GX Developer

Custom Application Development Toolkit
for extending Oasis montaj and standalone applications

USER GUIDE and
REFERENCE MANUAL

Manual Version:

v7.0, (November 17, 2009)

www.geosoft.com

Contents

Geosoft GX Developer License Agreement

GEOSOFT grants you a license to use GX Developer including all GX Developer tools and source code included with the GX Developer installation for whatever purpose you like. You may provide copies of any or all of GX Developer to anyone you wish, and you may freely modify GX source code to meet your own purposes and distribute any derivative products you create as you see fit.

1. **SERVICES**:

GX Developer is supported by the GX User Community through the [GXnet] list, which is maintained by Geosoft. To join the [Gxnet] user list, select *Help->User Forums->Gxnet* from the Oasis montaj menu.

GX Developer is not directly supported by Geosoft unless you have a separate agreement with Geosoft to provide you with support.

2. **WARRANTY**:

GEOSOFT does not warrant that the functions contained in GX Developer will meet your requirements or will operate in the combinations which may be selected for use by you, or that the operation of the GX Developer will be uninterrupted or error free or that all program defects will be corrected.

Each GX Developer shall be furnished to me in accordance with the terms of this Agreement. No warranties, either express or implied, are made to me regarding the Licensed Program.

THE FOREGOING WARRANTIES ARE IN LIEU OF ALL OTHER WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Year 2000 Date Considerations

The Licensed Programs have been tested conform to DISC PD2000 1:1998 Year 2000 Conformity Requirements (www.bsi.org.uk/disk/year2000/2000.html), with the exception of clause 3.3.2, paragraph b. Section 3.3.2 paragraph b) requires that inferences for two-digit year dates greater than or equal to 50 imply 19xx, and those with a value equal to or less than 50 imply 20xx. The Licensed Programs will recognise all two-digit years as 19xx. This is to prevent errors importing historical data that pre-dates 1950. All dates that follow 1999 must use four digit dates in the Licensed Programs.

Introduction

This manual is a programming reference guide to Geosoft's **GX Developer** toolkit — a development environment that enables you to customize the **Oasis montaj** environment, add your own programs, and create full custom application suites that can be run from **Oasis montaj**.

The latest release of this document as well as the full download for **GX Developer** is available at: http://www.geosoft.com/support/devtools/

This document also covers the use of **GX Developer** External API. If you intend to use External API, please see ***GX Developer – External API*** in **Part 5:– Working with Other Programming Languages** (page 91).

Who should use GX Developer?

You should use **GX Developer** if you want to do any of the following, which are listed in order of difficulty:

- You want to modify an existing GX to meet your needs. Usually this is a simple modification to improve the efficiency of your use of the system.
- You want to create your own custom GX to do something special in the system. Your GX will use the GX API.
- You have your own algorithm in a separate DLL and you want to call it from **Oasis montaj**.
- You have a separate processing environment and you want to create/open Geosoft files (databases, maps) and/or apply Geosoft processes to your data.
- You are a third party developer who wants to create and market products that work with **Oasis montaj** or with the external API.

A basic requirement for using the **GX Developer** is that you are an earth science professional responsible for developing and/or distributing data processing software. However, you do not have to be a computer scientist to develop GX applications. The **GX Developer** programs and GX Programming Language are intended to be easy-to-learn so that anyone with the equivalent of a university level programming credit should be able to develop their own applications with a minimum of effort.

Because the **GX Developer** enables you to create custom interactive components for user input and online help, you may find that interface design and help development skills are useful.

GX Developer takes advantage of many of the features of the **Oasis montaj** Data Processing System, and we recommend that you acquire a working knowledge of **Oasis montaj**. If you have not already done so, please read the corresponding manual. This documentation describes the system in detail, including the Graphical User Interface, database environment and processing functionality.

*Do not use **GX Developer** if…*

- If you do not want to write your own programs.

- If you are not already familiar and comfortable with some other programming language. **GX Developer** should not be your first programming experience.
- If you are a FORTRAN programmer and, for whatever reason, you are not able to learn and understanding a C-like, object-oriented programming environment.
- If you require programming support and you do not have a **GX Developer** support agreement from Geosoft. Please contact your regional Geosoft office for more information about this service.

How this manual is organized

This manual is organized into six parts:

- ***Part 1: GX Developer Basics*** provides reference material about **GX Developer**, the GXC language and GRC resources (page 7).
- ***Part 2: Working with menus*** provides more information about working with **GX Developer**, object-oriented programming, and how to work with key components of the system (page 45).
- ***Part 3: GX Function Libraries*** describes the libraries (classes and functions) available in the GX API (Application Programming Interface) (page 82).
- ***Part 4: GX Debugger*** provides debugging software modelled after the general style of Microsoft's Visual Studio (page 87).
- ***Part 5: Working with Other Programming Languages*** provides detailed information on how to use **GX Developer** from other programming languages (page 91).
- ***Part 6: Using the GX API Externally*** provides instructions on how to create applications that can run indpendently from **Oasis montaj**.

Hardware and Software Requirements

Any system running **Oasis montaj** is capable of running **GX Developer**. You must have **Oasis montaj** installed and licensed on your system.

Installing GX Developer

Geosoft **GX Developer Toolkit** is a separate install package freely available on the download page of the Geosoft web page (www.geosoft.com). The toolkit is installed by running the EXE that you find in the downloaded ZIP file.

Click the **[Next>]** button and the *License Agreement* will be displayed. Please read the agreement carefully and choose to either "accept" or "do not accept". If "accept" is selected, click the **[Next>]** button and the *Setup Type* dialog will be displayed. Selecting the *Complete* option installs close to 90MB to your system in the C:\Program Files directory. Selecting *Custom* provides two options as follows:

GX Development	This includes the GX compilers and documentation that enables you to create custom GX's that can be run from within **Oasis montaj**. Choosing GX Development will

	also install all of the sample GX source code.
External Application Development	This includes libraries and examples to write custom applications using C#, C++. Fortran, VB to run with **Oasis montaj**.

When you have made your selection, click the **[Next>]** button and a final screen will be displayed indicating you are ready to install, click the **[Install]** button. Once all of the files have been copied, click the **[Finish]** button.

After you complete your installation, depending on what you have installed, you will have the following directories and files in your C:\Program Files\Geosoft\GX Developer directory:

Directory	What it Contains
apps\dll	Redistributable DLL files that your clients require to execute your external Geosoft application. These DLLs contain the entire Geosoft API.
apps\examples	Various examples to aid in your development. C, C#, and Fortran are highlighted.
apps\include	Header files that your code must include to use the Geosoft API calls.
apps\lib	Completed DLL or EXE application will link with these library files.
gx\bin	Compilers here: gpp.exe, grc.exe, gxc.exe, viewgx.exe.
gx\include	Header files (GXH files) which provide GX prototypes for all functions in the GX API.
gx\src	Directory contains the source code for almost all Geosoft GXs.
hlp	PDF documentation outlining **GX Developer**. Also contains the GX Developer compiled help for all methods.

Obtaining Additional Information and Help

The GX Programming Language is a subset of the C Programming Language. This document does not deal with the C language in detail and you may wish to refer to a C language programming guide, such as:

Kernighan, B.W. and Ritchie, D. M., 1988. The C Programming Language, PTR Prentice Hall, Englewood Cliffs, N.J., U.S.A.

We recommend that you refer to the sample Source code shipped with the **GX Developer**. It will provide you with a wealth of information about how to program GXs and use the GX Programming Language. The GX Developer compiled help file documents all classes and methods. This help system is produced from our Header (.GXH) files but contains the definitive documentation.

As part of your support contract for **GX Developer**, Geosoft can help you with questions on how to compile and run your GXs, and we can answer all general questions about **GX Developer**. We can also address any problems or usage differences from this documentation. However, we cannot support programming

questions or usage of the language to solve specific problems. If such help is required, Geosoft Technical Services can provide training or specific help on a fee basis. Please contact your regional Geosoft office for more information about this service.

Another source of help is the GXNET list, which is available to all users that have e-mail access. GXNET provides GX Developers with a forum to help each other and share information. Geosoft monitors this discussion group but we do not necessarily provide technical support via the list and we are not responsible for the technical accuracy of information obtained from GXNET.

You can subscribe to GXNET from the **Oasis montaj** *Help* menu, or online at http://www.geosoft.com/support/forums/

You can also find out more information about our list servers from the support section of the Geosoft web site: http://www.geosoft.com/support/

Part 1 - GX Developer Basics

This part of the documentation describes how to use the **GX Developer** compilers and provides reference information about the language rules and syntax. ***Part 2 - Working with GX Developer*** provides a topic-oriented reference guide that explains how to use a number of key components in the system.

You will use GXC, the GX Programming Language, to create GX programs. GXC is a C-like language that includes the majority of C expressions and control statements together with a simplified set of data types. Since our objective is to keep the language easy-to-use, GXC does not include support for pointers, structures or callable functions other than those included in the Geosoft function libraries (the GX API) or in other DLL's that conform to the GX API requirements.

GXC source code is compiled using the GX source code compiler *gxc.exe*. As with standard C, your source code is first translated by a C-Pre-processor that modifies the code based on Pre-processor directives. Pre-processor directives can be used to control which code is compiled, define constants, define macro statements to be re-used in your code, and to include other files in the compilation. See ***Pre-processor Directives*** for more information.

This chapter describes the basic C language that is implemented in GXC. Anyone familiar with C-Language programming may wish to proceed immediately to ***Working with Menus***.

For reference, following is a very simple GXC program that displays a message to the user. This program contains all the minimum required elements of a GXC program:

Source Code	Comments

`NAME        = "Say hello"`
`VERSION     = "v1 Copyright Geosoft Inc. 1999"`
`DESCRIPTION = "Just say Hello"`
	Every GX begins with these three keywords (NAME, VERSION and DESCRIPTION), which are assigned to strings that identify and describe the GX.

`// --- include the GX API prototype headers` `#include <all.gxh>`	The GX API (Application Programming Interface) is described in *gxh* files in the **GxDev/gxh** directory. This statement includes the file *all.gxh*, which in turn includes all other *gxh* files. The gxh files provide prototypes of all functions together with and function documentation for the GX programmer. Note that comments begin with the characters "//" and are ignored. Standard C-style comments "/*…*/" are also supported.
`// --- declare required variables here  ---`	This simple example does not require any variables, but if it did, all variables would be declared here.

```
{  // --- start program statements ---
   DisplayMessage_SYS("Hello","Hello
world");
}  // --- program end ---
```

The program statements are enclosed in braces { }. This program contains a single statement that displays a message. The usage of the DisplayMessage_SYS statement is described in GxDev/gxh/sys.gxh.

Note: Unlike many other programming languages, the C and GXC languages are case sensitive. This means that the tokens "`DisplayMessage_SYS`" and "displayMESSAGE_sys" are different to the GXC compiler.

GXC Language

This section describes the elements contained in the GXC Language.

Elements of GXC

TOKENS

Tokens are the smallest indivisible elements recognised by the GX compiler. A token is the source-program text that the compiler does not break down into component elements. Tokens may be keywords, identifiers, constants, strings, operators, or punctuation characters. Each of these is described in the following sections.

WHITE-SPACE CHARACTERS

Space, tab, linefeed, carriage-return, form-feed, vertical-tab and newline characters are called "white-space characters" because they serve the same purpose as spaces between words in a sentence on a page. In GXC, they serve to separate tokens and make GXC source code more readable. When reading source code, the GXC compiler ignores white-space characters except when they are used to separate tokens and as part of character constants or strings. Note that the compiler also treats comments as white space.

COMMENTS

A comment is a sequence of characters that begins with a forward slash – asterisk (`/*`) and ends with an asterisk – forward slash (`*/`). For example:

```
/* This is a comment.
   This is a continuation of the comment. */
```

GXC also supports single-line comments preceded by two forward slashes as in the following example:

```
// This is a valid comment in GXC.
```

The next newline character that is not preceded by a backslash terminates comments in this format (\).

KEYWORDS

Keywords are are tokens that have special meaning to the GXC compiler. They consist of the following:

break	case	continue	default	DESCRIPTION
do	else	for	if	int
NAME	real	RESOURCE	sizeof	string
switch	typedef	VERSION	void	while

GX-specific keywords consist of the NAME, VERSION, DESCRIPTION and RESOURCE keywords listed above. These identify graphical resources and special sections in Source files to the compiler. The other keywords are implemented exactly as in the C language.

Keyword names cannot be used for any purpose other than the defined by GXC. However, identifier names can be replaced by the pre-processor if they have been redefined using a #define statement.

IDENTIFIERS

"Identifiers", or "symbols" are the names that you supply for variables and functions in your programs. You cannot use a keyword as an identifier name. You create an identifier by specifying it in the declaration of a variable or function. You cannot use an identifier unless it has been declared.

We strongly recommend that you begin all identifier names with a lowercase character that describes the data type, or in the case of functions, the function return value. This makes your code much easier to read and maintain. You will find the following recommended prefix characters and their data types used in the Geosoft GX code:

- `i` an integer
- `r` a real (floating-point)
- `s` a string
- `h` a class handle (This convention is often used in Geosoft source code, but not always. Identifiers that begin with an upper-case character can usually be assumed to be class handles.)

DECLARING VARIABLE IDENTIFIERS

All variables used in a GXC program must be declared immediately before the opening brace for the GXC program statements. Variables are declared by specifying a data type followed by a comma-separated list of variable names and a semicolon to end the declaration statement. The GX language supports a simplified selection of variable types - int, real, and string as well as Class handles:

int The int variable type stores integers, and replaces the short and long types found in normal C. It is equivalent to the 4-byte long type. `int` variables may also be declared and accessed as arrays, as in the following examples:

```
int iVal;
int(4) iNum;
{
   iVal = 2;
```

```
   iNum[0] = iVal + 1;
   iNum[1] = 2*iNum[0];
}
```

Note that array allocations are done with round brackets "(4)", while access is performed using square brackets "[0]". As with C variables, indexing begins at 0 and proceeds to one less than the allocated size. Note also that the declaration syntax for arrays is a different from standard C in that the array size is part of the variable type, not the variable name.

real The `real` variable type stores floating point numbers, and replaces the `float` and `double` types found in normal C. It is equivalent the 8-byte `double` type. `real` variables may be declared and accessed as arrays, as in the following examples:

```
real rVal;
real(4) rNum;
{
   rVal = rNum[0];
   rNum[2] = rNum[1];
}
```

string the `string` type is roughly equivalent to the `char` type in normal C. `string` variables must be declared with a maximum string length, as in the following examples:

```
string(DB_SYMB_NAME_SIZE)  sInCh,sChan;
string(80) sTextLine;
string(GS_MAX_PATH)   sFile,sData;
```

Unlike the `int` and `real` variables, `string` variables may not be accessed by element indices; for example the following is illegal:

```
sFile[6] = 'a';    // illegal statement
```

Instead, the STR library functions must be used to manipulate strings.

Class Handles are instances of Geosoft object classes that are defined in the GX API. To use a class you must declare a class variable as in the following examples:

```
DB  hData;          // hData is a handle to a DB class that deals with databases.

DGW hDialog;        // hDialog is a handle to a DGW class that deals with resource dialogs.

LST hChannelList;   // hChannelList is a handle to an LST class that deals with lists.
```

DECLARING FUNCTION IDENTIFIERS

GXs call functions to perform most of the real processing work and to control the **Oasis montaj** interface. The suite of functions available to the GX programmer is called the GX API, or GX Application Programming Interface. Before calling a function, the function identifier must be declared in a function prototype statement. All Geosoft function prototypes are defined in the GXH files in the GxDev/gxh directory. You can refer to ***GX Function Libraries*** for more information on working with GX functions in the GX API.

A function prototype statement is normally contained in a separate GXH file that is included (using the `#include` directive) in your source code immediately before your variable declarations. Following is the syntax for a function identifier:

[*dll_name*] *return_type function_name(arg_1_type,arg_2_type,...);*

dll_name or license_type	The name of the DLL that contains the function. Note that functions provided by Geosoft use this field to identify the type of license this function can used under. The valid license types for Geosoft are: "_public", "_licensed", "_extended" and their application versions: "_public_app", "_licensed_app" and "_extended_app". See the **Geosoft License Issues** section for more details.
return_type	The function return value type, which can be an **`int`**, a **`real`**, a ***class***, or **`void`**. The type **`void`** indicates that the function does not return a value. Note that in C and GXC, you can call a function that returns a type just as you would call a void function.
function_name	The function identifier name that you use to call the function.
arg_#_type	A list of types that correspond to the required argument types. Unlike C, GXC does not support functions with undefined argument lists. The "`var`" qualifier should precede any arguments that may be modified by the function.

For example, the following prototype defined the function identifier `ReadReal_BF` in the BF class, which can be used to read a real value from a binary file:

```
[_public] void
ReadReal_BF(BF,     // BF handle
            int,    // BF element type, one of GS_? types in Geosoft.gxh
        var real); // real data returned
```

This is a void function contained in the geogx dll. The function takes three arguments, and the third argument will be modified by the function to hold the real value read from the binary file. You can refer to the **Geosoft.gxh** file to find the basic data types that are supported in the read.

You can create function declarations to functions in your own DLLs by creating your own GXH file and including it in your GXC code. All of your functions must be contained in a Dynamic Link Library (DLL) that is accessible when your GX is run. Refer to **DLLs within Oasis montaj** for more information on working with your own libraries.

CONSTANTS

Constants are tokens that evaluate to a number or a string in your code. Constants can be integers, floating-point numbers or strings enclosed in double quotes.

INTEGER AND FLOATING POINT CONSTANTS

Numeric constants are either integer or floating point real numbers, as illustrated in the following example:

```
int iValue;
real rValue;
```

```
iValue = 1234;        // assigns integer to constant
iValue = 0777;        // assigns integer to constant octal 777 (decimal 511)
ivalue = 0xff;        // assigns integer to constant hex ff (decimal 255)
rValue = 32.1;        // assigns real to a constant
rValue = 3.21e1;      // Real constant in exponential notation
```

Note that numeric constants can only be used in context with the implied constant type. The implied type of a constant is integer unless there is a decimal place or the constant is in exponential format. You can use a type cast to change the type of a constant if required (see ***Casting Types***).

STRING CONSTANTS

A string constant is a sequence of characters enclosed in double quotes. Strings are always terminated by an implied null character. In GXC, you must use the STR methods defined in str.gxh to work with strings. For example, the Strcpy_STR function will copy a string constant to a string variable:

```
String(16) sName;
Strcpy_STR(sName,"Bill");     // copies the constant "Bill" to string variable sName
```

Special characters in a string can be defined using escape sequences. An escape sequence consists of a backslash character (\) followed by one or more characters. The following table defines the ANSI escape sequences supported in GXC:

Escape Sequence	Represents
\a	Bell
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\"	Double quotation
\\	Backslash
ooo	ASCII character in octal
\x*hhh*	ASCII character in hexadecimal notation

If a backslash character precedes a character not specified in the table, the undefined character is interpreted as the character itself. If a backslash appears as the last character in a line (immediately before a new line character), the next line is considered a continuation of the current line. This can be useful for defining long string constants.

DEFINING CONSTANTS

Constants that are used repeatedly in your GX source code can be defined once using a `#define` statement to establish a token name to represent the constant. The `#define` statement must start in column 1 of the source line, it must be the only statement on the line, and it must precede the first use of the constant token. For example, here we define the token "TWOPI" to represent the numeric constant 6.2831853:

```
#define TWOPI  6.2831853
```

It is a common convention to use all upper-case characters for the defined token, in this case `TWOPI`. All occurrences of `TWOPI` in the source code that follow this line will be replaced by the constant "`6.2831853`":

```
rCirc = TWOPI * rRadius;  // seen as rCirc = 6.2831853 * rRadius
```

The GX Programming Language predefines many constant values in the GXH header files, both in the **geosoft.gxh** file and in the individual function library header files. For example, the integer and real dummy values are defined in **geosoft.gxh** as follows:

```
#define iDUMMY -2147483647
#define rDUMMY -1.0E32
```

Note that `#define` statement lines cannot contain comments.

You can use the backslash character before the end of a line to define a long string constant that continues onto the next line:

```
#define LONG_STR "Long strings can be br\
oken into two or more\
pieces."
```

Note that if there are any white-space characters following the backslash, other than a newline character, the continuation is not recognised.

OPERATORS

Operators are symbols (both single characters and character combinations) that enable you to manipulate values. There are three types of operators. A unary operator consists of either a unary operator prefix added to an operand, or the `sizeof` operator followed by an expression. The expression can be either the name of a variable or a cast expression. If the expression is a cast expression, it must be enclosed in parentheses. A binary expression consists of two operands joined by a binary operator. A ternary expression consists of three operands joined by the ternary operator. Note that this latter type of expression is supported in the EXP class but not by the GX syntax. GXC includes the following operators:

Symbol	Type	Description
`+ -`	unary	Positive and negative operand.

`!`	unary	Logical NOT applied to operand.
`sizeof`	unary	Size of operand in bytes
`(int) (real) (`*class*`)`	unary	Cast (convert) operand on the right to the type specified. Casting a real type to an int will truncate the real to the integer closest to zero.
=	binary	Assignment of the evaluated operand on the right of the operator to a variable on the left.
++ --	unary	Increment and decrement assignment. If the operator precedes the operand, the operand is incremented or decremented before the expression is evaluated. If the operator follows the operand, the operand is incremented or decremented after the expression is evaluated.

= += -= *= /= %=	binary	Arithmetic operators evaluate the operand on the right of the operator and apply the arithmetic operator to change the operand on the left. For example, the following are equivalent:
		`iVal += 4 * iData;`
		`iVal = iVal + (4 * iData);`

*** / %**	binary	Multiply, divide and remainder. The remainder operator evaluates to the remainder after dividing the left operand by the right operand.
+ -	binary	Addition and subtraction.
< > <= >= == !=	binary	Relational evaluate to 1 if true, 0 if false
&&	binary	Logical AND evaluates to 1 (true) if left and right operands are non-zero, 0 (false) otherwise.
||	binary	Logical OR evaluate to 1 (true) if either left or right operand is non-zero , 0 (false) otherwise.
,	binary	Sequential evaluation evaluates expression on left, then expression on the right.

?:
[NOT IMPLEMENTED]
	ternary	The conditional expression evaluates the first operand (before the ?), and if true (non-zero), evaluates second operand (between ? and :), or if false evaluates the third operand (after the :). For example:
		iVal = (4>5)?5:-10;
		is equivalent to
		iVal = -10;

The following table gives the precedence order and association for the supported GXC operators:

Symbol in order of precedence	Type	Association
[] () .	Expression	left to right
Postfix ++ and postfix --	Unary	right to left
Prefix ++ and prefix --	Unary	right to left

sizeof + - !	Unary	right to left
(int) (real) (*class*)	unary cast	right to left
* / %	multiplication	left to right
+ -	additive	left to right
< > <= >=	relational	left to right
== !=	equality	left to right
&&	logical AND	left to right
||	logical OR	left to right
?:	conditional	right to left
= *= /= %= += -=	simple and compound assignment	right to left
,	sequential evaluation	left to right

PUNCTUATION CHARACTERS

Punctuation characters are used to control how the compiler processed the source code and evaluate expressions. The following table documents the punctuation characters.

Characters	Description
;	The semicolon marks the end of an expression. One expression is always completely evaluated before starting on the next expression.
{ }	Braces mark the start and end of a group of expressions that are evaluated in order. Braces are normally used to group statements that must be executes within a control structure.
( )	Parentheses control the order of evaluation in arithmetic and logical expressions such that everything inside parenthesis is evaluated before evaluating the next associated operator outside parentheses.

UNSUPPORTED C-LANGUAGE OPERATORS AND TOKENS

As you can see from this summary of operators, GX Programming Language provides almost full support of all C language operators. There are two sets of standard operators that you will not find here — bitwise operators and address operators.

Bitwise operators (<<, >>, &, |, ^ and ~) enable you to perform very low-level manipulations at the bit level. Address operators (& and *) are for manipulating the addresses of variables and for working with pointers, respectively.

Since GX programming does not require low-level functions, addresses or pointers, they are not included in the language. String literals (tokens) are also not supported. Instead we provide a class of library functions (STR) that you can use for working with text.

Defining Constant Expressions

Typically, you use constant expressions for both control flow (i.e. incrementing counters) and computational purposes. The format consists of a left-hand expression (variable) of a certain type, an assignment operator (=) and a constant of the matching type. Examples are:

```
iConstant = 1;
rConstant = 2.27052;
```

The system evaluates constants at compile time so that you will know immediately whether they are correct.

Defining and Using Math Expressions

A powerful capability in **Oasis montaj** is to apply some mathematical operation to selected rows in one or more channels of data, such as in the following expression:

```
Z = 2.5*sqrt(X*X + Y*Y)
```

where `X`, `Y` and `Z` are the names of channels.

The GX Programming Language provides this same functionality with the math expression object EXP. To define the above math expression within a GX you could type:

```
ZExp=Create_EXP(Data,"Z=2.5*sqrt(X*X + Y*Y);",64);
```

Note the semicolon ";" terminating the expression. The number `64` represents the maximum length of the expression string. A call to the Math_DU function is used to apply the expression to a given line:

```
Math_DU(Data,Line,ZExp);
```

In many cases, however, channel and variable values may not be known ahead of time to the programmer, but instead are determined at run-time, perhaps by being selected by the user. In this case, the expression may be defined using replaceable parameters, indicated by pre-pending dollar-sign characters "$" to the local variable names:

```
GetString_SYS("MYGX", "X_CHAN", sXCh);
GetString_SYS("MYGX", "Y_CHAN", sYCh);
GetString_SYS("MYGX", "Z_CHAN", sZCh);
rMult = 2.5;

ZExp = Create_EXP(Data,"$sZCh=
                  $rMult*sqrt($sXCh*$sXCh+$sYCh*$sYCh));",350);
```

The `350` value at the end of the expression, giving the maximum expression length in characters, recognises the fact that individual channel names may each contain up to 64 characters (defined by DB_SYMB_NAME_SIZE in db.gxh) characters. (See the EXP.GXH header file for details).

Math expressions may also be applied to grids. The expression syntax is identical, and the procedure is similar, but it uses the IEXP object. (See the IEXP.GXH header file for details).

CASTING TYPES

Arithmetic operations in **GX Developer** must remain true to type. Integer and real types can be mixed only if type casting is performed. The `(real)` and `(int)` operators are cast operators that convert the operand on the right to the specified type. Other programming languages (including C) will perform casting automatically, but this is a common source of programming errors. GXC requires that the GX programmer make all type casts explicit.

For example, the following operations will produce an error on compilation:

```
iVal = 10.0;         // should be:  iVal  = 10;
rVal = 0;            // should be:  rVal = 0.0;
iVal = rVal + 1;     // should be:  iVal = (int)rVal + 1;
rVal = iVal/rVal;    // should be:  rVal = (real)iVal/rVal;
```

Class handles and symbols can also be stored in an integer variable, which can be useful in some applications. However, to store the handle in an integer variable requires that the class be cast to an integer. When the class handle is used, it must be cast back to the type of the class. The following example illustrates this:

```
DB hDB;              // working database
int(10) iDB;         // array of 10 database handles
int i;
string(GS_MAX_PATH) sName;
{
   for (i=0;i<10;i++) {

      // --- code to get databases ---
      ...
      iDB[i] = (int)hDB;
   }

   // --- get the name of the 5'th database ---

   GetName_DB( (DB)iDB[4],DB_NAME_FILE,sName);

   ...
}
```

Statements

The statements of a GXC program control the flow of program execution. In GXC, as in other programming languages, several kinds of statements are available to perform loops, to select other statements to be executed, and to transfer control. This section describes the following GXC statements in alphabetical order:

Statement	Description
break	breaks control out of a do, while, for or switch statement

compound statement	sequence of statements enclosed in braces
continue	go immediately to the next iteration of a loop
do-while	repeat while a condition is true
expression statement	A single expression
for	loop a specified number of times
if	controls conditional branching
switch	transfer control based on a complex condition
while	while a condition is true, process a statement

THE BREAK STATEMENT

The **break** statement terminates the execution of the nearest enclosing **do**, **for**, **switch** or **while** statement in which it appears. For example:

```
switch (iTest)
{
      case 0:
            rX = rX + 0.25;
            break:
      case 1:
            rY = rY + 0.25;
            break;
      case 2:
            break;      // Does not increment rX or rY when iTest =
2
      default:
}

for (i=0;i<1000;i++) {
   if (rData[i] == rDUMMY) break;
   rData[i] = 0.0;
}
```

In a **switch** statement, **break** is commonly used as the last statement in each case block to prevent program flow moving into the next case block. In a **do**, **for**, or **while** loop, the **break** statement can be used to terminate the loop prematurely.

COMPOUND STATEMENTS

A compound statement (also called a “block”) typically appears as the body of another statement, such as the `if` statement. Compound statements are a sequence of statements enclosed by braces (`{ }`). An example is:

```
if (Tb != NULLTB) {
   ProgName_SYS("Load levels",1);
   LockSymb_DB(Data,Level,DB_LOCK_READWRITE,DB_WAIT_INFINITY);
   TableLineFid_DU(Data,Level,Ref,LTBL,Tb);
   UnLockSymb_DB(Data,Level);
}
```

Note: Unlike C, you cannot declare new variables within a compound statement.

THE CONTINUE STATEMENT

The **continue** statement passes control to the next iteration of the nearest enclosing **do**, **for** or **while** statement, bypassing any remaining statements in the **do**, **for** or **while** statement body.

Within a **do**, or **while** statement, the next iteration starts by re-evaluating the expression of the **do** or **while** statement.

A **continue** statement in a `for` statement causes the last expression of the **for** statement to be evaluated, then the conditional statement is evaluated. Depending on the result, the statement body is iterated or the loop terminates.

Following is an example of a **continue** statement:

```
for (i=0;i<1000;i++) {
   if (rData[i] == rDUMMY) continue;
   rData[i] = 0.0;
}
```

This example will set all non-dummy values in the `rData` array to 0.

THE DO-WHILE STATEMENT

The **do-while** statement lets you repeat a statement or compound statement until a specified expression becomes false. The statement or compound statement is always executed at least once. For example:

```
i = 0;
do {
   rY = rData[i];
   i++;
} while ( (i < iLength) && (rY != rDummy) );
```

THE FOR STATEMENT

The **for** statement has the following form:

`for ( `*init-expression*`; `*cond-expression*`; `*loop-expression*` ) statement`

Execution of a **for** statement proceeds as follows:

1. The *init-expression*, if any, is evaluated. This is normally used to establish the initial values for the variables used to control the loop.
2. The *cond-expression*, if any, is evaluated. If the expression evaluates to true (any non-zero value) the statement body is executed.
3. After each iteration of the statement body, the *loop-expression* is evaluated, then the *cond-expression* is evaluated. If the *cond-expression* is still true, the statement body is repeated. This process continues until the *cond-expression* evaluates to be false (0).

The most typical usage of a **for** loop is to repeat a statement body a fixed number of times as in the following example:

```
rXvalue =  2.5;
iNth = 5;
for (iPower=0; iPower < iNth; iPower++)
  rXvalue *= rXvalue;
```

In this example, the program calculates the N^{th} power (in this case N = 5) of a predefined r`Xvalue`.

EXPRESSION STATEMENTS

An expression statement is a single statement that is evaluated according to the operator rules defined in the ***Operators*** section of this manual. Expression statements always end with a '**;**'. Below are examples of expression statements:

```
rX = 4.5;                      // simple assignment
rD = rSqrt(rX*rX + rY*rY);     // function call returns a real value
rX = rY = 0.0;                 // 0.0 is assigned to both rX and rY
```

THE IF STATEMENT

The **`if`** statement tests a condition and if true (non-zero) executes the **`if`** statement body, or if false, and an **`else`** clause exists, executes the else statement body. Following are examples:

```
if (rData == rDUMMY)
   rX = rDUMMY;

if (rData == 0.0) {
   rY = 0.0;
   rX = rDummy;
} else {
   rY = rDUMMY;
   rX = 0.0;
}
```

When nesting **`if`** statements and **`else`** clauses, the compiler will associate an **`else`** statement with the most recent **`if`** statement that lacks an **`else`**. However, this can easily lead to errors in your code and we recommend that you use braces to clearly identify the associations of **`if`** and **`else`** statements.

THE SWITCH STATEMENT

The **`switch`** statement helps control complex conditional and branching operations. The **`switch`** statement defines a branching structure for testing integer values and performing certain actions depending on the integer value that is retrieved. An example is:

```
switch (iTest)
{
      case 0:
            rXvalue = rXvalue +  .25 ;
            break:
```

```
      case 1:
            rYvalue = rYvalue + .25;
            break;
      case 2:
            break;        // Does not increment x or y when iTest = 2

      default:            // control will come here if no cases match the iTest
}
```

The **switch** causes program control to move to the **case** line that has the same value as **iTest**. Note that once control has advanced to the appropriate case, each statement will be processed in order, including advancing into the next **case** or **default** section. The **break** statement is commonly used to limit execution to the statements within one **case**.

THE WHILE STATEMENT

The **while** statement lets you repeat a statement or compound statement until a specified expression becomes false (zero). For example:

```
i = 0;
while ( (i < iLength) && (rData[i] != rDummy) ) {
   rY = rData[i];
   i++;
}
```

Calling Functions

Most of the actual processing work in a GX is carried out by calling GX API functions, or perhaps functions from one of your own DLL libraries. All GX API functions are defined in GXH files contained in the GxDev/gxh directory. The documentation for these functions is found in the GxDev/hlp/GxDeveloper.chm as well as online at http://www.geosoft.com/support/devtools/

In GXC, functions either return a value or are “void”, which means they do not return a value. Functions that return a value can be used in statements anywhere that the value type can be used. Functions that do not return a value can only be used as statements. Functions that return a value can also be used as statements, in which case the return value is simply not used.

The following example illustrates use of functions in the RA class (see GxDev/gxh/ra.gxh), which lets you read ASCII files:

```
RA hRA;
string(128) sLine;
int i,iLines;
{
   hRA = Create_RA("test.txt");     // create handle to the file "test.txt"
   iLines = iLen_RA(hRA);           // get the number of lines in the file
   for (i=0;i<iLines;i++) {         // go through every line
      iGets_RA(hRA,sLine);          // read the next line, ignore return value
      DisplayMessage_SYS("test.txt",sLine);   // display the line
   }
   Destroy_RA(hRA);                 // destroy the handle to the file
}
```

Please refer to the section ***Working with Library Functions*** for more information about using the GX API functions.

Preprocessor Directives

Pre-processor directives, such as `#define` and `#include`, are processed in the first stage of compiling a GXC program. All directives are replaced by the result of acting on the directive. For example, the pre-processor can replace tokens in the text, insert the contents of other files into the source file, or suppress compilation of part of the file by removing sections of text. Pre-processor lines are recognised and carried out before macro expansion. Therefore, if a macro expands into something that looks like a pre-processor command, that command is not recognised by the pre-processor.

The GXC pre-processor recognises the following directives:

#define	#undef	#elif
#if	#include	#else
#ifdef	#endif	#ifndef

The number sign (#) must be the first character on the line containing the directive; white-space characters can appear between the number sign and the first letter of the directive. Some directives include arguments or values. Any text that follows a directive (except an argument or value that is part of the directive) must be enclosed in comment delimiters (/* */) (single-line comment delimiters (//) are not supported on directive lines). Lines containing pre-processor directives can be continued by immediately preceding the end-of-line marker with a backslash (\).

Pre-processor directives can appear anywhere in a source file, but they apply only to the remainder of the source file.

#DEFINE

You can use the **#define** directive to give a meaningful name to a constant in your program. The two forms of the syntax are:

#define *identifier token-string*

#define *identifier*[**(** *identifier***,** *...* **,** *identifier* **)**] *token-string*

The **#define** directive substitutes *token-string* for all subsequent occurrences of an *identifier* in the source file. The *identifier* is replaced only when it forms a token. (See

Tokens.) For instance, *identifier* is not replaced if it appears in a comment, within a string, or as part of a longer identifier.

A **#define** without a *token-string* removes occurrences of *identifier* from the source file. The *identifier* remains defined and can be tested using the **#ifdef** directive.

The *token-string* argument consists of a series of tokens, such as keywords, constants, or complete statements. One or more white-space characters must separate the *token-string* from the *identifier*. This white space is not considered part of the substituted text, nor is any white space following the last token of the text.

Formal parameter names appear in *token-string* to mark the places where actual values are substituted. Each parameter name can appear more than once in *token-string*, and the names can appear in any order. The number of arguments in the call must match the number of parameters in the macro definition. Liberal use of parentheses ensures that complicated actual arguments are interpreted correctly.

The second syntax form allows the creation of function-like macros. This form accepts an optional list of parameters that must appear in parentheses. References to the *identifier* after the original definition replace each occurrence of *identifier* (*identifier*, ..., *identifier*) with a version of the *token-string* argument that has actual arguments substituted for formal parameters.

The formal parameters in the list are separated by commas. Each name in the list must be unique, and the list must be enclosed in parentheses. No spaces can separate *identifier* and the opening parenthesis. Use line concatenation — place a backslash (\) before the newline character — for long directives on multiple source lines. The scope of a formal parameter name extends to the new line that ends *token-string*.

When a macro has been defined in the second syntax form, subsequent textual instances followed by an argument list constitute a macro call. The actual arguments following an instance of *identifier* in the source file are matched to the corresponding formal parameters in the macro definition.

This example illustrates the **#define** directive:

```
#define WIDTH 80
#define LENGTH ( WIDTH + 10 )
```

The first statement defines the identifier WIDTH as the integer constant 80 and defines LENGTH in terms of WIDTH and the integer constant 10. Each occurrence of LENGTH is replaced by (WIDTH + 10). In turn, each occurrence of WIDTH + 10 is replaced by the expression (80 + 10). The parentheses around WIDTH + 10 are important because they control the interpretation in statements such as the following:

```
iVar = LENGTH * 20;
```

After the pre-processing stage the statement becomes:

```
iVar = ( 80 + 10 ) * 20;
```

which evaluates to 1800. Without parentheses, the result is:

```
iVar = 80 + 10 * 20;
```

which evaluates to 280.

The following example of macros with arguments illustrate the second form of the **#define** syntax:

```
// Macros to return minimum and maximum values
#define MIN(a,b) (((a)<(b))?(a):(b))
#define MAX(a,b) (((a)>(b))?(a):(b))
```

Arguments with side effects sometimes cause macros to produce unexpected results. A given formal parameter may appear more than once in *token-string* (as in the previous example). If that formal parameter is replaced by an expression with side effects, the expression, with its side effects, may be evaluated more than once.

The **#undef** directive causes an identifier's pre-processor definition to be forgotten.

If the name of the macro being defined occurs in *token-string* (even as a result of another macro expansion), it is not expanded.

#UNDEF

As its name implies, the **#undef** directive removes (undefines) a name previously created with **#define**.

#undef identifier

The **#undef** directive removes the current definition of *identifier*. Consequently, the pre-processor ignores subsequent occurrences of *identifier*. To remove a macro definition using **#undef**, give only the macro *identifier*; do not give a parameter list.

You can also apply the **#undef** directive to an identifier that has no previous definition. This ensures that the identifier is undefined. Macro replacement is not performed within **#undef** statements.

#IF, #ELIF, #ELSE, AND #ENDIF

The **#if** directive, with the **#elif**, **#else**, and **#endif** directives, controls compilation of portions of a source file. If the expression you write (after the **#if**) has a nonzero value, the line group immediately following the **#if** directive is retained in the translation unit. The following summarises the syntax of these directives:

#if *constant-expression*
#ifdef *identifier*
#ifndef *identifier*

if parts…

#elif *constant-expression*

else-if parts ...

#else

else parts :

#endif

Each **#if** directive in a source file must be matched by a closing **#endif** directive. Any number of **#elif** directives can appear between the **#if** and **#endif** directives, but at most one **#else** directive is allowed. The **#else** directive, if present, must be the last directive before **#endif**.

The **#if**, **#elif**, **#else**, and **#endif** directives can nest in the text portions of other **#if** directives. Each nested **#else**, **#elif**, or **#endif** directive belongs to the closest preceding **#if** directive.

All conditional compilation directives, such as **#if** and **#ifdef**, must be matched with closing **#endif** directives prior to the end of file; otherwise, an error message is generated. When conditional-compilation directives are contained in include files, they must satisfy the same conditions: There must be no unmatched conditional-compilation directives at the end of the include file.

#INCLUDE

The **#include** directive tells the pre-processor to treat the contents of a specified file as if those contents had appeared in the source program at the point where the directive appears. You can organise constant and macro definitions into include files and then use **#include** directives to add these definitions to any source file. Include files are also useful for incorporating declarations of external variables and complex data types. You only need to define and name the types once in an include file created for that purpose.

#include "*path-spec***"**

The *path-spec* is a filename optionally preceded by a directory specification. The filename must name an existing file. The syntax of the *path-spec* depends on the operating system on which the program is compiled.

GRC Resources and Dialogs

GX Developer Resources are the visual elements of a GX — specifically, user-interface dialog windows. If your GX will present a dialog to the user to obtain information from the user, you will use a resource to define the dialog.

A Geosoft Resource file contains dialog resources and resource components. The Geosoft Resource Compiler (GRC) must compile a Resource file to produce a Geosoft Resource (GR) file and a Geosoft Resource Header (GRH) file. The resource file and file header must be included in the corresponding GXC source code file (using the `#include` Pre-processor directive) before declaring any variables.

You can work with any of the following resource types:

- FORM (dialog box containing edit fields and buttons).

- LIST (dropdown list containing item components)
- HELP (help text)

The FORM resource can contain edit fields and buttons that may be linked to LIST or HELP resources:

- EDIT simple edit field
- FEDIT file/browse field
- LEDIT list selection field
- CEDIT colour selection field
- EBUT exit button
- HBUT help button

This section describes each resource and component type and summarises the syntax for using them. A GRC component has the following basic syntax:

keyword,parameter,parameter,...

When you are referring to syntax descriptions or tables, remember that all items in {braces} are optional. The compiler ignores blank lines and lines that begin with "//" characters.

For example, following is a resource file that defines a simple dialog:

```
//
// DIFF.GRC
//-------------------------------------------------------------------------
RESOURCE,FORM,DiffForm,"Calculate differences",-1
LEDIT,,,16,"Channel to difference",R,FORCE,,CHAN
LEDIT,,,16,"Output difference channel",R,,,CHAN
EDIT,,,16,"Number of differences (>=1)",R,INT
LEDIT,,,16,"Normalize differences?",R,FORCE,no,NORM
EBUT,&OK,0
EBUT,&Cancel,1,CANCEL
HBUT,&Help,help

RESOURCE,LIST,CHAN

RESOURCE,LIST,NORM
ITEM,"No"
ITEM,"Yes"

RESOURCE,HELP,help,diff.rtf
```

The FORM Resource

A FORM is a dialog box that prompts you for information or displays information. In montaj, you see forms when you select a menu option followed by ellipses (...) or when you run a GX that requires interactive parameter entry. Form elements include a title bar, entry fields for text, numbers, lists of files, and push buttons.

When a dialog box is displayed, you can either type any required and optional information, and press a button or press <Esc> to close the box without making

changes. Many forms will include a <Cancel> button, which is equivalent to pressing <Esc>.

SYNTAX

```
RESOURCE, FORM, resource_name, "title", esc_val
```

PARAMETERS

The following table describes the parameters, purpose and/or values you can specify for this resource.

Parameter	Purpose and/or Allowed Values
resource_name	Identifies an individual resource. You must use a unique name for each resource in a GX.
Title	The text that you want to display at the top of the dialog (dialog banner). The title of a dialog can be changed from a GX using the SetTitle_DGW function as follows: `SetTitle_DGW(hDGW,"My new title");`
esc_val	Specifies the number that is returned if the user presses <Esc>. If this occurs, no validation is done (entered values are not stored or evaluated). This value should normally be set to –1, or the same as used for a [Cancel] button.

EXAMPLES

```
RESOURCE,FORM,SAMPLE,"Sample GX Dialog Title",-1
```

The above example defines a dialog with the resource name "SAMPLE". In your GX source code, it would be referenced as follows:

```
// --- Create the Dialogue ---

hDGW = Create_DGW("SAMPLE");
```

In this sample, `hDGW` has been declared to be a DGW class instance.

Resources have identifiers that are defined in the ".grh" header file produced when the resource is compiled. This file is included using the `#include` "*filename*`.gxh`" command.

FORM Components

A FORM resource will have one or more field and button components:

- Text entry fields (EDIT)
- File name and path entry fields (FEDIT)
- Static and dynamic lists (LEDIT)
- Colour selection tool (CEDIT)
- Pushbuttons (EBUT)

- Help pushbuttons (HELP)

Field and list resource components will appear on the dialog box in the order that you specify them, meaning that you can control their vertical positioning. Buttons appear at the bottom of the dialog, and are positioned horizontally left to right in the order specified.

FORM COMPONENTS — SYNTAX

The following shows the syntax for the each FORM components. Although they appear together, you can use them independently.

```
EDIT,,,width,{&}label,{required},{validation},{default_value}
FEDIT,,,width,{&}label,{required},{validation},{default_value}, {file_path},{file_mask}
LEDIT,,,width,{&}label,{required},{validation},{default_value}, list_resource
CEDIT,,,width,{&}label,{required},{default_value}
EBUT, {&}label{~},ret_val,{cancel}
HBUT, {&}label,help_resource
```

PARAMETERS

The following table describes the parameters, purpose and/or values you can specify for the various FORM resource components. The second and third parameters in the EDIT, FEDIT and LEDIT components are no longer used, but their positions, marked by commas without values, must be retained. (You may also refer to etc/grc.doc for more current information.)

Parameter	Purpose and/or Allowed Values
Width	Defines the width of an edit field in characters. The maximum width will depend on your system display limitations. It is a good idea to make all edit fields in a GX the same width because this improves the appearance of the dialog.
Label	Identifies the label associated with this object. The label is displayed on the left of edit fields, and as the text in a button. Labels can be changed from a GX by calling the SetInto_DGW as in the following example: `SetInfo_DGW(hDGW,_MYFORM_0,DGW_LABEL,"My new label");` The & character placed before a character of the label in a button will make that character the "hot" button that lets your user activate the button by pressing <Ctrl-*character*> Placing a ~ (tilde) character after a button label will make this button the default action when the user presses <return> or <enter>
Required	R if a user entry is required; N for a non-editable entry (appears in a gray background); blank for optional user entry.
Validation	Indicates the type of values, files or user selections you want to allow for the component. One of the following: INT – (EDIT) integer value REAL- (EDIT) real value PASSWORD – allows entry of a password in which user keystrokes are displayed

	as a '*' character. N – (EDIT) indicates read-only. Text appears in a grayed window NEW – (FEDIT) file does not yet exist and the user will be prompted to confirm. OLD – (FEDIT) the file must already exist FORCE – (LEDIT) forces user to select one of the items in the list
default_value	Default value to place in the field if the GX has not specifically set a value. You can also use a default value defined in **geosoft.ini** by specifying *%group.parameter%*". For example, "`%MONTAJ.DEFAULT_COLOUR%`" will place the default colour table in a field by default. (This if for colour file selection only, and not for use in the CEDIT (colour selection) tool .) Note that the `SetInfo_DGW` and `SetInfoSYS_DGW` methods will change the value in a dialog field.
file_path	Indicates the default path from which to create a file list. Use a tilde (~) to indicate the Geosoft directory. Multiple items may be specified, separated by commas.
file_mask	Defines the file name mask (default is *.*). "/" to look for directories only. "**" to allow multiple file selection (see note below) "*.grd" displays all data and image DAT types that are available. "*.GRDDAT" limits the list to data grid DATs. "*.IMGDAT" limits the list to image DATs. Multiple masks can be specified by separating each mask by a semicolon. To select multiple files of a specific type, make the first mask "**" followed by a semicolon and the mask for the file type wanted. For example "**;*.grd" allows the user to select multiple grid files. Extensions can be changed from a GX by calling the SetInto_DGW as in the following example: `SetInfo_DGW(hDGW,_MYFORM_0,DGW_EXT,"My new extension");`
list_resource	The name of the LIST resource for an LEDIT component. A LIST resource that matches this name must be defined in the same resource file.
ret_val	The value that is returned when a button is pressed.
cancel	If the word "cancel" is specified, this button is treated as a cancel button and no validation of the dialog box is performed. Pressing a cancel button is equivalent to pressing the <Esc> key.
Help_resource	Defines the name of the HELP Resource to display when a help button is selected.

EXAMPLES

There are two types of FEDIT components: those that create lists of already existing files, and those that are to specify new files. The type is specified using the OLD or NEW validation string. If NEW is specified, and the file already exists, the validation ensures that the user is prompted to verify whether the file is to be overwritten:

```
FEDIT,,,40,"Create a New File" ,R,NEW,,,*.dat
FEDIT,,,40,"Symbol Font Name",R,OLD,symbols,~,*.gfn
FEDIT,,,30,"Grid name",R,OLD,,,*.grd
```

In the second example, files of type ".gfn" in the user's Geosoft directory will be displayed, and the symbols.gfn file will be selected by default, if no previous font file has been selected in this field.

In the third example, the presence of the "*.grd" alerts the resource compiler to the fact that grid files are requested, and the special grid file browser is invoked when the "..." browse button is selected beside this field. This grid browser automatically appends to the grid name the file decorations specific to the chosen grid type required to correctly interpret the grid.

The EDIT components may be used in four basic ways: to handle strings, integers and real values, and to display a text label in a shaded window as a read-only field:

```
EDIT ,,,30,"A Real value",,REAL
EDIT ,,,30,"An Int value",R,INT,10
EDIT ,,,30,"A String value",R
EDIT ,,,30,"Channel Name",N
```

In the first example, no input is required, but if it is, it must be a valid floating-point number. In the second example, an integer value is required, and 10 will be offered as default if no value is passed into the GX for this field. The third example requires the user to enter any string value. The fourth example is for printing out a channel name (for example); the user cannot change this value; the programmer determines it in the source code.

The CEDIT components are used to specify colours for coloured entities such as symbols, symbol fills, and lines. In the dialog, the edit window will be painted with the currently selected colour, and when the user clicks on the window the colour selection tool comes up so the user can select a different colour. The following are examples of the use of the CEDIT component:

```
CEDIT ,,,30,"Line colour","K"
CEDIT ,,,30,"Line colour #2","R128"
CEDIT ,,,30,"Line colour"
```

The last (optional) field is the default colour string. Colour strings take the form "R#G#B#, where # is a value from 0 to 255, giving the intensity of the Red, Green or Blue colour. Black can be expressed in a number of ways, including "K", "R0" or "R0G0B0". (Colours not included are assumed to be at zero intensity.) Letters given without numeric qualifiers are assumed to be at full intensity; for instance white can be written as "R255G255B255" or "RGB".The default colour (third example above) is black "K". In addition, you can use "N" (None) to indicate no colour, which is the same as transparent –for instance for fill colours in symbols – as well as "C" (Cyan), "Y" (Yellow), and "M" (Magenta).

For an example showing use of the LEDIT component, see the LIST Components – EXAMPLES section below.

The LIST Resource

A LIST resource is a drop-down box containing a group of selectable parameters, such as channel names. The contents of the list can be defined as part of the list resource in the Resource file, or they can be defined at run time using the `GetList_DGW` method to the get the LIST object (see example below).

If there is a list available for the selected field, you see a drop down list box containing items with a down arrow to the left of the box.

When you select an item, it replaces any existing text in the field.

SYNTAX

`RESOURCE, LIST,` *resource_name*

PARAMETERS

The following table describes the parameters, purpose and/or values you can specify for the LIST resource.

Parameter	Purpose and/or Allowed Values
resource_name	Identifies an individual resource. You must use a unique name for each resource in a GX.

List Components — Description

Individual items in the dropdown LIST resource are specified using the ITEM statement. The items appear immediately following the LIST resource, one item per line.

SYNTAX

`ITEM,`*item_name*`{,`*alias*`}`

PARAMETERS

The following table describes the parameters, purpose and/or values you can specify for the ITEM resource component.

Parameter	Purpose and/or Allowed Values
item_name	Identifies the list item displayed to the user.
alias	Indicates the corresponding alias value actually passed back and forth from the dialog. If no alias is defined, the item_name is used.

EXAMPLES

The following is an example of an LEDIT component, with a list whose items are defined inside the resource file:

```
LEDIT,,,20,"Plot sizes?",R,FORCE,"Yes",YesNo
LEDIT,,,20,"Sizes",R,FORCE,"Large",SIZE
```

```
RESOURCE,LIST,YesNo
ITEM, Yes,1
ITEM, No,0

RESOURCE,LIST,SIZE
ITEM, Small
ITEM, Medium
ITEM, Large
```

The “R” parameter ensures that the user must enter a result in the field, and the “FORCE” parameter ensures that the entered text is one of the menu items.

The following is a commonly found example of an LEDIT component where the resource items are not defined in the resource file:

```
LEDIT,,,20,"Channel",,FORCE,,CHAN

RESOURCE,LIST,CHAN
```

This creates an empty list resource that will be populated by the GX at run-time. In this example, the CHAN resource will be populated by a list of channels in the current database using GX code similar to the following:

```
// --- Set up lists ---

List = GetList_DGW(hDGW,_MYFORM_3);     // retrieve list from dialog
SymbLST_DB(Data,List,DB_SYMB_CHAN);     // fill the list from the database
Sort_LST(List,0,0);                     // sort the list alphabetically
```

In the above example, the “_MYFORM_3” parameter indicates that the LEDIT component referenced is the fourth component (because indexing begins at 0) in the FORM resource whose *resource_name* is “MYFORM”.

The HELP Resource

Help resources are used to provide the user with context sensitive help about how to use your GX, or specifically how to use the dialog that is displayed. The HBUT button on a FORM (Dialog) resource requires a HELP resource. The HELP resource is displayed when the user presses the [Help] button.

Help information itself can be provided as a simple text file, an RTF (Rich Text Format) file, or you can use a *WinHelp* or *Compile HTML* help system.

SYNTAX

```
RESOURCE, HELP, resource_name, help_file_name
```

PARAMETERS

Parameter	Purpose and/or Allowed Values
Resource_name	The help resource name that is used in the HBUT control on the FORM resource.
Help_file_name	Identifies the ASCII or RTF (Rich Text Format) file to be used as a help resource. This text is merged into the GX. You can also create a WinHelp or Compile HTML help file for your GXs that will use the Windows Help system to display your help. See the next section for more information on using WinHelp or Compile HTML Help.

EXAMPLES

The following are examples showing the use of an ASCII text help file, and an RTF file:

```
RESOURCE,FORM,Dialog,"My dialog has 2 help buttons",-1
HBUT,"Help me",help1
HBUT,"More help",help2

RESOURCE,HELP,help1,simple_help.txt
RESOURCE,HELP,help2,more_help.rtf
```

Using WinHelp or Compile HTML (*.CHM) Help Files

You can also use a WinHelp (*.HLP) or Compile HTML (*.CHM) help system files to provide help information to any GX you create. The following section outlines the steps you need to follow to do link a WinHelp/Compile HTML help system to a GX.

Linking WinHelp or Compile HTML help to Oasis montaj GXs

This tutorial uses a number of sample files that are installed with your **GX Developer** installation. The following files will be copied to a gx\examples**winhelp** subdirectory located within your **GX Developer** directory:

myhelp.ini	This INI file is used to map the GX *Topic ID* to the help file.
myhelpgx.gx	This is a sample GX with a dialog box containing two help buttons.
myhelpgx.grc	This is the grc file for the sample GX *myhelpgx.gx*
myhelpgx.gxc	This is the gxc file for the sample GX *myhelpgx.gx*
myhelpfile.hlp	This is a sample WinHelp file with only two help topics, one for each help button on the GX dialog.
brokenlink.hlp	This help file is displayed if the link is broken.

Step 1 – The GRC File

The .grc file contains the help resource information for the GX. In this example, the GX has a dialog box that contains two help buttons.

```
//
// MyHelpGx.GRC
//-----------------------------------------------------------------------
----------
RESOURCE,FORM,MyHelpForm,"GX to test Winhelp Topics",-1
EDIT,,,16,"Bogus Edit Box",,
EBUT,&OK,0
EBUT,&Cancel,1,CANCEL
HBUT,&HelpTopic1,MyHelpTopicLink1
HBUT,&HelpTopic2,MyHelpTopicLink2

RESOURCE,LIST,CHAN

RESOURCE,HELP,MyHelpTopicLink1,BrokenLink.hlp
RESOURCE,HELP,MyHelpTopicLink2,BrokenLink.hlp
```

Parts of the GRC File

The help buttons are defined in the HBUT lines:

```
HBUT,&HelpTopic1,MyHelpTopicLink1
HBUT,&HelpTopic2, MyHelpTopicLink2
```

The `&HelpTopic2` text is the title text on the help button.

The `MyHelpTopicLink2` text is a **link ID** (or help resource). This ID is used in the myhelp.ini file to link to the real help **Topic ID** in the myhelp.hlp file (this is explained more in the next step).

Broken Links

The RESOURCE lines specify a file that is displayed if a user clicks on the help button and the link to your Winhelp topic is broken for any reason.

In this example, a broken link would display the brokenlink.hlp file. You can also specify a text file (brokenlink.txt) or an RTF file (brokenlink.rtf).

You should compose a message that tells your users what to do should when they find a broken link (contact you, for example). You can place this file in the gx\include directory so it can be shared by all your GXs. Geosoft uses a file named “nogx.hlp” exactly for this purpose (please do not use this file since it asks your users to contact Geosoft).

Step 2 – The myhelp.ini file (Mapping File)

The next step is to create a mapping file that **Oasis montaj** will use to link the **link ID** to a **Topic ID** in a specific WinHelp (*.hlp) or Compile HTML Help (*.chm) file.

In this example, we created a mapping file called myhelp.ini that contains the following text:

```
[GXHELP]
MyHelpTopicLink1="MyHelpFile.hlp:MyHelpTopic1"
MyHelpTopicLink2="MyHelpFile.chm:MyHelpTopic2.htm"
```

WHAT EACH LINE MEANS:

The first line in the file **[GXHELP]** identifies that this is a mapping file used to provide GX help.

The following two lines consist of three parts, illustrating WinHelp (*.hlp) and Compile HTML Help (*.chm) files.

WinHelp Example

`MyHelpTopicLink1="MyHelpFile.hlp:MyHelpTopic1"`

The (blue) text `MyHelpTopicLink1` is the **Link ID** for the first button as indicated in the myhelp.grc file.

The (red) text `MyHelpFile.hlp` or indicates the WinHelp file to look in for this information.

The (green) text `MyHelpTopic1` is the **Topic ID** associated with the topic text in the WinHelp file.

Compile Help (.chm) Example*

`MyHelpTopicLink2="MyHelpFile.chm:MyHelpTopic2.htm"`

The (blue) text `MyHelpTopicLink2` is the **Link ID** for the first button as indicated in the myhelp.grc file.

The (red) text `MyHelpFile.chm` or indicates the Compiled Help file to look in for this information.

The (green) text `MyHelpTopic2.htm` is the HTM file associated with the topic text in the Compile Help file. To determine the individual HTML files contained within a Compile Help file, use the following command:

```
hh.exe -decompile extracted GridAnalysisHelp.chm
```

WHAT ARE TOPIC IDS?

Each topic in a WinHelp file has a unique **Topic ID**. Topic ID names should not have any spaces in them. A Topic ID is *not* the same as a Topic name. A Topic Name is only the title text of the topic.

Tip: At Geosoft, to make things easier, we usually use a topic ID that is derived from or similar to the name of the GX. For example, the ARROW GX topic in the help system would have a topic ID called ARROW_GX.

Step 3 – Create a Custom Winhelp File

If you haven't already done so, you will now need to create your custom WinHelp file (Myhelp.hlp) using a help authoring software program (at Geosoft we use Robohelp).

Add custom Help INI to Oasis montaj

Before you can use your custom GX help file, you will have to tell **Oasis montaj** to use the custom .ini file you created. This will let **Oasis montaj** know which mapping files to check for links.

Simply, your custom .ini file needs to be placed into the spec\ini subdirectory in an **Oasis montaj** installation. This spec directory resides on the same level as bin, gx, and hlp.

Any number of files can be placed into this directory. With this modification, **Oasis montaj** will now search the spec\ini directory and load each file when it looks for a help topic to link to.

Step 4 – Copy the Files and Restart Oasis montaj

If you are using **Oasis montaj**, you should copy any help files you create to the hlp sub-directory, any INI files to the spec\ini sub-directory, and GXs to the gx sub-directory where your **Oasis montaj** is installed.

Whenever you change an INI or HLP file that **Oasis montaj** uses, you need to restart the software to have the changes take effect.

Final Note: The instructions in this note only apply to linking help from a GX. If you want to link help file topics to custom GUI (graphic user interface) objects or to dialogs created using visual basic, you will need to create a .hh mapping file and link them using a different method.

Combining Resources and Components

The following example shows how you would combine various resources and resource components to create a dialog box. To see how this dialog box appears on the screen, try loading and running the XLEVEL GX file in **Oasis montaj**:

```
RESOURCE,FORM,XLEVELForm,"Make a level channel",-1
FEDIT,,,16,"Intersection table",R,OLD,,,*.TBL
EDIT ,,,16,"Maximum gradient (Z/fid)",,REAL,0.0
LEDIT,,,16,"Process line types",r,FORCE,Tie,TYPE
LEDIT,,,16,"Unlevelled data channel",R,FORCE,,CHAN
LEDIT,,,16,"Output cross-level channel",,,,CHAN
LEDIT,,,16,"Output difference channel",,,,CHAN
LEDIT,,,16,"Output cross-gradient channel",,,,CHAN
EBUT,&OK,0
EBUT,&Cancel,1,CANCEL
HBUT,&Help,help

RESOURCE,LIST,CHAN

RESOURCE,LIST,TYPE
ITEM,Tie
ITEM,Line
ITEM,Selected

RESOURCE,HELP,help,nogx.hlp
```

Working with Menus

An **Oasis montaj** menu file (.omn or .smn file) defines a menu that can be added the **Oasis montaj** menu bar. Menus organise suites of GXs and internal commands into logical processing groups to be presented to your user. You will normally create a menu specifically for your own application suite, and the first thing your user will do is load your menu by clicking the *Load menu* option on the *GX* menu. The contents of your menu will be inserted into the displayed menu immediately before the "Window" menu.

Note that only menus with extension .omn will be displayed in the menu browse tool. Menu files with extension .smn are for menus that will only be included in other menus. A menu loads submenus using the LOADMENU directive followed by the submenu name.

For example, following is the menu file for the gravity reduction system (gravity.omn):

```
/
/ Gravity reduction system
/-------------------------------------------------------------------------------

MENU "&Gravity"
ITEM "Pro&ject information..."       ,grproj.gx
ITEM "&Processing parameters..."     ,grparm.gx
SEPARATOR
SUBMENU "&Base stations"
SUBMENU "&Calibration"
SUBMENU "&Locations"
SEPARATOR
SUBMENU "&Import"
ITEM "&Drift correction..."          ,grdrift.gx
ITEM "&Merge with master database...",grappend.gx
SEPARATOR
ITEM "Process &repeats..."           ,grrepeat.gx
SUBMENU "&Terrain corrections"
ITEM "&Free air, Bouguer anomaly..." ,grboug.gx
SEPARATOR
ITEM "Edit file..."                  ,edit.gx
ITEM "Sort all by &1 channel..."     ,sortall.gx
ITEM "Sort all by &2 channels..."    ,sortall2.gx
ITEM "E&xpression..."                ,math.gx
ITEM "Expressi&on file..."           ,mathfile.gx


MENU "&GravRed/&Import"
ITEM "&Import gravity survey..."     ,grload.gx
SEPARATOR
ITEM "&Edit data file..."            ,edit.gx
ITEM "&Download from CG3..."         ,grdnlcg3.gx
...
```

Each line of a menu file is either a comment, a blank line, or a keyword followed by parameters. The following table describes each of the supported keywords:

Keyword	Parameters	Description
CLEARALL	None	The CLEARALL keyword will clear all menus; excluding the coremenus.omn This enables your menu to completely control the **Oasis montaj** environment for your specific application. The CLEARALL keyword, if used, should be the first keyword in your menu file. If you use this keyword, you should provide your user with access the *ID_CLEARMENU internal command somewhere in your menu system. This command will allow your user to reset the menus to the Geosoft coremenus.omn.
MENU	Name	This identifies a new menu, which can appear on the **Oasis montaj** menu bar, or may be a submenu that will become part of a previously defined menu. Each MENU line is followed by one or more ITEM lines, SUBMENU lines or SEPARATOR lines. You can also specify the name of a pop-up menu that is activated when the right-mouse button is pressed. Pop-up menus are context sensitive, which means that the menu that appears depends on the current mode and the object that is under the mouse cursor at the time the right-mouse button is pressed.
SUBMENU	Name	This identifies a menu entry that will be a sub-menu in the menu list. There must be a MENU associated with the submenu later in the OMN file, and the name in the MENU must be “menu_name/sub_menu_name”. See the above example. Submenus may also be nested.
SEPARATOR	None	Places a separator to space items in the menu list when displayed.
LOADMENU	File	Load another menu file. If you need to share certain menu components among applications, it is easier to store the menus in a separate menu file that can be included in other menu files. If a menu file is never intended to be used by itself, you should give it extension “.smn” to prevent the file from appearing in the load menu browse tool. All menu files are located in the <geosoft>\omn directory by default. Menus that are not located in this directory must be specified explicitly (see **File Names** below).
LOCATE	TOP BOTTOM AFTER "*Name*" BEFORE "*Name*"	Controls the location of the menu items that follow. The AFTER and BEFORE require the name of a menu item that already exists in the menu.
ITEM	Name, Action	An ITEM in a menu identifies an action that will happen

	<*image.bmp[i]*> {?*context*}	when your user selects the item. *Name* is the text that appears in the menu. *Action* can be the name of a GX, a GS script, a Sushi PDF file, a Win-32 help file, or the name of an internal command. GX, GS, PDF and help files must be in the project directory or in the Geosoft directory, or the locations must be specified explicitly (see **File Names** below). Internal commands must be one of the commands listed in the Internal Command table below. *Action* can also be a SHELL command that will execute an operating system command. See the "SHELL item action below for more information. <*image.bmp[i]*> indicates this item will show an image in the menu to the left of the item. The [i] indicates the position of the specified icon or picture within the larger bmp file. {?*context*} is a context test that if not true will shade the item *Name* in the menu. The following contexts are supported: `{?map}` a current map is open `{?gdb}` a current database is open `{?map_or_gdb}` a current map or database is open `{?map_and_gdb}` a current map and database are open
LOADTOOLBAR	*"Name.geobar"*	Loads a toolbar into the project. See "Working with Toolbars" below.

SHELL item action

The item action SHELL will cause the "ShellExecute" function to be called with the parameters of the SHELL command passed as the arguments to the ShellExecute function. Syntax for the SHELL command is:

SHELL(*Operation;"*File*";"*Parameters*";"*Directory*";"*ShowCmd")

Only *Operation* and "File" are required.

The SHELL command is typically used to execute an external probram such as an operating system program as in the following example:

ITEM "Stop IIS Server" , SHELL(open;"IISRESET.EXE";"/stop")

Operation can also be "winhelp", in which case "File" will be a JumpID command as in the following example:

ITEM "FAQ" , SHELL(winhelp,"JumpID(\"%geopath%\\interface.hlp\",
\"Frequently_Asked_Questions\")".

Refer to the ShellExecute Function description in MSDN for more information about the ShellExecute command.

Working with Toolbars

An **Oasis montaj** toolbar file (.geobar file) defines a single toolbar that can be added to any **Oasis montaj** project. Toolbars organise suites of GXs and internal commands into logical processing groups to be presented to your user. You will normally create a toolbar specifically for your own application suite, and the first thing your user will do is load your toolbar by right-clicking the *Toolbars* section of the project explorer and selecting *Add toolbar(s)…*. The user can then browse to the bar directory and load your toolbar. The bar will be loaded and displayed in default fashion and added to the list in the project explorer.

Note that each toolbar needs to have its own .geobar file.

For example, following is the toolbar file for a small toolbar installed with **Oasis montaj** ***Viewer***:

```
TOOLBAR "Standard Bar"
ITEM  "Open database...", *ID_FILE_OPEN_DATA <standard.bmp[2]>
ITEM  "&Open map...",     *ID_FILE_OPEN_MAP  <standard.bmp[5]>
SEPARATOR
ITEM  "Run &GX...",       gxrun.gx           <standard.bmp[13]>
ITEM  "&Load menu...",    *ID_VIEW_LOADMENU  <standard.bmp[14]>
```

Each line of a toolbar file is either a comment, a blank line, or a keyword followed by parameters. The following table describes each of the supported keywords for toolbar files:

Keyword	Parameters	Description
TOOLBAR	Name	This identifies a new toolbar, which can appear on the **Oasis montaj** project explorer. Each TOOLBAR line is followed by one or more ITEM lines or SEPARATOR lines. Only one toolbar should be specified per .geobar file.
ITEM	Name, Action <*image.bmp[i]*> {?*context*}	An ITEM in a toolbar identifies an action that will happen when your user selects this button on the bar. *Name* is the text that appears in the tooltip when the user hovers over the button for a couple of seconds. *Action* can be the name of a GX, a GS script, a Sushi PDF file, a Win-32 help file, or the name of an internal command. GX, GS, PDF and help files must be in the project directory or in the Geosoft directory, or the locations must be specified explicitly (see **File Names** below). Internal commands must be one of the commands listed in the Internal Command table below. *Action* can also be a SHELL command that will execute an operating system command. See the SHELL command description for more information. <*image.bmp[i]*> indicates this button will show an image. The [i] indicates the position of the specified icon or picture within the larger bmp file.

		{?*context*} is a context test that if not true will shade the item *Name* in the menu. The following contexts are supported: `{?map}` a current map is open `{?gdb}` a current database is open `{?map_or_gdb}` a current map or database is open `{?map_and_gdb}` a current map and database are open
SEPARATOR	None	Places a separator to space items in the toolbar when displayed.

File Names

Files to be located by name, either in OMN files, GEOBAR files, or in scripts, will be located either in their default location (<geosoft>\omn for menu files, <geosoft>\gx for GX files), or in the location explicitly identified by a full path name. For explicitly located files, you may use a prefix <label> to identify a directory in which to locate a file. Allowable <label> values are:

<geosoft>	the main Geosoft installation directory.
<geosoft2>	the secondary Geosoft installation directory (typically user).
<geotemp>	the Geosoft temporary file directory.
<windows>	the operating system Windows directory.
<system>	the operating system SYSTEM directory.
<*other*>	other environment variables. The "Geosoft/Oasis montaj/Environment" registry is searched first. If the name indicated by "*other*" is not found, the user's environment is searched.

For example, if you install your custom system, including GXs, in the user\my_system directory, you can identify the GX in your OMN file as follows:

```
ITEM,"Run one of my GX's",<geosoft>\user\my_system\my_gx.gx
```

Internal commands

The list of internal commads can be found in the GX Developer.chm help file. They can be activated from an ITEM in a menu or toolbar. Note that internal commands are not stored in a script or in the processing log file. Only GX commands and GX parameters appear in a script of a log file. If scripting is required, most commands can be emulated by writing a GX that performs the required action.

Pop-Up Menus

A pop-up menu is a context sensitive menu that appears when the right mouse button is pressed. The menu that is displayed (popped-up) depends on what has been selected, where the mouse is located and/or the current mode of **Oasis montaj**. The

default oasis pop-up menus are defined in the file **montaj_popups.smn** in the Geosoft directory. The following table lists some of the available pop-up menus:

Popup Name	Context
@PopupSelLine	Line/group
@PopupSelChanNone	Empty channel
@PopupSelChan	Loaded channel
@PopupSelData	One or more date items selected in a channel/line
@PopupSelBDat	Data from all lines on this channel
@PopupSelProf	Cursor in a profile window
@PopupSelFid	Fid is selected
@PopupMap	Map with nothing selected
@PopupSelGroup	Group selected and in context
@PopupSelView	View selected and in context
@PopupSelItem	Group edited and item selected
@PopupSelGrid	AGG group selected
@PopupGroupEdit	Group opened for edit, nothing selected
@PopupMapGroup	Map in group select mode, no group selected
@PopupMapView	Map in view select mode, no view selected
@PopupSelCSymb	Colour symbol group selected
@PopupBoxMark	Group edited, multiple items selected
@PopupSelPAGG	Poly-aggregate (animation) selected
…	…

The SHELL command

An ITEM can use the SHELL action to identify an operating system command to be executed. The SHELL action syntax is as follows:

ITEM "*Text*" **,SHELL(***verb;file*[**;***parameters;directory;ShowCmd*]**)**
ITEM "*Text*" **,SHELL(WinHelp;***data*[**;***nCmd*]**)**

The Windows "ShellExecute" command is called with the specified parameters. For the **WinHelp** verb, the Windows WinHelp command is used. Please refer to your Windows API documentation for the use of these parameters. Note that the items in square brackets are optional and are for special use when full control of the Windows functions are required.. *ShowCmd* and *nCmd* are both integer values that can be set to the Windows manifest constant values as required. By default, *ShowCmd* will be **1** (SW_SHOWNORMAL), and *nCmd* will be **258** (HELP_COMMAND).

The following are common examples show how to use the SHELL command:

```
ITEM "&Geosoft Website",SHELL(open;http://www.geosoft.com)
```

```
ITEM "&Geosoft Website",SHELL(open;http://www.geosoft.com)

ITEM "&Send a message",SHELL(open;"mailto:geonet@lists.geosoft.com")

ITEM "&Subscribe",SHELL(open;
   "mailto:majordomo@lists.geosoft.com?subject=Subscribe%20to%20GEON
   ET%20user%20help%20list.?body=SUBSCRIBE%20gxnet")

ITEM "Ho&w to get help...",SHELL(WinHelp;
   "JumpID(\"oe32.hlp\",\"ID_HELP_TECH_SUPPORT_TOPIC\")")
```

Displaying images for items in menus

Menu items can display images (icons really) to stress what an item does or represents. Icons must be 18 x 18 in size (pixels) and can be stored individually or contained in a larger bitmap file and indexed by the image number in square brackets []. For example, if my bitmap file contains six 18 x 18 images and the item requires the 4th image in the file you would use <image.bmp[3]> to reference that position (index count begins at 0). Image files are .bmp files and are usually stored in the img subdirectory of **Oasis montaj**.

Examples:

```
ITEM "Run &GX...", gxrun.gx <standard.bmp[13]>

ITEM "&Load menu...", *ID_VIEW_LOADMENU <standard.bmp[14]>
```

Controlling item activation

Some items should only be active if the project has a database or a map open. You can specify the item dependency by placing one of the following qualifiers at the end of an ITEM line. The menu item will be greyed out if the dependancy is not true:

```
{map}
{gdb}
{map_or_gdb}
{map_and_gdb}
{packed_map}
```

Examples:

```
ITEM "New database line", newline.gx {gdb}

ITEM "Display on the map", display.gx {map}

ITEM "Draw database line on map", drawline.gx {map_and_gdb}
```

Note that dependencies {map} can be in any order at the end of a line.

How menus are loaded

The default **Oasis montaj** menu will contain the minimum menu you see when you do not have a project open. This includes a File menu, a Window menu and a Help menu. When you create a new project, **Oasis montaj** first loads the coremenus.omn file, then loads the menu(s) specified by the Default Menu entry in the settings.gx. When an existing project is opened, the same actions are executed and then any menus that were open when the project was last closed are also displayed.

If the user clears menus, all menus are removed except the coremenus.omn file.

If you want your application to completely control your users menu, you can change the Default Menus entry. You should look at the contents of montaj.smn and montaj_popups to see if there are any components of these menus that you want to include in your application.

Geosoft Environment Settings (geosettings.meta)

The **geosettings.meta** file in the Geosoft ini directory is used to maintain user preferences for a variety of default settings when your user is using the system. You can use this file to store your own GX default settings, and you can control the access you want to provide your users. The ASCII file **ini.doc** in the Geosoft etc directory documents all parameters that may appear in this file.

The following SYS functions can be used to access and change information in the **geosoft.ini** file:

`iGlobal_SYS`	Get a global parameter setting.
`GlobalSet_SYS`	Set a global parameter setting.
`GlobalReset_SYS`	Reset the global parameters to original values.
`WriteGlobal_SYS`	Modify the global parameters to current settings. (Rewrite the **geosettings.meta** file.)

Refer to sys.gxh for information on how to use these functions. You can also look at the SETTINGS GX source code to see how Geosoft modifies the global settings.

Part 2 - Working with GX Developer

This part of the documentation discusses object-oriented programming in general and provides a users guide for using some of the key components of the **Oasis montaj** system. ***Part 1 - GX Developer Basics*** provides reference information about how to use the compilers and specific language rules and syntax.

Building a GX Application Suite

GX Developer has been designed to enable you to build a complete application that can be run from **Oasis montaj**. With **GX Developer** you have access to almost all components of the **Oasis montaj** environment and you can customise most of what your end-user will see when using your application.

A GX application will normally consist of a menu and a set of GXs that perform specific functions. The menu serves to expose the structure and capabilities of your application and lets users run your GXs from the **Oasis montaj** menu. You may choose to completely replace the menus provided as part of **Oasis montaj**, or you may create an application that is simply added to the basic menus. See ***Working with Menus*** for more information on how to create menus.

You can also control certain **Oasis montaj** features and settings using settings in the **geosoft.ini** file located in the GEOSOFT directory. See ***How menus are loaded*** for more information about the Geosoft environment.

Object-Oriented Programming

The OASIS data processing system consists of a graphical user interface, a database system for very large data sets, an integrated mapping and visualization system, and a processing systems that processes data and creates maps. There are also many layers of functionality that enable you to interact with the databases, maps, the processing engine and the outside world. In order to simplify this world, **Oasis montaj** was designed and written using object-oriented programming techniques.

The object structure of the system is also apparent to the **GX Developer**. For application developers experiencing object-oriented programming for the first time this can be confusing since there are significant differences from purely procedure-based programming. This section is intended to acquaint you with the object-oriented structure of the OASIS database environment and introduce you to some of the key concepts that will influence your GX programming.

A fundamental component of an object-oriented system is a ***Class***. A Class encapsulates the information, features and functions (also called methods) that are used to manipulate the contents of the Class. A good analogy to a Class is a House. When you think of a House, you think of a building that houses a single family, contains a kitchen, bedrooms, family members, front door, etc., and when you knock on the door (execute a method), someone answers. This is a Class called House. Note that a Class does not exist; it is just the set of characteristics that make up what you think of as a House and the methods that you can use to interact with a House.

A class ***instance*** does exist – it is a specific house, say 2128 Langtry Drive in Oakville, Ontario, Canada. There can be many instances of the class House, each with its own unique contents and characteristics. According to our definition, all instances of a House will have a front door, and when you knock on it someone will answer. For a class instance to exist, it must be created, at which time any resources required to create the class are consumed (bricks, wood, etc for a House; memory, disk space, etc for a GX class).

In **GX Developer**, a variable must be declared to hold an instance of a class. The variable type will be the class name, and the variable name will be used to reference that instance of the class. The variable is often called a class handle. Note that declaring a class variable does not create the class – it just creates a name that can be used for an instance of that class. All classes will have a "Create_ " method to create an instance of a class, and a "Destroy_" method to destroy an instance of a class and return its resources to the system. Other class methods will only work on classes that have been created and not yet destroyed.

A GX programming example is a Class named VV. This class is used to hold a single array of any type of data. One of its key characteristics is that it can hold and efficiently manipulate extremely large data arrays of data. It also comes with a rich set of methods that can be used to manipulate the data in the array (such as filter it, enlarge it, get and set values in it, etc). The VV class and the methods that manipulate the class are described in the prototype file vv.gxh in the GxDev/gxh directory. There are also many other classes that can accept a VV instance as an argument to one of their class methods. For example, the DB class (the database class) has a method (PutChanVV_DB) that stores a VV instance in a channel of a database.

In summary, a Class has a name and describes a set of information (data) and what you can do with it. A Class instance is a specific unique instance of a class, and methods are the things you can do to manipulate an instance of a class.

Differences between Procedural and Object-Oriented Programming

In an object-oriented world, you will notice many differences between procedural languages, such as FORTRAN, and object-oriented languages, such as the GX Programming Language. While a full discussion of these differences is beyond the scope of this manual, there are a few basic concepts that you should remember:

- Data hiding
- Instantiation
- Objects versus OASIS symbols

Data hiding is a fundamental characteristic of objects. This means that you cannot directly access or view the data within an object except through the methods (functions) of the class object. For example, the VV class deals with 1-dimensional data arrays, and it can efficiently store and manipulate very large array data sets. The VV class hides how it does this from you, and it hides how it stores the data in the computer memory because this is really of no concern to you. To access data in a VV array, you will use VV methods such as iLength_VV, which tells you the length of

the VV, and rGetReal_VV or SetReal_VV to get and set a real value in a VV array efficiently. Such as with virtual vectors (which replace arrays in the GX environment). To see all of the data in the vector, you require two methods — one for determining the length of the vector (`iLength_VV`) and one for extracting the data (`rGetReal_VV`). You also require a loop to extract each data point until the end of the vector (i.e. length) is reached.

Instantiation is the process of creating instances and returning handles to objects. To understand this concept, consider the dialog box – an interface component that lets you accept or display information to users. In GX programming, we provide a method for creating a dialog box (`Create_DGW`). When you use this method, the system assigns memory to a database object (i.e. creates an instance of the object) and returns a handle to the dialog box.

You can continue creating instances of other dialog boxes until you run out of memory. The key point to remember is that each instance has its own handle — the connection that enables you to access the dialog box and perform various actions with it. Handles are discussed in the next section.

Another key element of the object-oriented programming component in the OASIS environment is the treatment of objects and symbols. As we learned above, objects are unique components of the OASIS system and interface that we can manipulate. Symbols, on the other hand, are the named parts of the database consisting of lines and channels. Each line and channel has its own identifier.

When working with symbols, keep in mind that they are defined in the database. When you use a symbol (line, channel), the system returns a keyword (e.g. a line or channel id). You don't work with the symbol directly. Symbols are permanent

In contrast, objects can be any part of database or system (database itself or a part of the Graphical User Interface). When you use an object you must associate a handle with it. When you destroy an object, the instance disappears. In other words, objects are not permanent – they are tools that exist only as long as you require them.

Working with Library Functions

When you are starting to program GXs, one key point our application developers stress is to know the libraries (classes) and functions (methods) in each library. These provide you with all of the functionality you require to use the OASIS interface, get data and perform actions (i.e. processing) on data.

Although it is a good idea to be familiar with all libraries, there are certain ones you will use more than the others. These are shown in the following table:

Tasks	GXH Library
Database (lines, channels and database)	DB, DU
File handling	RA, WA, BF
Strings	STR
System	SYS

Maps, Views	MAP, MVIEW
Vectors (arrays)	VV

GX Structure and Program Flow

It is rare that a new GX is created completely from scratch. Instead, a new GX is created by modifying an existing GX, even if only for the general outline.

The COPY GX

The following is a walk-through of a typical, well-designed GX function, the COPY GX, which is used to copy one channel to a new one — if the new channel does not exist, it will be created, and the channel data can be decimated during the copy.

The walk-through is intended to show the general outline of a GX, and establish some general principals for writing a GX; a fuller description of individual processes found within the GX may be found later in this section.

Step 1: Set the header information: `NAME`, `VERSION` and `DESCRIPTION`.

- In addition to being useful for source control, these parameters are recognized by the VIEWGX program.
- Note that multi-line text is allowed (see the `DESCRIPTION` statement.)
- Parameters residing in the system parameter block, used and/or modified by this GX, are explained.
- As in normal C, "//" indicates the start of a comment, which is ignored during compilation.

```
//=====================================================================
NAME          = "Copy one channel to another"
VERSION       = "v1.01.00  Copyright Geosoft Inc. 1999"
DESCRIPTION   = "

Copies one channel to another.  If the new channel does not exist,
it
will be created with the same definition as the original channel.
The channel data can be decimated during the copy.

Parameters:

COPY.FROM     - Original channel
    .TO       - Destination channel
    .DECIMATE - Decimation factor, default 1
    .FIDSTART - New fiducial start, default is current start.
    .FIDINCR  - New fiducial increment, default is current
increment.
"
```

Step 2: Include the GX resources.

- The compiled resource is contained in a Geosoft Resource "GR" file, while resource identifiers are contained in the Geosoft Resource Header "GRH" file.

```
//================================================================
//                    RESOURCES
//================================================================

RESOURCE = "copy.gr"
#include "copy.grh"
```

Step 3: Included GX Function Header files. These contain the necessary library function prototypes.

- GX library functions may not be used unless their prototypes are included.
- The "catch-all" header ALL.GXH includes all the regular, non-mapping Geosoft function prototypes; ALL32.GXH is now functionally equivalent to ALL.GXH (it is now just an #include "all.gxh" statement) and is no longer required for mapping prototypes. Constants using the #define pragma may also be defined here.

```
//================================================================
//                    INCLUDE
//================================================================

#include <all.gxh>       // system
```

Step 4: Declare variables.

- These include all object handles, as well as the `int`, `real` and `string` types.
- Though not required, it is common practice to declare variables in the following order: handles, integers, reals, strings.

```
//================================================================
//                    VARIABLES
//================================================================

EDB          EData;          // Database handle
DB           Data;           // Database handle
DB_SYMB      InCh;           // Channel Handle
DB_SYMB      OutCh;          // Channel Handle
DB_SYMB      Line;           // Line Handle
DGW          Diag;           // Dialogue handle
LST          List;           // List handle

int          i;              // Utility
int          iN;             // Decimation factor
int          iLines;         // Number of Lines Processed
int          iTotLines;      // Total Number of Lines to Process
real         rFidStart;      // Fid start
real         rFidIncr;       // Fid increment
real         rNewStart;      // New fid start
```

```
real         rNewIncr;          // New fid increment
string(50)   sInCh;             // Channel Names
string(50)   sOutCh;            // Channel Names
string(32)   sTemp;             // Temp string
string(60)   sLabel;            // Label for progress bar
```

Step 5: Enclose the main code block.

- This includes the remainder of the source file.
- The code is enclosed in a pair of curly brackets.

```
//====================================================================
//                      CODE
//====================================================================

{
```

Step 6: Do the initial setup.

- This usually involves getting the current database or map, and setting up information required later for user dialogs.
- Some GXs may also recover parameters from a control or initialization file here, and set various default values.

```
// --- Get database ---

  EData = Current_EDB();
  Data = Lock_EDB(EData);
```

Step 7: Perform interactive processes.

- Interactive processes should be enclosed by the `if(iInteractive_SYS()){}` control block.
- These processes generally involve creating, loading, displaying, then interrogating a dialog object for information modifiable by a user.
- Variables are usually obtained from, and loaded back into, the workspace parameter block, using the `SetInfoSYS_DGW` and `GetInfoSYS_DGW` functions.

```
// --- Are we running interactively ? ---

  if (iInteractive_SYS())
  {

    // --- Create the Dialogue ---

    Diag = Create_DGW("COPYForm");

    // --- Set up lists ---
```

```
  List = GetList_DGW(Diag,_COPYFORM_0);
  SymbLST_DB(Data,List,DB_SYMB_CHAN);
  Sort_LST(List,0,0);
  List = GetList_DGW(Diag,_COPYFORM_1);
  SymbLST_DB(Data,List,DB_SYMB_CHAN);
  Sort_LST(List,0,0);

  // --- Set any Defaults ---

  SetInfoSYS_DGW(Diag,_COPYFORM_0,DGW_TEXT,"COPY","FROM");
  SetInfoSYS_DGW(Diag,_COPYFORM_1,DGW_TEXT,"COPY","TO");
  SetInfoSYS_DGW(Diag,_COPYFORM_2,DGW_TEXT,"COPY","DECIMATE");
  SetInfoSYS_DGW(Diag,_COPYFORM_3,DGW_TEXT,"COPY","FIDSTART");
  SetInfoSYS_DGW(Diag,_COPYFORM_4,DGW_TEXT,"COPY","FIDINCR");

  // --- Run the Dialogue ---

  i = iRunDialogue_DGW(Diag);
  if (i != 0) Cancel_SYS();     // The user hit cancel


  // --- Get the Strings ---

  GetInfoSYS_DGW(Diag,_COPYFORM_0,DGW_TEXT,"COPY","FROM");
  GetInfoSYS_DGW(Diag,_COPYFORM_1,DGW_TEXT,"COPY","TO");
  GetInfoSYS_DGW(Diag,_COPYFORM_2,DGW_TEXT,"COPY","DECIMATE");
  GetInfoSYS_DGW(Diag,_COPYFORM_3,DGW_TEXT,"COPY","FIDSTART");
  GetInfoSYS_DGW(Diag,_COPYFORM_4,DGW_TEXT,"COPY","FIDINCR");

  // --- Destroy the Dialogue ---

  Destroy_DGW(Diag);

}
```

Step 8: Retrieve information from the workspace parameter block and verify it.

- Verification means ensuring that the data exists, and that it falls within acceptable limits for the process that must be performed.

```
// --- Get Parameters ---

GetString_SYS("COPY","FROM",sInCh);
GetString_SYS("COPY","TO",sOutCh);
iN = GetInt_SYS("COPY","DECIMATE");
rNewStart = GetReal_SYS("COPY","FIDSTART");
rNewIncr  = GetReal_SYS("COPY","FIDINCR");

// --- Verify parameters ---

if (iN==iDUMMY) iN = 1;
if (iN<= 0)
   Abort_SYS("Decimation factor must be > 0.");
if ((rNewIncr!=rDUMMY)&&(rNewIncr<=0.0))
   Abort_SYS("Fid increment must be > 0.");
```

Step 9: Do setup for processing.

- Create handles, retrieve and verify symbols needed for processing.
- Initialize the progress indicator, as well as variables necessary to track progress.

```
// --- Does the Input Channel Exist ? ---

if (!iExistSymb_DB(Data,sInCh,DB_SYMB_CHAN))
   Abort_SYS("channel does not exist.");
InCh = FindSymb_DB(Data,sInCh,DB_SYMB_CHAN);

// --- Does the Output Channel Exist ? ---

if (!iExistSymb_DB(Data,sOutCh,DB_SYMB_CHAN)) {
   OutCh = DupSymb_DB(Data,InCh,sOutCh);        // Create it
   UnLockSymb_DB(Data,OutCh);
} else
   OutCh = FindSymb_DB(Data,sOutCh,DB_SYMB_CHAN);


// --- Lock the channel symbols ---

if (InCh != OutCh)
   LockSymb_DB(Data,InCh,DB_LOCK_READONLY,DB_WAIT_INFINITY);
LockSymb_DB(Data,OutCh,DB_LOCK_READWRITE,DB_WAIT_INFINITY);

// --- Prepare to do the work ---

iLines = 0;
iTotLines = iCountSelLines_DB(Data);
Progress_SYS(1);
```

Step 10: Process the data or map.

- Processes often loop through selected database lines, or call functions which do so.

```
// --- Go through all selected Lines ---

ProgName_SYS("",1);
Line = FirstSelLine_DB(Data);
while (iIsLineValid_DB(Data,Line))
{
   // --- Update the Progress Bar ---

   LockSymb_DB(Data,Line,DB_LOCK_READONLY,DB_WAIT_INFINITY);
   GetSymbName_DB(Data,Line,sTemp);
   UnLockSymb_DB(Data,Line);
   Strcpy_STR(sLabel,"Copy line: ");
   Strcat_STR(sLabel,sTemp);
   ProgName_SYS(sLabel,0);
   ProgUpdate_SYS( (int) ((real) iLines /
```

```
                                (real) iTotLines * 100.0) );

      // --- Copy/Decimate ---

      Decimate_DU(Data,Line,InCh,OutCh,iN);

      // --- Correct the Fiducial Start ---

      if ((rNewIncr!=rDUMMY)||(rNewStart!=rDUMMY)) {

         if (rNewStart==rDUMMY)
            rFidStart = rGetFidStart_DB(Data,Line,InCh);
         else
            rFidStart = rNewStart;
         if (rNewIncr==rDUMMY)
            rFidIncr  = rGetFidIncr_DB(Data,Line,InCh);
         else
            rFidIncr  = rNewIncr;

         SetFid_DB(Data,Line,OutCh,rFidStart,rFidIncr);
      }


      // --- Advance to Next Line ---

      Line = NextSelLine_DB(Data, Line);
      iLines++;

   }

   // --- Add maker ---

   EasyMakerSymb_DB(Data,OutCh,"Copy channel","COPY;");

   // --- Unlock the Channel Symbol ---
   if (InCh != OutCh) UnLockSymb_DB(Data,InCh);
   UnLockSymb_DB(Data,OutCh);
```

Step 11: Clean up.

- Destroy any created objects which should not exist once the GX terminates.
- Unlock databases and maps.
- Turn off the progress indicator.

```
   // --- done ---

   Progress_SYS(0);
   UnLock_EDB(Edata);          // --- Release the database ---
}
```

Working with Databases

The following sections describe the following procedures.

- Opening and locking a database
- Selecting lines for processing
- Locking and unlocking lines and channels
- Applying math expressions to database

Opening and Locking a Database

An important distinction exists in **Oasis montaj** between a database (DB) and an edited database (EDB). An EDB is what the user sees on the screen – it is a representation of the data (the DB) which exists somewhere on disk. With the EDB object is possible to do a number of things, including looking at, loading or unloading channels of data, creating profiles, and marking ranges of data. The one thing that cannot be done with the EDB is retrieve or change data, or select individual lines for processing. To do this, the database must be made exclusively available to the individual user. This is performed by locking the database to other users. The locking function returns the DB handle, which can be used to access the data directly:

```
// --- Get database ---

Edata = Current_EDB();
Data = Lock_EDB(Edata);

// --- Access data using the DB handle “Data” ....
.
.
.

// --- Release the database ---

UnLock_EDB(Edata);
```

Selecting Lines for Processing

Many processes access the database one line of data at a time. This is performed using a loop structure and the FirstSelLine_DB and NextSelLine_DB functions, which allow the process to step through all the selected lines in the database:

```
Line = FirstSelLine_DB(Data);
while (iIsLineValid_DB(Data,Line))
{

   // --- Process the line ---
   .
   .
   .

   // --- Advance to the next line ---

   Line = NextSelLine_DB(Data, Line);
   iLines++;       // increment a counter for the progress bar
```

```
}
```

Line selection is usually done from the OASIS interface, for instance by using the selection tool.

A common GX dialog option, found, for instance, in data export GXs, allows the user to process all the lines, just the selected lines, or the currently displayed line. The following annotated example, excerpted from the EXPDB GX, shows how this selection process is implemented within a GX.

- Within the variable declaration section, declare a DB_SELECT object. This will be a handle to the current selection list within the database; because it already exists, it is not an object that must be "Created" or Destroyed".

```
DB_SELECT   Select;        // current selection mode
```

- The line selection choice is retrieved from the workspace parameter block, and the selection is verified and interpreted as one of the line selection options.

```
GetString_SYS("EXPDB","LINE",sLine);

if (iChar_STR(sLine) == iChar_STR("D"))
   iLine = DU_LINES_DISPLAYED;
else if (iChar_STR(sLine) == iChar_STR("S"))
   iLine = DU_LINES_SELECTED;
else if (iChar_STR(sLine) == iChar_STR("A"))
   iLine = DU_LINES_ALL;
else
   Abort_SYS("Line selection invalid");
```

- The identity of the currently viewed line is determined from the edited database (EDB) object

```
Edata = Current_EDB();
```

// --- Get the name of the currently selected line ---

```
GetCurLine_EDB(Edata,sCurLine);
```

- Get the database (DB) handle by locking the edited database.

```
Data = Lock_EDB(Edata);
```

- Perform the selection; if the user wishes to use all the selected lines, nothing needs to be done before processing begins. The selection list handle is obtained using the GetSelect_DB function.

```
if(iLine != DU_LINES_SELECTED) {
   Select = GetSelect_DB(Data);
   if (iLine == DU_LINES_DISPLAYED) {
      Select_DB(Data,"",DB_LINE_SELECT_EXCLUDE);
      Select_DB(Data,sCurLine,DB_LINE_SELECT_INCLUDE);
```

```
   }
   else if (iLine == DU_LINES_ALL)
      Select_DB(Data,"",DB_LINE_SELECT_INCLUDE);
}
```

- Now that the correct set of lines is selected, do the processing.

```
// --- process the data ---
.
.
.
```

- Finally, return the line selection list to its original state.

```
// --- reset starting selections

if (iLine != DU_LINES_SELECTED)
   SetSelect_DB(Data,Select);
```

Locking and Unlocking Lines and Channels

The OASIS database is currently designed for a single-user environment, but contains features that will smooth the possible future transition to a multi-user environment. For this reason, you may be required to lock and unlock objects for use.

For example, if you are filtering the lines in a database, you require an input channel (with read status) and an output channel (with read/write status). When you are reading and writing to a channel, the system requires that you lock it, perform the reading or writing process and unlock it. This prevents the data from being altered by another process or user while you alter some of the data.

For certain operations, such as applying math expressions, the same requirements apply. However, when you look at sample code, you will notice that channels are not locked or unlocked. Because the locking and unlocking can be quite complex for math expressions, the system is designed to manage this task for you.

The following examples of locking and unlocking lines and channels are from the COPY GX, shown above.

- Creating a line or channel symbol using the DupSymb_DB function automatically locks it. (The CreateSymb_DB function does *not* lock the new symbols.) In the following code, if the output channel doesn't already exist, it is created from an already existing symbol using the DupSymb_DB function. Since it will be locked later, the lock should be temporarily "undone". It is possible to lock an object more than once, but care should be taken that an equal number of unlock operations have been performed upon completion.

```
// --- Does the Output Channel Exist? ---

if (!iExistSymb_DB(Data,sOutCh,DB_SYMB_CHAN)) {
   OutCh = DupSymb_DB(Data,InCh,sOutCh);       // Create it
   UnLockSymb_DB(Data,OutCh);
} else
```

```
   OutCh = FindSymb_DB(Data,sOutCh,DB_SYMB_CHAN);
```

- The same channels are locked throughout the line-by-line processing, so it makes sense to lock them just once, outside the loop. Notice that the input channel is locked "READONLY", since no changes will be made to it. It is a wise practice never to give more access to an object than the process requires.

```
// --- Lock the channel symbols ---

if (InCh != OutCh)
   LockSymb_DB(Data,InCh,DB_LOCK_READONLY,DB_WAIT_INFINITY);
LockSymb_DB(Data,OutCh,DB_LOCK_READWRITE,DB_WAIT_INFINITY);
```

- Within the line loop, the line symbols are locked momentarily in order to determine the line name for use in the progress bar. (See the important note about line locks below).

```
   // --- Update the Progress Bar ---

   LockSymb_DB(Data,Line,DB_LOCK_READONLY,DB_WAIT_INFINITY);
   GetSymbName_DB(Data,Line,sTemp);
   UnLockSymb_DB(Data,Line);
   Strcpy_STR(sLabel,"Copy line: ");
   Strcat_STR(sLabel,sTemp);
   ProgName_SYS(sLabel,0);
   ProgUpdate_SYS( (int) ((real) iLines / (real) iTotLines * 100.0)
);
```

- Finally, after the loop is completed, unlock the channel.

```
// --- Unlock the Channel Symbol ---

if (InCh != OutCh) UnLockSymb_DB(Data,InCh);
UnLockSymb_DB(Data,OutCh);
```

Note: It is necessary to lock a line only when accessing information about the line symbol itself (as in the example above, where the line name is retrieved). However, a lock is *not* required when accessing line data itself, for instance in the calls to `Decimate_DU` and `SetFid_DB` in the COPY GX above, even though these functions take the line handle as an argument. It is possible that in the future, if **Oasis montaj** becomes truly multi-user capable, that this 'hole" will be filled and a line lock will be required for all functions taking the line symbol as an argument. We strongly recommend that in new code that the line lock and unlock statements be placed to enclose all functions using the line symbol to access data to within the loop structure, as in the following example:

```
while (iIsLineValid_DB(Data,Line))
{
```

```
    // --- Lock the line ---

    LockSymb_DB(Data,Line,DB_LOCK_READONLY,DB_WAIT_INFINITY);

    // --- Update the Progress Bar ---
    .
    .
    .

    // --- Process the line ---
    .
    .
    .

    // --- Unlock the line ---

    UnLockSymb_DB(Data,Line);

    // --- Advance to the next line ---

    Line = NextSelLine_DB(Data, Line );
    iLines++;       // increment a counter for the progress bar
}
```

Parameter Storage in Oasis montaj

One of the more subtle features of the Oasis processing environment is the strategy employed for storing and retrieving settings, such as channel names and processing parameters that are used in your GXs. When designing applications, you need to be aware of how Oasis manages these data storage and retrieval operations. Data can be stored in a number of different places, depending on its context and purpose. Proceeding from most general to most specific, information can be stored in the geosettings META file, in an individual project, or in a database line or channel. Information can also be saved to a map, or an individual view within the map. Choice of location is dependent on how widely the user wishes the information to be made available, and how specific it is to the object with which it is stored.

Parameter Storage in the geosettings META file

The geosettings META file in the Geosoft ini directory contains global default settings for use throughout the **Oasis montaj** environment. This is the type of parameter set using the “Oasis montaj Settings” menu item (the SETTINGS GX) or the Advanced Settings (the ADVSETTINGS GX). To set a parameter in the geosettings META file, examine the following example:

```
GlobalSet_SYS("MYGROUP.MYPARAM", “Example”);
```

This command sets the system parameter “MYPARAM” in the group “[MYGROUP]” to the value “Example”. Once all values are set, it is necessary to use the following command to save the changes to the actual file:

```
WriteGlobal_SYS("");
```

Stored values may be retrieved in the following manner:

```
if (iGlobal_SYS("MYGROUP.MYPARAM",sTemp) == 0) {
      // Do something...
}
```

For more information on storing and retrieving global parameters, see the SYS.GXH header file.

Parameter Storage in the Project

Each **Oasis montaj** project has a designated area for storing program information called the parameter block. Data stored in this area is unique to the project — it cannot be shared with other projects directly.

When you run a GX, typically you will use functions, such as `SetInfoSYS_DGW` or `GetInfoSYS_DGW` to set parameters in dialog boxes and to retrieve them for processing. The values of these parameters are saved in the workspace for future use.

Saving parameters within the workspace gives you flexibility when running GX processes using Geosoft Scripts.

USING SETINFOSYS_DGW AND GETINFOSYS_DGW

The `SetInfoSYS_DGW` or `GetInfoSYS_DGW` functions transfer values between the workspace parameter block and the dialog. Without these functions, the transfer would require two steps. The following two lines:

```
GetString_SYS("MYGX", "PARAM", sBuff);      // --- retrieve from parameter block
SetInfo_DGW(Diag, _MYFORM_0, DGW_TEXT, sBuff);   // --- set in dialog
```

are replaced by...

```
SetInfoSYS_DGW(Diag, _MYFORM_0, DGW_TEXT, "MYGX", "PARAM");
```

Perusal of GX code reveals that `SetInfoSYS_DGW` (and by extension `GetInfoSYS_DGW`), is used mainly for passing four different types of information: real, integer and string variables, read-only text, file paths, and list values.

REAL, INT AND STRING VARIABLES

Most commonly, parameters are numeric or text variables. Parameters are always stored as text strings, even if they are later interpreted as `real` or `int` values using the `GetReal_SYS` and `GetInt_SYS` functions. The source line would look like:

```
SetInfoSYS_DGW(Diag, _MYFORM_0, DGW_TEXT, "MYGX", "PARAM");
```

while the corresponding lines in the resource (GRC) file, for `real`, `int` and `string` variables, could be:

```
EDIT ,,,30,"A Real value",,REAL
EDIT ,,,30,"An Int value",,INT
EDIT ,,,30,"A String value"
```

The "DGW_TEXT" parameter indicates that the parameter is to be interpreted as a text string when it is passed back and forth between the parameter block and the dialog. In the resource file the "REAL" and "INT" parameters ensure that the value is correctly interpreted and validated by type.

READ-ONLY TEXT

EDIT resources may be used to print read-only information to a greyed field in the dialog, such as a statistical value, or a channel name. Values are passed into the dialog using SetInfoSYS_DGW, or SetInfo_DGW, but there is no corresponding GetInfoSYS_DGW or GetInfo_DGW call:

```
SetInfo_DGW(Diag, _MYFORM_0, DGW_TEXT, sChan);
```

while the corresponding line in the resource file could be:

```
EDIT ,,,30,"Current Channel",N
```

FILE PATHS

The "DGW_FILEPATH" argument is used when working with file names. Even though just the local name of the file is displayed in the dialog field, the full path, including disk and directories, is passed back and forth. The source code could be:

```
SetInfoSYS_DGW(Diag, _MYFORM_0, DGW_FILEPATH, "MYGX", "FILENAME");
```

The resource lines could be either of the following:

```
FEDIT,,,40,"Create a New File",R,NEW,,,*.dat
FEDIT,,,40,"Open an Old File",R,OLD,,,*.dat
```

LIST VALUES

Individual values from lists of items are passed as text strings, exactly as in the "Real, Int and String Variables" section above. Therefore the "DGW_TEXT" parameter is used:

```
SetInfoSYS_DGW(Diag, _MYFORM_0, DGW_TEXT, "MYGX", "SIZES");
```

The corresponding lines in the resource file, along with the list items, could be:

```
LEDIT,,,20,"Sizes",R,FORCE,"Large",SIZE

RESOURCE,LIST,SIZE
ITEM, Small
ITEM, Medium
ITEM, Large
```

LIST ALIAS VALUES

Often, lists items have two parts, the value (the name which appears to the user), and the alias (which we wish to work with). An example is a choice of "yes" or "no", where the item "yes" could have as its alias the number "1". To the programmer, it is the alias that is useful. The "DGW_LISTALIAS" argument ensures that the list alias is passed back and forth to the dialog, even though the list value is seen by the user (If you wish the value instead, use the "DGW_TEXT" option). The following is an example using list aliases:

```
SetInfoSYS_DGW(Diag, _MYFORM_0, DGW_LISTALIAS, "MYGX", "LEGEND");
```

The corresponding lines in the resource file, along with the list items, could be:

```
LEDIT,,,20,"Plot Legend?",R,FORCE,"Yes",YN

RESOURCE,LIST,YN
ITEM,"Yes"          ,1
ITEM,"No"           ,0
```

COLOUR VALUES

Before v5.0.7, colour values were specified using the LEDIT component and list mappings between names like "Dark Red" and list values "R64". This cumbersome method has now been superseded by the CEDIT resource. In the GRC file, you would replace the old combination of LEDIT and LIST resource:

```
LEDIT,,,20,"Colours",R,FORCE,"Black",COLOURS

RESOURCE,LIST,COLOURS
ITEM,"Black"          ,K
ITEM,"Dark Red"       ,R64
ITEM,"Green"          ,G
ITEM,"Blue"           ,B
```

with the new CEDIT resource:

```
CEDIT,,,20, "Colours","K"
```

The old "COLOURS" LIST is no longer necessary.

In the GXC code, the CEDIT resource is accessed using the "DGW_TEXT" option, since it works directly with the colour string. When updating an older GXC file which previously used the LEDIT resource for specifying colours, you would replace

```
SetInfoSYS_DGW(Diag, _MYFORM_0, DGW_LISTALIAS, "MYGX",
"COLOURPARM");
```

with

```
SetInfoSYS_DGW(Diag, _MYFORM_0, DGW_TEXT, "MYGX", "COLOURPARM");
```

No other GXC code need be altered to make use of the CEDIT resource.

Parameter Storage in Oasis montaj objects

A special object, call the “REG” is used to store data within an individual database line, database channel, map or map view. An example is measurement units (such as “m” for meters), which are unique to a particular channel and which are stored within the channel REG. The following example illustrates how the channel REG is obtained and how parameter information is exchanged with it:

```
// --- Create (an empty) REG object ---

ChReg = Create_REG(128);

// --- Get the channel symbol, and the channel parameter data ---

Ch = FindSymb_DB(Data,sCh,DB_SYMB_CHAN);
GetRegSymb_DB(Data, Ch, ChReg);

// --- Get values from the REG ---

GetInt_REG(ChReg,"INTPARAM", iVal);
GetReal_REG(ChReg,"REALPARAM", rVal);
Get_REG(ChReg, "STRINGPARAM", sVal, sizeof(sVal));

// --- Use values...
.
.
.

// --- Set values back into the REG ---

SetInt_REG(ChReg,"INTPARAM", iVal);
SetReal_REG(ChReg,"REALPARAM", rVal);
Set_REG(ChReg, "STRINGPARAM", sVal);

// --- Set the REG data back into the channel ---

SetRegSymb_DB(Data, Ch, ChReg);

// --- Destroy the REG object ---

Destroy_REG(ChReg);
```

Note that the REG object is created, then filled with values from the channel using the `GetRegSymb_DB` function. If the REG object is altered, the new settings must be loaded back into the channel using `SetRegSymb_DB` for the changes to take effect in the channel itself. Finally, the REG must be destroyed.

The line REG is accessed in the same way as the channel REG. A new REG object is created, then filled, and must be destroyed in the end. It is accessed by using a line symbol:

```
GetRegSymb_DB(Data, Line, LineReg); // Get REG from a line symbol
SetRegSymb_DB(Data, Line, LineReg); // Set REG to a line symbol
```

Access to other objects' REGs is handled differently. For a MAP or MVIEW object, a handle to the object's own REG is returned to the user, and the user works directly with the object's REG. No Create or Destroy is performed, and no "SetREG_MAP" (for example) function is required:

```
// --- Get the handle to the map REG ---

MapReg = GetREG_MAP(Map);

// --- Set a value in the map REG ---

Set_REG(MapReg, "MAP.MAPPARAM", sVal);
```

The REG in a map's view is accessed using the `GetREG_MVIEW` function.

Data string values obtained from a REG may be loaded to, and retrieved from a dialog using the `SetInfo_DGW` and `GetInfo_DGW` commands. (When using REG-derived data in dialogs, it is most practical to work with all data, including real and integer variables, as strings using the `Get_REG` and `Set_REG` functions.) Alternatively, the REG settings can be added to the workspace parameter block using the `SetREG_SYS` function, so that the `SetInfoSYS_DGW` function can be used, as in the following example:

```
GetRegSymb_DB(Data, Ch, ChReg);
SetREG_SYS(ChReg);

// --- REG parameters are now available as SYS parameters ---

SetInfoSYS_DGW(Diag, _MYFORM_0, DGW_TEXT, "MAP", "MAPPARAM");
.
.
.
```

Note that the REG parameters should be stored in the form "GROUP.PARAM" so that both the group and parameter values are defined in the system parameter block when `SetREG_SYS` is called.

Working with Maps

Maps in **Oasis montaj** are graphic documents (.map files) that basically represent a drawing that can be displayed in a map window or printed on paper. To work effectively with maps, you need to be familiar with the purposes of maps in the system as well as the role of views and groups. An excellent introduction to Geosoft Maps and their components may be found in ***Chapter 5: Map Editing and CAD Tools*** of the ***Oasis montaj Quick Start Tutorials*** book.

Views and Groups

A Geosoft map is a document that is designed to hold spatially referenced graphical information. A map represents "sheet of paper", that when printed, is a true map. Everything that you see on a map is drawn in a map ***View***, which is fundamentally a

coordinate system that is located somewhere on the map drawing area. A map may contain any number of Views, each with its own coordinate system and located as required. Views normally represent a 2D coordinate system, but may also represent a 3D coordinate system (version 5.1.2 and later).

Graphics features that are drawn in a map View are placed in named ***Groups***. For example, contours may be drawn in a "Contour" group, survey locations may be drawn in a "Survey" group, and images are each placed in their own group. Organizing graphical information into groups allows us to deal conveniently with certain types of information. For example, by selecting an image group we are able to bring up an image manipulation tool; by selecting a flight-line plan we can activate a direct database line link; or by selecting a graphics group we can edit graphic features in the group.

3D Views also contain groups, but groups in a 3D View must be drawn on a specific ***Plane*** within the view. 3D Views may contain any number of drawing planes, and each plane may be oriented independently in space and have a relief surface defined by a relief grid.

Base and Data Views

Most standard maps created by Geosoft applications will have at least two views – a ***Base*** view and a ***Data*** view. The Base View uses the paper coordinates, with the origin in the bottom left corner, and the units of measure is mm, and the Data View normally uses the default ground coordinate system of the project, which is most often a projected coordinate system. In some cases the Data view may use an arbitrary coordinate system in which the "X" and "Y" scales are different, which would be the case when plotting an X-Y graph of a function or data distribution.

It is possible to attach a coordinate map projection to a view. The view then becomes "aware" of data in other projections. This extremely powerful mechanism allows images created with one projection to be rendered in the view's projection automatically, without the intervention of the user. Similarly, links between different maps and databases are aware of projections, and make any necessary conversions so that the cursor positions are consistent between maps and data.

Opening and Locking a Map

As with databases, a distinction exists between the edited map object (EMAP), and the map object itself (MAP). An EMAP is a handle to a map already open in **Oasis montaj**, and under the control of the **Oasis montaj** interface. Through EMAP, you can obtain information about how the map is displayed and you can control user interface features, such as getting the current cursor position, determining the currently selected view, or requesting the user to draw a polygon on a map.

To make changes to the map requires you to get a MAP handle, which is a two step process:

```
EMap = Current_EMAP();
```

```
Map  = Lock_EMAP(EMap);
```

While you have the MAP handle, you cannot use any EMAP functions. On completion of the GX, or when you need to use EMAP functions, the map can be released:

```
UnLock_EMAP(EMap);
```

This allows **Oasis montaj** to update the map as it is displayed to the user.

Accessing Views

Whether you work with the Base view, Data view or some other view depends on the type of operation to be performed. Annotations such as labels, title blocks, legends, scale bars are usually done in the Base view, while geographically located objects, such as postings, images, symbols, are drawn in the Data view. Access to a view is obtained with a call to `Create_MVIEW`, as in the following example:

```
View = Create_MVIEW(Map,"*Data",MVIEW_WRITEOLD);
```

Often maps contain more than one view for drawing data.. The "*" in "*Data" means that you want to access the currently selected data drawing view, which is the last view selected by the user, but never the "Base" view.

The MVIEW_WRITEOLD opens the view so you can to the existing view. Specifying MVIEW_WRITENEW will create a new view (replacing an view if it already exists), and MVIEW_READ lets you read from a view. When creating a new view, the coordinate system will be in millimetres relative to the map sheet, which is the same as the Base view. Use the MVIEW coordinate scaling functions to establish a coordinate system for a view (`TranScale_MVIEW, SetWindow_MVIEW, FitWindow_MVIEW, ScaleWindow_MVIEW`).

Starting a Drawing Group

A group is collection of related objects plotted in a view. Each group in a view has a unique name. Individual groups are normally created for things like images, sets of symbols, line paths and annotations. It is recommended that you create a new group each time you add a distinctive "feature" to the map. Because groups may be conveniently selected as a whole, operations such as moving, hiding or changing particular plotting attributes may be easily applied to all items in the selected group or groups in a single operation.

Although groups may be named arbitrarily, each group in a view must have a unique name. When Geosoft applications create groups, we normally place a type prefix at the beginning of the name:

TYPE_my_name

Where

TYPE Group type:

AGG	aggregate
CSYMB	ITR based symbol plot
SYMB	other symbol plots
SLEG	symbol legend
POST	posting

Groups derived from a database will usually include the database name, and possible channel in the group name. For example, an ITR-based (colours based on data value ranges) symbol plot produced from the database "chem.gdb", produced from selected values from the channel Ag, would be named "CSYMB_chem_Ag". Methods have been created to automate this procedure. The naming process and group creation process is illustrated in the following posting example:

```
GetName_DB(Data,DB_NAME_FILE,sDB);                  // Get database name
FileNamePart_STR(sDB,sDB,STR_FILE_PART_NAME);       // Get root name of
                                                    // database
GenGroupName_STR("POST",sDB,sChan,sGroup);          // Make a group name
StartGroup_MVIEW(View,sGroup,MVIEW_GROUP_NEW);      // Start the group
```

Setting Group Attributes

Group Attributes are drawing attributes that describe how to draw things in a group. A new group has default attribute values, and normally you will want to reset some yourself, using the appropriate MVIEW functions. There are no fewer than 24 attributes currently defined for map objects, and more may be added in the future. These are listed in the following table, along with default values, and the function to call to change the value. Note that linear dimensions are always specified in the coordinates of the view, which may be ground units in the case of a "Data" view. For example, if the view unit is metres, then setting the font size to 1.0 would mean that the text height would be 1 metre according to the view scale. Angles are measured in degrees, counter clockwise from the X-Axis.

Attribute	MVIEW function	Default Value
Line Thickness	LineThick_MVIEW	0.1
Line Style	LineStyle_MVIEW	0 (solid)
Line Pitch	LineStyle_MVIEW	5
Line Colour	LineColour_MVIEW	black
Line Smoothing?	LineSmooth_MVIEW	0 (do not smooth)
Font Name	TextFont_MVIEW	default from Settings
Font Size	TextSize_MVIEW	2.5
Font Angle	TextAngle_MVIEW	0
Font Colour	TextColour_MVIEW	black
Font Reference Position	TextRef_MVIEW	0 (bottom left corner)

Symbol Font	SymbFont_MVIEW	default from Settings
Symbol Number	SymbNumber_MVIEW	0
Symbol Size	SymbSize_MVIEW	2.5
Symbol Angle	SymbAngle_MVIEW	0
Symbol Colour	SymbColour_MVIEW	black
Symbol Fill Colour	SymbFillColour_MVIEW	none (transparent)
Pattern Number	PatNumber_MVIEW	0 (no pattern fill)
Pattern Style	PatStyle_MVIEW	MVIEW_TILE_RECTANGULAR
Pattern Size	PatSize_MVIEW	2.0
Pattern Line Thickness Ratio	PatThick_MVIEW	0.05
Pattern Density Ratio	PatDensity_MVIEW	1.0
Pattern Angle	PatAngle_MVIEW	0
Fill Colour	FillColour_MVIEW	none (transparent)
Clip Mode	ClipMode_MVIEW	0 (clipping off)
Rendering Transparency	sSetTransparency_MVIEW	1.0 - Opaque 0.0 - Transparent

Additional Notes

In a 3D view, the `StartGroup_MVIEW` will create a new group on the default drawing plane, which will normally be the last plane created. You can change the default drawing plane to a different existing plane by calling `SetDefPlane_MVIEW`.

If you apply a 3DN to a view that already contains groups, the groups will not have an assigned plane, and will not appear in the 3D view. You should create a drawing plane as required and call SetAllNewGroupsToPlane_MVIEW to place the unassigned groups on the new plane.

Groups drawn in 3D vies cannot be edited. It is common to let your user create what they want to see in 3D on a 2D View first so that they can edit the material before it is placed in 3D. Refer to the V3DIMG GX for an illustration of how to do this.

After you have created a 3D view, you can activate the 3D viewer to allow your user to manipulate the view right away. To do this, call ActivateView_EMAP after unlocking the map.

Adding an Image to a Map

Image data is handled using the AGG class. Images may have one or more layers, and the brightness and colour sequence of layers may be altered using the AGG class functions. Below is an example, adapted from the GRIDIMG1 GX, of how to place an image of a grid into a map. Remember that the grid name requires whatever file type decorations are necessary to identify its format for the call to `LayerIMG_AGG`. (See the FEDIT examples in the GRC Resources and Dialogs section of Part 1.)

```
// --- create aggregate ---

Agg = Create_AGG();

// --- add grid to the AGG ---

Progress_SYS(1);
ProgName_SYS("Layer",1);
LayerIMG_AGG(Agg,sGrid,0,sColor,rDUMMY);
Progress_SYS(0);

// --- open the current data view ---

View = Create_MVIEW(Map,"*Data",MVIEW_WRITEOLD);

// --- create a group name for the AGG, using the grid name without decorations ---

Strcpy_STR(sAgg,"AGG_");
FileNamePart_STR(sGrid,sGrid,STR_FILE_PART_NAME);
ToLower_STR(sGrid);
Strcat_STR(sAgg,sGrid);

// --- put the AGG in the view ---

Aggregate_MVIEW(View,Agg,sAgg);
```

Clipping Objects in a View

Each view contains a "poly-polygon" object (PLY) which specifies the area or areas where plotting is to occur. Initially, when a view is created, this PLY is not defined and no clipping occurs. Both inclusive and exclusive PLY objects can be created (and individual polygons in the PLY object can be of either type). Once a clipping PLY is added to the view, clipping can be turned on or off for individual groups within the view. Whether clipping is applied to a new group when it is created can be controlled using the `GroupClipMode_MVIEW` function. The following is an example, adapted from the TINVORONOI GX, of how a clipping region is defined and added to a view, and how the clipping may be applied to a group. In this case, the Voronoi cells of a Triangular Irregular Network (TIN) are plotted, and the user can specify whether to clip the cells to the outside boundary (convex hull) of the nodes:

```
// --- Get the Voronoi Cell edges (returned as line segments) ---

VVv = Create_VV(-32,0);      // sizeof(GS_D2LINE)
GetVoronoiEdges_TIN(Tin, VVv);

// --- Get the Convex Hull ---

Ply = Create_PLY();
GetConvexHull_TIN(Tin, Ply);

VVx = CreateExt_VV(GS_DOUBLE, 0);
VVy = CreateExt_VV(GS_DOUBLE, 0);

GetPolygon_PLY(Ply, VVx, VVy, 0);
```

```
// --- open the data view ---

View = Create_MVIEW(Map,"*Data",MVIEW_WRITEOLD);

Progress_SYS(1);

if(iClip) GroupClipMode_MVIEW(View, CLIP_ON);

// --- create path group ---

StartGroup_MVIEW(View,"Voronoi_Cells",MVIEW_GROUP_NEW);

// --- set line characteristics ---

LineColor_MVIEW(View,iColor_MVIEW(sLineColor));
LineThick_MVIEW(View,rThickness*rScale);
LineStyle_MVIEW(View,0,0.0);

// --- Set the view's clipping PLY ---

if(iClip) SetClipPLY_MVIEW(View, Ply);

// --- Plot the Voronoi Cells ---

LineVV_MVIEW(View, VVv);

// --- Turn off clipping for subsequent plotting ---

if(iClip) GroupClipMode_MVIEW(View, CLIP_OFF);
```

To set clipping for one or more groups that already exist, use the `MarkGroup_MVIEW` function to select the groups, and then call the `ClipMarkedGroups_MVIEW` function.

Version 6.0. introduced group based clipping that allows multiple clipping PLYs masks to be added to a view. These masks are named by using a text description or identifier and can be assigned to one or more groups in the view. The following functions manipulates and sets this inside a view:

Function	Description
iNumExtClipPLY_MVIEW	Get the number of extended clip PLY objects in a view
ExtClipPLYList_MVIEW	Get the names of existing extended clip PLY objects in a view as list
GetNameExtClipPLY_MVIEW	Get the name of the extended clip PLY object in a view.

GetExtClipPLY_MVIEW	Get an extended clip PLY object used by a view.
iSetExtClipPLY_MVIEW	Set an extended clip PLY object used by a view.
DeleteExtClipPLY_MVIEW	Deletes an extended clip PLY object used by a view.
SetGroupExtClipPLY_MVIEW	Sets extended clip information for group in a view.
GetGroupExtClipPLY_MVIEW	Gets extended clip information for group in a view.

Creating a Maker

A “maker” is information added to the group, which allows the user to recreate the group, using the same process which created it initially. For instance, the following maker is created in the POST GX:

```
Maker_MVIEW(View,1,1,"POST",MAKER_GX,"Posting...","POST;");
```

This maker indicates that both a map and database are required, that the POST GX was run, and that the “POST” settings from the workspace parameter block are required. When the user selects this group and brings up the right mouse-button menu, the item “Posting...” appears at the bottom, and by selecting it the POST GX will be run again, allowing the user to alter settings.

Working with 3D Views

A 3D View represents a 3D coordinate system and were introduced in version 5.1.2. 3D views may contain one or more drawing “planes”, on which normal 2D drawing can take place. A drawing plane represents a 2D coordinate system oriented in the 3D space of the 3D view. The drawing plane surface may be flat, or it may have relief defined by grid, which defined the relief in the plane’s Z axis direction.

3D views are created from 2D views by providing a 3D viewing object called an 3DN. To create a 3D view, first create a 2D view, establish the 2D coordinate system of the view (this will represent the X,Y axis of the 3D coordinate system), create a 3DN that defines the starting 3D viewing parameters, and apply the 3DN to the view:

```
// --- create and scale a 2D view ---

View = Create_MVIEW(Map,s3DView,MVIEW_WRITENEW);
TranScale_MVIEW(450000.0,6000000.0,10.0,10.0);

// --- create a 3DN ---
```

```
h3DN = Create_3DN();
SetPointOfView_3DN(h3DN,6.0,20.0,25.0);
SetRenderControls_3DN(h3DN,TRUE,FALSE,"X","Y","Z");

// --- apply the 3DN to the 2D view ---

SetH3DN_MVIEW(View,h3DN);
```

Now that the view is a 3D view, it must be located on the map using FitMapWindow3D_MVIEW. By locating a 3D view on a map, we make the view appear to be a 2D view for any functions that need to work with the view in 2D. For example, a 3D view can be selected, moved and resized on a 2D map just like any other 2D view.

In the following example, we have previously determined a location and data range that we would like to fit. The data range is not really important, but we normally choose a range that is similar to the X,Y range of the 3D view:

```
// --- locate the 3D view on the map ---

FitMapWindow_MVIEW(View,
                   rMinX,rMinY,rMaxX,rMaxY,
                   rDminX,rDminY,rDmaxX,rDmaxY);
```

Before drawing to the 3D view, we must create a drawing plane. It is then oriented with respect to the 3D coordinate system by specifying:

1. The pitch (rotation around X), yaw (rotation around Y) and roll (rotation around Z) of the new plane surface relative to the 3D coordinates system of the 3D view.
2. The X, Y and Z offset of the plane origin relative to the origin of the 3D view.
3. The X, Y and Z axis scale of the plane surface relative to the Plane coordinate system.

Examples:

```
// --- Create an X-Y drawing plane on the X-Y axis of the view --
-

iPlane = iCreatePlane_MVIEW(View,"X-Y");
SetPlaneEquation_MVIEW(View,iPlane,
                       0.0,0.0,0.0,   // no rotation
                       0.0,0.0,0.0,   // no offset
                       1.0,1.0,1.0);  // unit scaling
```

```
// --- Create an Y-Z drawing plane offset 100 units in X ---

iPlane = iCreatePlane_MVIEW(View,"Y-Z");
SetPlaneEquation_MVIEW(View,iPlane,
                       0.0,90.0,0.0,  // Y-Z plane
                       100.0,0.0,0.0, // offset 100 units in X
                       1.0,1.0,1.0);  // unit scaling
```

If you would like the plane to have a relief surface, this can be established from any grid file that registers within the plane 2D coordinate system. You specify the grid file, then the relief parameters:

```
// --- use an SDTS DEM to define surface relief ---

SetPlaneSurface_MVIEW(View,iPlane,"topo.sdt(SDT)");


// ---
// Apply 2x vertical exaggeration and remove 150m base.  Sample
// the grid to produce a 400x400 resolution relief surface.
// ---

SetPlaneSurfInfo_MVIEW(View,iPlane,400,
                       150.0,2.0,rDUMMY,rDUMMY);
```

Now you can create new groups and draw to this plane using the normal 2D drawing commands in MVIEW. Anything that can be drawn on normal 2D view can be drawn on a plane in a 3D view.

Termination and Error Handling

With normal execution of a GX, no explicit termination statement is required. The process proceeds to the last executed statement in the main program block. A number of cases exist, however, perhaps due to errors in execution, unexpected parameter values, or user intervention, when the GX must terminate prematurely. Several functions exist to handle the various scenarios:

The Exit_SYS function

The `Exit_SYS` function is a "soft landing" early exit. Execution ceases as if it were the end of the GX, and no message is written to the user or log file. No error is registered, so if the GX has been called from another GX, the calling GX will proceed normally.

The Cancel_SYS function

The `Cancel_SYS` function is normally used in response to a user selecting the "Cancel" (or corner "X" button) in a dialog. No message is output, and, if this GX is called from another GX a value of 1 will be returned to the calling GX, which should be handled as necessary (See ***Calling GXs from within a GX*** below), usually with its own call to `Cancel_SYS`.

The Abort_SYS function

The `Abort_SYS` function terminates the GX with a message. It is normally used in the non-interactive section of a GX when a bad or missing parameter is detected, and the GX may not safely continue. A call to `Abort_SYS` terminates not only the current GX, but any GXs which happen to have called the current GX.

Messages and Warnings to Users

Within interactive sections of the GX it is possible, through careful design, to "trap" bad or missing parameter values, or unusual conditions, and alert the user, without terminating the GX with a call to one of the above three functions. One method is to enclose the interactive portion of the GX within a while(){} statement, and break out of the loop only when all necessary conditions have been met. A `DisplayMessage_SYS` (or `DisplayInt_SYS` or `DisplayReal_SYS`) message can be used to alert the user. Once the user reads the message and selects the "Ok" button, the dialog comes up again so the value can be altered. The following is an example:

```
if(iInteractive_SYS()) {
   iDone = 0;
   while(iDone!=1) {

      // --- Do dialog stuff here ...
      .
      .
      .

      // --- Retrieve and validate a variable ---

      GetReal_SYS("MYGX","MYPARAM",rVal);

      if(rVal<=0.0)
         DisplayMessage_SYS("Bad value: MYPARAM", "Must be > 0.0");
      else
         iDone = 1;    // A "break;" statement would also work
   }
}
```

Using Progress Indicators

For processes that may take a long time to complete, it is important to provide feedback to the user about the current state of progress. The `Progress_SYS` functions are designed to create a progress bar with a message and "Stop" button to allow premature termination of the process. The following example illustrates the various aspects of setting up a progress indicator:

- The progress bar shows the percentage of the process which has been completed. To calculate the percentage requires that some progress variable be defined. Often, this is the number of selected lines in a multi-line database. Alternatively, it could be the number of rows in a single line of data.

```
iLines = 0;
iTotLines = iCountSelLines_DB(Data);
```

- Turn on the progress indicators. Many Geosoft functions contain their own progress reporting capabilities, which are normally disabled. Calling Progress_SYS(1) activates not only the progress indicators for the current GX, but for the called functions as well.

```
Progress_SYS(1);
```

- Set a progress name. This may be done once, or may be changed on each update of the progress indicator. In this case it will be changed with every line. This call, without a name, is used to set the initial indicated percentage to 0.

```
ProgName_SYS("",1);
```

- Update the progress bar within the process loop. The message is changed to reflected the current line, without affecting the displayed percentage. The `iCheckStop_SYS` function allow the user to break prematurely from the process.

```
// --- Go through all selected Lines ---

Line = FirstSelLine_DB(Data);
while (iIsLineValid_DB(Data,Line))
{
   // --- Update the Progress Bar ---

   LockSymb_DB(Data,Line,DB_LOCK_READONLY,DB_WAIT_INFINITY);
   GetSymbName_DB(Data,Line,sTemp);
   UnLockSymb_DB(Data,Line);
   Strcpy_STR(sLabel, "Copy line: ");
   Strcat_STR(sLabel,sTemp);
   ProgName_SYS(sLabel,0);
   ProgUpdate_SYS( (int) ((real) iLines / (real) iTotLines *
100.0));

   // --- Allow user a premature exit ---

   if(iCheckStop_SYS()) Cancel_SYS();

   // --- Do processing ...
   .
   .
   .

   // --- Advance to the next line ---

   Line = NextSelLine_DB(Data, Line );
   iLines++;       // increment a counter for the progress bar
}
```

```
// --- Set the progress indicator to 100% ---

ProgUpdate_SYS(100.0);
```

- Finally, after all processing function calls, turn off the progress indicator

```
Progress_SYS(0);
```

Creating a “Wizard” GX

A “wizard” is a GX that uses a series of dialogs, often with different branches depending on selected parameters. Examples include the NEWMAP GX, for creating a new map, and IPJSET GX, for defining projections. It is even possible to call one wizard from another (as NEWMAP calls IPJSET). Wizard dialogs generally have “Back” and “Next” buttons, except for the last dialog, which usually has “Back” and “Finish” buttons. Control may terminate with the last dialog in the series, or be returned to the first dialog. The following is an outline of a simple wizard with two subdialogs, with control returned to the main dialog on completion. There is an “Options” button in the main dialog, which begins the wizard.

- Set up some predefined values before the main code block; ensure that the buttons in the resource file contain the same values.

```
#define BACK 0
#define NEXT 1
#define FINISH 2
#define OPTIONS 1

#define DONE 99
#define MAIN_FORM 100
#define WIZARD1_FORM 101
#define WIZARD2_FORM 102
```

- Put the following “while” structure within the interactive block. The “//--- Set info...” and “//--- Get info...” comments indicate code that has been removed for the sake of brevity.

```
i = MAIN_FORM;
while (i != DONE) {

   switch (i) {

      case MAIN_FORM:    // Main Dialog has “Ok” and “Options” buttons

        Diag = Create_DGW("MAIN");

        // --- Set info....

         iD = iRunDialogue_DGW(Diag);
         if (iD == -1) Cancel_SYS();

         // --- Get Info...

         Destroy_DGW(Diag);
```

```
            if(iD==0)
               i = DONE;
            else if(iD==OPTIONS)
               i = WIZARD1_FORM;

            break;

         case WIZARD1_FORM:       // Wizard 1 has “Back” and “Next” buttons

            Diag = Create_DGW("MAIN");

              // --- Set info...

            iD = iRunDialogue_DGW(Diag);
            if(iD == -1) {
               i = MAIN_FORM;
               break;
            }

            // --- Get Info...

            Destroy_DGW(Diag);

              if(iD==BACK)
                 i = MAIN_FORM;
              else if(iD==NEXT)
                 i = WIZARD2_FORM;

            break;

      case WIZARD2_FORM:          // Wizard 2 has “Back” and “Finish” buttons

            Diag = Create_DGW("MAIN");

            // --- Set info....

            iD = iRunDialogue_DGW(Diag);
            if(iD == -1) {
               i = MAIN_FORM;
               break;
            }

            // --- Get Info...

            Destroy_DGW(Diag);

              if(iD==BACK)
                 i = WIZARD1_FORM;
              else if(iD==FINISH)
                 i = MAIN;

            break;

         case default:
```

```
         Abort_SYS("I'm lost");

} // end while(i!=DONE)
```

- Note that the "Cancel" return from the wizard dialogs returns control to the MAIN form, instead of exiting directly with a `Cancel_SYS` call. Breaking before the "Set info" commands ensures that no changes are made when the dialog is cancelled out of. Changes made by previous dialogs within the wizard are retained, however.

Calling GXs from within a GX

Occasionally it is necessary to call one GX from another, using the `iRunGX_SYS` function. This can eliminate the need to write additional code, and promotes standardization of methods.

For example, many processes, such as those that import data, may require a new database or ask the user whether or not to overwrite the current database. The following code fragment (from the IMPASC GX) performs this function:

```
// --- Get OASIS Database ---

if (iHaveCurrent_EDB() && iInteractive_SYS()) {

   if (DisplayQuestion_SYS("Import ASCII","Import data into the
current database ?")==0)
   {
      if (iRunGX_SYS("create.gx"))  Cancel_SYS();
   }

} else {
   if (iRunGX_SYS("create.gx")) Cancel_SYS();
}
```

```
// --- get database ---

EData = Current_EDB();
Data = Lock_EDB(EData);
```

Several important points should be noted:

- Databases cannot be doubly locked. If the called GX uses `Current_EDB` and `Lock_EDB` to obtain the DB database handle, be sure that the calling GX has freed its own DB handle with a call to `Unlock_EDB`. For the same reason, it is important that databases always be released after having been locked.

```
// --- Get database ---

Edata = Current_EDB();
Data = Lock_EDB(Edata);

// --- Access data using the DB handle "Data" ....
```

```
.
.
.

// --- Release the database ---

UnLock_EDB(Edata);

// --- Call the GX ---

iRunGX_SYS("AnotherGX");

// --- Lock the database again, and continue ---

Data = Lock_EDB(Edata);
```

- The iRunGX_SYS function returns 0 if it completes without an error. If an error occurs, or if the user has "cancelled" out, a value of 1 is returned, and it may be necessary to handle this. The following code queries the database to get the current range of X and Y values, then creates a new map based on the range. If any error occurs, the `Cancel_SYS` function is called to terminate execution of the GX.

```
if (iRunGX_SYS("xyrange.gx")) Cancel_SYS();
if (iRunGX_SYS("newmap.gx"))  Cancel_SYS();
```

- Sometimes it is necessary to call a GX non-interactively. In this case it is important that the called GXs parameters have been correctly set before it is called. To call a GX non-interactively, turn off the interactive mode as in the following example:

```
// --- turn off interactive mode ---

SetInteractive_SYS(0);     // 0 - interactive off

// --- call a GX non-interactively ---

if (iRunGX_SYS("mygx1.gx")) Cancel_SYS();

// --- restore interactive mode ---

SetInteractive_SYS(1);     // 1 - interactive on

// --- call a GX interactively ---

if (iRunGX_SYS("mygx2.gx")) Cancel_SYS();
```

Of course, if this GX were run in batch mode, the `SetInteractive_SYS` command would have no effect, and both called GXs would be run non-interactively, as would the calling GX.

The following is a brief list of commonly called GXs, the GXs that call them, and a brief explanation of the purpose of the call.

Called GX	Example Calling GX	Purpose
CREATE	IMPASC	Create a new database.
RANGEDB	NEWMAP	Determine X, Y range of a database
IMGRANGE	NEWMAP	Determine X, Y range of a grid
DEFMAP	GRIDIMG1	Create a new map with no scale
SCLMAP	POST	Define a scale for a map with no scale
IPJSET	NEWMAP	Set up a map projection

Preparing your GX to run as a Script

Some GXs cannot be run from a script. These include those that call GUI elements such as the histogram tool, or the colour symbol tool. Most, however, can be recorded and run successfully in a batch mode, provided that the suggested program flow outline is adhered to. In particular, the following points are important:

- Keep interactive elements sequestered from the processing code by using the "if `(iInteractive_SYS()` " statement.
- Keep non-scripting style functions (EDB,EMAP) sequestered from scriptable code by using the "if (iScripting_SYS()" statement.
- Make use of the workspace parameter block to store required values. The `SetInfoSYS_DGW` and `GetInfoSYS_DGW` functions use the workspace parameter block, so once the GX is run once interactively the required parameters are set. The parameters are retrieved using the `GetString_SYS`, `GetReal_SYS` and `GetInt_SYS` functions.
- Ensure that the parameters are properly checked and verified, as an aid in debugging and running the GX.

Compilation and Debugging

The sections to follow discuss the following compilation and debugging topics.

- Command Line Compilation
- Debugging tips and suggestions

Command-Line Compilation

The following GX.BAT is a simple command-line batch file, which may be used to compile your GX. It compiles the resource (.GRC) file first, the source (.GXC) file. It assumes that both the resource compiler (GRC.EXE) and the source compiler (GXC.EXE) are in the user's environment path, and that the user's Geosoft directory is c:\Geosoft.

@if exist %1.grc grc %1

```
gxc %1
copy *.gx c:\Geosoft
```

It may be run from the command line using the following syntax: gx mygx

Debugging Tips and Suggestions

The following are some last-minute tips from our GX developers. When fixing errors in GXs, we recommend:

ENABLE ALL ERRORS

When testing GXs in **Oasis montaj**, select the "Edit|Oasis montaj Settings" menu item, and set the "Error report level" to "All errors". Your GX may fail during testing, and not all errors are always reported to the user. Setting "All errors" ensures that you can view the reason for any premature stoppage.

REUSE HANDLES WITH CAUTION

Watch out for reuse of handles to created objects. Anything you create with a call of the form "`Create_...`" sets aside storage for the handle and a "`Destroy_...`" call releases that storage.

But before you reference that handle again, you must reinitialise it to `NULL`. This avoids bugs associated with trying to use a handle to something that no longer exists!

For instance, if you have code similar to the following:

```
hVV = Create_VV(real,10);
i = iLength_VV(VV);
Destroy_VV(hVV);
.
.
if (hVV) Destroy_VV(hVV);
```

You will see an error because the hVV handle still has a value. To resolve this situation, change the code to the following:

```
hVV = Create_VV(real,10);
i = iLength_VV(VV);
Destroy_VV(hVV);
hVV = NULL; .
.
.
if (hVV) Destroy_VV(hVV);
```

Setting the `hVV` handle to `NULL` solves the problem for you!

TRACK VARIABLE VALUES WITH DISPLAYXXX_SYS

Whenever you need to determine the value of a variable, you can use the `DisplayXXX_SYS` functions: `DisplayMessage_SYS`, `DisplayInt_SYS`, and `DisplayReal_SYS`. You can do this before and after function calls, as shown below, to track how a value changes:

```
DisplayInt_SYS("iResult before",iResult);   // Display the value
iResult = iN + iM;                          // Set value somehow
DisplayInt_SYS("iResult after",iResult);    // Display the value
```

RETRIEVE THE CORRECT VALUES AND SPELL CAREFULLY

When you retrieve values from a dialog box, always make sure that the parameters match the dialog element it is supposed to get the value from.

Also, make sure that the parameter label is spelled correctly. It is legal to try to get values for a parameter which has not been explicitly defined in the workspace parameter list, so if you put a value into a *misspelled* parameter, the parameter you meant to change will NOT be changed. If you try to obtain a value from a non-existant variable using one of the `GetXXX_SYS` functions, the returned value will be either an empty string, or the real or integer dummy values (the same as if the variable exists, but is not defined).

Part 3 – GX Function Libraries

The classes and functions in the GX API have been documented in the GXDeveloper.chm help file. This documentation is generated from our existing GXH files and compiled into HTML. This information is also available online at: http://www.geosoft.com/support/devtools/

Classes and Handles

Oasis montaj is an object-oriented system, and **GX Developer** is an object-oriented development environment. Most of the functionality in the system requires the creation of instances of a class, then manipulation of that instance through calling class methods. For example, the VV class deals with arrays of data, and the various methods in VV class (described in VV.GXH) allow you to work with data arrays.

In order to work with a class, a handle to the class must be obtained. Some objects, (such as databases, lines, channels, and maps) may already exist, and functions exists (such as Current_EDB and Lock_EDB) to return their handles. Other objects, such as the DGW dialog object, or the very large vector VV object, must be newly created, and have a Create_XXX() function, where XXX is the class name (e.g. Create_VV creates an instance of the VV class). The Create function returns an Object Handle, which can be used in class methods. Class methods are described in GXH files that bear the class name (VV.GXH describes all VV methods, and VVU describes VV utility methods). An Object Handle is always a long (32-bit) integer - it is not a pointer. An Object Handle is always passed to a class method by reference, just like any other argument to the method.

An instance of a class created within the GX should be destroyed when it is no longer required, and always before exiting your program. The Destroy_SYS method can destroy any class instance, and each class also has a Destroy_XXX method (e.g. Destroy_VV(hVV) destroys the hVV instance of a VV class).

Methods libraries are typically collections of functions with similar purposes that may require one or more class objects. Although they also carry "_XXX" tags, there is no "hXXX" instance to create (though most require at least one instance of a particular class as an argument).

Geosoft Licensing Issues

All methods available through the GX interface fall under one of the license classes below:

Type	Description
Public	Available with the free ***Viewer***, the **GX Developer** interface. No licenses are needed at all to access this method. Note, that some methods are public but offer reduced functionality.
Licensed	These methods require **Oasis montaj** to be installed and licensed.

	As long as base **Oasis montaj** license is available these methods will work.
Extended	These methods require specific licenses to execute. Since the license structure that enables these methods is dynamic it is recommended that GX Developers not use Extended methods.

Also, to protect against malicious code and viruses, Geosoft has instituted a Signed GX system. All GX's compiled by Geosoft are signed and will only execute if they have been licensed for execution on your system. If a signed GX is modified in any way it will not longer be considered safe code and will not execute.

Oasis montaj users can still execute unsigned GX's but a warning will appear indicating that this GX was not signed and asking the user to:

Run Once (Run the GX but only this time)

Run Always (Store the GX's signature and always allow that GX to run)

Deny (Do not execute this GX at all)

At this time, only members of the Geosoft Partners program can have their GX's signed by Geosoft.

VIEWGX – License Analysis

To help GX developers determine under what licenses their GX will execute, the VIEWGX program has been upgraded to a license analysis of a GX. To view this analysis run:

VIEWGX –l MyGX.GX

This will produce a listing of licenses that this GX will run under:

```
This GX will execute with the following licenses:

 10000 Oasis montaj™ Mapping and Processing System
 10100 montaj™ Geophysics
 10101 montaj™ Chimera Geochemistry
 10102 montaj™ Drillhole Plotting
 10103 montaj™ Induced Polarization
 10104 montaj™ Geophysics Leveling
 10105 montaj™ MAGMAP Filtering
 10106 montaj™ Grav/Mag Interpretation
 10107 montaj™ Airborne Quality Control
 10108 montaj™ 256-Channel Radiometric Processing
 10109 montaj™ Gravity and Terrain Correction
 10110 montaj™ Gridknit
 10111 montaj™ UX-Detect
 10200 montaj™ DAP Administrator
 10500 montaj plus™ Modeling Lite
 10520 montaj plus™ GMSYS Basic Profile Modeling
 10521 montaj plus™ GMSYS Intermediate Profile Modeling
 10522 montaj plus™ GMSYS Advanced Profile Modeling
 10523 montaj plus™ Modeling 3D
```

```
 10524 montaj plus™ Depth To Basement
 10525 montaj plus™ Isostatic Residual
 10540 montaj plus™ Grav/Mag Filtering
 10541 montaj plus™ Compudrape
 30000 Target™ Surface and Drillhole Mapping
 30101 Target™ Chimera™ Geochemical QA and Analysis
```

If a more detailed analysis is required, the –L options can be used to produce output as follows:

```
This GX will execute with the following licenses:

 0 Public License

   missing Wrapper [Create_BIGRID] Marble
   missing Wrapper [Destroy_BIGRID] Marble
   missing Wrapper [iLoadParms_BIGRID] Marble
   missing Wrapper [Run_BIGRID] Marble

 10000 Oasis montaj™ Mapping and Processing System

   OK

 10100 montaj™ Geophysics

   OK

 10101 montaj™ Chimera Geochemistry

   OK

 10102 montaj™ Drillhole Plotting

   OK

 10103 montaj™ Induced Polarization

   OK

 10104 montaj™ Geophysics Leveling

   OK

 10105 montaj™ MAGMAP Filtering

   OK

 10106 montaj™ Grav/Mag Interpretation

   OK

 10107 montaj™ Airborne Quality Control

   OK
```

```
10108 montaj™ 256-Channel Radiometric Processing
  OK
10109 montaj™ Gravity and Terrain Correction
  OK
10110 montaj™ Gridknit
  OK
10111 montaj™ UX-Detect
  OK
10200 montaj™ DAP Administrator
  OK
10500 montaj plus™ Modeling Lite
  OK
10520 montaj plus™ GMSYS Basic Profile Modeling
  OK
10521 montaj plus™ GMSYS Intermediate Profile Modeling
  OK
10522 montaj plus™ GMSYS Advanced Profile Modeling
  OK
10523 montaj plus™ Modeling 3D
  OK
10524 montaj plus™ Depth To Basement
  OK
10525 montaj plus™ Isostatic Residual
  OK
10540 montaj plus™ Grav/Mag Filtering
  OK
10541 montaj plus™ Compudrape
  OK
30000 Target™ Surface and Drillhole Mapping
```

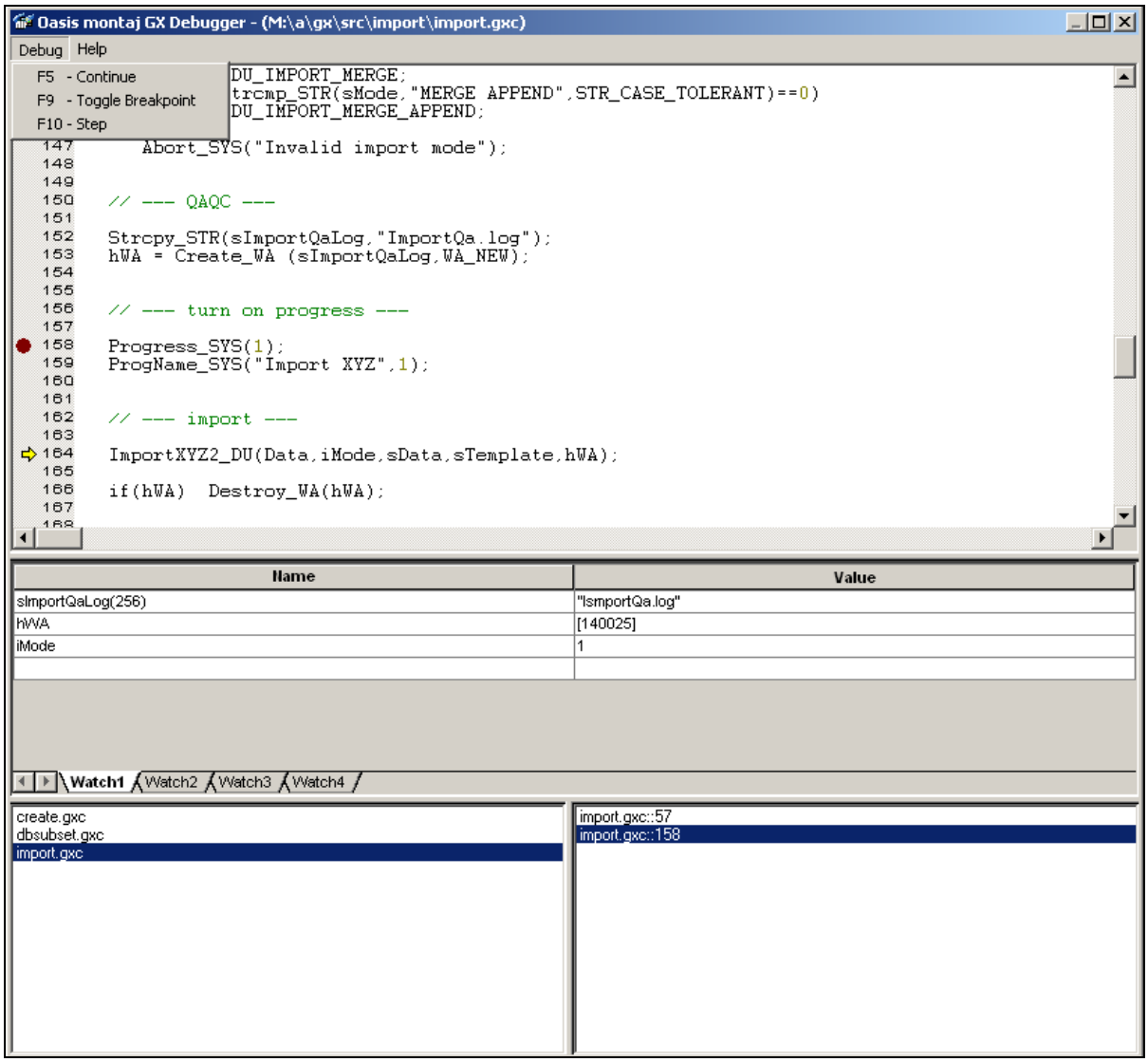
```
  OK

30101 Target™ Chimera™ Geochemical QA and Analysis

  OK
```

Each general license supported by Geosoft will be listed along with either “OK” or the reasons this GX will not execute under this license. This program can be very useful in determining what licenses will be required to run your GX. Note that the license tables can and do change between versions so the 6.0 version of VIEWGX is only accurate for the 6.0 version.

Part 4 – GX Debugger

As of **Oasis montaj** version 5.1.8, the **GX Developer** is endowed with a debugger, modelled after the general style of Microsoft's Visual Studio. Although the GX debugger does not offer the full range of the Visual Studio functionalities, it is a flexible and well adequate tool for debugging GXs, and will tremendously speed up your development.

By enabling the debugger on a GX, the user can step at run-time through the GXC code of the GX. The user may place break points in the code, and search for strings. Furthermore, this process gives the user access to the memory stack as well as the ability to modify the stack at run-time. The illustration below displays a typical GX debugger session. If you are familiar with Microsoft's Visual Studio or a similar development environment, you will easily adapt to using this debugger.



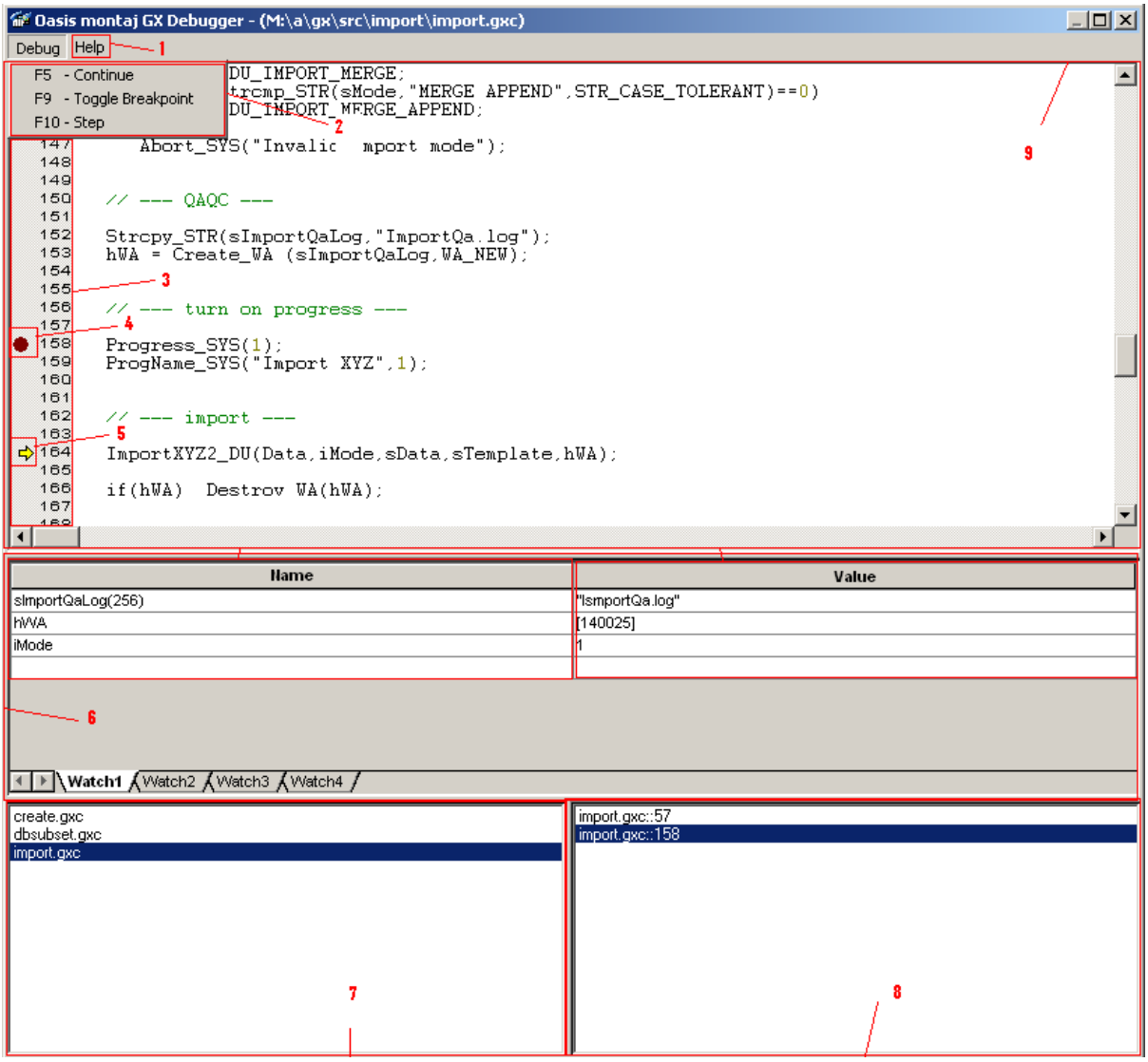
To enable the GX Debugger, first load the menu ***"Debug"*** from the list of **Oasis montaj** menus. This menu offers two GXs:

- ***DBGEnable.gx:*** enables the debugging of the specificied GX, and
- ***DBGDisable.gx:*** turns off the debugger of the specificied GX.

DBGEnabled prompt the user for a directory name, and scans it to find all GXC source files under its directory tree. These are the files that will be available for viewing and debugging in the debugger interface. The second prompt determines the name of the GX in which to place the first break. The next time the user runs directly or indirectly thisGX, the process will realise the GX Debugger process, and stop at the first execution line of the GX. If the specified GX is not found, the user is given an appropriate message, and is placed back in the GX Debugger dialogue. DBGDisable simply removes the ability to break at the first line of the first GX.

Usage

The illustration below displays the various parts of the GX Debugger.



The toolbar consists of the ***Debug*** menu, and the ***Help*** menu.

1 - The ***Help*** menu displays a summarized version of the GX Debugger usage.

2 - The ***Debug*** menu enable the user to toggle (F9) on/off break points in the source code, continue (F5) to the next break point, or proceed (F10) at the rate of one execution line at a time. These short keys are the same as the Visual studio keys.

3 – The ***Gutter*** displays the line numbers, breakpoints and the current line of execution.

4 – The user can set/unset ***Breakpoints*** at the current cursor position by hitting F9. A brown circle identifies the breakpoints in the gutter. The debugger allows you to set breakpoint on any line in any of the files preloaded into the file list. Many lines do not actually contain any execution code and the debugger will skip over break points in these lines even if you set a breakpoint on them. Problems also arise on multi-line statements where the breakpoint will only be caught if set on the last line of the GX.

5 – A yellow arrow indicates the ***Current line of execution***. If no such symbol exists in the file you are currently viewing, the current position of the GX execution is elsewhere in one of the other files in the File List. When you press F9.

6 – The ***Watch Window*** contains four tabs where you can add watches to variables in the debugged GX's. Add a watch by typing the variable's name in the Name column. If a variable is not in the current scope, it won't be displayed. The size of strings and arrays are indicated in parentheses following the variable name. The second column (Value) contains the values of the watch variables. In the case of *string*, *int* and *real* variables you will be able to edit the values before stepping to the next line of execution or continuing the GX run. Object handles are treated differently. The numeric value of the objects handle is shown in square braces. When a handler for an object exists (at the moment only *IPJ, PLY* and *META* is supported) a double click on the handle value will pop up an editor or viewer for the object. In some cases, only a simple string representation will be presented in a read only text box (currently the case for IPJ and PLY), and in other cases a full-fledged editor will be started (*META* is a good example).

7 – All the files found underneath the directory you specified to the DBGENABLE GX will be displayed in the ***File list***. You can view the source code of any file in the list by double clicking on it.

8 – The ***Breakpoint list*** shows all the breakpoints active in the session. Double clicking on these will take you to the location in the source code.

9 – The ***Source Code*** displays the colorized source code of one of the files in you File List. The colourized scheme follows the Visual Studio standards.

Notes

There are dividers between the different parts of the debugger window that allow the resizing of the parts to individual preference. The whole window can also be resized. The positions will be remembered for the duration of the debugging session.

The watch windows are not the only way to inspect variables' values. If you hover over a variable in the source code, a tool tip will pop up displaying its value.

When stepping (F10) to the next line of execution, and the GX is terminated, the debugger will retain the stepping mode. This means as soon as on of the GX's in the file list is run the debugger will pop up and display the current line of execution at the first executing line in that file. Another handy feature of stepping is that you may 'step into' GX's in your file list by hitting F10 on a iRunGX_SYS call to such a GX.

Currently there is no checking for consistency between the GX binaries and the source code. If an outdated binary is used with newer source code, the breakpoints and source window may display wrong information and can cause confusing situations. For example, a blank line may be the current line of execution or none of your breakpoints will stop the execution of the GX. In this eventuality stop the process, reload the GX to debug, and run it again.

Dummy values will always be shown as an asterix character '*'. Similarly, you can change a numeric value of a variable to a dummy by typing an asterix in the Values column.

A specific element in the array may be indicated using square braces containing the 0-based index into the array. When you attempt to view an array by typing its name in the Name column it will show the first element (VariableName[0]) in it. The alternative is to hover over the variable in the source code window; the tool tip will display the entire array.

Part 5 – Working with other languages

Other Language Support

The **GX Developer** environment also provides an API for accessing Geosoft technologies from other programming languages. This allows the creation of DLL's that can run inside **Oasis montaj** and interact with the system through the same interface that GX's do. Other languages can also be used to produce external applications (see ***Part 6 - Using the GX API Externally***).

Oasis montaj 6.0 installs all necessary components needed to run other language applications from GX's. No additional software is needed. Other language DLL's will run under the **Oasis montaj** ***Viewer*** and **Oasis montaj** without modification.

Oppi applications are usually placed in the <*geosoft*>*bin* directory but can be placed in a different location if desired. A registry setting is required to locate each DLL that is not placed in the <*geosoft*>*bin* directory. Add the following registry key to the system running the application:

HKEY_LOCAL_MACHINE\SOFTWARE\Geosoft\{KEY}\3rdPartyDLLs.

The {KEY} is the version of oasis montaj that will be running. In the standard Oasis montaj version this key is "Oasis montaj" and for the Viewer it is "Oasis montaj Viewer". However, some special versions of montaj for our CS customers do have unique key names.

Once this key is created, populate it with the names of each DLL needing to be redirected to a different location. For example, if a custom GX called a method in my geocustom.dll file installed in c:\program files\custom, I would create a new string in the registry:

Name: geocustom.dll

Data: <Program Files>\custom\geocustom.dll

The first time your GX is run, montaj will try to load the DLL from <geosoft>\bin first. If it is not found there, it will check this registry setting for any matches of the DLL name and load it from the specified path instead. Until montaj has been restarted, the DLL will always be loaded from the alternate location.

C Programmer Support

Introduction

This section describes how to call GX functions in **Oasis montaj** from external DLLs that have been called from a GX, and from stand-alone external 32-bit Windows programs using the External API.

External applications and GX Developers access Geosoft data processing DLLs through the GX API (Application Programming Interface). This interface performs thorough parameter checking and passes calls on to the appropriate processing DLL.

Programmers using this interface must be experienced in using their own C programming environment. **Oasis montaj** 6.0 was developed using Microsoft Visual Studio .NET 2005, and the sample programs are supplied with Microsoft compatible make files and library files. Programmers using other development environments must modify their environment somewhat to address their own situation.

Installation

The **GX Developer** installation already contains all that is required to build other language DLL's and external applications. The following are the directories:

apps\lib\geogx.lib	This is the main library file that contains all functions in the **GX Developer**. This library files should be included in your list of link libraries.
apps\lib\geodist.lib	This library is required for external applications. It provides the very small set of methods in the External API (see Part 6).
apps\include	C Header files: gx_lib.h — C header file for all GX functions in the current release. This header file is only for the purpose of providing prototypes to C compilers. The original GXH files (gxdev\gxh*.gxh) should be referred to for information on actual function usage. Note that there are both standard calling convention (GX_STANDARD_FUNC) and C calling convention (GX_WRAPPER_FUNC) wrappers for every function. C programmers should use the GX_WRAPPER_FUNC functions. gx_define.h — #define statements for all defined constants in the **GX Developer** GXH files. These are for use in your programs, but the original GXH files should be used to obtain information on usage. If the constant definitions conflict with definitions used in your programs you should not include this file and instead use the explicit form of the constants. gx_extern.h — C header file includes functions to create and destroy the GX Object Pointer, and display or retrieve error message. These functions are required by stand-alone programs.

apps\examples\c	C Application Examples:	
	gridstat	Console program to compute grid statistics.
	gridcopy	Console program to copy/convert grids using IMG class interface.
	chanadd	Console program to add a value to a database channel.
	chanstd	Identical to chanadd but uses the "stdcall" function interface.
	callfunc	A sample DLL that is called from a GX running under **Oasis montaj**.
	licensecheck	Console program to detect if a license is present. Works in conjunction with **Oasis montaj**.
	OMScript	GUI program that can execute GS Scripts and GX's in GUI mode.
apps\examples\CSharp	C# Samples using the new .NET interface:	
	chanadd	Console program to copy/convert grids using IMG class interface. (C# version).
	OMScript	GUI program that can execute GS Scripts and GX's in GUI mode (C# version).

Compilation Environment

If working with Visual Studio .NET, setting up your compilation environment requires the following:

1. Add the <*Program Files*>*Geosoft\GX Developer\apps\include* directory to the list of include directories.
2. Add the <*Program Files*>*Geosoft\GX Developer\apps\lib* directory to the list of library directories.
3. Add one of the the *geogx[_xxxx].lib* file to the list of link libraries. If you will be calling GX functions from other DLL (See **API Interfaces** for mor info on this).
4. If you are working in another environment, you must set up your system in a similar way according to your own requirements. Also, it may be necessary to modify the definition of certain values at the beginning of the gx_lib.h file that will customize the prototype syntax to your own requirements.

External Stand-Alone Applications

You can create separate programs completely independent of **Oasis montaj**. These programs must create a handle to the Geosoft function libraries and use that handle in all their calls. It is important to note that an external program may not be able to call "app" functions (they begin with App_ in gx_lib.h). Although some of these methods can be called externally they may not behave in the same way they do under **Oasis montaj**.

Please note that all paths used below are relative to the main **GX Developer** installation path:

<Program Files>\Geosoft\GX Developer

Refer to *apps\examples\c\gridstat\gridstat.c* for a simple example of a console program that calls GX functions. The basic procedure is:

1. Declare a GX_OBJECT_PTR (GX Object Pointer) and any GX_HANDLE variables that may be required for classes you will be using.
2. Create a GX Object Pointer by calling Create_GEO. You can only create one GX object pointer in your application.
3. Call GX functions as required. The *apps\include\gx_lib.h* file provides the C compiler prototypes. The documentation for the functions can be found in *hlp\GX Devloper.chm*. Note that the C functions always take the GX Object Pointer passed by value as the first argument. The GXH prototypes do not declare the GX Object Pointer because it is implied.
4. Always check for errors immediately after calling a GX function. If an error occurs, deal with the error.
5. Destroy the GX Object Pointer.

DLLs within Oasis montaj

To write a DLL that runs inside montaj, just set your compiler to build a DLL instead of an EXE. It will also be necessary to create a GX that will call your DLL. When your GX calls your DLL, the first argument will be the GX Object Pointer needed to call all Geosoft functions.

Refer to *apps\examples\c\callfunc\callfunc.c* for a simple example of a DLL that is called from a GX within **Oasis montaj**. The basic procedure is:

1. Create a DLL with functions with the first parameter as a void pointer and returning a long (or double if it returns floating point) with the C calling convention (see callfunc.c):

```
// --- iSum_CALLFUNC ---

__declspec(dllexport)
long __cdecl iSum_CALLFUNC(  // returns sum of two numbers
         GX_OBJECT_PTR pGeo,  // geosoft handle
         const long *pl1,     // first number
         const long *pl2)     // second number
{
   return(*pl1 + *pl2);
}


// --- Sum_CALLFUNC ---

__declspec(dllexport)
void __cdecl Sum_CALLFUNC(
         GX_OBJECT_PTR pGeo,  // geosoft handle
         const long *pl1,     // first number
         const long *pl2,     // second number
               long *plSum)   // returned sum
{
   *plSum = *pl1 + *pl2;
}


// --- ChanBase_CALLFUNC ---

__declspec(dllexport)
void __cdecl ChanBase_CALLFUNC(      // add two channels
         GX_OBJECT_PTR pGeo,     // geosoft handle
         const long *phDB,       // database handle
         const char *pcChan,     // channel name
         const double *pdValue)  // base value to add

{
```

2. Create a GX prototype of your DLL function placing the DLL name in brackets so **Oasis montaj** will know where to find it (see callfunc.gxh):

```
//-------------------------------------------------------------
// iSum_CALLFUNC    Return the sum of two numbers

[callfunc]                    // name of the DLL
int iSum_CALLFUNC( int,       // first number
                   int );     // second number

//-------------------------------------------------------------
// Sum_ADD_     Sum two numbers

[callfunc]                   // name of the DLL
void Sum_CALLFUNC( int,      // first number
                   int,      // second number
                var int);    // returned first+second.

//-------------------------------------------------------------
// ChanBase_CALLFUNC  Add a base value to a named channel.

[callfunc]               // name of the DLL
void ChanBase_CALLFUNC(
                 DB,     // database
                 string, // channel name (must exist)
                 real);  // base value to add
```

3. Call the DLL function within your GX (see test1.gxc):

```
if (iSum_CALLFUNC(iVal1,iVal2) !=  3)
      Abort_SYS("there is an error in iSum_CALLFUNC");
```

It is important to note that DLLs cannot call Create_GEO to make another GX Object Pointer. Only one such object can exist in any application and any attempt to create the object will abort the application.

In a DLL running inside montaj, the “app” functions are accessible. App functions are those that control or use the **Oasis montaj** GUI and are indicated by the [_public_app], [_licensed_app] or [_extended_app] identifier on the function prototype in the *.gxh files, and by the prefix “App_” in the function names in gx_lib.h.

Passing Arguments

All GX functions defined in GXH files require that the GX Object Pointer be passed by value as the first argument. This is an implied argument that is not specified in the GXH prototypes.

The only possible argument types (other than the GX Object Pointer) are:

long int (GX_LONG, 32 bits)

double (GX_DOUBLE, 64 bits)

strings (null terminated char arrays)

handle (GX_HANDLE, which is a long int)

All arguments are passed by reference. This includes any required constants. To pass an integer constant, use the _l() macro, which is defined in geoextern.h. To pass a double constant, use the _d() macro. For example:

```
hImg = CreateFile_IMG(pGX,_l(GS_DOUBLE),"test.grd");
```

Accessing Data

Data is either accessed by passing arguments through the GX functions, or with the use of a VV or VA. Data in a database can also be modified using database functions described in **gx\include\du.gxh**.

Before accessing data directly, you should try to find a function that will do exactly what you want to do to the data. The best way to do this is to find an existing GX that does almost what you are trying to do and see how it works. All GX source code is included in the **gx/src** directory.

If you must access the data directly, the most common way is as follows:

1. Create a VV (or VA) class instance to hold the data:
   hVV = Create_VV(pGX,_l(REAL),_l(0));
2. Read data from a data source, either a database:
   GetChanVV_DB(pGX,&hDB,&hLine,&hChan,&hVV);
   Or read data from a grid image:
   ReadX_IMG(pGX,&hIMG,&lX,_l(0),_l(0),&hVV);
3. Get a pointer to the data. This requires a VM class to hold the data:
   hVM = Create_VM(pGX,_l(REAL),_l(0));
   GetVM_VV(pGX,&hVV,&hVM,_l(0));
   pdData = GetMR_VM(pGX,&hVM);
   The pdData is now a double pointer to the data.
4. Process the data (in this example, add 1000.0 to each value):
   lLen = iLength_VM(pGX,&hVM);
   for (l=0;l<lLen;l++) pdData[l] += 1000.0;
5. Put the data back in the VV:
   SetVM_VV(pGX,&hVV,&hVM,_l(0));
6. Put the VV back in the database:
   PutChanVV_DB(pGX,&hDB,&hLine,&hChan,&hVV);
   Or write the data to a grid image:
   WriteX_IMG(pGX,&hIMG,&lX,_l(0),_l(0),&hVV);
7. Destroy each class instance (handles):
   Destroy_VV(pGX,hVV);
   Destroy_VM(pGX,hVM);

The ChanBase_CALLFUNC function in the **apps\examples\c\callfunc\callfunc.c** provides a working example of how to do this.

A common question at this point is why have a VV and a VM? A VV is intended to store very large arrays of data, and all VV methods are optimized to work efficiently regardless of the array size. Even very large VV instances will only use a very small

amount of system RAM, which allows OASIS to work with many VV instances at the same time. A VM on the other hand actually allocates the required memory from the operating system, and it will therefore only be efficient when the requirement fit comfortably within the available RAM.

Error handling

If an error occurs in a GX function, the error condition is registered internally and the method will fail. The only way to determine if an error occurred is to call the iCheckError_SYS() function. If errors have occurred, you must deal with them.

In a stand-alone program, you can call ShowError_GEO() to directly display the error or sGetError_GEO() to retrieve the errors one by one. In DLLs, you can call the ShowError_SYS function (an App function) to display the error to the user.

GUI Applications

If your program is build using a GUI of some type and is not a Console application we recommend you enable Geosoft GUI support by passing the hWND handle of the main window of your program to the Create_GEO function. This will enable you to call ShowError_GEO and see a GUI error message appear (as opposed to writing to the console). It will also enable progress bar support when you use the Progress_SYS() functions.

Licensing Issues

Functions that are not licensed to run on the computer at run time will produce an error when the function is called. Note that this means a function may work on a development computer that is licensed to use a particular function, but when it is run on a user's computer it will fail if that user is not licensed to use that function.

Calling Conventions

The **GX Developer** libraries are built using both the Microsoft C "_cdecl" calling convention and standard calling conventions (Microsoft "_stdcall"), which is used by other languages such as Visual Basic and Fortran. Standard calling convention versions of functions are indicated by the "Std_" prefix for the function name in gx_lib.h.

Visual Basic Programmer Support

Introduction

This section describes how to call GX functions in **Oasis montaj** from stand-alone external 32-bit Visual Basic programs as well as Visual Basic DLL's running under **Oasis montaj**.

Visual Basic users will access **GX Developer** functions through the same GEOGX.DLL that all developers use. All methods in GEOGX.DLL are exported both in the "cdecl" and "stdcall" calling convention.

Programmers using this interface must be experienced in using their own Visual Basic programming environment.

As we are moving to a .NET platform, we not longer provide Visual Basic samples. Visual Basic applications will continue to execute in **Oasis montaj** 6.0 if they are relinked to the new GEOGX dll using the headers provided.

FORTRAN Programmer Support

Introduction

Currently, it is not possible to link directly with DLLs created from FORTRAN code. However, using the Geosoft Inc. version of the "F2C" FORTRAN-to-C translation program with carefully modified FORTRAN source code, it is possible to create "C" DLLs which may be called from a GX. This may normally be accomplished with a fraction of the effort that would be required to rewrite the FORTRAN code in C. The F2C program recognises standard "vanilla" FORTRAN-77 code, with the exception of certain I/O functions noted below.

It is also not possible to create a stand-alone FORTRAN program. However, you can create a simple stand-alone C program that created the Geosoft object pointer and calls a FORTRAN derived subroutine to do all the work of a FORTRAN program. See the section on C programming for more information on how to create a stand-alone C program.

Installation

FORTRAN support is available with the **GX Developer** installation. No additional modules must be installed. The main directories are in **apps\examples\fortran**:

f2c\	Contains the f2c program to convert Fortran programs to C code: f2c.exe f2c program f2clib.h f2c standard header file for f2c converted C files. forlib.h f2c in-line fortran functions.
example\	A sample fortran subroutine that reads and writes a grid file.

Preparing your FORTRAN code for F2C

It is essential that the FORTRAN code be carefully examined and altered, not only for compatibility with the Geosoft programming environment, but to allow for successful conversion by the F2C utility. The following steps must be taken:

REMOVING FORTRAN I/O

All input/output statements, whether to the screen or to a file, must be eliminated or replaced with calls to Geosoft library functions. Library functions are accessed through intermediate routines called “wrappers”. A “wrapper” function goes “around” a library function, be it a Geosoft subroutine or one provided externally, to make it callable by the GX or the converted FORTRAN routine (See ***Creating Wrapper Functions*** below).

All parameters formerly requested at run-time through a screen prompt must now be passed as parameters into the subroutines. These parameters are normally obtained using a GX dialog. In some cases it may be necessary to add new functionality (such as defining default values). Likewise, output parameters, formerly written to the screen, must be passed out, or written to a file via its Geosoft object handle.

Illegal FORTRAN commands (i.e. those not supported by this version of F2C) include: read, write, print, open, close, rewind. Access to individual files is accomplished using the handle to the file object, and calling the appropriate Geosoft library function.

It is at the linking stage that you will discover any illegal FORTRAN I/O functions that have not been removed.

CREATING WRAPPER FUNCTIONS

There are two distinct types of “wrapper” functions. The first lies between the GX and the converted FORTRAN code. It is called by the GX, and itself calls the converted-to-C FORTRAN subroutine(s). The example file GXX_EXAMPLE.C demonstrates one of these wrapper functions.

The second type of wrapper function is called by the converted FORTRAN subroutines, and provides access to Geosoft library functions. In our example, those routines have the suffix "_WF". All are contained in the file WFUNCS.C.

ENABLING ERROR HANDLING

All routines which can fail should pass the error variable ("ierr" in our examples). Upon failure, an error message should be registered (see below), and ierr is set to 1. Control is immediately returned to the calling function. The calling function always checks the value of ierr returned, and passes control up, if necessary. **The FORTRAN “STOP” is forbidden.**

If an error occurs in a Geosoft GX library function, the error condition is registered internally and the method will fail. In the WFUNC.C example, various Geosoft GX library functions are called in order to handle (for instance) reads and writes to grids. The `sCheckTerminate_GEO` function is called (using the CHECK_STOP macro) after each function to determine if an error has occurred. The value of ierr is set to 1, and control is returned to the calling function.

Error messages are normally stored in a text file (see EXAMPLE.ERR below). When an error is detected, the relevant error message is registered, and optionally, one or more numeric or string parameters may be set within the message. Once control has

been passed out the routine and up the calling chain, the registered error message will be displayed to the user via a dialog.

To register an error and set values within the error message, see the following functions in WFUNCS.C:

registererr_wf__ : Register the error

seterrparmi_wf__ : Use an integer as a replaceable parameter
seterrparmr_wf__ : Use a floating point value as a replaceable parameter

seterrparms_wf__ : Use a character string as a replaceable parameter

DEFINING GLOBAL VARIABLES

The first operation performed within the wrapper function is to define as a global variable the handle to the Geosoft object. The Geosoft object is required by all Geosoft library functions. In WFUNCS.C, this is accomplished with a call to the InitGlobals_WF function, which defines the global structure WF_GLOBALS. This function also sets up the registration for error messages, using the user-supplied error message file.

OPENING GRIDS, FILES FOR READING AND WRITING, AND OTHER GEOSOFT OBJECTS

FORTRAN code uses the OPEN statement to assign a unit number to a file. Subsequently, this unit number is used to read or write data to the file. In converted code, a wrapper function is called instead. In place of FORTRAN unit numbers, a handle to a Geosoft grid object (IMG), or file object (BF) is returned and used in subsequent calls for reading and writing. These objects must eventually be destroyed with calls to the `CloseXXX` functions (which call the `Destroy_OBJECT` GX functions). Normally, the destruction occurs before control passes from the routine in which the object is created.

Note: It is important, in functions that return an object handle (such as occurs in the WFUNCS.C function `newggrid_wf__`), to reset a returned object's handle to 0 if an error occurs in the object creation. This is so the calling function will not try to destroy the returned object when it exits.

CALLING GEOSOFT LIBRARY FUNCTIONS

The range of GX library functions listed in the library header (GXH) files can be accessed from a "C" routine by making a few changes in the way they are called. Here is an example, using the `WriteY_IMG` function used in the WFUNCS.C function `PutRow_WF`:

The prototype as it appears in IMG.GXH:

```
[geogx] void
WriteY_IMG(IMG,     // IMG handle
           int,     // element # in y (row #)
           int,     // beginning element # in x to write (0 is the first)
           int,     // # elements to write (0 for whole vector)
           VV);     // VV handle
```

As it appears in GX_LIB.H:

```
GX_WRAPPER_FUNC GX_HANDLE GX_WRAPPER_CALL
WriteY_IMG( GX_VAR   GX_OBJECT_PTR,
            GX_CONST GX_HANDLE_PTR,
            GX_CONST GX_LONG_PTR,
            GX_CONST GX_LONG_PTR,
            GX_CONST GX_LONG_PTR,
            GX_CONST GX_HANDLE_PTR);
```

Finally, as it is used in WFUNCS.C:

```
WriteY_IMG(Globals.pGeo,      // GX object pointer handle
                              //   (defined in InitGlobals())
           long *plIMG,       // handle to the IMG object
           long *plRow,       // Row #
           long *plCol0,      // First column
           long *plNCol,      // Columns to write
           long *plVV);       // handle to the VV object
```

There are a couple of points to note:

- A new, first argument must precede all those listed in the GXH file. This is the Geosoft object handle; in our examples it is defined within the global structure as "`Global.pGeo`". The GX variable types "int", "real" and "string" are replaced in the "C" wrapper with the pointers "long *", "double *" and "char *".
- In WFUNCS.C, all the Geosoft functions use variables accessed by reference. Since the FORTRAN calls typically use float (REAL*4) variables, you will have to use the following procedure to convert from a float to a double variable:

```
dTempDouble = *fFortranReal4;
GeosoftFunction(Globals.pGeo, &dTempDouble);
```

CALLING THE CONVERTED FORTRAN ROUTINES

The file GXX_EXAMPLE.C illustrates the GX wrapper function used to call the converted FORTRAN routine. The call to the routine is preceded by a call to `InitGlobals_WF`. This MUST be called in order to initialise the global structure with the value of the Geosoft object pointer, if any call to a Geosoft library function is to made from within the routine, including the creation of grid and file resources, and the registration of error messages.

As in the WFUNCS.C example above, some type conversion may be necessary between the input parameters and the arguments required by the converted FORTRAN routines. In may be necessary to pad string variables with spaces to their full length, because comparisons of strings in the F2C'd versions of FORTRAN files compare the entire length of the string.

Function prototypes of converted "C" functions should be taken directly from the F2C'd version of the FORTRAN routine. They will contain additional underscores in the name, and may contain additional arguments, such as the lengths of character strings being passed in.

CREATING A PROGRESS INDICATOR

An example of how the progress indicator is implemented is found in the file EXAMPLE.F. For the progress meter to function, the metering must be turned on in the original calling GX by calling `Progress_SYS(1)`, or by calling the appropriate GX wrapper function in your C code: `Progress_SYS(Globals.pGeo, 1).` Updates are accomplished using the `ProgUpdate_SYS` or the `ProgUpdateL_SYS` functions. Finally, `iCheckStop_SYS()` can be placed following calls to `ProgUpdate_SYS` to stop the process when the user selects "Cancel" on the progress update window.

INCLUDE AND ERROR MESSAGE FILES

EXAMPLE.GXH

Prototypes for the wrapper functions called from the GX (from the EXAMPLE.GXC code) reside in GX header (GXH) files. Note that two versions of some exist, with and without an initial "I". The F2C functions generally require that the actual string length be passed along with the string, but for convenience, the version called from the GX can determine the string length itself, and reduce the number of variables the user must specify for the call. The name of the function has been pre-pended with the tag “YOUR” to distinguish it from regular Geosoft library functions. It is a good practice to use a unique beginning identifier for your own collection of functions to ensure uniqueness among the larger family of GX functions, as their number continues to expand.

Note that the name of the DLL must appear in the function prototype, enclosed in square brackets.

WRAPPERS.H

This file is included by both wrapper function files: WFUNCS.C and GXX_EXAMPLE.C. . It should remain in the same directory as the above “C” code. It contains the definition of the Global structure, function prototypes (both for the “_WF” functions and the converted FORTRAN routine), and any pre-defined constants.

GX_DEFINE.H, GX_LIB.H, GX_EXTERN.H

These files may be found in the GxDev\c\include directory. This location should be placed in the “Include Files” path, specified in the compiler’s project workspace. They contain information on the Geosoft function libraries. The GX_DEFINE.H file contains definitions of constant values and parameters. The GX_LIB.H file contains “C” prototypes for the library functions. Both GX_DEFINE.H and GX_LIB.H are derived directly from the Geosoft library GXH header files, and so each function in GX_LIB.H has a corresponding entry in the relevant header file, which should be consulted for help on function usage. The GX_EXTERN.H file is used by third party developers wishing to build stand-alone applications that access Geosoft functions. This module holds the functions needed to start a Geosoft session. It also contains prototypes for several functions for reading from and writing to binary files, which

are not found in the regular GX libraries, but which are useful in the context of third party software applications that need access to binary data.

F2C.H

This include file is included automatically within the F2C output files. It should remain in the same directory as the converted “C” code. If an original FORTRAN routine is duplicated over two or more files, the duplicates should be commented out, and a "C" prototype included here. This prototype should be taken directly from the F2C output.

FORLIB.H, F2CLIB.H

These files may be found in the GxDev\gxh directory. This location should be placed in the “Include Files” path, specified in the compiler’s project workspace. They include function prototypes and definitions required by the converted FORTRAN “C” files.

EXAMPLE.ERR

This file contains the error messages. The "%" variables are replaced parameters, which may be set using the “seterrparm” functions defined in WFUNCS.C. The error file itself is registered inside the InitGlobals function.

Running F2C and Building the DLL

The batch file DOF2C.BAT demonstrates the call required to convert the EXAMPLE.F file to EXAMPLE.C. The user should ensure that F2C.EXE is in the current path. The “C” output is not pretty, but it works. You should never change the C created by F2C. If you have problems in your final program, always fix the problems in your FORTRAN source code.

A list of errors, warnings and message may be produced. In the absence of errors, a listing of the subroutines converted is output.

The Microsoft “C” Version 6.0 build file EXAMPLE.DSW is included. It compiles the “C” files EXAMPLE.C, GXX_EXAMPLE.C and WFUNCS.C and produces the dynamic link library file EXAMPLE.DLL. The only “special” compiler setting, other than the defaults given for a new “WIN32 Dynamic Link Library”, is that the Geosoft wrapper function library GEOGX.LIB should be included in the “Object/library modules” field in the “General” category of the “Link” tab in “Project Settings”. Remember to add the location of the include files to the “Include Files” section of the “Directories” tab in the “Options” dialog. The DLL should be written to the Geosoft directory, or at least be included in the path.

Running the EXAMPLE GX

The Example GX opens a grid, multiplies it by a given factor, and creates a new grid. The GRC, GXC and RTF files necessary to create the GX are included for your inspection, along with the GX itself. To run the GX from within **Oasis montaj**, select “Run GX” from the “GX” menu, or select the “GX” button on the toolbar, and select the EXAMPLE.GX file using the open file dialog. Select a grid to open, and a new

grid name to create. Input and output grid types are selected as usual. Select a multiplication factor, and press "Ok" to begin. The progress indicator will indicate the current progress, as the grid is processed.

Licensing Issues

Functions that are not licensed to run on the computer at run time will produce an error when the function is called. Note that this means a function may work on a development computer that is licensed to use a particular function, but when it is run on a user's computer it will fail if that user is not licensed to use that function.

Programming Support

At this time, Geosoft is not in a position to offer programming support to FORTRAN programmers except on a casual basis. Licensed GX Developers may obtain support for GX development questions, and these may be useful for supporting FORTRAN programming requirements, but our support channels cannot support FORTRAN questions. We are currently reviewing our support strategy for this development and we hope to announce a support offering some time in the future.

.NET Programmer Support

This section describes how to call the GX function library from a managed assembly. The assembly may be a shared library that is invoked by Oasis montaj or a stand-alone windows application.

Managed applications invoke GX functions through the GX.NET API (Application Programming Interface) which is distributed through the geonet.dll assembly, which is distributed with GX developer in the apps\dll folder and installed in the Microsoft Global Assembly Cache (GAC) with instances of montaj.

Note that, unlike other GX languages, development against the geonet assembly is linked to a specific version of geonet. Oasis montaj extensions will work with different versions of Oasis montaj as long as the version that was originally compiled against is available in the GAC. We recommend that GX developers install the version they used in the GAC when distributing montaj extensions, or place the geonet.dll in the same directory as their standalone executables.

Developers using this interface must be familiar with Microsoft .NET, C# and the Microsoft .NET Framework. Oasis montaj was developed using Microsoft Visual Studio.NET 2005 and the sample programs are supplied with a C# project files. Programmers using other development environments must modify their environment somewhat to address their own situation.

Geosoft recommends either the commercial version of Visual Studio or Visual Studio Express for developing .Net solutions.

External Stand-Alone Applications

You can create programs that run independent of Oasis montaj, yet utilize Oasis's powerful GX library. These programs must first initialize GX.NET library before

they begin calling any GX functions. It is also important to note that an external program may not have access to all member functions that might be available to an assembly running within Oasis montaj.

Refer to *<Program Files>\Geosoft\GX Developer\apps\examples\CSharp\chanadd* for a simple example of a console program.

1. Instantiate the GX.NET libraries by calling the 5 parameter CGX_NET constructor.

```
CGX_NET hGXNet = new CGX_NET (programName, version,
                              maxMemory, windowHandle, flags);
```

2. Call GX functions as required. (See the GX.NET API documentation for a full list of all the classes and their associated methods.)

Assemblies within Oasis montaj

Refer to *<Program Files>\Geosoft\GX Developer\gxnet\src\gxnet.csproj* for an example of a C# assembly that is called from within Oasis montaj (it includes all the Oasis montaj .Net GXs). Note that you will likely need to fix the locations of the referenced assemblies in this project.

1. Create a new Class Library in Visual Studio and add assemblies from *<Program Files>\Geosoft\GX Developer\apps\redist\bin* as references. See *gxnet.csproj* for an example.
2. By double clicking on the following file in that project some templates for Visual Studio will be installed to quickly create .Net GXs from scratch: *Templates\GeosoftTemplates.vsi*
3. This will allow you to add well documented starting code for one of three styles of GXs. This can be done by adding a new item through the context menu in the Solution Explorer and picking one of "Geosoft GX", "Geosoft Advanced GX" or "Geosoft Wizard GX" in the Geosoft section. Follow the comments in the generated classes and form to start creating your GX.
4. Create a new menu item (see the section on Menus in Oasis montaj for a full description) with the second parameter being in the form <assembly name>(<class name>|<method>). The <class name> must be fully qualified. E.g: **dllname.dll(MyAssembly.MyGX;Run)**.
5. Place your assembly in the Oasis montaj *bin* directory and execute the menu item to run the code. When redistributing your assembly to other machines be sure to install the version of geonet you referenced into the GAC as mentioned above to ensure that it will work with any version of Oasis montaj.

It is important to note that you cannot call the 5 parameter CGX_NET constructor as this will try to initialize the GX.NET API which is already initialized. This could result in your application crashing. This constructor is for use by external applications like the example found in *<Program Files>\Geosoft\GX Developer\apps\examples\csharp\chanadd*.

Note, in an assembly running inside montaj, all methods beginning with “App” are accessible. The App methods control or use the Oasis montaj GUI and hence rely on Oasis montaj being present.

Error Handling

If an error occurs in one of the GX.NET API functions, the error condition is registered internally and a Geosoft.CERROR exception is thrown. This will enable you to wrap your code with try/catch blocks in order to handle the error as you deem fit.

Stand Alone GUI Applications

If your program is a windows application we recommend that you enable Geosoft GUI support by passing the handle of your main window into the CGX_NET constructor as the fourth parameter. This will allow you to call CGX_NET.ShowError() and see a GUI error message. It will also enable progress bar support when you use the CSYS.Progress() function.

Licensing Issues

If a licensed method is called and the computer that is executing your assembly does not have the required license and error will be thrown. This might result in your program running successfully on one computer but may not on another computer with a different license.

Part 6 - Using the GX API Externally

Introduction

To create applications that run outside **Oasis montaj**, developers use the External API, which consists of a set of redistributable files that must be installed with your application. Redistributing your application and the 3rd party components can usually be achieved fairly easily by making use of the Windows Installer XML (WIX) Toolset or another 3rd install creation utility but this is outside the scope of this document.

The recommended way to create an application (which only makes use of unlicensed functionality) that can operate completely independent of an **Oasis montaj** installation is to install all the files and *GeosoftFiles* directory in *<Program Files>\Geosoft\GX Developer\apps\redist\bin* right next to the binaries that would be making use of the API externally. Although it might be possible to remove some files that might not be used it cannot reliably be supported, nor will any guidelines be provided on how to do this.

It is also possible to make use of of an **Oasis montaj** or **Oasis montaj Viewer** installation to reduce the application footprint and make use of licensed functionality. To do this the *GeosoftFiles* directory above can be omitted and a *geosoft.key* file then placed next to the applications and other files containing a single text line with the name of the application as found in the registry (e.g. “Oasis montaj”). See the Registry section below for more info on what this means.

In addition to these files your application will need to install and/or check for the following 3rd party redistributables. Note that this is not necessary when making use of an Oasis montaj installation as described in the paragraph above.

DLL	Description
Microsoft Visual Studio 2005 Redistributable Package	DLL’s that provides the C Runtime functions, MFC and STL functionality on which the Geosoft dll’s rely. Available from here.
XCeedCry Component	Found in the installation at: *...\redist\system32_register* This is a COM component and needs to be installed and registered as a shared component in *<system32>*
Microsft .Net 2.0 Framework	Found here.

No registry settings of any kind are required to be setup, but the directory where the files are installed should be writable to the Windows user as temporary files and a user folder will be created underneath.

We can also recommend the Depends.exe utility for debugging any dll dependency problems (see http://www.dependencywalker.com/) that might arise.

Registry

Although no registry settings are required to create external applications, we do recommend that you register specific directories for your application if you want to control where temporary files and user directories will be located. During execution, applications using the external API will create log and temp files in the current working directory. To have the Log and temp files sent to a different directory, it is required that you create a key file for your application. This file, called *geosoft.key* must be in the same directory as your EXE. It contains the name of your application {MyApp}. With this key file, the Create_GEO method will look in the registry under the following key for settings:

HKEY_LOCAL_MACHINE\SOFTWARE\Geosoft\{MyApp}\Environment

GEOTEMP	Where all temp files will be placed.
GEOSOFT	Where the contents of the GeosoftFiles directory is (which can be omitted if next to the application).
GEOSOFT2	Where all user files will be placed.

This will force all temp and log file to be created in the GEOTEMP directory. Please note that the key names can conflict with Geosoft keys as well as other developer keys so please use descriptive names.

Licensing

The functions available in this API are the same ones available under the GX programming interface for **Oasis montaj**. All the [_public] methods are provided free and can used by anyone without restriction by Geosoft. However, [_licensed] and [_extended] methods do require that **Oasis montaj** be installed with an appropriate license. The “app” methods are a special group that may or may not be provided outside **Oasis montaj**. If they are provided, they may not have the same behaviour as they would if run inside **Oasis montaj**.

Part 7 – UNICODE

Introduction

In versions of **Oasis montaj** prior to 6.2, all characters were stored as single-byte characters and used as part of the local code page. This implied that maps and databases created on a computer with a specific code page would not display properly on a computer with a different code page. To resolve these issues and allow the use of complex languages (Chinese), **Oasis montaj** was updated in 6.2 to use Unicode. This now allows **Oasis montaj** to store and display characters of any language consistently.

Implementation

To reduce the implementation costs a hybrid Unicode/UTF-8 approach was used. The GUI elements based on MFC (dialogs, text boxes, menus) are fully Unicode aware and enabled. This allows the application to receive input and display the full Unicode specification.

The rest of the code source has been upgraded to the UTF-8 standard. This allows the storage of Unicode strings in standard strings without requiring a large investment in code modifications. File formats have been updated to support UTF-8 encoded strings. This allows existing file formats to change only a little while providing the full benefits of Unicode.

Text files have also been upgraded to the UTF-8 standard. New text files generated by the system will contain the UTF-8 signature characters at the beginning of the file.

Compiler

The Geosoft GX compiler (GXC) has been upgraded to support UTF-8 encoded source. Thus, users can create GX's that have special characters in their GUI elements (dialogs), in their parameters (INI settings) and even in their variable names. The COPYCH gx found in the GX Developer source code section is an example of a UTF-8 encoded GX in Chinese characters.

GX Developers

For GX developers the changes should have a very limited impact. We recommend that all GX's be recompiled for 6.2 to ensure that any special characters inside the GX (usually the degree symbol) are converted to the proper UTF-8 encoding inside the GX. Otherwise, existing GX's should execute without problem.

Note that any code that manipulates strings directly should be reviewed as UTF-8 encoded strings can have between 1-6 bytes representing one character. UTF-8 special characters are always above 127 and as such should be left alone in your code.

API Interfaces

Any code making calls to the Geosoft API either from a DLL inside **Oasis montaj** or from an 3rd party application must be aware of the interface they will use. In the versions prior to 6.2, the GEOGX dll was the standard interface library for all calls into the Geosoft API.

To provide the most compatibility to existing solutions Geosoft has created three API interfaces:

GEOGX (ANSI)

The GEOGX interface provides ANSI support to all Geosoft API functions. All strings used in this interface are single byte characters in the ANSI code page and are converted to UTF-8 internally. Any result strings are internally converted from UTF-8 into the ANSI code page. Any characters that cannot be converted to the ANSI code page will be converted to question marks (?).

Although this is the default interface (to support existing applications), it suffers performance penalties during the conversion between ANSI and UTF-8. Also, any special characters that cannot be converted to ANSI will be lost. We recommend that all applications use one of the other interfaces if possible.

GEOGX_U (UNICODE)

The GEOGX_U is a native Unicode interface for those applications that run completely in Unicode mode. All strings used in this interface wide character Unicode and will be converted to UTF-8 internally. Any result strings are internally converted from UTF-8 to Unicode.

There is no loss of characters in this interface and only a small performance penalty during the conversions. This interface is recommended for full Unicode applications.

GEOGX_UTF8 (UTF-8)

The GEOGX_UTF8 is a UTF-8 interface for those applications that use UTF-8 as their standard. No conversion at all is needed as **Oasis montaj** runs in this mode making this interface the fastest. This interface is recommended for applications that use UTF-8 as their standard.

MFC DLLs Inside Oasis Montaj

For those developing DLLs that execute inside **Oasis montaj** it is important to understand that 6.2 uses the Unicode version of MFC. We strongly recommend that all third party DLLs also use the Unicode version of MFC to ensure a consistent interface. Otherwise, your application will display GUI elements that are not Unicode aware and enabled and this will cause confusion.

Part 8 – Efficient coding techniques

Introduction

This special section gives some examples on how to work efficiently with the GX API. The code used in the examples uses GX.Net but keep in mind that the concepts introduced also applies to GXC and other languages because of overhead incurred by switching into GX API calls regardless of where the execution happens.

Mixed Code Efficiency

When working with mixed managed and unmanaged code performance becomes an issue when switching from managed to unmanaged code all the time. In this case the GX.Net API is a wrapper to unmanaged libraries so every call switches from managed to unmanaged and back.

One should always try to minimize switching between the libraries by trying to get data in bulk and/or doing the processing without multiple function calls to and from managed code.

Examples

Say for instance we want to apply a special equation to values in two VVs and put it in a third e.g.:

```
double dSpecialEquation(double dX, double dY)
{
   // Not so special after all
   return dX + dY;
}


void Run()
{
   int iN = 10000000;

   DateTime dtStart;
   TimeSpan tsDuration;

   CVV oVVx = CVV.CreateExt(Constant.GS_DOUBLE, iN);
   CVV oVVy = CVV.CreateExt(Constant.GS_DOUBLE, iN);
   CVV oVVz = CVV.CreateExt(Constant.GS_DOUBLE, iN);

   oVVx.MakeMemBased();
   oVVy.MakeMemBased();
   oVVz.MakeMemBased();
   oVVx.FillReal(30.0);
   oVVy.FillReal(60.0);

   dtStart = DateTime.Now;
   for (i = 0; i<iN; i++)
      oVVz.SetReal(i, dSpecialEquation(oVVx.rGetReal(i),
         oVVy.rGetReal(i)));
   tsDuration = DateTime.Now - dtStart;
```

```
        double dZ = oVVz.rGetReal(200);
        System.Diagnostics.Debug.WriteLine("That took " +
              tsDuration.TotalSeconds + " seconds!");
    }
```

The output of Run gave me:

```
That took 372.045536 seconds!
```

That seemed very slow so I looked and found a VV function that does exactly what my "special" equation does. This is the first thing a GX programmers should do, always scan the API for some function that does exactly what he wants to do without getting/setting single values one-by-one. The following change to the relevant section results in:

```
        dtStart = DateTime.Now;
        oVVx.Add(oVVy, oVVz);
        tsDuration = DateTime.Now - dtStart;
        double dZ = oVVz.rGetReal(200);
        System.Diagnostics.Debug.WriteLine("That took " +
              tsDuration.TotalSeconds + " seconds!");
```

```
That took 0.8281356 seconds!
```

Note that this gives us the fastest time possible for adding 10 million numbers and placing in a result vector in this code. Two more examples are given because there may not always be a function readily available in the GX API for some cases. The first example uses one of the powerful EXP classes (note that there is one for DU and also IEXP that works on IMG objects). Also keep in mind that any equation can be used, even an algorithm with multiple lines (see the classes' documentation for more details).

```
        dtStart = DateTime.Now;
        CVVEXP oExp = CVVEXP.Create();
        oExp.AddVV(oVVx, "X");
        oExp.AddVV(oVVy, "Y");
        oExp.AddVV(oVVz, "Z");
        oExp.DoFormula("Z = X + Y;", Constant.STR_VERY_LONG);
        tsDuration = DateTime.Now - dtStart;
        double dZ = oVVz.rGetReal(200);
        System.Diagnostics.Debug.WriteLine("That took " +
              tsDuration.TotalSeconds + " seconds!");
```

```
That took 1.1093608 seconds!
```

We could also use the GX.Net vector classes that give direct access to the internal data through simple typed indexes. NOTE: these only work if VVs (or VMs) are memory based, which is why the MakeMemBased calls were used above.

```
        dtStart = DateTime.Now;
        GXNet.DoubleVector vDoublex = CGX_NET.GetDoubleVV(oVVx, iN);
        GXNet.DoubleVector vDoubley = CGX_NET.GetDoubleVV(oVVy, iN);
        GXNet.DoubleVector vDoublez = CGX_NET.GetDoubleVV(oVVz, iN);

        for (i = 0; i<iN; i++)
           vDoublez[i] = dSpecialEquation(vDoublex[i], vDoubley[i]);
        tsDuration = DateTime.Now - dtStart;
        double dZ = oVVz.rGetReal(200);
        System.Diagnostics.Debug.WriteLine("That took " +
              tsDuration.TotalSeconds + " seconds!");
```

```
That took 0.937494 seconds!
```

This shows that even when doing the number crunching in managed code comparable results can be achieved to doing it in unmanaged code.