sound Editor.

`E_FNCI` Set if the function is valid in the Image Editor.

`E_FNCR` Set if the function is valid in the Resource Editor.

`E_FNCK` Set if the function is valid in the Keyboard Editor.

`E_FNCALL` Set if the function is valid in all editors.

`E_FNCH` Set if the function is to be entered in the history buffer.

`E_REPEATFUNC` Set if the function can be repeated using the center mouse button.

T_FUNCTION

Structure:

```
typedef struct
   {
   long              Function;
   short             PropVal;
   } T_FUNCTION;
```

Description: When functions (as generated by the keyboard, Spacemouse, or similar device) are processed, they are placed in a buffer. Each entry in the buffer has this structure.

`Function` The actual function number.

`PropVal` In the case of a proportional device, a value reflecting the position of a particular axis, from 1 to 32767. For function codes where this is not relevant, or where they have been generated by a non-proportional device, this field is 0.

T_FVECTOR

Structure:

```
typedef struct
   {
   float             x,y,z;
   } T_FVECTOR;
```

Description: This is a three-dimensional floating point vector. It can also be used to specify a point in an unrestricted area of 3D space.

T_IMAGEINFO

Structure:

```
typedef struct
   {
   unsigned char     *Data;
   unsigned char     *Palette;
   unsigned short    Width,Height,Pitch;
   unsigned char     BitsPerPixel;
   unsigned char     Id;
   } T_IMAGEINFO;
```

Description: VRT supports different graphics file formats which are handled, by default, by the image library supplied with it. You can use this library from within your SDK applications, or replace it with an SDK application if you want to use other graphic formats.

The image library is loaded on demand. To make sure that it is loaded call `ImageLoadLibrary`. Saving and loading is mediated using five vectors: `C_ImageAlloc`, `C_ImageFree`, `C_ImageLoad`, `C_ImagePalette`, `C_ImageSave`. If you want to write an application to load a new image file format, it must replace all of these vectors together to eliminate the problem of one application allocating memory while another tries to deallocate it. If any of these vectors are `NULL`, the image library is not yet available, and should be loaded using `ImageLoadLibrary`.

`Data` Points to the image data. This is arranged as a rectangular array of pixels, `Pitch` pixels across and `Height` pixels high. The amount of data in each pixel is governed by `BitsPerPixel`.

`Palette` Points to the palette information for the picture, organized as an array of 8-bit RGB values.

`Width`, **`Height`** The size of the image in pixels.

`Pitch` The width of the rectangle used to store the image. This is always at least as large as `Width`, but may be more to meet alignment in memory.

`BitsPerPixel` The number of bits used to represent each pixel.

`Id` A handle used internally by the image library.

T_LONGORPTR

Structure:
```
typedef union
   {
   long              l;
   void              *p;
   short             w[2];
   char              b[4];
   unsigned short    uw[2];
   unsigned char     ub[4];
   float             f;
   } T_LONGORPTR;
```

Description: This union is used wherever different data types must be accommodated in the same space (for example, the SCL calculation stack must be able to hold pointers, longs and floats). The union defines a set of mutually residing data types.

T_LONGVECTOR

Structure:
```
typedef struct
   {
   long              x,y,z;
   } T_LONGVECTOR;
```

Description: This is a three-dimensional vector. It can also be used to specify a point in an unrestricted area of 3D space.

T_MATRIX

Structure: `typedef short T_MATRIX[3][3];`

Description: This is a 3x3 transformation matrix, used by VRT to calculate rotations and viewpoint transformations. It is rescaled so that a value of 1.0 is represented by 16384 in the matrix. This means that the identity matrix is:

16384	0	0
0	16384	0
0	0	16384

T_OBJFAC

Structure:

```
typedef struct
{
short                Object,Facet;
} T_OBJFAC;
```

Description: A structure containing the object number and facet number within that object, returned by `FindFacet` when looking for objects under the mouse pointer.

T_PICKITEM

Structure:

```
typedef struct
   {
   char              *Name;
   T_LONGORPTR       Value;
   char              Exclude;
   } T_PICKITEM;
```

Description: A list of `T_PICKITEM`s is passed and returned to the VRT routine `Pick`, which displays a dialog box allowing you to select one or several items from a displayed list.

`Name` The name to display for this item in the dialog box.

`Value` A user-defined value relating to the item. It could be a chunk type, a pointer to an object, or any other value that is convenient to represent this item.

In addition, there are several related constants defining what is to be selected in the pick list (See `Pick`):

`E_PICKFILTER`	Show ‘filter’ button.
`E_PICKONE`	Pick only one item from the list.
`E_PICKSOME`	Pick any number of items from the list.
`E_PICKNOSORT`	Do not sort the list.
`E_PICKSORT`	Sort the list alphabetically.

`Exclude` Excludes or includes an item from the list. Set to 0 if you want to include the item in the list, or 1 to exclude it. This value must be included so that the structure is 9 bytes long.

T_POINTREC2D

Structure:

```
typedef struct
   {
   short             x,y;
   } T_POINTREC2D;
```

Description: This structure defines a point in 2D space, and is usually used for specifying positions on the screen.

T_POINTREC3D

Structure:

```
typedef struct
   {
   short             x,y,z,uz;
   short             ScaleFactor;
   short             Spare1,Spare2,Spare3;
   } T_POINTREC3D;
```

Description: Hold definitions of points in 3D Its additional fields are used by the processing routines for storage of additional information to speed up the processing.

x, **y**, **z** The x, y, z coordinates of the point after normalization and scaling.

uz The z coordinate before scaling is applied.

ScaleFactor A shift value used in the normalization process. The actual coordinate of the point is x,y,z shifted left by `ScaleFactor`.

Spare1, **Spare2**, **Spare3** Not used.

T_VECTOR

Structure:

```
typedef struct
   {
   short             x,y,z;
   } T_VECTOR;
```

Description: A short three-dimensional vector, that can also be used to specify a point in a restricted area of 3D space.

T_VPENTRY

Structure:

```
typedef struct
   {
                      unsigned short  ObjNum,VPNum;
   } T_VPENTRY;
```

Description: Defines the object and viewpoint number which are used by a given user in the VRT.

ObjNum The number of the controlled object for that user.

VPNum The viewpoint the user is controlling.

T_SCLFILE

Type: **E_SCLFILE**

Structure:

```
typedef struct
   {
   int                URL;
   int                Browser;
   } T_SCLFILE;
```

Description: Dictates if a file can be displayed in a browser.

URL True if the file pointed to by `C_SCLFile` is a URL, or false if not.

Browser True if the target browser is open, or false if it is not.

T_THUMBNAIL

Type: **E_CTTHUMBNAIL**

Structure:

```
typedef struct
   {
   unsigned short    ChkType, Length;
   unsigned short    ImageLen, MaskLen;
   unsigned short    ImgEsc, MaskEsc;
   unsigned short    UncompLen;
   unsigned short    Spare[3];
   unsigned short    Data[1];
   } T_THUMBNAIL;
```

Description: Stores the thumbnail image used by VCA files in the Drag ‘n’ Drop Warehouse.

The compression routine uses Run Length Encoding on the bitmap data stored in each Device Independent Bitmap (DIB). A sequence of 3 or more repeated words is replaced with `<esc><k><val>` where `esc` is `ImgEsc` or `MaskEsc`, `k` is the repeat count, and `val` is the repeated value. When decompressed, two standard DIBs will result. The first is a 256 color thumbnail image, and the second is a monochrome mask where a value of 1 indicates a background pixel and 0 a foreground pixel.

ChkType The chunk type (`E_CTTHUMBNAIL`).

Length The length of the chunk including `ChkType` itself.

ImageLen The number of bytes occupied by the image DIB when uncompressed.

MaskLen The number of bytes occupied by the mask DIB when uncompressed.

ImgEsc The value used when compressing the image DIB.

MaskEsc The value sued when compressing the mask DIB.

UncompLen The total length of the chunk when uncompressed.

Spare Spare

Data Image DIB (compressed) followed by mask DIB (compressed).

Chapter 8 - Data converter

Introduction

VRT includes a data converter module which consists of a number of functions that convert the local data format into a rich intermediate format. This data is modified to conform to Superscape data format restrictions and the user specified requests. The intermediate data format is very similar to the Superscape structure but with a lot more information and a greater degree of accuracy.

The data converter module is written mainly for converting data from three dimensional based formats, but it could also be used to convert other data formats—for example you could write a converter module that converts bitmap images into outline shapes.

Intermediate data format

The basic format of the intermediate data consists of a `T_ELEMENT` definition for each shape or item in the data file. The converter must create the element by calling `Conv_NewElement()`, which returns a pointer to the newly created element.

The converter fills in a number of `T_CNVFACET` structures for each facet related to the shape or item. This list can easily be created using the `Conv_NewFacet()` function which returns a pointer to the facet. The number of points in a facet is passed as an argument to `Conv_NewFacet()`, and must have a color related with it. If the color is zero then the color for the facet comes from the color of its related element; if it is 256 the color is taken from the layer.

Once the list of facets is complete a call to the `Conv_Element()` function converts the facets into a shape, and performs any necessary cutting and lighting on the shape (or shapes) created.

If a nested data structure is used, other elements can be referred to as children of an element by calling `Conv_NewChild()` for the parent element, and filling in the `T_CHILD` structure that is returned. The child structure must contain the address of a valid element to use as the child, and relative position within the parent element to place it. As well as a position there is space for rotation and scale information in the child structure.

When the function `Conv_Element()` has been called to generate the shape (or shapes) needed for the element definition, an instance of this element in the world can be created by getting a pointer to the relevant element using the `Conv_FindElement()` call, and filling in the position, scale, rotation and color information, and then call `Conv_MakeWorldObject()`. This function creates all the objects required for a complete instance of the element, including any children. Any number of instances of an element may be created in this way, simply by using different position, scale and rotation values.

`Conv_ModifyPosition()` is called after the Superscape World object has been created to handle any additional repositioning of the object. For example, in the DXF converter `Conv_ModifyPosition ()` takes into account any position modification according to the ECS of an object.

Defining the data converter module

The relevant data converter module for a file is identified by its name. All converter modules are called CONV_???.DLL where ??? is the extension of the file it converts. For example, CONV_DXF.DLL converts DXF files.

The converter module must define the following seven functions which VRT calls during data conversion. The functions must be in the standard SDK format, using the `__vrtcall` convention.

```
Conv_Open
Conv_FirstPass
Conv_Scan
Conv_Close
Conv_ModifyPosition
Conv_SkipElement
Conv_Validate
```

These functions are registered to VRT by inserting the relevant function addresses into the API vector table during the `App_Init` function, as follows:

```
short  App_Init(void)
{
   API_Vectors->_Conv_Open=FileOpen;
   API_Vectors->_Conv_FirstPass=FirstPass;
   API_Vectors->_Conv_Scan=ScanItem;
   API_Vectors->_Conv_Close=FileClose;
   API_Vectors->_Conv_ModifyPosition=ModifyPosition;
   API_Vectors->_Conv_SkipElement=SkipElement;
   API_Vectors->_Conv_Validate=Validate;
}
```

The functions are called by VRT in the following order.

1. When you select a file to convert, VRT attempts to find a suitable converter module based on the file extension. If it finds a suitable converter, it calls `Conv_Validate()` with the name of the file.

2. If the module confirms that the file is of the correct type, VRT prepares to convert it and then calls `Conv_Open()`.

3. Once the file is opened and the converter module has completed the preparation, VRT calls `Conv_FirstPass()`.

4. The converter module makes a first conversion pass. What the module does during this pass depends on the module, but generally it produces a list of elements contained in the file. For example, the DXF converter produces a list of elements, but only fills in the name of the elements.

5. VRT displays the list of defined elements, and allows you to select what you want to do to them—such as Light or Cut. When you have made the selection, VRT calls `Conv_Scan()`.

6. The converter module can be set to process the data file completely in one scan, or convert a single part of the file at a time. VRT repeatedly calls `Conv_Scan()` while the return value is `E_OK`.

7. Within the `Conv_Scan()` function the converter module may call the VRT functions `Conv_Element()`, `Conv_MakeWorldObject()` and all other related minor functions any number of times to create shapes and world objects.

8. When VRT receives a return value of `E_ERROR`, it presumes the conversion process is complete, and stores the generated world and shape files.

Data converter module - example

This section contains a simple example of the converter module interface that converts data in a fictional, extremely simple, data format called EDF (Example Data Format). The data format consists of just two simple predefined primitives, a Cube and a Pyramid, and can specify any number of these primitives placed anywhere with any size, rotation and color. Although extremely simple, it covers most aspects of data conversion. The example code for this device is included in the CONV_EDF.C file in the EXAMPLES>CONV_EDF directory.

An EDF file consists of a number of tokens (a single word at the start of a line) and their associated parameters (Integer, Float, String) which follow on the same line, separated by either a comma or blank space. The file type is identified by an EDF token on the first line, which `Conv_Validate()` uses to verify the file type. Each token has a fixed number and type of token list associated with it. Comments are on a single line and start with a # character. Valid tokens are:

Object:	String Name
Position:	Integer X Position, Integer Y Position, Integer Z Position
Size:	Integer X Size, Integer Y Size, Integer Z Position
Rotation:	Float X Rotation, Float Y Rotation, Float Z Rotation
Colour:	Integer Base Color
End:	NONE
EndFile:	NONE
EDF:	NONE

Note: The comments in the example have been omitted for brevity.

⇒ First include the standard header files.

```
#include     "APP_DEFS.H"
#include     <stdio.h>
#include     <math.h>
#include     <stdlib.h>
#include     <string.h>
#include     <ctype.h>
```

⇒ Undefine the `isspace` macro to make sure that the library equivalent is used.

```
#undef       isspace
```

⇒ Define an enumerated type to specify the values associated with the various tokens.

```
enum  Tokens {
             Token_OBJECT=0,
             Token_POSITION,
             Token_SIZE,
             Token_ROTATION,
             Token_COLOUR,
             Token_END,
             Token_ENDFILE,
             Token_EDF,
             Token_INVALID
             };
```

⇒ Define prototypes for the converter module functions.

```
long __vrtcall  FileOpen(char *filename);
long __vrtcall  FirstPass();
long __vrtcall  ScanItem(void);
void __vrtcall  FileClose();
void __vrtcall  GetRotation(T_ELEMENT *Element,
                    T_ELEMENT *ParElement);
void __vrtcall  SkipElement();
short __vrtcall Validate(char *filename);

enum Tokens GetToken(void);
short GetValue(T_LONGORPTR *Value, short ValueType);
```

⇒ Define the number of valid tokens and the parameter types (this makes things easier).

```
#define      E_NUMTOKENS  8
#define      E_VALLONG  1
#define      E_VALSTR  2
#define      E_VALFLOAT  3
```

⇒ Define some common data: the handle for the file to be converted, and its length, storage for the tokens during processing, and a pointer to the parameter list for the current token.

```
FILE  *File;
long  FileLen;
char   Line[256],*Parameter;
```

⇒ Define an array of token names.

```
char  Tokens[E_NUMTOKENS][32]={
                                "Object:",
                                "Position:",
                                "Size:",
                                "Rotation:",
                                "Colour:",
                                "End:",
                                "EndFile:",
                                "EDF",
                                };
```

⇒ Define two arrays for each of the Cube and Pyramid primitives. The first array specifies the number of points in each facet, this value is used in the call to `Conv_NewFacet()`. The second array is two dimensional and contains the co-ordinate values for each vertex on each facet for the primitive. The minimum and maximum co-ordinates of the primitives are 0.0 and 1.0 because the scale factor used in the `Size`: token then specifies the actual size in world co-ordinated of the object prior to scaling by VRT.

```
short  CubeNumPoints[6]={4,4,4,4,4,4};
T_CNVPOINT  CubePoints[6][4]=
                {
                  {
                     0,0.0,0.0,0.0,
                     1,1.0,0.0,0.0,
                     2,1.0,1.0,0.0,
                     3,0.0,1.0,0.0,
                  },
                  {
                     1,1.0,0.0,0.0,
                     4,1.0,0.0,1.0,
                     5,1.0,1.0,1.0,
                     6,1.0,1.0,0.0,
                  },
                  {
                     4,1.0,0.0,1.0,
                     7,0.0,0.0,1.0,
                     8,0.0,1.0,1.0,
                     5,1.0,1.0,1.0,
                  },
                  {
                     7,0.0,0.0,1.0,
                     0,0.0,0.0,0.0,
                     3,0.0,1.0,0.0,
                     8,0.0,1.0,1.0,
                  },
                  {
                     0,0.0,0.0,0.0,
                     7,0.0,0.0,1.0,
                     4,1.0,0.0,1.0,
                     1,1.0,0.0,0.0,
                  },
                  {
                     3,0.0,1.0,0.0,
                     6,1.0,1.0,0.0,
                     5,1.0,1.0,1.0,
                     8,0.0,1.0,1.0,
                  }
                  };

short PyramidNumPoints[5]={ 3,3,3,3,4 };
T_CNVPOINT PyramidPoints[5][4]=
                  {
                     {
                       0,0.0,0.0,0.0,
                       1,1.0,0.0,0.0,
                       2,0.5,1.0,0.5,
                       0,0.0,0.0,0.0,
                     },
                     {
                       1,1.0,0.0,0.0,
                       3,1.0,0.0,1.0,
                       2,0.5,1.0,0.5,
                       0,0.0,0.0,0.0,
                     },
                     {
                       3,1.0,0.0,1.0,
                       4,0.0,0.0,1.0,
                       2,0.5,1.0,0.5,
                       0,0.0,0.0,0.0,
                     },
                     {
                       4,0.0,0.0,1.0,
                       0,0.0,0.0,0.0,
                       2,0.5,1.0,0.5,
                       0,0.0,0.0,0.0,
                     },
                     {
                       0,0.0,0.0,0.0,
                       4,0.0,0.0,1.0,
                       3,1.0,0.0,1.0,
                       1,1.0,0.0,0.0,
                     }
                  };
```

⇒ Define the first function—the standard initialization function—which fills in the relevant converter module API vectors as previously explained. It is not necessary to store the old contents as this is done by VRT.

```
short  App_Init(void)
{
    API_Vectors->_Conv_Open=FileOpen;
    API_Vectors->_Conv_FirstPass=FirstPass;
    API_Vectors->_Conv_Scan=ScanItem;
    API_Vectors->_Conv_Close=FileClose;
    API_Vectors->_Conv_ModifyPosition=ModifyPosition;
    API_Vectors->_Conv_SkipElement=SkipElement;
    API_Vectors->_Conv_Validate=Validate;

   return(E_OK);
}
```

⇒ The exit routine does not need to do anything as VRT restores the old contents of the API vector table.

```
short  App_Exit(void)
{
   return(E_OK);
}
```

⇒ Use the `Validate` function to call `Conv_Open()` to open the specified file. This makes sure that there is only one way to open the file, and changes need only be made once. This is the same for `Conv_Close()`. To validate the file, the function reads the first token in the file, and checks if it is of type `Token_EDF`. If it matches the token, VRT presumes that the file is valid.

```
short __vrtcall Validate(char *filename)
{
    enum Tokens Token;
    if(FileOpen(filename)==E_ERROR)
         return(E_ERROR);

    if((Token=GetToken())==Token_INVALID)
      {
        FileClose();
        return(E_ERROR);
      }
     else
      {
        if(Token!=Token_EDF)
      {
      FileClose();
      return(E_ERROR);
      }
    }
 FileClose();
 return(E_OK);
}
```

⇒ Use the `Conv_Open()` function to open the requested file, in the correct mode (as text) and store the handle in the global variable `File`, for the functions such as `GetToken()` and `GetValue()` to use. Fill in the length of the EDF file for the precentage complete calculation.

```
long __vrtcall FileOpen(char *filename)
{
  if ((File=fopen(filename, "r"))==NULL)
  {
    Conv_Fatal("File not found");
    return(E_ERROR);
  }
  fseek(File, 0, SEEK_END);
  FileLen=ftell(File);
  fseek(File, 0, SEEK_SET);
  return(E_OK);
}
```

⇒ Use the `Conv_Close()` function to close the file opened by `Conv_Open()`.

```
void __vrtcall FileClose()
{
    if(File)
        fclose(File);
}
```

⇒ The `Conv_FirstPass()` function creates a list of empty element structures with just the names of the primitives you are going to use (Cube and Pyramid). VRT displays the list and allows you to select the elements to convert. Generating this list is made simpler by using the `Conv_NewElement()` function. Using this function means that the module need not concern itself with the number of elements already defined, the position in memory of the element list, and so on. Simply call the `Conv_NewElement()` function and fill in the structure returned.

The first pass also creates a single layer in the layer list for cases where facets are marked as having the same color as the layer they are on. In this case the existence (or not) of a layer list is not be a problem, as all coloring is done by using the color of a facets element definition and not by layer. However, take care.

```
long __vrtcall FirstPass()
{
   T_ELEMENT *pElement;
   T_CNVLAYER *pLayer;
    if((pElement=Conv_NewElement())==NULL)
         return(E_ERROR);
   else
   {
      strcpy(pElement->Name,”Cube”);
      pElement->Flags|=*C_CNVFlags;
   }
    if((pElement=Conv_NewElement())==NULL)
         return(E_ERROR);
   else
   {
      strcpy(pElement->Name,”Pyramid”);
      pElement->Flags|=*C_CNVFlags;
   }
   pLayer=Conv_NewLayer();
   if(pLayer!=NULL)
   {
      strcpy(pLayer->Name,”__Superscape_VRT__”);
       pLayer->Colour=16;
   }

   return(E_OK);
}
```

⇒ Set the `Conv_Scan()` function to process the entire EDF file in one go, returning `E_ERROR` on completion, as the primitives are created at the start of the `Conv_Scan()` function. If you call the `Conv_Scan()` function repeatedly to process the objects, the primitives would be created each time, causing some nasty problems. Alternatively, you could create the primitives during the first call of `Conv_Scan()` and then make sure that during subsequent calls, the EDF file is processed, creating 1 object each time. As you can see from the `while(ScanOK)` loop, this possibility is already catered for; resetting the `ScanOK` variable at the processing of the `End` token would do this, but the creation of the primitives would have to be placed within an `if` statement.

```
long __vrtcall ScanItem(void)
{
  enum Tokens TokenID;
  short ScanOK=1;
  T_ELEMENT *pElement=NULL;
  T_LONGORPTR Value;
  T_CNVFACET *pFacet;
  short Facet, Point;
}
```

⇒ Call `Conv_InitElement()` to prewarn VRT that you are about to begin processing a new element, and allow VRT to prepare any information it needs. Then find the previously defined empty element structure Cube, and create a list of facets using the `Conv_NewFacet()` function. The contents of the previously defined facets for the Cube shape are copied into this list. Once complete the origin of the element is filled in—this is the point used as the handle when creating an instance of the element. In this example the origin is the bottom, front left corner, the same as the Superscape system. Call `Conv_Element()` to generate the shapes for this element, and `Conv_CleanUpElement()` to make sure that the facet list is cleared ready for any further elements.

```
if((pElement=Conv_FindElement(“Cube”))!=
    NULL && (pElement>Flags&E_CNVSELECTED))
{
  Conv_InitElement();
  for(Facet=0; Facet<6; Facet++)
  {
     if((pFacet=Conv_NewFacet(CubeNumPoints[Facet]))==NULL)
        return(E_ERROR);

     pFacet->NumPoints=CubeNumPoints[Facet];
      pFacet->Special=0;
      pFacet->Colour=0;
     for(Point=0; Point<CubeNumPoints[Facet]; Point++)
       pFacet->Points[Point]=CubePoints[Facet][Point];
  }

  pElement->OriginX=0.0;
  pElement->OriginY=0.0;
  pElement->OriginZ=0.0;
  pElement->Colour=16;
  Conv_Element(pElement);
  Conv_CleanUpElement();
}
```

⇒ Use the same method to generate the shapes for the Pyramid primitive, except in this case you need to copy the facets from the array defined for the Pyramid shape.

```
if((pElement=Conv_FindElement(“Pyramid”))!=
    NULL && (pElement>Flags&E_CNVSELECTED))
{
  Conv_InitElement();

  for(Facet=0; Facet<5; Facet++)
  {
    if((pFacet=Conv_NewFacet (PyramidNumPoints[Facet]))==NULL)
       return(E_ERROR);

    pFacet->NumPoints=PyramidNumPoints[Facet];
     pFacet->Special=0;
     pFacet->Colour=0;
    for(Point=0; Point<PyramidNumPoints[Facet]; Point++)
      pFacet->Points[Point]=PyramidPoints[Facet][Point];

  }
  pElement->OriginX=0.0;
  pElement->OriginY=0.0;
  pElement->OriginZ=0.0;
  pElement->Colour=16;
  Conv_Element(pElement);
  Conv_CleanUpElement();
}
pElement=NULL;
```

⇒ Scan through the EDF file, searching for valid tokens. The first token required to create a world object is `Object:` as this specifies the type of primitive to be used. Then scan for other related tokens such as `Position:`, `Size:` and `Rotation:`, and fill in the information in the element from the parameters of these tokens. When you reach an `End:` token, and a valid `Object:` token has been found and the information filled in, call `Conv_MakeWorldObject()` to generate an instance of the element in the world.

```
while(ScanOK)
{
  TokenID=GetToken();

  switch(TokenID)
  {
      case  Token_OBJECT:
          GetValue(&Value, E_VALSTR);
          pElement=Conv_FindElement(Value.p);
          if(pElement!=NULL)
          {
              pElement->Insert_X=0.0;
              pElement->Insert_Y=0.0;
              pElement->Insert_Z=0.0;
              pElement->Scale_X=1;
              pElement->Scale_Y=1;
              pElement->Scale_Z=1;
              pElement->UserData
              pElement->Rotation_X=0.0;
              pElement->Rotation_Y=0.0;
              pElement->Rotation_Z=0.0;
              pElement->Colour=16;
          }
          break;

      case  Token_POSITION:
          if(pElement!=NULL)
          {
          if(GetValue(&Value, E_VALLONG)!=E_ERROR)
               pElement->Insert_X=(double)Value.l;
          if(GetValue(&Value, E_VALLONG)!=E_ERROR)
               pElement->Insert_Y=(double)Value.l;
          if(GetValue(&Value, E_VALLONG)!=E_ERROR)
               pElement->Insert_Z=(double)Value.l;
          }
          break;

      case  Token_SIZE:
          if(pElement!=NULL)
          {
              if(GetValue(&Value, E_VALLONG)!=E_ERROR)
                    pElement->Scale_X=(double)Value.l;
              if(GetValue(&Value, E_VALLONG)!=E_ERROR)
                    pElement->Scale_Y=(double)Value.l;
              if(GetValue(&Value, E_VALLONG)!=E_ERROR)
                    pElement->Scale_Z=(double)Value.l;
```

```
                }
                break;

          case  Token_ROTATION:
              if(pElement!=NULL)
                {
                   if(GetValue(&Value, E_VALFLOAT)!=E_ERROR)
                         pElement->Rotation_X=(double)Value.f;
                   if(GetValue(&Value, E_VALFLOAT)!=E_ERROR)
                         pElement->Rotation_Y=(double)Value.f;
                   if(GetValue(&Value, E_VALFLOAT)!=E_ERROR)
                         pElement->Rotation_Z=(double)Value.f;
                }
                break;

          case Token_COLOUR:
              if(pElement!=NULL)
               {
                   if(GetValue(&Value,  E_VALLONG)!=E_ERROR)
                         pElement->Colour=Value.l;
                }
                break;

          case Token_END:
                if(pElement!=NULL)
                   Conv_MakeWorldObject(pElement, 0,
                        *C_CNVLast_Child, NULL);
                break;
          case Token_ENDFILE:
                ScanOK=0;
                    return(E_ERROR);

          default:
                break;
        }
      }

      return(E_OK);
    }
```

⇒ The `Conv_SkipElement()` function is not required with this data format. It scans the data file to the end of the element currently being processed. Call it when the user has pressed escape and chosen to skip the current element.

```
  void __vrtcall SkipElement()
  {
  }
```

⇒ The `Conv_GetRot()` function is not required, as the coordinate system and rotation system used in the EDF format is identical to the Superscape data structure. The two lines of code in this function stop the compiler complaining about defined but not used variables.

```
void __vrtcall GetRotation(T_ELEMENT *Element,
      T_ELEMENT *ParElement)
{
      Element=Element;
      ParElement=ParElement;
}
```

⇒ Set the internal functions, as opposed to API vectors. The `GetToken()` function reads a line from the file, then compares the first word in the line with all valid tokens until it finds a match. The return value is the enumerated value for the valid token, or `Token_INVALID` if it is not recognized. After the token is parsed the function prepares the `Parameter` global variable to point to the first parameter after the token. As this is the only place in the converter module which reads information from the EDF file, this is also where the `C_CNVPercentage_Complete` value is updated, according to how far through the file you are in relation to the total length of the file.

Call `Conv_UpdateProgress()` to make sure that the changes in this value are displayed.

```
enum Tokens GetToken(void)
{
   short Token;

   if(File!=NULL && !(feof(File)))
   {
      fgets(Line, 256, File);
      *C_CNVPercentage_Complete=((float)100/FileLen)*ftell(File);
      Conv_UpdateProgress();
      while(Line[0]==’#’)
      {
            fgets(Line, 256, File);
          *C_CNVPercentage_Complete=((float)100/FileLen)*ftell(File);
            Conv_UpdateProgress();
      }

       for(Token=0; Token<E_NUMTOKENS; Token++)
        {
          if(strncmp(Tokens[Token], Line, strlen(Tokens[Token]))==0)
                  break;
        }
       if(Token!=Token_INVALID)
        {
             Parameter=Line+strlen(Tokens[Token]);
             while(isspace(Parameter[0]) && Parameter[0]!=’\0')
              Parameter++;
```

```
     }
    return(Token);
  }
  else
    return(Token_INVALID);
}
```

⇒ The `GetValue()` function parses the next parameter in the parameters list for the current token. If the requested type is an integer or float, then the function stores the numerical value of the parameter in the passed value storage address. If it is not, the function stores a pointer to the string parameter in the value. The function then prepares the `Parameter` global variable to point to the next parameter in the list.

```
short GetValue(T_LONGORPTR *Value,short ValueType)
{
    short ParamLength;
    static char ParamStr[32];

    if(Parameter!=NULL)
     {
        ParamLength=strcspn(Parameter,", \n\t\f\r\0");
        if(ParamLength==0) return(E_ERROR);
        strncpy(ParamStr,Parameter,ParamLength);
        ParamStr[ParamLength]='\0';
         if(Value!=NULL)
          {
              switch(ValueType)
               {
                   case E_VALLONG:
                      Value->l=(long)atoi(ParamStr);
                       break;
                   case E_VALSTR:
                      Value->p=ParamStr;
                       break;
                   case E_VALFLOAT:
                      Value->f=(float)atof(ParamStr);
                       break;
                   default:
                       Value->l=0;
                        break;
               }
           }
      Parameter+=ParamLength+1;
      while(isspace(Parameter[0]) && Parameter[0]!='\0')
          Parameter++;
        return(E_OK);
      }
  return(E_ERROR);
}
```

Sample EDF file

```
EDF
#
# An example EDF (Example Data Format) file for
# testing purposes.
#
Object: Cube
Position: 0,0,0
Size: 100,100,100
Rotation: 0,45,0
Colour: 20
End:

Object: Pyramid
Position: 500,0,0
Size: 100,100,100
Rotation: 0,22.5,0
Colour: 52
End:

EndFile:
```

Installing a new data converter module

To make a new data converter module available for automatic use the filename must follow the previously explained standard, and must be placed in the same directory as your VRT programs. If a module is not found in this directory to process a selected file type, VRT displays a dialog box from which you may select a different converter module. This allows modules to be stored in a different directory, but loses the convenience of automatic operation.

Chapter 9 - SDK examples

Introduction

The SDK comes with several simple example applications, each of which illustrates a useful technique for programming with the SDK. The first example application (SCL1) illustrates the basic techniques of registering a new SCL command for use in VRT. Since this is the first example, the entire file is examined line-by-line, and the function of each part explained. In the subsequent examples, only the significant changes are considered. Comments in the code are ignored.

The examples are provided as a set of projects in the EXAMPLES directory where you installed the SDK software; each example has its own directory. Open the workspace SDK.DSW in Microsoft Developer Studio and load each project file (.DSP) as necessary. Alternatively, you can start a single project by double-clicking its .DSP file.

SCL1: A simple SCL command

SCL1 simply registers a new SCL function called `Sequence` which returns a number starting at one, then one higher every time it is used.

At the top of the file is a comment containing information about the file, its author, and versions. Like all other comments, it is ignored by the compiler, but is a useful record for the author and users.

Next comes a section headed by the comment ‘`INCLUDES`’. This lists the files to be included first. It contains the line:

```
#include "APP_DEFS.H"
```

which includes the file APP_DEFS.H. This is a header file which defines various useful shortcuts and values for use in the application. It contains definitions for the structure of all the different data types an application may encounter when communicating with VRT, values of flags, and short cuts for all the calls to and from VRT. All applications must include this file.

Immediately after this is a section headed `DEFINED FUNCTION PROTOTYPES`, that contains:

```
short App_Init(void);
short App_Exit(void);
```

These prototypes specify the form of each function which may be referenced outside this file. Large applications may consist of several files which are all linked together. In this case, the only things which are needed outside this file are the entry and exit points, `App_Init` and `App_Exit`. In each case, they take no arguments (`void`) and return a `short` value. All functions must have prototypes.

The next section, `EXTERNAL FUNCTION PROTOTYPES` is blank, since the application does not need to rely on any functions in another file. Similarly, `EXTERNAL VARIABLES` is blank since no variables outside this file are required.

The `INTERNAL FUNCTION PROTOTYPES` section contains the following:

```
static void __vrtcall SCL_Sequence(void);
```

This defines a prototype for a function called `SCL_Sequence`, which takes and returns nothing (`void`), and is available for use only in this file (`static`). This function, when it is completely defined later, holds the actual code which is executed when the new SCL instruction is obeyed. `__vrtcall` is the VRT calling convention specifier used to call all VRT functions.

The next section is headed `DECLARATIONS`, and contains any variables or `#defines` to be used in the file. In this case it contains the following:

```
T_COMPILEREC  NewSCL={“Sequence”,
                       0,
                       0x01,
                       0,
                       E_PROCEDURE,
                       E_SSINTEGER,
                       };
short  SCLCode;
```

The first part defines a variable called NewSCL, which is a compiler record (type T_COMPILEREC). This is a structure containing information about the new SCL function to be registered, and since this is only used once, can be initialized at the start. The elements of the structure are initialized in order, and are:

- The name of the instruction (sequence).
- The preferred operation code (0, as there is no preference). This is only used when there are several SDK applications loaded at once, to make sure that the same opcodes are used each time.
- The number of inputs and outputs (0x01, 0 inputs, 1 output).
- A reserved value used by the compiler (0).
- The type of function (E_PROCEDURE).
- The type of value returned (E_SSINTEGER).

Following this would be a list of input parameter types, but there are no inputs so this is left out.

The short variable SCLCode is used to store the handle (operation code) of the new SCL instruction so that it can be removed when the application is removed. Since this is filled in later, it need not be initialized.

After this comes the main part of the module, the DEFINED FUNCTIONS section. This contains the actual definitions of the function prototypes in the DEFINED FUNCTION PROTOTYPES section. The first of these is called App_Init and is called when the application is first loaded. It looks like this (without comments):

```
short  App_Init(void)
{
     SCLCode=RegisterSCL(&NewSCL,SCL_Sequence);

    if(SCLCode<0)
      return(E_ERROR);

    return(E_OK);
}
```

The definition (taking no arguments and returning a `short`) must necessarily be exactly the same as the prototype. When it is executed, `RegisterSCL` is called first. This takes the address of the compiler record, which is called `NewSCL`, and the address of the code to call when the new SCL instruction is executed, `SCL_Sequence`. This registers the SCL, and returns either an opcode number or an error. Opcode numbers are positive and indicate a successful registration. Error numbers are negative and indicate that it could not be registered for some reason. This result is stored in `SCLCode`.

There are two main reasons that the SCL registration might fail; the maximum number of new SCL instructions (256) has already been registered, or the requested name already exists. Both of these cases are unlikely here, but it is good practice to check for errors. The next line does this, and if the returned value is less than zero, returns with an exit code of `E_ERROR`. This indicates to VRT that the initialization code has failed in its task and that the application has failed to install correctly.

If, however, no error occurred, execution falls through to the next line which returns with a code indicating that all is well (`E_OK`). The initialization is now complete. The SCL has been registered, and can be recognized by both the SCL compiler and executor.

The next function indicates what to do when the application is removed, either by exiting from VRT or using Clear Applications on the File menu. It is called `App_Exit` and looks like this:

```
short  App_Exit(void)
{
     UnRegisterSCL(SCLCode);

     return(E_OK);
}
```

This is very simple. The SCL code which was returned from the `RegisterSCL` instruction (`SCLCode`) is passed back to `UnRegisterSCL`, which removes it from the list of registered SCL instructions. Since this can never fail, the exit routine is terminated by returning with a code of `E_OK`, indicating that all is well.

The next section is headed `INTERNAL FUNCTIONS`, and contains any functions that are only referenced from within this file. In this case, only the body of the new SCL, `SCL_Sequence`, is included in this section:

```
static void __vrtcall SCL_Sequence(void)
{
     static long Number=0;
     Number++;
     PushN(E_SSINTEGER,Number);
}
```

The function is declared as `static` since it is not referenced outside this file. It takes no arguments and returns none using the C calling conventions. Returning arguments to SCL is done slightly differently.

The first line defines a `static` variable (it holds its value between calls to the function) called `Number`, which is initialized to 0 at startup.

When the function is called, the next line increments `Number` by one. This gives the sequential characteristic of the new SCL function. The new value of `Number` is returned to SCL in the next line using a function called `PushN` (for ‘Push Number’). SCL uses a stack to store intermediate results and return values, and pushing a value onto the stack is the way to return values from a function. This line pushes a number, of type `E_SSINTEGER` (an integer), the value of which will be `Number`. The total effect is to return the next highest number each time the function is called.

This completes the first application.

⇒ Build the application

Choose Build>Update All Dependencies, and then Build>Build SCL1.DLL to build the application.

⇒ Check the application

1. Start VRT and go to the World Editor.
2. Choose File>Other Files>Open Application.

The Open Application dialog box is displayed.

3. Select SCL1.DLL from the correct directory, and click OK.
4. The application is loaded and started.

If all is well, no error message is displayed and the World Editor is displayed.

5. Select the Anchor object, and choose SCL>Edit.
6. Add the following short SCL program to the Anchor object:

```
print(Sequence);
```

7. Switch to Visualiser [F2].

The values displayed advance by one every frame, showing that the application is working as expected.

Although this is a very simple example, it does illustrate the function of the initialization and exit functions, the way to register functions with SCL, and how to return values from the new SCL function.

Do not exit from VRT; one of the effects of the application is demonstrated at the start of the next section.

SCL2: Resetting SCL1

Make sure that SCL1 is running, displaying the numbers as it goes. Press F12, to reset the world. If the viewpoint has been moved, it is reset, but the numbers displayed continue as before. This is because the SCL code registered by the application has no mechanism for detecting a reset and resetting the `Number` variable to zero.

Exit VRT and load SCL2.DSP into Developer Studio. Superficially, it is much the same. The first difference comes in the `INTERNAL FUNCTION PROTOTYPES` section, which defines a new function:

```
static void __vrtcall ResetSequence(void);
```

This is a function which is called during the initialization process to reset the `Number` variable to 0.

The next difference is in the `DEFINITIONS` section, where the variable `Number` is defined. It has been moved here so that both the `SCL_Sequence` and `ResetSequence` functions have access to it.

In the `DEFINED FUNCTIONS` section, `App_Init` now looks like this:

```
short App_Init(void)
{
     SCLCode=RegisterSCL(&NewSCL,SCL_Sequence);
     if(SCLCode<0)
        return(E_ERROR);
     VecB4Init=ResetSequence;
     return(E_OK);
}
```

The new line sets a vector, `VecB4Init` to point to the function `ResetSequence`. A vector is generally the address of a function, and usually points to a ‘stub’ which, in effect, says ‘do nothing’. There are several vectors defined, most of which are called during processes in Visualiser. `VecB4Init` is called immediately before initializing the world. The application is redirecting it so that rather than doing nothing it now calls the function `ResetSequence`.

`App_Exit` is the same—the change made to `VecB4Init` by `App_Init` is undone automatically by the interface code.

The `SCL_Sequence` code is very similar to before, except that the declaration of `Number` has been moved outside, as was noted above:

```
static void __vrtcall SCL_Sequence(void)
{
    /* Number declaration was in here */
    Number++;
   PushN(E_SSINTEGER,Number);
}
```

Finally, there is the complete new function, `ResetSequence`. This resets the value of `Number` to 0, then calls the old reset vector, `OldVecB4Init`:

```
static void __vrtcall ResetSequence(void)
{
    Number=0;
    OldVecB4Init();
}
```

The call to the old vector is required in case the previous owner of the vector needs to perform any initialization too. In this case, there is only one application on the system, so this is not strictly necessary. However, it is very good practise since it makes the application ‘future proof’. When you use this application in conjunction with others at a later date, defensive programming can make errors much less likely.

⇒ Check the application

Build SCL2.DLL, and then repeat the sequence of operations exactly as for the SCL1 example above—the numbers displayed should be constantly increasing by one. However, when you press F12 (reset), they should restart at 1 again. This is confirmation that the new code in SCL2 is working.

SCL3: Returning more than one result

Often, a new SCL command needs to return more than one result. The example here is a command called `ScrnSize`, which gets the size of the screen in pixels, and fills in two variables with its width and height.

The function called by the SCL is called `SCL_ScreenSize`, and is declared first in the `INTERNAL FUNCTION PROTOTYPES` section as follows:

```
static void __vrtcall SCL_ScreenSize(void);
```

Like all SCL functions, the C code takes no parameters and returns no result (`void`). Any required parameters are passed on SCL's internal stack.

The declaration of the compiler record for this function is slightly different to the one used in SCL1 and SCL2, since the function takes a different number of arguments. This is the declaration:

```
static T_COMPILEREC NewSCL=
                    {
                      "ScrnSize",
                      0x400,
                      0x20,
                      0,
                      E_PROCEDURE,
                      E_SSNONE,
                      E_SSPOINTER+E_SSWRITE,
                      E_SSPOINTER+E_SSWRITE,
                    };
static short SCLCode;
```

The name of the function has changed to '`ScrnSize`'. Following this are:

- The preferred operation code `0x400`.
- The number of inputs and outputs (`0x20` - 2 inputs, no return value).
- The value used by the system (`0`).

The function type is still `E_PROCEDURE`, but it returns no value (`E_SSNONE`), and takes two variable addresses to fill in (`E_SSPOINTER+E_SSWRITE`). These addresses must be writable, otherwise it would be unwise to modify their contents.

`static short SCLCode` is used to store the handle of the new SCL instruction.

`App_Init` is very similar to the one in SCL1, except that the function being registered is `SCL_ScreenSize` instead of `SCL_Sequence`:

```
short App_Init(void)
{
      SCLCode=RegisterSCL(&NewSCL,SCL_ScreenSize);
       if(SCLCode<0)
          return(E_ERROR);
       return(E_OK);
}
```

App_Exit is identical.

The new SCL is called from an SCL program by supplying the addresses of two short variables, for example:

```
short Width, Height;
scrnsize (&Width, &Height);
```

The code that is called is in the function SCL_ScreenSize:

```
static void __vrtcall SCL_ScreenSize(void)
{
     short *pWidth, *pHeight;
     pHeight=PopP(E_SSPSHORT);
     pWidth =PopP(E_SSPSHORT);
     if(pWidth!=NULL)
          *pWidth=(C_ScreenParams[*C_GraphicsDD])->Width;
     if(pHeight!=NULL)
          *pHeight=(C_ScreenParams[*C_GraphicsDD])->Height;
}
```

The first line defines two pointers to short variables, pWidth and pHeight. Values are popped from the SCL stack in reverse order. Although the function is called from SCL with the Width first, then the Height, the pointers to these variables are popped off Height first, then Width. The type passed to PopP (for ‘pop pointer’) is the required type of argument, in this case E_SSPSHORT, the type code for a pointer to a short value.

Now that these pointers have been retrieved, the contents can be filled in. Firstly, however, it is good practice to check for NULL pointers. A reference in SCL to the address of variable null will result in a C style NULL pointer being passed. SCL commands should not attempt to write to NULL pointers, and should ignore them (do nothing). This is what in fact happens. This allows the user to get just the width of the screen using:

```
short Width;
ScrnSize(&Width, &null);
```

The width and height of the screen are read from the structure pointed to by *C_ScreenParams. This is filled in and set up by the Superscape system, and is just one of the many items of useful data available to applications.

The function does not return any values, and therefore does not need to push anything back onto the SCL stack before it finishes.

⇒ Check the application

1. Build SCL3.DLL.
2. Start VRT, and load the application using File>Other Files>Open Application in the World Editor.
3. Select the Anchor object, and add the following SCL program to it:

```
short Width, Height;
ScrnSize(&Width,&Height);
print(Width);
```

4. Switch to Visualiser [F2].

The width of the screen, in pixels, is continually displayed. Change the resolution and note how the width value changes in response.

By `print`ing the `Height` value instead, it can be seen that both values are filled in correctly with the current screen size.

This sort of technique is very useful when communicating with external devices that return several values. If the number of values is large (greater than about 5), it may be worth considering passing the address of a single array rather than several individual variables. This is explored in the “SCL7” example.

SCL4: Calling Superscape routines

Many of the most useful routines in the Superscape system are available to applications. In this example, a new SCL function is defined which prints a text string on the screen at a given position. It is called `Sprint` (string print), and is used as follows:

```
Sprint(<x position>,<y position>,<string>,<fg colour>,<bg colour>);
```

The positions are specified in pixels from the top left of the screen to the top left of the text string to be drawn. The foreground and background colors are color indices into the palette.

The source file, SCL4.C is again quite similar to the previous examples. The SCL function to be called is called `SCL_PrintString`:

```
static void __vrtcall SCL_PrintString(void);
```

The compiler record reflects the new function also:

```
static T_COMPILEREC NewSCL=
                    {
                        “Sprint”,
                        0,
                        0x50,
                        0,
                        E_PROCEDURE,
                        E_SSNONE,
                        E_SSINTEGER,
                        E_SSINTEGER,
                        E_SSPOINTER,
                        E_SSINTEGER,
                        E_SSINTEGER,
                    };
static short SCLCode;
```

The name is different, and the number of inputs and outputs has changed to `0x50` (5 inputs, no return value). It takes as inputs the integer x and y coordinates, a pointer to the string to print, and integer color indices.

`App_Init` is again very similar to the one in SCL3, except that the function being registered is `SCL_PrintString`:

```
short App_Init(void)
{
     SCLCode=RegisterSCL(&NewSCL,SCL_PrintString);
     if(SCLCode<0)
         return(E_ERROR);
     return(E_OK);
}
```

`App_Exit` is identical.

The actual print string function reads as follows:

```
static void __vrtcall SCL_PrintString(void)
{
    short x,y;
    unsigned char FGColour,BGColour;
    char *String;
    BGColour = (unsigned char) PopN(E_SSINTEGER);
    FGColour = (unsigned char) PopN(E_SSINTEGER);
    String   = PopP(E_SSPCHAR);
    y        = (short) PopN(E_SSINTEGER);
    x        = (short) PopN(E_SSINTEGER);
    if(String!=NULL)
    {
        Text(x, y, FGColour, BGColour, 6, -1, 0, 3 ,String);
    }
}
```

The variable declarations just declare spaces for the arguments to be stored. They are popped from the stack, in reverse order. The functions `PopN` (pop number) and `PopP` (pop pointer) require the type of value to expect, in this case either integers or a pointer to character. Note the additional casts `(unsigned char)` and `(short)` which dictates the type of variable expected.

If the string pointer is non-`NULL`, the Superscape routine `Text` is called. This takes a number of parameters. The first two are the x and y position of the string, as passed through from the SCL. The next two are the foreground and background colors for the string, again passed through from SCL. The next argument is the font number to use, in this case 6 (the 6x8 font).

Following this is the maximum length allowed for the string. In this case, there is no maximum length so a value of `-1` is used to signify this. The next parameter is the alignment, `0`, meaning that the x and y position marks the top left of the text to print. Then there is a flags value, `3`, which indicates that the text is to be drawn on both the foreground screen (the one being displayed) and the background screen (the one on which the next picture is being drawn).

Finally, the actual string pointer is passed. The `Text` routine now draws the string on the screen with the specified position and color. Since no values are returned, nothing is pushed back on to the stack before the function finishes.

⇒ Check the application

1. Build SCL4.DLL.

2. Start VRT, and load the application in the World Editor using File>Other Files>Load Application.

3. Select the Anchor object and add the following SCL program to it:

```
Sprint(100,100,”Hello”,15,0);
```

4. Switch to Visualiser [F2].

The string “Hello” is printed on the screen at 100 pixels in from the top and left hand edge, in bright white (color 15) with a transparent background (0).

Notice that even as the viewpoint is moved around, the view in the window appears to be behind the string. This is because it is being printed every frame over the top of the picture.

As with any other SCL function, any argument can be replaced with a variable or expression.

⇒ Edit the SCL

1. Switch to the World Editor, and edit the SCL program on Anchor to read:

```
Sprint(100+random(100),100+random(100),”Hello”,15,0);
```

2. Switch to Visualiser.

Each frame, the string is drawn in a different, random position between 100 and 200 pixels from the top and left edges of the screen.

The string is removed from the screen each frame because it is printed within the window onto the world, which is cleared every frame.

3. Move the view until there is nothing on the screen, and press the B key.

This turns the screen clear mechanism off, and each string is left on the screen.

4. Press B again until the background returns to its original state (the strings all disappear as the background is again cleared).

For this reason also, any writing to the screen outside the window will not be erased each frame.

SCL5: Altering data using variables

It is sometimes useful to use a new SCL function to return the address of a variable using the SCL variable functions such as `xpos`. The address of the variable should be calculated and pushed onto the stack, as in this example. It registers a new SCL function called `BmEdge`, which returns as a variable the beam edge value of a light source on a given object (the same as built-in function `bmedge`). This is called from SCL as follows:

```
BmEdge(<object number>)
```

In order to work this out, it is necessary to find the address of the light source attribute on a given object—a very useful technique.

The source file, SCL5.C, is very similar to the previous ones. The SCL function is called `SCL_BeamEdge`, and is used in `App_Init` to register the SCL. The compiler record reads as follows:

```
static T_COMPILEREC NewSCL=
                    {
                        "BmEdge",
                        0,
                        0x11,
                        0,
                        E_VARIABLE,
                        E_SSPOINTER+E_SSWRITE,
                        E_SSOBJNUM,
                    };
static short SCLCode;
```

The SCL name is `BmEdge`, and it takes 1 value and returns 1 value (`0x11`). The function is treated as a variable, so its type is `E_VARIABLE`, and it returns a writable pointer to the variable (`E_SSPOINTER+E_SSWRITE`). Finally, it takes an object number (`E_SSOBJNUM`) as its one argument.

The function `SCL_BeamEdge` looks like this:

```
static void __vrtcall SCL_BeamEdge(void)
{
    short Object;
    T_WORLDCHUNK *Light;

    Object=PopN(E_SSOBJNUM);
    Light=ChunkAdd(Object,E_CTLIGHTSOURCE);

    if(Light==NULL)
        PushP(E_SSPSHORT,NULL);
    else
        PushP(E_SSPSHORT+E_SSWRITE, &Light->Lig.BeamEdge);
}
```

Firstly, variables are defined for the object number to look at, and a pointer to the world chunk (the data structure that defines a world attribute) for the light source.

The object number is popped from the stack, using `PopN`. This checks that the number on the stack is actually an object (`E_SSOBJNUM`), and returns the object number as an integer.

The next line uses the Superscape routine `ChunkAdd` to find the address of a chunk of type `E_CTLIGHTSOURCE` (the light source attribute type), on object `Object`. If none is found, this returns a `NULL` pointer, otherwise a pointer to the attribute is returned.

The return value is checked, and if `NULL`, a `NULL` pointer is returned. This is interpreted by SCL as equivalent to `&null`, and the function behaves like the `null` variable (it always returns 0 when read, and writing to it has no effect). `PushP` pushes this value as a pointer to a short value (`E_SSPSHORT`).

If a light source was found, the address of the `BeamEdge` field in the light source is found and pushed as a writable short pointer. This pointer is dealt with by the SCL executor according to whether the variable is being read from or written to. The actual read/write process is handled entirely outside this piece of code.

⇒ Check the application

1. Build SCL5.DLL.
2. Start VRT, and load the world SCL5.XVR using File>Open in the World Editor.

This contains a few objects to make the effects of changing the beam width visible.

3. Switch to Visualiser.

The light is suspended above a floor made up of many small facets. This is lit by a medium beam with a soft edge. The two nearby arrows change the beam edge value, when activated.

4. Switch to the World Editor, and load SCL5.DLL using File>Other Files>Open Application.
5. Select the downward pointing arrow, and add the following SCL to it:

```
if(activate(me,0) && BmEdge('Light')>=100)
     BmEdge('Light')-=100;
```

This decreases the beam edge value by 100 every time the arrow is activated, as long as the beam edge value is at least 100 to start with. This stops the beam edge value ever going negative, which is illegal.

6. Select the upward pointing arrow, and add the following SCL:

```
if(activate(me,0))
     BmEdge('Light')+=100;
```

This simply increases the beam edge value when the object is activated.

7. Switch to Visualiser, and click on the two arrows.

The edge of the beam becomes more or less sharp as the beam edge value is changed.

8. Switch to the World Editor and use File>Save As to save this new world as MYSCL5.XVR . The application is bound into the .XVR file, and this is the copy that will be used when this VRT file is loaded in future.

⇒ To use a new version of the application with the VRT file

1. Load the XVR file, and go to the World Editor.
2. Choose File>Other Files>Clear Applications.
3. Choose File>Other Files>Open Applications.
4. Load the new version of the application module.
5. Choose File>Save.

SCL6: Altering data directly

In the last example, the SCL code returned the address of some data which was manipulated by other SCL instructions. In this example, the application alters the world data directly. It defines a new SCL command called `SetSize`, which is used as follows:

```
SetSize(<object>,<x size>,<y size>,<z size>);
```

This sets the size of the object <object>, altering all three axes in one go. It is possible to do this in other ways, but this is neater.

The C routine called to perform this function is `SCL_SetSize`, and is defined as usual in the `INTERNAL FUNCTION PROTOTYPES` section:

```
static void __vrtcall SCL_SetSize(void);
```

Its compiler record looks like this:

```
static T_COMPILEREC NewSCL=
                    {
                        “SetSize”,
                        0,
                        0x40,
                        0,
                        E_PROCEDURE,
                        E_SSNONE,
                        E_SSOBJNUM,
                        E_SSINTEGER,
                        E_SSINTEGER,
                        E_SSINTEGER,
                   };
static short SCLCode;
```

The name is `setsize`, it takes 4 inputs and returns nothing (`0x40`); it is a procedure type (`E_PROCEDURE`); returns nothing (`E_SSNONE`); takes an object number (`E_SSOBJNUM`), and three integers (`E_SSINTEGER`) as arguments.

The `App_Init` and `App_Exit` code is the same as before, except that the C routine called by the SCL is registered as `SCL_SetSize`.

The actual function itself looks like this:

```
static void __vrtcall SCL_SetSize(void)
{
    long          x,y,z;
    short         ObjNum;
    T_WORLDCHUNK  *Object;
    z=PopN(E_SSINTEGER);
    y=PopN(E_SSINTEGER);
    x=PopN(E_SSINTEGER);
    ObjNum=PopN(E_SSOBJNUM);
    Object=ChunkAdd(ObjNum,E_CTSTANDARD);
    if(Object!=NULL)
    {
        Object->Std.XSize=x;
        Object->Std.YSize=y;
        Object->Std.ZSize=z;
    }
}
```

The variables `x`, `y` and `z` are set up to hold the required size of the object, `ObjNum` holds the number of the object, and `Object` is a pointer to the standard chunk of the object.

These are filled in by popping the values from the stack. Note how `ObjNum` is filled in by popping a number of type `E_SSOBJNUM` from the stack. This returns the number of the object as an integer.

The next line calculates the address of the standard chunk (`E_CTSTANDARD`) of object `ObjNum`, placing the address in `Object`. If this fails (`ObjNum` does not exist), this returns a `NULL` pointer, which is checked for in the next line. If the pointer is `NULL`, nothing happens.

Otherwise, the `x`, `y` and `z` sizes are filled into the object's standard chunk from the variables where they were stored. Since no values are returned, the function terminates there.

⇒ Check the application

1. Build SCL6.DLL.

2. Start VRT, and load the application in the World Editor using File>Other Files>Load Application.

3. Select the Anchor object and add the following SCL program to it:

```
if(activate(me,0))
    SetSize(me,1000,1000,1000);
```

4. Switch to Visualiser, and click on Anchor.

It resizes itself into a cube 1000 units on a side. Since its original size was 2000 x 30 x 2000, this shows that all three axes have been changed at once.

SCL7: SCL and devices

The next SCL example shows how communication is handled with the device drivers. In this example, a new SCL command `getprpos` is registered which reads the current proportional device and returns the state of its axes and buttons. `GetPrPos` is used like this:

```
<buttons>=getprpos(<array address>);
```

The function returns the state of all the buttons as an integer, bit 0 of which is set if button 1 is pressed, bit 1 of which is set if button 2 is pressed, and so on. The array of `shorts` at <array address> is filled in with the six axis values returned from the device. There must be at least six elements in this array, since the `getprpos` function always fills in six values. If less are defined, the end of the array is overwritten and the following data corrupted.

The source code is in SCL7.C. The function definition for `SCL_GetPosition` is:

```
static void __vrtcall SCL_GetPosition(void);
```

Its compiler record is:

```
static T_COMPILEREC NewSCL=
                    {
                        “GetPrPos”,
                        0,
                        0x11,
                        0,
                        E_PROCEDURE,
                        E_SSINTEGER,
                        E_SSPOINTER,
                    };
static short SCLCode;
```

The name field is obvious, and the function takes one argument and returns one value directly (`0x11`). It is a procedure type, returns an integer and takes a pointer to the array as its one argument.

`App_Init` and `App_Exit` are the usual pair, registering the SCL function and unregistering it again at the end.

The function itself looks like this:

```
static void __vrtcall SCL_GetPosition(void)
{
    short *Array;
    T_PRPOS *Position;
    short i;
    Array=PopP(E_SSPSHORT+E_SSWRITE);
    if(*C_SCLError<0)
        return;
    Position=PrGetPos(NULL);
    for(i=0;i<6;i++)
       Array[i]=Position->Axis[i];
    PushN(E_SSINTEGER,Position->Buttons);
}
```

The variable declaration defines `Array`, the pointer to the start of the array to fill in, `Position`, which is a pointer to a proportional position structure (`T_PRPOS`), which is returned by the device driver, and a loop counter, `i`.

The address of the array is popped from the stack, with a type specifying a writable short pointer. Any other type of pointer causes this to fail. In the previous examples, a failure to pop a value correctly was not disastrous, and would be caught after the function completed. Here, however, misreporting the address of an array to fill in could have serious consequences, and so the variable `*C_SCLError` is checked to see if an error has occurred. If so, the function returns immediately, allowing the SCL error handling mechanism to take over before anything is written to the bogus address.

Assuming that no error has occurred, the function `PrGetPos` is called. This is a device driver function, which returns a pointer to a structure which has been filled in with the axis values for all the axes of the device. All device driver functions take and return one pointer, but since this function does not actually require any data to be passed to it, a `NULL` pointer is passed instead. The returned structure consists of the button state and the axis values.

The axes are copied from the Position structure into the array at `Array`, in the loop that follows. The loop counter, `i`, is used to index into each array and transfer each axis in turn. The axes for most devices are defined as follows:

0: X position	1: Y position	2: Z position
3: X rotation	4: Y rotation	5: Z rotation

The button status is retrieved, and pushed back onto the stack as an integer.

⇒ Check the application

1. Build SCL7.DLL.

2. Start VRT, and load the application in the World Editor using File>Other Files>Load Application.

3. Select the Anchor object and add the following SCL program to it:

```
short Axis[6];
long Buttons;
Buttons=GetPrPos(&Axis);
print(Axis[0]);
```

4. Switch to Visualiser. The current X translation value is printed in the top left corner of the screen.

5. Move your proportional device, such as Spacemouse or mouse movement, from left to right (Note: You must use the first enabled device which is usually mouse movement.). If you have mouse movement enabled, move the mouse left and right with the right hand button pressed. The value changes, negative in one direction, positive in the other.

Printing the other elements of the array allows you to see the effects of the other axes, and printing the value of `Buttons` reflects the current state of the depressed buttons.

VEC1: Intercepting vectors

The SDK allows you to intercept several vectors which are called by the Superscape system at various times. In SCL2 you saw the use of the `VecB4Init` vector for resetting the counter used in SCL1.

Most of the vectors are called every frame, however, and can be used for a variety of purposes. In this example, you will use the vector `VecB4Instrs`, which is called just before drawing the instruments, to display a running facet count.

The file VEC1.C starts, as usual, with the header comment, followed by the inclusion of APP_DEFS.H. Prototypes for `App_Init` and `App_Exit` are defined, followed by the prototype for the new routine, `PrintFacCount`:

```
static void __vrtcall PrintFacCount(void);
```

This will be hooked onto the vector and called every frame. `App_Init` simply does this:

```
short App_Init(void)
{
     VecB4Instrs=PrintFacCount;
     return(E_OK);
}
```

The `VecB4Instrs` vector is set to the address of `PrintFacCount`, and the routine returns an OK report.

`App_Exit` is extremely short, since the original vector contents are restored by the Superscape system when the application is dumped:

```
short App_Exit(void)
{
     return(E_OK);
}
```

It simply returns an OK value.

The actual vector routine, `PrintFacCount`, takes the current facet count (as drawn in the last window onto the world), converts it to a string, and displays it at the bottom right of the window:

```
static void __vrtcall PrintFacCount(void)
{
     char String[9];
     *LongToDec(*C_FacetCount,String,8)='\0';
     Text((short)(C_Console->WindXMax-2),
          (short)(C_Console->WindYMax-2),
          E_COLWHITE, E_COLTRANSPARENT,
          6, -1, 10, 3, String);
     OldVecB4Instrs();
}
```

The first line defines some space for the string to be built, and the second uses `LongToDec` to build it from the value in `*C_FacetCount`. This is the number of facets drawn in the last window onto the world. The last argument to `LongToDec` is the number of characters to convert, in this case 8. `LongToDec` returns the address of the character after the last character converted, so this is filled in with a `0` to terminate the string properly.

The string is printed onto the screen. `C_Console` points to a structure defining the last window onto the world (`T_CONSOLE`). It contains, amongst other things, the minimum and maximum extents of the window in X and Y. These are used to place the string 2 pixels in from the right (maximum x), and 2 pixels up from the bottom (maximum y). The color of the displayed string is set to bright white on a transparent background, and the font set to `6` (this is the 6x8 font used on the loading screen). The `-1` indicates that there is no maximum length for the string, and `10` is the alignment value. In this case, the bottom right corner of the string is aligned with the x and y coordinates specified as the first two arguments. The flags value, `3`, indicates that the value should be drawn on both foreground and background screens. Finally, the string itself is specified, and `Text` draws it on the screen.

The previous owner of the vector is called, in case this needs to do anything.

⇒ Check the application

1. Build VEC1.DLL.
2. Start VRT and, in the World Editor, load one of the demonstration files using File>Open.
3. Load the application VEC1.DLL, using File>Other Files>Open Application.
4. Switch to Visualiser.

The current facet count is displayed in the bottom right hand corner of the window.

VEC2: Vectors and SCL

SCL2 showed how intercepting a vector could alter data used by the SCL. This example shows the converse. It sets up a vector very similar to the one in VEC1, except that a string is displayed instead of the facet count. This string is set using a new SCL instruction, `Display`.

The file consists of the usual header and prototypes for `App_Init` and `App_Exit`. There are three internal functions:

```
static void __vrtcall SCL_Display(void);
static void __vrtcall PrintString(void);
static void __vrtcall ResetString(void);
```

`SCL_Display` is the SCL function, `PrintString` prints the string on the screen, and `ResetString` removes it when the world is reset.

The `DECLARATIONS` section contains the following:

```
static T_COMPILEREC NewSCL=
                    {
                        “Display”,
                        0,
                        0x10,
                        0,
                        E_PROCEDURE,
                        E_SSNONE,
                        E_SSPOINTER,
                    };
static short SCLCode;
static char* DisplayString=NULL;
```

The compiler record `NewSCL` should be familiar from the SCL examples, describing the function `Display`, with 1 input and 0 outputs (`0x10`), being a procedure returning nothing and taking a pointer (to a string). `SCLCode` is storage for the SCL handle for unregistering it again, and `DisplayString` is the address of the string to display. This is initially set to `NULL` to indicate that no string should be displayed at all.

`App_Init` looks like this:

```
short App_Init(void)
{
     SCLCode=RegisterSCL(&NewSCL,SCL_Display);
     if(SCLCode<0)
         return(E_ERROR);
     VecB4Instrs=PrintString;
     VecB4Init=ResetString;
     return(E_OK);
}
```

It registers the new SCL instruction using `RegisterSCL`, storing the SCL instruction code in `SCLCode`. If this is less than 0, an error has occurred and the routine returns to the main program with an error report.

`VecB4Instrs` is set up to point to the `PrintString` routine, and `VecB4Init` to `ResetString`. This makes sure that they are called at the required points. Finally, the function returns an OK report.

All `App_Exit` needs to do is unregister the SCL instruction—all the vectors are restored by the Superscape system.

The `PrintString` function is:

```
static void __vrtcall PrintString(void)
{
     if(DisplayString!=NULL)
     {
          Text((short)(C_Console->WindXMax-2),
               (short)(C_Console->WindYMax-2),
               E_COLWHITE, E_COLTRANSPARENT,
               6, -1, 10, 3,
               DisplayString);
     }
     OldVecB4Instrs();
}
```

The first line checks to see if there is a valid string to display, and if not, skips the call to `Text`. This has the same parameters as the previous example, and thus displays the string `DisplayString` 2 pixels in from the bottom right corner of the current window, in white on transparent, using font 6 (6x8).

The old vector owner is called, and the function finishes.

The SCL instruction `display` calls the following function:

```
static void __vrtcall SCL_Display(void)
{
     DisplayString=PopP(E_SSPCHAR);
     if(*C_SCLError<0)
          DisplayString=NULL;
}
```

The address of the string to display (a pointer to char) is popped from the stack and placed in `DisplayString`. If an error occurred, this address is invalid and must be set back to `NULL` to prevent problems when printing the string.

Finally, the function `ResetString` is defined:

```
static void __vrtcall ResetString(void)
{
    DisplayString=NULL;
    OldVecB4Init();
}
```

This resets the variable `DisplayString` to `NULL`, turning off the string being displayed. The old vector is called as usual.

⇒ Check the application

1. Build VEC2.DLL.

2. Start VRT, and load the application in the World Editor using File>Other Files>Load Application.

3. Select the Anchor object, and add the following SCL program to it:

```
if(activate(me,0))
    Display("Hello, world!");
```

4. Switch to Visualiser, and click on Anchor.

The message "Hello, world!" is displayed in the bottom right of the screen. Unlike `print`, this information persists even when the SCL is not being executed (when Anchor is not being activated).

5. To clear the message, press F12.

The function `ResetString`, on the `VecB4Init` vector, is executed at this point and removes the string from the screen.

VEC3: The unknown vectors

In addition to the vectors that are called every frame, or at specific times, there are a set of vectors defined that are called when the Superscape system comes across something unusual. This example shows how to intercept the `UnknownInstr` vector, and define a completely new instrument type.

VEC3.C starts in the usual way, with the header, includes and prototypes for `App_Init` and `App_Exit`. The `INTERNAL FUNCTION PROTOTYPES` section reads:

```
static void __vrtcall NewInstrument(T_INSTRUMENT *Instr);
```

This is the routine that draws the new instrument. It is passed the address of the unknown instrument as an argument.

The instrument defined is a ‘seven segment’ display, similar to that found on most calculators. In the `DECLARATIONS` section, a pair of tables is set up showing the positions of each segment, and which segments are required for each number from 0 to 9:

```
static char PosTable[]=  { 1,  0,  6,  0,
                           7,  1,  7,  7,
                           7,  9,  7, 15,
                           1, 16,  6, 16,
                           0,  9,  0, 15,
                           0,  1,  0,  7,
                           1,  8,  6,  8,
                         };
static char Segments[]= {  0x3F,
                           0x06,
                           0x5B,
                           0x4F,
                           0x66,
                           0x6D,
                           0x7D,
                           0x07,
                           0x7F,
                           0x6F,
                         };
```

`PosTable` contains the positions of the start and end of seven lines, in $^{1}/_{16}$ths of a character height. These are used to draw the segments on the screen. `Segments` contains ten bit vectors, with bits set according to which segments are to be drawn to represent the numbers from 0 to 9.

`App_Init` simply sets up the unknown instrument vector, then returns an OK report:

```
short  App_Init(void)
{
     UnknownInstr=NewInstrument;
     return(E_OK);
}
```

App_Exit just returns OK. The vector is removed by the Superscape system.

The actual code to draw the instrument is the most complex function so far:

```
static void __vrtcall NewInstrument(T_INSTRUMENT *Instr)
{
    short CharHeight, CharWidth, Length, x, y, i, Mask, Stroke;
    char String[80];
    T_GRLINE Line;
    if(Instr->Type!=21)
    {
        OldUnknownInstr(Instr);
        return;
    }
    CharHeight=Instr->Height;
    CharWidth =Instr->Height*2/3;
    Length =Instr->Width /CharWidth;
     *LongToDec(C_InsVal[Instr->Index].l,String,
        (short)Length+0x8000)=0;
    x=Instr->XPos;
    y=Instr->YPos;
    for(i=0;i<Length;i++)
    {
        Mask=Segments[String[i]-’0'];
        for(Stroke=0;Stroke<28;Stroke+=4)
            {
                Line.XStart=x+(CharHeight* PosTable[Stroke  ])/16;
                Line.YStart=y+(CharHeight* PosTable[Stroke+1])/16;
                Line.XEnd  =x+(CharHeight* PosTable[Stroke+2])/16;
                Line.YEnd  =y+(CharHeight* PosTable[Stroke+3])/16;
                Line.Illumination=255;

                if(Mask&1)
                  Line.Colour=(C_InsVal2[Instr->Index].l>>24)&0xFF;
                else
                  Line.Colour=(C_InsVal2[Instr->Index].l>>16)&0xFF;
                GrDrawLine(&Line);
                Mask>>=1;
            }
        x+=CharWidth;
    }
}
```

After defining some variables, the first thing to do is to check to see if the instrument type is one that is recognized. In this case, this is (arbitrarily) 21. Types 0 to 10 are used by the system, and 11 to 20 are reserved for future expansion. Any value from 21 to 99 can be used as a new instrument type. If the type is not 21, the old owner of this vector is called to see if it knows how to draw the instrument. The function then returns.

If it is type 21, the function continues. The height of one character is set to the height of the instrument and placed in `CharHeight`, and the width, `CharWidth`, to $^2/_3$ of that. The total number of characters across the instrument is calculated and placed in `Length`.

The instrument value is looked up from the array `C_InsVal`, using the index from the instrument itself, and converted to a string of length `Length` using `LongToDec`. The additional `+0x8000` added to `Length` is to make sure that leading zeros are included in the string.

The current x and y positions for the character to draw are set up in `x` and `y`, and a loop started that prints each character in turn.

The segments required are read from the array `Segments`, indexed by the character code from 0 to 9. This value is placed in `Mask` for later use. Another loop is started, going round once for each segment, or `Stroke`. There are seven segments, each represented by four values in the `PosTable` array, so `Stroke` takes values from 0 to 24 (less than 28), in steps of 4.

The position of the start and end points of the line is set up, using the values from `PosTable`, scaled by the height of the character, and added onto the x and y positions of the top left of this character. The illumination value is also set up. This is a reserved value and should be set to 255.

The color in which to draw the line depends on whether the segment is required in this character or not. The lowest bit of `Mask` is checked, and if set, the color is set to foreground. The foreground color is placed in the top byte of this instrument's `C_InsVal2` element, and needs to be shifted 24 times to bring it into the correct range.

If the segment is not required, the color of the line is set to the background color, which is in the upper mid byte of the `C_InsVal2` value, and thus needs to be shifted 16 bits to get it to the bottom of the value. The line is drawn using the graphics device driver call `GrDrawLine`.

The `Mask` is shifted along to the next bit ready for the next stroke, and the `Stroke` loop goes round again.

When this finishes, the `x` position is advanced by the width of one character, and the next character is drawn by the same process. After all the characters have been drawn, the function finishes.

⇒ Check the application

1. Build VEC3.DLL.

2. Start VRT, and load the application in the World Editor using File>Other Files>Load Application.

3. Choose Editor>Layout Editor to switch to the Layout Editor.

4. Drag the bottom line of the large blue box (the window onto the world) up a little to make room for the new instrument.

5. Choose Item>Create Instrument, and select User from the Select Instrument dialog box.

The User Instrument dialog box is displayed.

6. Enter 21 as the Type, and 251723776 as Value2 (0x0F010000, which indicates white—color 0x0F—on black—color 0x01).

7. Click OK, and drag a rectangle at the bottom of the screen for the instrument to be drawn into.

8. Switch to the World Editor [F4].

9. Select the Anchor object, and add the following SCL program to it:

```
instr(1)+=7;
```

This increases the displayed value of instrument 1 (our new instrument) by 7 every frame.

10. Switch to Visualiser to see the effect [F2].

This technique can be used to customize screen displays by adding new instrument types which may be more suited to the display of information specific to a particular world.

TIM1: Using the timer device

This example shows how to use the timer device. It maintains a list of ‘vents’—descriptions of code to call and when to call them. The application TIM1 uses this mechanism to increase SCL counter 0 by one every second.

The basic structure of TIM1.C should be familiar by now. The prototypes for `App_Init` and `App_Exit` having been declared, our new function’s prototype is:

```
static void __vrtcall IncCounter(void);
```

Since you can call this at any time, it takes no parameters, and returns none. The timer device handles the precise details of calling it. To tell the device how often you wish to call the routine, a timer node (an element of the list of events) must be set up. This is done in the `DECLARATIONS` section:

```
static T_TIMENODE TimerEvent= { 1000,
                                0,
                                NULL,
                                IncCounter,
                              };
```

This specifies that the function `IncCounter` is to be called every `1000` milliseconds (once per second). The two fields with `0` and `NULL` in must be set up like this, since they are filled in by the system.

The `App_Init` function adds this timer node to the active list with the timer device function `TmAddList`:

```
short App_Init(void)
{
     TmAddList(&TimerEvent);
     return(E_OK);
}
```

The address of the timer event is passed to the timer device which performs the necessary manipulation to add it to the list. `App_Exit` removes it again:

```
short App_Exit(void)
{
     TmRemList(&TimerEvent);
     return(E_OK);
}
```

This removes it from the active list, deactivating the function to which it refers.

The new timer function is simple. It merely increments counter 0, which can be found as the 0 element of the array `C_Counters`:

```
static void __vrtcall IncCounter(void)
{
    C_Counters[0]++;
}
```

⇒ Check the application

1. Build TIM1.DLL.

2. Start VRT, and load the application in the World Editor using File>Other Files>Load Application.

3. Select the Anchor object and add the following SCL program to it:

```
print(counter(0));
```

4. Switch to Visualiser [F2].

The numbers being printed are counting at one second intervals, indicating that the counter is being increased under timer control.

5. Press F12. The numbers reset to 0, and start counting again. Since the counters are automatically reset to 0 when you press F12 there is no need for specific reset code as required to reset the sequence instruction in SCL2.

TIM2: Timer device driven SCL

The TIM1 example altered a variable (the counter) which was easily accessible to SCL. In this example, TIM2 creates and keeps updated a string which contains the current time. This is of the form HH:MM:SS, so five past nine in the morning would be 09:05:00. To read the string, a new SCL function, `TString`, is registered, which returns the address of the string.

The source file, TIM2.C, starts in the usual way, and presents the prototypes for our functions in the `INTERNAL FUNCTION PROTOTYPES` section:

```
static void __vrtcall IncTString(void);
static void __vrtcall SCL_GetTString(void);
```

`IncTString` is called by the timer device to increment the value in the string, and `SCL_GetTString` is the function called by the SCL to get the string address.

The `DECLARATIONS` section looks like this:

```
static char     TString[20];

static T_TIMENODE TimerEvent= {
                              1000,
                              0,
                              NULL,
                              IncTString,
                              };

static T_COMPILEREC NewSCL= {
                              "TString",
                              0,
                              0x01,
                              0,
                              E_POINTER,
                              E_SSRPOINTER
                             };
static short    SCLCode;
```

The first line declares space for the time string, `TString`. This is filled in later. `TimerEvent` is a timer node structure showing that you wish to call the function `IncTString` every `1000` milliseconds.

`NewSCL`, the compiler record, defines the new `SCL` instruction as `tstring`, which has no inputs and one return value; is a pointer type function (`E_POINTER`); returns a 'real' pointer (`E_SSRPOINTER`). When an SCL instruction returns a pointer, it is usually taken as a pointer to a variable, and is dereferenced immediately. In this case you do not want this to happen, so the return type `E_SSRPOINTER` is used.

App_Init registers the SCL and timer events:

```
short App_Init(void)
{
     SCLCode=RegisterSCL(&NewSCL,SCL_GetTString);
     if(SCLCode<0)
         return(E_ERROR);
     TmAddList(&TimerEvent);
     return(E_OK);
}
```

App_Exit removes them again:

```
short App_Exit(void)
{
     TmRemList(&TimerEvent);
     UnRegisterSCL(SCLCode);
     return(E_OK);
}
```

The timer function, IncTString, reads the variables containing the current time and assembles them into a string:

```
static void __vrtcall IncTString(void)
{
     char *p;
     p=LongToDec(*C_Hour, TString, (short)0x8002);
     *p++=':';
     p=LongToDec(*C_Minute, p, (short)0x8002);
     *p++=':';
     p=LongToDec(*C_Second, p, (short)0x8002);
     *p=0;
}
```

The first line declares a pointer variable, p, which is used to point to the next character to fill in. The first call to LongToDec reads the value from *C_Hour and converts it to a two digit number at TString. The length is set to 0x8002 rather than 2 so that the leading zeroes are included (if not, the 09:05:00 example would read 9: 5: 0). After this call, p is set to point to the next character in the string. This is filled in with a ':' character. The same then happens for minutes and seconds, except that at the end, the last character is set to **0** to terminate the string correctly.

The SCL function, SCL_GetTString is:

```
static void SCL_GetTString(void)
{
     PushP(E_SSPCHAR,TString);
}
```

This pushes the address of TString onto the stack as a pointer to char type.

⇒ Check the application

1. Build TIM2.DLL.

2. Start VRT, and load the application in the World Editor using File>Other Files>Load Application.

3. Choose Editor>Layout Editor to switch to the Layout Editor.

4. Drag the bottom line of the large blue box (the window onto the world) up a little to make room for the new instrument.

5. Choose Item>Create Instrument, and select Text from the Select Instrument dialog box.

The Text Instrument dialog box is displayed.

6. The default values are correct, so click OK.

7. Drag a rectangle at the bottom of the screen for the instrument that is large enough to take eight characters.

8. Switch to the World Editor [F4].

9. Select the Anchor object, and add the following SCL program to it:

```
instr(1)=TString;
update(1);
```

This points the new instrument to the string, and makes sure that it is redrawn every frame. Instruments are only redrawn if their values change, but since the value of instrument 1 is the address of the string, which does not change, it would not usually be drawn—although the value contained in the string is changing, its address is not. The update command forces it to update every frame.

10. Switch to Visualiser to see the effect [F2].

TIM2 and VEC2: Using combinations of applications

Up to eight applications may be loaded at one time; if they are correctly written they should not interfere with each other. In the previous example, you set up a text instrument to view the time string, but in VEC2 you defined an SCL function, `Display`, which displays a string permanently in the bottom right corner of the display. There is no reason why you cannot load TIM2 and VEC2 together, so that you can use the benefits of both.

⇒ Check the applications

1. Start VRT, and load TIM2.DLL in the World Editor using File>Other Files>Load Application.
2. Use the same function to load VEC2.DLL.
3. Select the Anchor object, and add the following SCL progam to it:

   ```
   if(activate(me,0))
       Display(TString);
   ```

4. Switch to Visualiser, and click on Anchor.

   The current time is displayed in the bottom right of the screen, as expected.

Further examples

Some additional examples have been supplied with the SDK. They are written in the same style as the rest of the examples, with extensive comments, and are designed to illustrate additional functions or techniques possible with the Superscape SDK.

FFT1: Fast Fourier Transform

A demonstration of intensive mathematics using the SDK. The Fast Fourier Transform function would be possible to code in SCL, but it would run extremely slowly. By using the SDK, the power of C can be harnessed, making the function considerably faster.

IMG1: Saving a GIF image

A demonstration of using the image library to save a file in GIF format. Registers a new SCL function called MakeGif. As a first example, MakeGif(1000,1000) produces an interesting effect. You can view the GIF file without leaving the VRT by loading it into the Image Editor (using File>Import/Export>Import Picture).

MAKEOBJ: Creating an object

A demonstration of the SDK CreateVRTObject command, in which a nominated object is replaced with a small, rotated cube.

MEM1: Dynamic memory allocation

A demonstration of dynamic memory allocation under SCL. It registers two new SCL functions for allocating and freeing memory blocks, and also has the support code required to clean them up when the world is reset.

REG1 and REG2: Sharing data

A demonstration of the SDK's RegisterShare function, which can be used to share information between SDK applications which are loaded at the same time. Here, REG1 acts as an information provider, and REG2 uses the information that REG1 provides.

SCL8: Calling SCL functions directly

A demonstration of calling SCL functions directly. The new SCL command MoveMe effectively performs a `moveby(1000,2000,3000,me)` by calling the SCL functions me and moveby directly.

SER1: Using the serial device

A demonstration of using the serial device. New SCL functions are registered to read bytes from serial port COM2.

TEXT: Getting a string from the user

A demonstration of using the edit text routines to get a string from the user.

VEC4: Intercepting calls to a device driver

A demonstration of intercepting calls to device drivers. The graphics device function `GrPreCopy` is intercepted, and the picture to display modified before being copied to the graphics card.

Index