Institut für Datentechnik
und Kommunikationsnetze

Technische Universität
Braunschweig

GSEOS
Language Description

Revision 1.11

IDA-GSEOS-0003

March 2008

Prepared by
Tim Wittrock

Table of Contents

Tables and Figures

1 Scope

1.1 Purpose of this Document

This document is a short description of the GSEOS language (called G). The language is used to create functions that decode data blocks from external or internal sources, to send messages to external devices, to evaluate QLook Items and to create and run batch files.

This document applies to GSEOS with compiler version 20080313 or later (you can find the compiler version number in Control Panel→Compiler→Debug Settings, called VersionID). Older versions may differ in some points from this description.

Chapter 2 will provide a short introduction into the tasks and qualities of the language. After that, chapter 3 describes the main features of G for structured programming (like conditional execution and loops). In chapter 4 you will find information of built-in data types and how to define new types on demand. The next chapter (5) shows, how to use these types by declaring variables, and in chapter 6 you will learn how to access the data represented by these variables. Since not all values have to be variables, chapter 7 provides information for using constant values in your program. Since you probably want to manipulate data inside your programs, chapter 8 explains data manipulations like calculations and array operations. Chapter 9 will teach you how to use functions (user defined as well as built-in and DLL), decoders, and QLook items. Special commands that are used for data block operations or time and event controlling are shown in chapter 10.

After that, we leave the description of the GSEOS language itself and read about the built-in preprocessor, which is used for commenting and definition of symbol replacements in chapter 11. Chapter 12 helps to understand the messages you may see during compilation or at runtime. The customization of the compiler and virtual machine is the subject of chapter 13. Having presented a list of all keywords used by G in chapter 14, chapter 15 finally mentions some tasks that are not finished yet.

1.2 Change Record

Table 1 Change Record

Date	Revision	Author	Affected Sections
6/10/1999	1.0	Wittrock	All sections
7/6/1999	1.1	Wittrock	8, 10, 11
7/27/1999	1.2	Wittrock	3, 10, 11
8/5/1999	1.3	Wittrock	9,10
11/18/1999	1.4	Wittrock	3.2.2, 4.1.2, 5.2, 7, 11.1.3, 10, 12
07/04/2000	1.5	Wittrock	6,7, 10, 12 (new), 13

Date	Revision	Author	Affected Sections
07/11/2000	1.6	Wittrock	
10/25/2000	1.7	Wittrock	6 (new), 7 (new), Index (new), complete revision
03/09/2001	1.8	Wittrock	Control panel description
01/10/2002	1.9	Wittrock	New: postfix increment / decrement
10/25/2002	1.10	Wittrock	5.4.2 (readonly): Example corrected
03/14/2008	1.11	Wittrock	4.2.2 (endianness): new keywords “little-endian” and “big-endian” added, complete revision of endian keywords 4.2.3 (bit-precise access): Support for 32-bits IEEE, 32-bits and 48-bits MIL floating point values in blocks added 12.1: Warning “Union coverage differs” added

1.3 Abbreviations

Table 2 Abbreviations

BDM	Block Data Manager
DLL	Dynamic Link Library
GSEOS	Ground Support Equipment Operating System
RTL	Runtime Library

1.4 Documentation Conventions

The following text will specify the typing conventions that are used to explain the GSEOS language syntax.

- `Keywords` have to be typed exactly as shown in the documentation.
- ***References*** refer to other elements. It can be for example a user-defined name or a complex expression defined in the documentation.
- **:=** following a ***reference*** specifies the definition of the ***reference***.
- **[** ***Expressions*** **]** enclosed in square brackets are optional and may be omitted.

- Enclosing **(*expressions*)** in brackets is used for grouping.
- **(*Expressions*)+** in brackets followed by a plus may appear one or more times.
- **(*Expressions*)*** in brackets followed by a asterisk may be omitted or appear one or more times.
- ***Statements*** are lists of expressions separated by semicolon.
- The vertical rule **|** between two expressions indicates an alternative. You may choose either the expression on the left or on the right side.

```
Examples are printed this way   // comments are placed behind "//"
```

2 Introduction

This is the documentation of the common configuration language for GSEOS. This language allows the creation of complex programs to solve even complicated tasks. The programs are processed by a built-in two-pass-compiler and executed by a built-in virtual machine. The language supports multiprocessing and time- or event-driven execution of special functions (like decoders). It is used for

- Data block definitions (based on C-like type definition) for easy processing data from external hardware, even if its data-format is quite unusual.
- Data decoding (done by event-driven functions)
- Commanding and batch processing
- Monitoring
- System control (logging, data recording, interfacing)

To provide easy programming, the GSEOS language syntax is similar to the syntax of the well-known programming language C. All of the tasks mentioned above can use common functions for data processing, so the user does not have to solve the same problems multiple. The main differences to C are the following:

+ Data driven functions (realized through special function headers)
+ Bit based memory placement and endian conversion for data blocks from external sources
+ Full runtime range checking of array indices.
+ Additional data types (bitmaps, Boolean)
+ Easy value-to-colored-text translation (text references)
+ System function library (for dialog boxes and interfacing)
+ Complex data types allowed as return values of functions
+ Mathematical operations on all elements of an array at once
– No dynamic memory management functions and pointers
– Identifiers have to be unique. No multiple use of the same identifier for a type and a variable at the same time.

Programming plain G using the built-in runtime library functions can solve most problems, but for special cases it is even possible to use any external DLL-functions.

3 Basics

As mentioned before, the GSEOS language is used for many different tasks. Because of this, the configuration is divided in different parts.

- The main data types, external data block formats, functions and decoders are defined in the compiler main file (its default name is `main.g`, see section 13.1.1 for further information). This file may include several other files.

- The code for the interface to QLook is stored inside the QLook configuration files (see Data Display Language Description, IDA-GSEOS-0005).

- Each batch program is stored as a particular file inside the batch file directory. Batch files are compiled and executed on demand.

At startup the compiler main file will be processed to define the data blocks from external sources and to create decoders that will work on these blocks. If the initialization file contains a function called `void _main ()`, this function will be called automatically at startup. It should be used for all user defined initialization purposes. The initialization file should contain all functions that provide globally used functionality. Since all symbols once defined stay resistant inside the symbol table as long as GSEOS runs, you may call these functions later from any other file or from inside QLook.

The counterpart of the initialization function is `void _exit ()`. If this function is defined, it will be executed on quitting GSEOS. This can be used for clean-up or other purposes.

3.1 Notation of Numbers

There are four different ways to specify constant integer values:

- Decimal, the standard, consisting of digits from `0` – `9`.
- Hexadecimal, specified by leading `0x` (zero x) followed by a string of `0` – `9`, `a` – `f` or `A` – `F`.
- Octal, specified by a leading `0` (zero) followed by a string of `0` – `7`.
- Dual, specified by leading `0b` (zero b) followed by a string of `0` and `1`.

The decimal number 29 for example may be represented by `29`, `0x1d`, `035` or `0b1101`. These notations are all equivalent.

3.2 Symbols Scope

Of course, you do not have to work with fix values, but you can use abstract symbols. These symbols can be used anywhere inside that scope in which they have been defined (or in a sub-scope of it). Symbol definitions are not valid outside the scope. It is not allowed to define a symbol multiple times inside the same scope, but you can overwrite the definition of a symbol inside of a sub-scope. Such a redefinition does not affect the definition of the embedding scope but only inside the sub-scope. After leaving the sub-scope, the former definition of the symbol is restored and can be used as before.

Whenever a symbol is used, the compiler looks for its current definition. It starts in the current scope. As long as the symbol is not found, the search will be continued in the next upper scope. A scope is enclosed by braces like that:

{ ***scope*** **}**

A scope may contain as many sub-scopes as desired, for instance:

{ ***scope*** **{** ***sub-scope1*** **}** **{** **{** ***sub-sub-scope*** **}** ***sub-scope2*** **}** **}**

❖ **Example**

```
{
  char A;         // define A to be an character variable
  A = 0;          // set the value of A to zero
  {               // enter a sub-scope
    long B;       // define B to be an long integer variable
    A = 2;        // set the symbol A (char) from the outer scope to 2
    B = A;        // assign A's content to the long integer B
  }               // leave the scope, A contains 2, B is lost.
  {               // enter a new sub-scope
    long A;       // create a new symbol A. The A from the outer scope is not
                  // accessible inside this scope any more
    // A = B;     // since B is neither defined inside this sub-scope, nor inside
                  // the outer scope, it is not accessible here.
    A = 12345;
  }               // leave this scope, the new definition and value of A is lost.
  A = A + 1;      // increment the character variable A. It now contains the
                  // value 3 (remember the change from 0 to 2 inside the first
                  // sub-scope!)
}
```

3.3 Conditional Execution

The syntax of conditional execution of program sequences is similar to C. The only exception is, that there are no **goto**s implemented in the GSEOS language.

3.3.1 if then else

if **(** ***expression*** **)** ***compound_statement*** **[** **else** ***compound_statement*** **]**

This statement is equivalent to the ANSI C statement. If the value of ***expression*** is non-zero (or **true**), the first ***compound_statement*** is executed. Otherwise (if ***expression*** is zero resp. **false**) the optional **else**-branch is executed. In nested ***if***-statements the first ***else*** is assumed to belong to the last ***if***.

3.3.2 .. ? .. : ..

The result of an expression like this

bool_expression **?** ***yes_expression*** **:** ***no_expression***

is ***yes_expression***, if the ***bool_expression*** is true (or non-zero, anyway) and ***no_expression*** otherwise. Only the resulting expression will be evaluated. The ***a ? b : c*** expression cannot be used as destination of an assignment or as non-constant reference parameter.

```
long lMax, l1, l2;
...
lMax = (l1 > l2) ? l1 : l2;   // this assigns the maximum of l1 and l2 to lMax
```

The type of the result of this expressions is the smallest common type of both possible results, if both are simple types. If the ***yes_expression*** and the ***no_expression*** are more complex, they have to be exactly of the same type. If necessary, use explicit type conversions to get fitting types.

```
unsigned char auc[4];
bool          b;
...
auc = b ? "yes" : "no\0";   // both arrays have to be of the same size!
```

3.3.3 switch - case

`switch` **(** ***switch_expr*** **)** **{** **(** **`case`** ***case_expr*** **`:`** ***statement_list*** **)*** **[** **`default`** **`:`** ***statement_list*** **]** **}**

This control flow statement is similar to the C switch-command (but not 100% equivalent).

case_expr **:=** ***expression*** **|** **(** ***low_limit*** **`..`** ***high_limit*** **)** **(** **`,`** ***case_expr*** **)***

low_limit **:=** ***high_limit*** **:=** ***expression***

The values of the ***case_expr***s are successively evaluated and compared to the value of the ***switch_expr*** (which is evaluated only once, on entering the switch statement). The first fitting **`case`**'s ***statement_list*** will be executed. Opposite to C, the ***expression***s may be variables as well and the ***switch_expr*** does not have to be of a simple type.

statement_list may contain any code as well as the **`break`** command, which causes the machine immediately to continue the execution after the **`switch`** statement. If the **`break`** command is omitted, the execution will run through all following **`case`** ***statement_list***s (without evaluating the ***case_expr***) until a **`break`** or **`return`** command is reached.

The ***case_expr***s may be any constant or variable value as well as a range, defined by **`..`** between the lower and the upper limit (in this order!):

`case` **`2`** **`..`** **`5`** **`:`** ***statement_list***

This executes the ***statement_list*** on 2, 3, 4 and 5. You may also define comma-separated lists of constant or variable values or ranges as ***expression*** like this:

`case` **`2`** **`..`** **`5`** **`,`** **`7`** **`:`** ***statement_list***

This executes the ***statement_list*** on if the ***switch_expr*** is equal to 2, 3, 4, 5 or 7.

Of course it is possible to use the normal C-like syntax for specifying lists instead of using commas:

`case` **`2`** **`..`** **`5`** **`:`**
`case` **`7`** **`:`** ***statement_list***

These two lines have the same effect as the line above.

The first case that fits the switch ***expression*** will be executed, regardless of any other fitting case expression following. Therefore the **`default`** statement has to be the last in the list, because it fits any value and would render all further **`case`**s useless.

3.3.4 for

`for` **`(`** **[** ***init*** **]** **`;`** ***condition*** **`;`** **[** ***modify_expression*** **]** **`)`** ***statement***

The ***init*** expression normally initializes the counter variable before the first pass. The counter variable is checked inside the ***condition*** expression at the beginning of each pass. The ***statement*** inside the loop will be repeatedly executed, as long as the ***condition***-expression is true. The optional third ***modify_expression*** modifies the counter variable at the end of each pass.

statement may contain any code as well as the **`break`** and the **`continue`** commands. The **`break`** command causes the machine to leave the complete **`for`** loop immediately and continue with the next command. In contrast to that, the **`continue`** command causes the machine to leave the ***statement*** of the current pass only and continue with the execution of the loop at the ***modify_expression*** and ***condition***.

3.3.5 while

`while` **`(`** ***condition*** **`)`** ***statement***

The ***condition*** is evaluated at the beginning of each pass. The loop ***statement*** will be repeatedly executed as long as the ***condition*** is true (i.e. non-zero).

statement may contain any code as well as the **`break`** and the **`continue`** commands. The **`break`** command causes the machine to leave the complete **`while`** loop immediately and continue with the next command. The **`continue`** command causes the machine to leave the ***statement*** of the current pass and continue with the execution of the ***condition***.

3.3.6 do while

`do` ***statement*** **`while`** **`(`** ***condition*** **`)`**

The loop will be executed while ***condition*** is true. Since condition is evaluated at the end of each pass, ***statement*** will be executed at least one time.

The **`break`** and **`continue`** commands work as described before (see 3.3.5).

3.4 Functions

You can build functions that provide actions and calculations that have to be done by multiple program parts. This allows using the same code from different parts without duplicating. It is possible to deliver variables into the functions and receive results from it, so the code of the function itself can be hold quite versatile.

resulttype functionname `(` ***parameters*** `)` `{` ***body*** `}`

The function can be called using the ***functionname*** from any other code. The ***body*** contains all desired actions and calculations and can access the ***parameters*** provided by the calling code.The function can return a result of the type ***resulttype*** that can be used in further operations of the calling code. The calling code can use the function call like any other value of the type ***resulttype***.

It is even possible to call the function's code recursively. Functions will be subject of section 9.3.

❖ **Example**

The fibonacci numbers are recursively defined as sum of the two predecessors:

```
fib(0) := fib(1) := 1
fib(x) := fib(x-1) + fib(x-2)
```

This definition can be put into a simple function:

```
long fib (long x)
{
  if (x < 2)
    return 1;                        // let the result be 1
  else
    return (fib (x-1) + fib (x-2));  // return the sum as result of fib (x)
}
...
{
  long Value;

  Value = fib (12);     //  assign the fibonacci number of 12 to Value
}
```

3.5 Time- and Event-Controlled Execution

Some tasks (like decoding data or updating the display) may have to be executed whenever a new data block arrives. To realize a function that is started always when a new data block of a certain type arrives, you may define decoder functions (see 9.1):

`decode on` `(` ***blocks_that_should_trigger_execution*** `)` `{` ***actions*** `}`

The second task you probably want to be started automatically on arriving data is the update of displayed QLook items. In most cases, you won't have to worry about that, because it is default that the arrival of any block mentioned inside triggers the updating. But it is also possible for the user to disable this mechanism and specify explicitly, which blocks should trigger the update (see 9.2).

A third field for time- or event-controlled execution is waiting for a certain time (**`sleep`**, see 10.2) or for the arrival of a certain data block (**`wait`**, see 10.4).

4 Data Types

Like C, the GSEOS language operates with variables of previously (user-) defined types. The type of a variable has to be declared explicitly before the first use (see section 3.2 for more information about the scope of type and variable definitions). Unlike C, you may assign variables of different types, as long as they only differ in the type *names*. Types that consist completely of elements of the same type at the same relative positions are accepted as equal.

Different types of the same length may be assigned, but cause warnings.

There are two different types of data types in the GSEOS language. The first type is used for internal calculations with local variables and is almost identical to C data types. The second one is used for handling external data blocks. The second type is based on the first type, but has some extensions for automated complex data handling.

The GSEOS language does not allow forward declaration of data structures.

4.1 Data Types for Local Variables

GSEOS data types are very similar to C data types with only one major exception: the GSEOS language contains no pointers. For creating function calls with returned parameters references of variables are used (similar to C++). In addition to that there are some extensions of standard C, like the new data type bitmap.

Data types for local variables may contain extensions that refine the type definition for external data block types (see 4.2 for further information), but these extensions will take no effect in declarations of local variables. This allows using the same base type for generating local variables as well as external data blocks.

The GSEOS language does not allow explicit type conversions of complex data types. Only simple types or arrays of simple types may be converted explicitly by the user to prevent compiler warnings. If variables of different types are mixed in an expression, the smaller type will be converted implicitly to the larger type to prevent information loss. Variables of different types with the same size may be assigned to each other but will cause warnings.

The following description will explain the main syntax of definitions.

4.1.1 Simple Data Types

Type	Description
[`signed` **|** `unsigned` **] (** `char` **|** `short` **|** `long` **)**	8, 16 or 32 bit values. If **[** `signed` **|** `unsigned` **]** is omitted, the default `signed` is used.
`double`	64 bit precision floating point value

`bool`	1 bit value (true == 1, false == 0)
`void`	Placeholder for “no value”.

The `void` type can be useful as type of a reference parameter, because this prevents type checking in calls to external functions. Although it is possible to assign anything to a void reference, it is not possible to assign a void type to any other variable. It is not even possible to do an explicit type conversion from void to any other type. That means, it is not possible to access data of void variables by using the GSEOS-language.

4.1.2 Enumeration

Enumeration types may be defined like in C:

`enum` *enumname* `{` *valuename* [`=` *constant*] (`,` *valuename* [`=` *constant* `]`)* `}`

The ***valuename***s have to be unique. They represent long integer values. If no ***constant*** is specified, the value will be set to the value of the previous ***valuename*** incremented by 1.

Enumeration variables or constants (like the ***valuename***s) may be assigned to long variables without problems, but assigning a long variable to an enumeration variable will cause warnings.

Please note that the increment and decrement operators do **not** evaluate the predecessor or successor of the current value but only increment or decrement its integer value by one.

❖ **Example**

```
enum Colors { BLACK, RED = 10, ORANGE, YELLOW, GREEN, BLUE, VIOLET, WHITE = 20 };
```

This defines the data type `Colors` and the constants `BLACK` (value 0), `RED` (defined to represent 10), `ORANGE` (11), `YELLOW` (12), `GREEN` (13), `BLUE` (14), `VIOLET` (15) and `WHITE` (20). You may use these constants instead of their values in your program whenever you want. Please note that the constants are no strings. Even when you use a enumeration constant (like `YELLOW`), the compiler

4.1.3 User Defined Types

The user may define types of specified names by using a statement like this:

`typedef` *basetype newtypename* (`[` *number_of_elements* `]`)*

This statement defines a new type named ***newtypename*** that matches the ***basetype***. The ***basetype*** may be any simple or more complex or even user-defined type. If the type definition is followed by one or more **`[` *number_of_elements* `]`**, the new type becomes an array of ***basetype*** elements with the given dimensions.

4.1.4 Arrays

Array types can be defined like in C by a variable declaration followed by one or more pairs of square brackets defining the dimension. To define an array type you have to use **`typedef`**:

`typedef` ***basetype newtypename*** **(`[` *number_of_elements* `]`)+**

This creates a new data type named ***newtype***, that is an array of ***basetype*** elements. For more information about arrays see section 5.7.

❖ **Example**

```
typedef unsigned char textlinetype [80];
typedef textlinetype  textscreentype [25];
```

The first line defines a type named `textlinetype` that consists of 80 unsigned characters. This type can be used to represent a line of a conventional text-terminal. The second line defines a type `textscreentype` that consists of 25 of those textlines. If you do not need the definition of a single textline, you may specify the `textscreentype` with the following line instead:

```
typedef unsigned char textscreentype [80][25];
```

4.1.5 Structs and Unions

Structures are bundles of variables that will be placed one after another in the memory. Unions are bundles of variables that will be placed at the same start position in the memory. The definition of structures and unions is identical to C syntax:

(`struct` | `union`) [***typename*** **] `{` (*vardefinition* `;`)+ `}`**

vardefinition may be any definition of variables. It may be a structure or union itself. In this case, omitting the name of ***vardefinition*** will put its elements flat into the hierarchy.

If you want to use the data structure multiple times, you should define an own type. You can do this either using the keyword **`typedef`** (see 4.1.3) or specifying ***typename***.

❖ **Example**

```
struct TwoChars   // definition of the type TwoChars
{
  char c1;
  char c2;        // the address of c1 is (address of c1)+1
};
```

This is equivalent to

```
typedef struct  // definition of the type TwoChars
{
  char c1;
  char c2;
} TwoChars;
```

The definition of a union may look like that:

```
union DemoUnion
{
  struct
  {
```

```
    char c1;
    char c2;
  };
  short s;    // c1 is the low byte of s1
};
```

You will find examples for even more complex structures in section 6.4.

4.1.6 Bitmaps

The bitmap type has been added for displaying pictures with QLook. It allows specifying two-dimensional arrays of pixel.

`bitmap` ***depth*** **`[`** ***width*** **`,`** ***height*** **`]`**

Allowed values for the color depth ***depth*** are 1 (black and white), 4, 8 (color table), 16, 24 and 32 (RGB). ***width*** and ***height*** specify the dimension of the bitmap in pixel. Please note that the order of width and height are composite to the order of standard C array elements. The two dimensions are used only for specifying display properties. The internal representation of a bitmap is a one-dimensional array whose elements can be accessed like any other one-dimensional array elements. See section 6.6 for further information of accessing elements of bitmaps with G.

4.1.7 "Constant" Variables

Variables and parameters can be declared to be constant values by using the keyword **`const`**. Constant variables cannot be altered. The keyword **`const`** is important if you want to allow constant values as reference parameters. The declaration of constant parameters may look like this:

```
void Something (unsigned char& auc1[], const unsigned char& auc2[]);
```

This line declares the parameter `auc2` of the function `Something` to be constant. That means, it is not allowed to use any write access to `auc2` inside the function.

4.2 External Data Block Types

Since GSEOS is supposed to be used for testing external hardware, it provides easy access to data sent by this hardware. The external data format may be quite complex and can be defined bit-precisely. It is also possible to specify the endianness of the data. Accesses to this data will be converted automatically to fit the internal data format of GSEOS without expensive user defined manipulations. This section describes the extensions that are used for the bit-precisely definition of data blocks.

4.2.1 Terminology

4.2.1.1 Data Blocks

Data received by GSEOS is stored in memory areas called data blocks. The incoming blocks are queued and distributed by the BDM.

A data block is a container for one or more values, packed together in a more or less complex data structure (described before). It is not possible to do calculations with a complete data block. For calculations you have to specify an element included in this container.

In GSEOS language, data blocks can be seen as top level of a structure. You cannot do calculations with a whole data structure but have to specify a single element of it. Due to this similarity, data block types are based on the data types for local variables but contain some extensions that allow automated data manipulation like conversion from big-endian to little-endian or placing data bit-precisely into memory. These extensions may be used for local data type definitions too, but will be ignored.

4.2.1.2 Data Block Elements

The elements of the data block container appear as parts of a structure and are called block elements. Whenever you specify a block element for read access, data will be extracted from the data block. When specifying a block element for write access, the data will be inserted to the data block. Extraction and insertion is done automatically and obeys the type extensions described in the following sections.

4.2.1.3 Data Block Handles

The variables that are used to access the memory occupied by data blocks are called data block handles. Although there might be just one single data block stored in memory, it is possible to have multiple data block handles referring to the same data block.

4.2.2 Endianness

The byte order of data blocks may be little-endian (used by Intel) or big-endian (Motorola). Since GSEOS runs on an Intel machine, data is stored in little-endian format by default. For best performance all internal calculations are done with the endianness of the machine.

Whilst all local variables are stored in little-endian format, blocks from external sources may be in big-endian format. To handle these different data formats it is possible to specify the format of data blocks. All accesses to data blocks will obey the endianness definition and convert the data automatically. To specify the endianness, the desired endian has to be specified directly before the type:

[`little-endian` **|** `big-endian` **]** ***type varname***

(For compatibility reasons it is also possible to use the deprecated keywords `intel` and `motorola` instead)

type may be any type definition. If type itself is a structure or union, the endianness of all of its sub-elements will be inherited from the parent structure, unless specified explicitly. If the endianness of sub-elements has been specified before, it will not be altered. If no endian is specified, it will be assumed to be little-endian unless the endianness of the containing symbol will be specified explicitly.

Remember that endianness specifications only take effect on block variables. All other local variables and function parameters ignore any endianness specification!

❖ **Example**

Now we add some endianness specifications to our data types. The modified types still can be used to declare local variables. Endian specifications will be **ignored** by declarations of **local variables**.

```
Bit struct TwoEndianedChars   // definition of the type TwoEndianedChars
{
  char c1;                // endianness of c1 depends on endianness specified later
  little-endian char c2;  // endianness of c2 is set to little-endian and will not
                          // be altered
                  // since a single byte has no byte order, you may specify
                  // any endianness you like – it just makes no difference...
};

big-endian union  // definition of the type EndianUnion with big-endian default
EndianUnion
{
  TwoEndianedChars; // var name omitted, creating EndianUnion.c1 (big-endian)
                    // and EndianUnion.c2 (little-endian)
  short s;          // creating EndianUnion.s (big-endian)
  little-endian struct  // creating EndianUnion.TwoUnsigned (little-endian default)
  {
    unsigned long ul1;              // EndianUnion.TwoUnsigned.ul1 (little-endian)
    big-endian unsigned long ul2;   // EndianUnion.TwoUnsigned.ul2 (big-endian)
  } TwoUnsigned;
};
```

4.2.3 Bit-precise Memory Access

Since the data format of external equipment often does not fit the normal 8-/16-/32-bit sizes, GSEOS offers the feature to define data types bit precisely.

simpletype* [(*bitsize* [, *gap*])] *varname

simpletype may be any build-in type (see simple data types, 4.1.1). ***bitsize*** specifies the number of bits used for representing the value in the external data block. ***gap*** specifies the number of unused bits that follow the value. The default value of ***bitsize*** depends on the size of ***simpletype*** (8 bits, 16 bits, 32 bits or 64 bits for char, short, long or double), the default value of ***gap*** is zero.

Opposite to integer formats, where the bitsize can be specified freely, there is a very limited number of sizes available for floating point values, as shown in the following table:

Type	Allowed Bitsize	Comment
char	1 - 8	
short	1 - 16	
long	1 – 32	
double	32	IEEE 754 32 bit floating point
	64	IEEE 754 64 bit floating point
	4	MIL-STD-1750a 32 bit float bit floating point (4 bytes)
	6	MIL-STD-1750a 48 bit floating point (6 bytes)

As you may see, the "bitsize" of floating point values is the actual bitsize only for IEEE-formatted floating point values. Opposite to this, all MIL-formatted floating point values are specified by using their size in bytes as "bitsize".

Bit-precise definitions as well as MIL formatting are followed in declarations of external data blocks only. If types with specifications of bit-precise memory placement are used to declare local variables, these specifications are ignored.

The format of bit-precisely stored values is like follows:

	Little-endian (Intel)	Big-endian (Motorola)
Byte 0	...LSB, predecessor	predecessor, MSB...
Byte 1	successor, ***gap***, MSB...	...LSB, ***gap***, successor

In little-endian format, the byte at the lowest address contains at its low bits the rest of the previous value and continues with the least significant bit of the current value. The significance of the bits increases with the address of the bits. After the most significant bit of the value is reached, a number of bits specified by gap is untouched before the value of the successor begins with its least significant bit at the high bytes of the byte at the higher address.

In big-endian format, the byte at the lowest address contains at its high bits the rest of the previous value and continues at the lower bits with the most significant bit of the current value. The significance of the bits decreases until the least significant bit is reached.

Since the position of the valid bits differs from little- to big-endian format, it is not possible to mix up these formats, if the values would have to share a byte.

❖ **Example**

```
struct Precise   // definition of the type Precise (length 41 bits)
{
  long(30)  l;  // l starting at bit 0 with length 30 bits
  char(8,3) c;  // c starting at bit 30 with length 8 valid bits + 3 gap
};

struct PreciseUnion
{
  long (24,12) l[10];  // Bit0-23 is l[0], Bit36-59 is l[1] etc.
  struct
  {
    long (0,24) dummy;    // Bit0-23 unused – it is already used by l[0]
    short (12,24) s[10];  // Bit24-35 is s[0], Bit60-71 is s[1] etc.
  };
} BitPreciseUnion;
```

4.2.4 Block Type Definition

The extensions on standard-C-like definitions only have effect on accessing data blocks. To define a data block you have to define a data block type first. Any declaration using a data block type defines a data block handle variable with endianness and bit-precise memory placement, any declaration using any other type defines local variables and ignores these extensions. Block type

definitions must be globally. Since the BDM has to know all block types at startup, defining block types has to be done inside the initialization-file.

`blockdef` ***type blocktypename blockvarname***

`xblockdef` ***type blocktypename blockvarname***

The keyword **`blockdef`** causes the compiler to create a data block type named ***blocktypename*** of the type ***type***. Simultaneous the global read-only data block handle ***blockvarname*** is declared. All uses of ***blockvarname*** will access the current block of the type ***blocktypename*** (currently processed by the BDM).

The keyword **`xblockdef`** has the same effect, but creates an additional entry for the previous BDM block (processed directly before the current block). The previous block is accessible by placing the keyword **`prev.`** directly before the ***blockvarname***. To save memory space you should use this feature only if you really need direct access to the previous block.

A third variation of block definition is the following:

`global` ***type*** **[** ***blocktypename*** **]** ***blockvarname*** **(** **`,`** ***blockvarname*** **)***

This line defines global static data blocks that may be changed from any code. See 5.5.2 for further information. The ***blocktypename*** may be omitted, if you do not need the block type in your program.

It is not possible to use block types inside of structures. Nevertheless you may create arrays of data blocks.

❖ **Example**

```
xblockdef EndianUnion UnionBlock CurrentBlock;
blockdef EndianUnion CreatedBlock DecodedBlock;
```

This first line specifies the data block type UnionBlock as a data structure of the type `EndianUnion`. The currently processed Block of this type is accessible via the globally defined read-only variable `CurrentBlock`. The second line defines the data block type `CreatedBlock` as a data structure of the type `EndianedUnion`, too. Blocks of the type `UnionBlock` will become accessible through the global block handle `CurrentBlock` if sent to the BDM, blocks of the type `CreatedBlock` through `DecodedBlock`. Although both data block types represent exactly the same data structure, this definition of two data block types allows distinguishing between block types and the actions that are triggered by them.

Please note that the BDM does not allow accessing previously processed blocks of the type `CreatedBlock`, since only the previous values of blocks defined with **`xblockdef`** are held by the BDM.

For instance, let an external hardware send data blocks of the type `UnionBlock` to the BDM. These blocks should trigger a decoder that generates a new block, just by decrementing all elements of the incoming block, and send the new block for further calculations to the BDM. Obviously the data structures of the incoming block is the same as the structure of the block sent by the decoder. But of

course the block sent by the decoder has to be distinguished from the incoming block, because should not be processed by the decoder once again. Therefor the type of the block created by the decoder has to differ from the type of the incoming block: Whilst the incoming block type is `UnionBlock`, the decoder should create a new block of the type `DecodedBlock`.

5 Variable Declaration

The GSEOS language contains a C-like variable declaration. The only major exception is the absence of pointers, because these variables could easily cause program crashes or unexpected side effects. Use of global variables is limited to data block handles. Please remember that using global variables may cause unexpected side effects, especially in multithreading environments like GSEOS.

5.1 Local Variables

Local variables may be declared like this:

type variablename* [= *constant*] (, *variablename* [= *constant*])

The ***type*** may be any of the data types mentioned in the previous chapter (4, Data Types).

The ***constant*** is used to define an initial value for the declared variable. Static variables will be set to this value only once (before the first execution). If the initial value is omitted, the variable will not be initialized and may contain random values (static variables will be initialized with 0, however)

Normally, local variables are located on the local stack, which allows fast accesses for calculations. Unfortunately, the size of the local stack is limited. To prevent early stack-overflows, large variables will be stored automatically in dynamically allocated memory. Remember that this causes (internally) a little bit more overhead on accesses to elements of large variables. You can adjust the stack size and the threshold value with the compiler control panel, described in section 13.1.2.

5.2 Global Variables

Global variables are declared in the main scope like local variables in local scopes, but cannot be initialized with a special value immediately:

type variablename* (, *variablename*)

All global variables are initialized with zero. Block handles are marked as invalid.

Please remember that using global variables may cause unexpected results in multithreading environments like GSEOS.

5.3 Extern Variables

This feature is used to access memory provided by the hardware driver directly without copying it into data block memory.

`extern` ***type variablename* (, *variablename*)***

The ***variablename*** has to fit the identification of memory that is mapped by a driver. The size of the ***type*** should fit the size of the mapped memory area. Nevertheless, the ***type*** may be smaller, but it must not be larger.

5.4 Local Data Block Handles

As the GSEOS language allows specification of external data blocks, it contains extensions to declare variables that access these data blocks. To avoid massive memory duplication, data blocks are not copied. Instead of that all data block variables are references (called block handles) to the memory containing the actual data block. If there are several block handles at once, all of them might access the same memory. As long as there is at least one block handle referring to a certain data block memory, this memory will be hold. When the last handle is deleted, the memory itself will be freed automatically. The GSEOS user does not have to pay attention to this memory management.

To avoid manipulation of multiple used data, there are two different types of data block variables: read-only data block handles on the one hand, and work data block handles on the other hand.

Block handles may be global variables (like the variables generated by `blockdef`, see 4.2.4), but of course it is possible to generate local data block handles, too.

5.4.1 Work Data Blocks

Work data blocks are represented by block handles the GSEOS user can freely work with. That means the user may modify single elements of the block.

When a work data block handle is declared, a new data block of the specified type will be created. The elements of the block are not initialized. The declared handle is the only reference to the memory used by the actual data block. You can modify any element of the block by using the handle. If another block is assigned to the handle, the complete data block memory will be copied.

The default for data block variables is to be a work data block handle. You may declare a work data block handle as follows:

`work` ***blocktype blockvarname*** **[** **`=`** ***initialCharacter*** **] (`,` *blockvarname* [`=` *initialByte*])***

or simply

blocktypename blockvarname **[** **`=`** ***initialCharacter*** **] (`,` *blockvarname* [`=` *initialByte*])***

Both declarations are equivalent.

The memory of a work data block may be byte filled with a user-specified value by assigning the constant char value to the whole block (using the block handle like that: ***handle*** **`=`** ***constant_char***). This byte filling does not take care of bit-precise placement.

❖ **Example**

To create a local work block handle, you can use the following declarations (see **Error! Reference source not found.** and 4.2.4 for the definition of the type `UnionBlock`):

```
...
{
  UnionBlock  uBlk1;         // create an un-initialized block
  UnionBlock  uBlk2 = 0;     // create an block with all bits initialized to 0
  UnionBlock  uBlk3 = 0xff;  // create an block with all bits initialized to 1

  uBlk1.TwoUnsigned.ul1 = 0x12345678;  // although the block item has not been
                                       // initialized, the block handle is valid.
  ...
}
```

5.4.2 Read-only Data Blocks

If you do not need to modify a data block but want to preserve the data for later use, you should use a read-only data block handle. You cannot modify the data referenced by a read-only data block handle, but you can change the reference itself by assigning another block handle to the read-only data block handle.

The declaration of a read-only data block handles is very similar to the declaration of work handles, but they require the keyword **`readonly`**. Thy syntax is as follows:

`readonly` ***blocktype blockvarname*** **[** **`=`** ***initialCharacter*** **] (`,` *blockvarname* [`=` *initialByte*])***

The handle will be initialized as invalid. You have to assign a valid block handle to it before using the read-only block handle.

It is not possible to modify parts of a read-only block. You may assign complete blocks to read-only block handles. Since read-only data block handles are not copies of the source but only references to the same memory, modifying the source will change the read-only also.

Since modification of data blocks that are managed by the BDM is not allowed, it is prohibited to send read-only data blocks (otherwise it would be possible to change a still existing work block that uses the same memory as the already sent block). For further information see 10.1, Special Commands: Send.

❖ **Example**

To create a local readonly block handle, you can use the following declaration (see **Error! Reference source not found.** and 4.2.4 for the definition of the type `UnionBlock`):

```
...
{
  readonly UnionBlock  uBlk; // create an un-initialized readonly block handle

  uBlk = CurrentBlock;       // link the local handle to the current BDM block
  if (uBlk.TwoUnsigned.ul1 < 0x12345678) // now the handle uBlk is valid and items
                                         // can be accessed (read only, of course)
    { ... }
  else
    { ... }
  ...
```

```
}
```

5.5 Global Data Block Handles

There are two different kinds of global data block handles: The first kind of data blocks is queued by the BDM and can be used to trigger operations. The other kind is not queued and cannot be used as trigger.

5.5.1 Queued Data Blocks

All global data blocks declared by **`blockdef`** or **`xblockdef`** (see 4.2.4) are queued data blocks and read-only block handles. Only the global block handles of queued data blocks can be used for triggering. You are not allowed to assign anything to these global data blocks. This behavior ensures that all decoders triggered on the same data block process the same data.

Global data block handles always refer to the currently processed block (the block at the top of the BDM queue). Updating is done automatically after all decoders that are triggered on this block have processed it completely and a new block arrives.

If the global block has been defined using **`xblockdef`**, adding the prefix **`prev.`** to the global data block variable name allows accessing the previously processed block. Earlier blocks are not accessible. If you need to access older blocks, you have to create a copy of it for your own in time. To save memory and performance, you should hold your copy in a local read-only data block handle. Since you might want to use your copy in a later run, the copy should be stored inside a static handle (see 5.6).

Each access to a global data block causes a look-up for a new block in the BDM. Especially in complex loops it may be possible to speed up the code by creating a local read-only copy of the global block handle first and operate with this copy inside the loop.

5.5.2 Non-queued Data Blocks

Global data blocks declared by using the keyword **`global`** (see 4.2.4) can be accessed for read and write at any time from any code. All changes of a global date block will take effect immediately to all users of this data block. Please note that using this kind of data blocks may lead to unexpected results, since multiple processes may access the same memory simultaneously.

It is not possible to use this kind of data blocks as trigger (with **`decode`** **`on`**, **`on`**, **`wait`**) and it is not allowed to use the queuing commands **`send`** and **`sendcopy`**, since there is no queue for non-queued data blocks.

Please note that it is not guaranteed that the data set in non-queued data blocks is processed completely by all decoders, since it may be altered before all decoders have finished processing. If you want to ensure the complete processing, you have to use queued data blocks.

5.6 Static Variables

Variables may be declared static by using the keyword **`static`**. While 'normal' variables do not hold their values when the scope is left, static variables remain in memory and do not alter the stored value between leaving and re-entering the scope. This behavior is similar to C.

Since accessing static variables is slower than accessing internal variables you should use static variables only if necessary.

- Static non-block variables will be initialized with zero, unless another constant value is specified inside the declaration. The initialization of the static variable is done only once (at compile-time).
- Static read-only-blocks will be initialized as invalid. It is not possible to access this variable before another block has been assigned to it.
- Static work-blocks will be initialized as valid. The elements of the block will not be initialized and may contain random values.

5.7 Arrays

Arrays may be declared like follows:

type arrayname **(** **`[`** ***number_of_elements*** **`]`** **)+**

As you see, it is possible to declare one-, two- or any other multidimensional array variable just by specifying a sequence of desired number of elements.

```
unsigned char asz[20][80];
```

This line specifies a two-dimensional array of characters. It may be seen as a list of 20 lines with 80 columns each. Using `asz[5]` would specify the whole sixth line.

number_of_elements may be omitted in special cases:

- Inside of function headers to allow passing arrays of variable length to functions. If the number of elements is not specified, the parameter has to be declared as reference (using **`&`**). Runtime range checking will prevent illegal memory accesses.
- Inside of unions. The size of the array will be set to the maximum size that fits into the union without changing the union's size. Runtime range checking is not affected; it is not possible to access memory outside the union. The actual size of the array is known at compile time, so range checks are done at compile time already.
- If you specify a default value for the array variable, you may omit the number of elements of the outermost dimension, since it is given by the size of the default value. The actual size of the array is known at compile time, so no runtime checks are needed.

Note that only the size of the first dimension may be omitted, anyway. So the syntax of the declaration of arrays with unknown size at compile time is one of the following:

type varname **`[]`** **(** **`[`** ***length*** **`]`** **)***

type arrayname **[ ]** **([** ***number_of_elements*** **])*** **=** ***default_value*** **;**

For instance, you may specify an array of three strings like this:

```
unsigned char asz[][80] = {“first”, “second”, “third” };
```

This is equivalent to the following declaration:

```
unsigned char asz[3][80] = {“first”, “second”, “third” };
```

Please note that the strings `“first”`, `“second”` and `“third”` are assumed to be arrays with 80 unsigned chars each. Missing characters of strings are padded automatically, but all other types of arrays have to fit exactly for assignments.

6 Accessing Data

Generally, all accesses to local or block memory are handled identically. It makes no difference for the user to use local variables or external data blocks. Basically, accessing variables is very similar to C, with a few exceptions that will be described in this chapter

6.1 "Standard" Variables

The standard variables (local, global or external variables that are no data block handles) can be accessed like in C.

Although local variables use memory in the machine stack, global variables use own memory and external variables use memory mapped by the hardware driver, the user does not have to pay attention for it – the syntax of all accesses is identically.

6.2 "Constant" Variables

Variables that are declared to be constant (**`const`**) can be used only for read accesses. It is not possible to modify their contents (write access). Constant variables will probably be in most cases parameters of a function that may not be modified in any way inside the function.

```
void Something (unsigned char& auc1[], const unsigned char& auc2[])
{
  auc1[0] = '1';     // allowed, since auc1 is not const
  // auc2[0] = '2';  // error, auc2 is constant and must not be altered.
  auc1[1] = auc2[2]; // ok, auc1 may be altered and auc2 may be read
}

void Anything ()
{
  unsigned char auc[100];
  const unsigned char cauc[] = "constant";  // assignment in definition is ok

  // Something ("forbidden", "allowed");  // since the first parameter is not
                                          // declared to be constant, it is not
                                          // allowed to use a constant here.
  // Something (cauc, auc);     // same as before: first parameter must not be
                                // constant. The second parameter may be variable,
                                // but can not be altered inside the function
  Something (auc, auc);  // ok, but the second parameter is not initialized yet.
                         // Since both parameters are references, auc1 and auc2
                         // will access the same memory inside the function.
  Something (auc, cauc);

  // cauc[3] = 'a';      // not allowed, since cauc is constant!
}
```

6.3 References

References to variables are allowed only in function headers. All accesses to the referenced parameter inside the function will be accesses on the outside variable itself. A parameter is defined as reference by following the type by **`&`**:

type **`&`** ***parametername***

The ***type*** may be any simple or user defined type as well as a data block type. Note that **`&`** is not an address operator. Its use is allowed only in the parameter list of function declarations, not to build a reference return type. Generally, the source type and the destination (reference) type have to fit exactly.

6.3.1 Using Constant Values as Reference

You cannot assign constant values to reference-parameters as long as the parameter is not specified to be constant by using the keyword **`const`** in the function declaration. Parameters declared to be constants may not be altered inside the function (see 4.1.7).

6.3.2 Using Data Block Handles as Local Type Reference

Generally it is not allowed to use data block handles (see 4.2) as references to local memory, because the format of data blocks is quite different and can not be accessed like local memory.

There are only few exceptions: If the memory format of the data block element is exactly the same as the format of its equivalent in local memory (without any gaps or unusual sizes of simple types), it is possible to

- use any fitting data block (read only as well as work data block) element for constant references or
- a valid work block element for non-constant references

There is a third exception: Any data block element without gaps and fitting to byte borders may be used as void reference parameter.

6.3.3 Using Data Block Handles as Data Block Reference

If you use a data block type inside the parameter declaration, you should be aware that data block type reference parameters could only take data block handles.

It is not allowed to use a read-only data block handle for a work block reference parameter, since this would allow modification of the data block inside the function.

You may not use a work block handle as non-constant read-only data block reference parameter. Doing that would allow re-linking the work block handle from the outside to a read-only block handle. That seems to be all right so far, but after leaving the function it would be possible to modify this former read-only block like any other work block.

❖ **Example**

```
void Demo (long& l, readonly EndianUnion blk1, const readonly EndianUnion blk2)
{
  l = 0x12345678;     // this changes the value of the variable outside
  blk1 = blk2;        // re-linking the handle blk1 is allowed
  // blk1.s = 123;    // this would be an illegal access to the read-only handle
  // blk2 = blk1;     // any modification of constant handles is not allowed
}


...
```

```
{
  long                 lDummy = 0xcafe;
  UnionBlock           blkA;
  readonly UnionBlock  blkB;
  readonly UnionBlock  blkC;
  Demo (lDummy, blkB, blkA);   // ok
  // now lDummy contains the value 0x12345678, and blkB refers to blkA

  // Demo (lDummy, blkA, blkC);   // not allowed, because blkA is not readonly.
                                  // the following line would cause problems:
  blkA.s = 5;       // since blkA would refer to the readonly block blkC, this
                    // would allow a modification of blkC items!
}
```

6.4 Structs and Unions

Elements of complex data structures (struct and union) can be accessed like in ANSI C:

datastructure **.** ***elementname***

The ***datastructure*** may be an element of any other data structure (array, struct or union) itself. Since elements of unnamed substructures are put flat into the superior data structure, they are accessed like any other element of the superior structure (without an additional **.** that would indicate a deeper hierarchy).

You can copy whole data structures. If the data structure is local memory, the data is copied at once. Data structures that are part of a data block are copied element after element, because the bit-precise data placement (with gaps) inside of data blocks could cause unexpected errors in combination with byte based memory moves. As this element-wise copying of data structures should not cause any unexpected result, this behavior may result in strange effects on unions, because the rear elements will (possibly partly) overwrite the data first members. But since it is possible to place the elements of a union in a way that they do not overlap but are placed into each others gaps, this behavior has benefits, too.

Local structures and unions may be byte-filled with a constant value by assigning a constant char (no variables and no values that do not fit into 8 bits!) to the whole structure or union. Because of the bit-precise data placement inside of data blocks, it is not possible to use this (byte based) feature for parts of data blocks. This feature is supposed to be used for fast initialization of all bits to a defined state, so the constant will be probably in most cases be zero (no bit set) or 0xff (all bits set). The syntax is the following:

datastructure **=** ***constant_char_value***

❖ **Example**

(Note: The type `TwoChars` has been defined in section 4.1.5 already.)

```
union  // definition of the variable SimpleUnion
{
  TwoChars;    // var name omitted, creating SimpleUnion.c1 (offset 0) and
               // SimpleUnion.c2 (offset 1)
  short s;     // creating SimpleUnion.s (offset 0)
  struct       // creating SimpleUnion.TwoUnsigned (offset 0)
  {
    unsigned char uc1;   // SimpleUnion.TwoUnsigned.uc1 (offset 0)
```

```
    unsigned char uc2;   // SimpleUnion.TwoUnsigned.uc2 (offset 1)
  } TwoUnsigned;
  char  ac[];  // create an array of characters that is as large as possible to
               // fit inside this union without enlarging it. The resulting
               // array ac will consinst of 2 elements here.
} SimpleUnion;

SimpleUnion.TwoUnsigned.uc1 = SimpleUnion.c2;
```

You can also use a previously user-defined type:

```
union  SimpleUnionType // definition of the type SimpleUnionType
{
  TwoChars SubChars;   // var name given, creating XYZ.SubChars.c1
                       // (offset 0) and XYZ.SubChars.c2 (offset 1)
  short s;     // creating XYZ.s (offset 0)
  struct       // creating a substructure without name. (offset 0)
  {
    unsigned char uc1;   // XYZ.uc1 (offset 0)
    unsigned char uc2;   // XYZ.uc2 (offset 1)
  };
  char  ac[];  // create an array of characters that is as large as possible to
               // fit inside this union without enlarging it. The resulting
               // array ac will consinst of 2 elements here.
};
// since this is just a type definition, there are no sub-elements accessible.
// After declaring a variable of this type, you are able to access its elements
// by replacing the XYZ with the variables name.

SimpleUnionType  SecondSimpleUnion;  // declare a variable of this type

SecondSimpleUnion.uc1 = SecondSimpleUnion.SubChars.c2;
```

Please note that `SimpleUnion` and `SecondSimpleUnion` contain different unnamed substructures, but represent the same internal structure.

6.5 Arrays

The access on single elements of Arrays is just like handled in C:

array **[** ***index*** **]**

The ***array*** may be an element of any other data structure itself.

Arrays may be byte-filled with a constant value by assigning a constant char value to the whole array. It is not possible to use variables (even if they are declared to be constant) or non-char values as source for byte filling. Since data inside of data blocks may be not byte based, it is not possible to use this feature for parts of data blocks.

6.5.1 Accessing Ranges

You may access a subset of an array at once by specifying a range of elements. Since all specified elements has to build one block without gaps, the range definition is allowed only for the last specified dimension and may not be followed by any further element specification. There are two different ways to specify array ranges:

array_variable **[** ***first*** **..** ***last*** **]**

The ***first*** value has to be less than ***last***. This returns the elements of the array starting at index ***first*** and ending with index ***last***. The other way is to specify the starting element and the number of elements inside the range:

array_variable **[** ***first*** **,** ***size*** **]**

size specifies the number of elements and has to be a value equal or greater than zero, ***first*** may be a variable that contains the value for the index of the first element of the specified range.

It is possible to use ranges for copying a part of an array into another part of the same array, even if the source and destination areas overlap.

❖ **Example**

```
union DummyUnion  // size specified by the included structure: 7 bytes
{
  char ac[];      // length will be set to 7 elements  (7 bytes)
  long al[];      // length will be set to 1 element   (4 bytes)
  short as[];     // length will be set to 3 elements  (6 bytes)
  char aac[][3];  // will be set to aac[2][3] (6 bytes)

  struct   // size is 7 bytes
  {
    short s;
    long  l;
    char  c;
  };
};

void DummyFunction (char& aac[][3])   // a variable number of 3-character strings
{
  DummyUnion u;
  long i = 1;
  short as[8];  // array of 8 short elements, the resulting size is 16 bytes
  short as2[20];
  DummyUnion aaU[10][8];

  u.aac[i] = aac[4];  // accessing the parameter acc[4] may exceed the range!
                      // accessing u.aac[i] is range checked at runtime (access is
                      // limited to element 0 and 1)
  as = 0xAA;  // fill all 16 bytes of 'as' with value 0b10101010
  as2[10..17] = as;  // assignment to an 8 short long subset of as2

  aaU[3][3..5] = aaU[2][1..3];  // ok
  aaU[1..4] = aaU[5..8];        // ok
  aaU[1..4] = aaU[3,4];         // ok, the overlapped copy is handled correctly
  // aaU[3][3..5].s;  would cause an error, since it is not allowed to refine
  //                  specification (.s) after a range ([3..5])
  // aaU[1..2][2];    is not allowed for the same reason.
}
```

6.6 Bitmaps

The bitmap type will probably only be used for displaying purposes and not for manipulating data using code written in G, because of the large amount of data generally used by bitmaps. Nevertheless, it is possible to access parts of a bitmap. You should remember that the internal structure of bitmaps is a one-dimensional array. If you want to access a single pixel, you have to calculate its position in the one-dimensional array first.

bitmap type	internal representation
`bitmap 1`, `bitmap 4`	Array of bytes; since the bytes are filled with several pixels, the pixels have to be extracted by the user (take the complete byte containing the pixel and extract the desired bit/nibble). The Byte containing the desired pixel(***x***, ***y***) is accessible through ***picture*`[ (`*width* `* `*y* ` + `*x*`) / 8]` for one bit color depth or ***picture*`[ (`*width* `* `*y* ` + `*x*`) / 2 ]`
`bitmap 8`	Array of bytes. Each byte contains the color index of a single pixel. Pixel(***x***, ***y***) is accessible through ***picture*`[`*width* `* `*y* ` + `*x*`]`
`bitmap 16`	Array of shorts. Each short contains a single pixel. Pixel(***x***, ***y***) is accessible through ***picture*`[`*width* `* `*y* ` + `*x*`]`
`bitmap 24`	Array of bytes, one pixel consists of three subsequent bytes. Pixel(***x***, ***y***) starts at ***picture*`[ (`*width* `* `*y* ` + `*x*`) * 3 ]`
`bitmap 32`	Array of longs. Each long corresponds to exactly one pixel. Pixel(***x***, ***y***) is accessible through ***picture*`[`*width* `* `*y* ` + `*x*`]`

Although it is possible to access parts of a bitmap this way, it may be useful to build a union that provides an alternative two-dimensional array for the bitmap.

❖ **Example**

```
union
{
  bitmap 16 [640,480] bmp;          // format for displaying 480 lines and
                                    // 640 columns with 16 bit color depth
  short               hex[480][640]; // format for 2-dimensional access
} graphic;                          // declare a variable of this type
```

`graphic.bmp` is accessible like a one-dimensional array of unsigned short integer values. `graphic.bmp[0]` is the upper left corner of the bitmap (`graphic.hex[0][0]`), `graphic.bmp[639]` is the upper right corner (`graphic.hex[0][639]`) and `graphic.bmp[307199]` is the lower right corner (`graphic.hex[479][639]`).

6.7 Data Blocks

When accessing data from inside of data blocks, the user does not have to pay attention for the bit precise placement inside the block, since this is handled automatically. That means, all accesses to parts of a data block are handled exactly like accesses to local memory described before.

Nevertheless, there are some differences between using local variables and data block handles that will be described in this section.

Accessing external block data is much more expensive than accessing internal variables because all block data accesses are bit precise operations that need more overhead than byte based accesses. In addition to that, all block accesses are indirect accesses in contrast to local memory accesses. Therefore you should avoid multiple access to the same block element for internal calculations. Copy the desired element into a local variable and use this copy for calculations instead.

Of course you may use data block types for parameters, too. As long as you do not use references, the delivery of data blocks to functions is handled like any other assignment of blocks. If you want to deliver data block handles as reference parameter to a function, you should read sections 6.3.2 and 6.3.3.

6.7.1 Work Data Blocks

Elements of work data blocks can be used for read and write accesses. The complete block handle can be used for read and write accesses, too. Assignments to work data blocks copy the complete data, not only the reference. Since this is slower than just re-linking the reference to memory, you should use work blocks only if you really need it for modifying some values inside the data block. For storing data without modification you should use read-only data blocks.

The copying instead of re-linking prevent creation of more than one work block handle referring to identical memory. That means, concurring write accesses to memory cannot occur.

❖ **Example**

```
work UnionBlock blkWork1;
UnionBlock blkWork2;

blkWork2.TwoUnsigned.ul2 = 0x12345678;
blkWork1.TwoUnsigned.ul1 = blkWork2.TwoUnsigned.ul2;
```

This assignment converts a big-endian unsigned long value into a little-endian value and copies it into another block element.

6.7.2 Read-only Data Blocks

Elements of read-only data block allow only read accesses, so it is not possible to modify the data block referenced by the block handle. But it is possible to use the complete block handle in read accesses as well as in write accesses. Write access to a read-only block handle re-links the destination block handle to the memory referenced by the source handle. Since read-only data block handles are initially marked as invalid, you have to link it to a valid block by such an assignment before the first read access. If you assign a work data block to a read-only block, it is possible to modify the content of the memory linked by both block handles by using the work block handle.

Please note that read-only block handles are not constants! It is possible to link the handle to another block, as long as it is not declared explicitly to be constant. On the other hand all constant block handles are read-only, too.

It is prohibited to use a work block as non-constant read-only reference parameter, since this would allow linking the handle inside the function to a real read-only block (like queued blocks from the BDM) and modifying its data after returning from the function. Of course it is forbidden to use

constant block handles as non-constant reference parameter, since this would allow manipulation inside the function.

❖ **Example**

```
work     UnionBlock blkWork1;
readonly UnionBlock blkRead;
readonly UnionBlock blkHistory[10];   // array of 10 blocks.

blkRead = CurrentBlock;       // let blkRead refer to the current UnionBlock
blkHistory[4] = CurrentBlock; // let the fifth block of the array refer to it, too
blkWork1.s = blkRead.s;
```

The declaration creates an empty read-only data block handle. As long as there is no valid data block assigned to it, you cannot access `blkRead` for reading purposes.

6.7.3 Queued Data Blocks

The variables containing the queued data blocks are created during the block definition with **`blockdef`** and **`xblockdef`** (see 4.2.4). The last parameter specifies the variable name that can be used for accessing the current block at any time like any other read-only data block.

Accessing queued data blocks consumes slightly more overhead than non-queued data blocks, since the data block handle has to be hold up to date (in opposite to non-queued block variables, the handles of queued blocks do not refer to fix memory areas unless the user re-links it). It might be better to use a local read-only block instead of the global variable, if there are very much accesses (for instance in loops) to the same block inside a function.

❖ **Example**

Assume the following lines have defined two queued data blocks before (`EndianUnion` is defined in section **Error! Reference source not found.**):

```
xblockdef EndianUnion UnionBlock CurrentBlock;
blockdef EndianUnion CreatedBlock DecodedBlock;
```

You can access elements of these blocks like that:

```
unsigned long ul;
ul = CurrentBlock.TwoUnsigned.ul2;      // extract from the current data block
ul = prev.CurrentBlock.TwoUnsigned.ul1; // extract from the previous data block
//ul = prev.DecodedBlock.TwoUnsigned.ul1; // error, because DecodedBlock is
                                          // defined by blockdef (not xblockdef)
```

The first assignment takes the big-endianed unsigned long out of the `CurrentBlock` that is currently processed (i.e. placed at the top of the queue) and puts it as little-endianed unsigned long into the local memory. The second assignment takes the little-endianed unsigned long out of the previously processed `CurrentBlock` memory and copies it into local memory. The third assignment is not allowed, since there is nor previous block held by the BDM.

Let's now assume, that an external hardware sends blocks of the type `UnionBlock`. We want to write a decoder that creates a block containing the differences of the current and previous block:

```
decode on (CurrentBlock)     // execute when a new CurrentBlock arrives
{
  CreatedBlock NewBlock;     // use this block for the result
```

```
  NewBlock.TwoUnsigned.ul1 = (CurrentBlock.TwoUnsigned.ul1 –
                              prev.CurrentBlock.TwoUnsigned.ul1);
  NewBlock.TwoUnsigned.ul2 = (CurrentBlock.TwoUnsigned.ul2 –
                              prev.CurrentBlock.TwoUnsigned.ul2);
  send (NewBlock);           // put the new block into the BDM queue
}
```

6.7.4 Typecasting

Variables of a certain type can be converted into other types explicitly by entering a type cast command:

(*typename*) *variable*

Explicit type casting can be used to suppress compiler warnings that would occur on implicit conversions. It also may be useful in QLook Items to make sure that the returned value is of the same type used in the format string.

Typecasts are possible in the following cases:

- The source type has the same physical size in bytes as the destination type. The source data itself will not be changed but handled as if it is of the destination type. Please note that all integer calculations on arrays and bitmaps will result in arrays of (physically) long integer elements, although the resulting type may remain a virtually smaller type. Because of this, the physical size of the array may be two or four times larger than it seems. In that case you have to downcast the resulting array to its virtual type at first. This cast shrinks the physical size to the virtual size.

- Source and destination types are simple types. The source will be shrunk or enlarged to fit the destination type.

- Source and destination types are arrays with the same number of elements. The elements are converted like described before. Please note that the number of elements of bitmaps is not the number of pixels but the number of elements of the internally used array (see 4.1.6 for further information).

Other type conversions are not handled.

It may be helpful to use a typecast for specifying the data type of constant values. Since the data type of 0xffff can be either signed (-1) or unsigned short (65535), the result of an operation can be quite different, depending on the interpretation of the constant type. The G compiler takes all constants as unsigned values, as long as no negative sign occurs.

❖ **Example**

```
void castdemo ()
{
  char   c;
  short  s = -1234;  // = 0xfb2e
  double d;

  d = s & 0xffff;   // since calculations are processed in long values and 0xffff
                    // seems to be unsigned short, the result will be
```

```
                    //   d = 0xfffffb2e & 0x0000ffff = 0x0000fb2e = 64302.0
  d = s & (short) 0xffff;  // d = -1234 & 0xffffffff = 0xfffffb2e = -1234.0
  c = s & 0xff;     // causes a warning, since s is too large
  c = (char) (s & 0xff);   // suppresses the warning. c = 0x2e = 46
}
```

To display a one-dimensional array of short integer as a 16-bit bitmap with 32 columns and 4 lines, you may specify the following type cast inside a QLook item:

```
(bitmap 16 [32,4]) aus[56,128]
```

7 Constant Values

Constant values can be used instead of variables for read-only accesses. It is not possible to use constant values as destination of data. Constants may not be used as non-constant reference parameter to functions.

7.1 Simple Type Constants

The format of constant values of a simple type is described in section "Notation of Numbers" (3.1). These constants may be used as initial value, as parameter, as assignment source and even as operand of any calculation.

Constant values between 0x00000000 and 0x7fffffff have no fixed sign and may be used as signed or unsigned values, unless a negation (**-**) or type cast occurs. Values between 0x80000000 and 0xffffffff are assumed to be unsigned integers.

7.2 Character Constants

Character constants are single unsigned character values. A character constant is specified as follows:

`** ***character*** **`

The character may be any ASCII character as well as the special characters described below.

Some special characters can be specified inside of a string by a leading backslash:

newline (line feed)	NL (LF)	**\n**
horizontal tab	HT	**\t**
vertical tab	VT	**\v**
backspace	BS	**\b**
carriage return	CR	**\r**
form feed	FF	**\f**
alert	BEL	**\a**
backslash	\	****
question mark	?	**\?**
single quote	’	**\’**
double quote	”	**\”**
octal number	***ooo***	*******ooo***
hex number	***hh***	**\x*****hh***

The octal and hexadecimal representation can be used to specify any character. Octal numbers may consist of up to three digits between 0 and 7; hex numbers may consist of one or two digits between 0 and f (you may use capital letters also).

7.3 String Constants

String constants are zero-terminated one-dimensional arrays of unsigned characters. A string constant is specified as follows:

" *string* "

The string may be any sequence of characters (see 7.2 for information on non-ASCII-characters). The resulting array will contain the whole sequence with an additional zero byte at its end. The constant string source may be shorter than the destination without causing any warning or error.

A string constant may be handled like any other array.

```
...
unsigned char  szText[8] = "abcdefg"; // fill szText with 'abcdefg\000'
szText = "abcdefghijklmnop"[2,8];     // fill szText with 'cdefghij'
if (_memcmp ("bcdefgh", szText))      // compare szText with 'abcdefgh\000'
  { ... }
...
```

A string constant may be defined over multiple lines in two different ways:

- It may be ended with a double quote, intercepted by white spaces or comments and continued with a further double quote. All white spaces or comments between the inner double quotes are ignored; all other texts will cause errors.

- The string may be stopped at end of line by a single backslash and continued directly at the beginning of the next line. All white spaces in front of the second line are included into the string. It is not possible to place any ignored text inside the string with this method.

```
sz = "first part "  // this is continued...
     "second part"; // ...here. sz contains now "first part second part"
sz = "first part\
      second part"; // sz contains now "first part       second part"
```

7.4 Complex Constants

Complex constants can be used as initial value, as parameter or as assignment source without problem. Since the type of complex constants is not determined non-ambiguous, it is not possible to use as operand of a calculation. In that case, you have to execute an explicit type cast on this constant.

It is possible to specify complete constant data structures (struct, union or array) by putting the desired values inside of braces.

7.4.1 Structures and Arrays

Since structures and arrays consist of multiple elements, it is necessary to define a list of all elements as follows:

{ *const_element* (, *const_element*)* }

This defines a constant structure or array consisting of the specified ***const_element***s. Any of the ***const_element***s may be a complex constant itself.

Please note that `{ 0x12345678, 0x1234, 0x12 }` could specify an array of three long integer values as well as a structure of a long, a short and a character value.

❖ **Example**

```
struct strConstDemo
{
  long l;
  unsigned char sz[6];
  struct
  {
    short s;
    char  c;
  };
};

void ConstCall (const strConstDemo& p)
{
  // initialize static variable v.  v.sz is set to 'GSEOS\000'
  static strConstDemo v = {0x12345678, "GSEOS", {12345, 0b10101101}};

  // now set v to a new value. v.sz is set to 'GSEOS5' without trailing zero byte
  v = {0x87654321, {'G', 'S', 'E', 'O', 'S', '5'}, {1, 2}};

  ConstCall ({0xaffe, "call", {0xcafe, 0xae}});
}
```

7.4.2 Unions

The definition of constant unions consists of the specification of the first sub-symbol's value only, because all sub-symbols share the same memory and cannot be set independently. Since constants are handled like local memory, it is not possible to interlock data with gaps like in external data block handles.

{ *const_element* }

This expression defines a constant union with its first element set to ***const_element***. If the first element of the union is shorter than the complete union, the upper bytes of the union are filled with zero.

❖ **Examples**

```
union uniConstDemo
{
  struct
  {
    unsigned char sz[6];
    short s;
    char  c;
  };
  long l;
};

void ConstCall2 (const uniConstDemo& p)
{
  // initialize static variable v.  v.sz is set to 'GSEOS\000'. sz.l is not
  // accessible for union-constants
```

```
  static uniConstDemo v = {{"GSEOS", 12345, 0b10101101}};

  // now set v to a new value. v.sz is set to 'GSEOS5' without trailing zero byte
  v = {{{'G', 'S', 'E', 'O', 'S', '5'}, 1, 2}};

  ConstCall2 ({{"call", 0xcafe, 0xae}});
}
```

If you want to use complex constants in calculations, you have to specify the type of the constant using type conversion:

```
typedef long tal[4];
tal   al;
short s;

al = s * (tal) {123, 456, 789, 101112};
```

This assigns the result of the array multiplication to `al`. Array operations like this and other data manipulations will be subject of the next chapter.

8 Data Manipulation

Data manipulation may be any calculation or re-arranging of arrays.

Internally all calculations are executed in four bytes long integer or eight bytes long floating point variables. The operands are converted into the according type before the operation.

8.1 Operations

8.1.1 Dyadic Operators

The following table lists all operators that need two operators. The resulting type of these operations is the smallest type that can store both operands.

	priority	$a \times b$	$a = a \times b$
Addition	3	a `+` b	a `+=` b
Subtraction	3	a `-` b	a `-=` b
Multiplication	2	a `*` b	a `*=` b
Division	2	a `/` b	a `/=` b
Modulo	2	a `%` b	a `%=` b
Power (a^b)	1	a `**` b	
And (bitwise) (logical)	5 9	a `&` b a `&&` b	a `&=` b
Or (bitwise) (logical)	7 10	a `|` b a `||` b	a `|=` b
Exclusive or	6	a `^` b	a `^=` b
Equal		a `==` b	
Not equal		a `!=` b	
Less than		a `<` b	
Less or equal		a `<=` b	

Greater than		a `>` b	
Greater or equal		a `>=` b	
Shift left	4	a `<<` b	
Shift right	4	a `>>` b	
Maximum	8	a `>?` b	
Minimum	8	a `<?` b	

The operands ***a*** and ***b*** may be simple types or even arrays of simple types. See the section 8.2 (Array Operations) for further information. Bitwise operations cannot be used on floating point values. The operands of logical operations are converted to boolean before execution (non-zero becomes true, zero values become false). The result type of logical operations and comparisons is boolean.

8.1.2 Monadic Operators

The built-in operators that need only one operand are shown in the following table.

	operator	annotation
Negation (numeric)	`-` a	result type is signed
Negation (bitwise)	`~` a	undefined on floating point values
Negation (logic)	`!` a	result type is boolean
Increment (prefix)	`++` a	increments the variable ***a*** by 1, the value of the expression is the **new** value of ***a***
Increment (postfix)	a `++`	increments the variable ***a*** by 1, the value of the expression is the **old** value of ***a***
Decrement (prefix)	`--` a	increments the variable ***a*** by 1, the value of the expression is the **new** value of ***a***
Decrement (postfix)	a `--`	increments the variable ***a*** by 1, the value of the expression is the **old** value of ***a***
Absolute value	`abs` a	result type is unsigned
sine	`sin` a	result is floating point
arcsine	`asin` a	result is floating point
cosine	`cos` a	result is floating point
arccosine	`acos` a	result is floating point

tangent	`tan` a	result is floating point
arctangent	`atan` a	result is floating point
natural logarithm	`ln` a	result is floating point
exponential e	`exp` a	result is floating point
decimal logarithm	`log` a	result is floating point
exponential 10	`10 **` a	see 'power' operation in section 8.1.1 (Dyadic Operators)

The ***unary_expression*** "a" may be any single value or array (see section 8.2) as well as more complex expressions within brackets `( )` (except for the increment and decrement operations which require non-constant integer variables).

❖ **Example for Calculations**

```
{
  long l1 = 12345;
  long l2 = 24680;
  long l3;
  double d;

  l3 = l1 <? l2;       // l3 = 12345 (= minimum of 12345 and 24680)
  d = l3 * 0.321;
}
```

8.1.3 Random Values

Pseudo random values can be generated using the keyword

`random`

It returns a double value $0 \leq x < 1$. It is also possible to use this command to generate values between 0 and a specified value:

`random` `(` ***limit*** `)`

The result will be a random value of the same type as ***limit***. It will be inside the interval **[0, *limit*[**. This command can be used as well to generate arrays of random values. In this case ***limit*** has to be an array of the desired type, and its elements represent the upper limit for the value of the corresponding random element. That means, the limit of the random value may be specified separately for each element.

The random number generator can be initialized using the command

`seed` `(` **[** ***expression*** **]** `)`

This command sets the starting point of the random number generator to the value of ***expression***. If ***expression*** is omitted, the generator will be initialized with a value depending on the system time, the time since GSEOS start and a pseudo random number generated before the re-initialization.

❖ **Example**

```
{
  long l1 = 12345;
  unsigned char auc1[10] = "abcdefghi";
  double d;

  auc1 = random ("123456789" - '1'); // no zero termination!!!
  d = random * 10;                   // 0 <= d < 10
  l1 = random (l1);
}
```

8.2 Array Operations

All mathematical and logical operations may be used on arrays as well as on single elements. This method is faster than a self-made loop over all elements.

An operation between a simple type value and an array will evaluate the operation on the value and each element of the array. The resulting array will contain the same number of elements as the input. The type of the elements is the same as the type of the result of the according non-array-operation.

An operation between two arrays will process the corresponding elements of both arrays.

Multidimensional arrays will automatically be flattened before processing. That means, that the result of an array operation will always be a one-dimensional array, regardless of the number of dimensions of the operands.

Remember: These array operations are no matrix- or vector-operations!

It is not possible to use functions that are defined for scalar parameters for processing whole arrays instead. This is a feature of built-in operations only.

❖ **Example**

```
{
  unsigned char auc1[10] = "abcdefghi";
  unsigned char auc2[10] = "ihgfedcba";
  unsigned char auc3[10];
  double   ad[10];

  auc3 = auc1 >? auc2; // auc3 = "ihgfefghi"
  auc3[0,5] = auc3[0,5] + ('A' - 'a');  // auc3 = "IHGFEfghi"
  ad = sin auc2;    // calculate the sine of the ASCII values
}
```

8.3 Assignment Operator

The assignment operator can be used for assignments of simple type values as well as arrays and complex data structures. Even the assignment of array ranges is allowed.

Besides of this normal assignment of values to variables of the same type, the assignment operator **=** may be used to fill all bytes of a complex data structure (like a union, structure or an array) with a user defined value. This value has to be a character constant. Since data block elements may be

placed bit precise and may contain gaps, it is not possible to use this feature on data block elements but only on complete data blocks.

complex_data* = *constant_char

complex_data may be a union, structure or an array and must not be a *part* of a data block. All bytes of it will be set to the value ***constant_char***.

data_block* = *constant_char

All bytes of the memory of the work data block will be set to the value ***constant_char***. Bit precise placement is ignored by byte filling. The ***data_block*** must not be read only, of course.

❖ **Example**

```
{
  unsigned char auc1[10] = “abcdefghi”;
  unsigned char auc2[10] = “ihgfedcba”;
  unsigned char auc3[10];
  unsigned long aul[10];

  auc3 = auc1 >? auc2; // auc3 = “ihgfefghi”
  auc3[0,5] = auc3[0,5] + (‘A’ – ‘a’);  // auc3 = “IHGFEfghi”
  auc1 = rand (“123456789” – ‘1’); // no zero termination!!!
  auc1[1,9] = auc1[0,9];  // move all elements one step up

  aul = 0xba;    // fill all elements with 0xbabababa
}
```

9 Modules and Functions

This section describes all kinds of code that creates executable units that may be called by GSEOS modules or by other G-language functions.

9.1 Decoders

Decoders are nameless functions that are called by the BDM if the trigger condition is true. Normally decoders work on data blocks and create new data blocks. They do not return any value and get no parameters. Decoders can only be called by the BDM.

`decode on (` ***blockvarname*** **[(\| *blockvarname*)+])** ***compound_statement***

The decoder will be started by the BDM if any of the specified global block variables (***blockvarname***) arrives. The ***compound_statement*** may contain any code like declarations, calculations, loops, function calls etc. to create new blocks.

Nevertheless, there are some points that you should keep in mind writing a decoder:

- Decoders should not contain any wait command. Since all decoders run inside the same thread, a wait would cause all decoders to wait. Waiting in decoders may cause runtime warnings.
- Remember that sending a block that triggers the sending decoder may cause endless recursion. This is checked at runtime by the BDM. If a certain recursion depth is exceeded, all decoding will be shut down.

❖ **Example**

```
decode on (CurrentBlock)           // execute on every arriving CurrentBlock
{
  CreatedBlock NewBlock;           // variable for the result of the decoder
  NewBlock.s = CurrentBlock.s / 2;
  send (NewBlock);                 // put the generated block into the BDM
}
```

9.2 QLook Items

The QLook items that can be configured by G can be divided in two different groups: Command buttons, which are used to initiate actions by mouseclick, and Active QLook Items, which visualize data with automatic update on change of the displayed data.

QLook items may contain all code that is allowed for function bodies except the preprocessor commands **`#define`** and **`#undef`**.

9.2.1 Command Buttons

Command Buttons may contain any code that is allowed as body of a function with void return value. Whenever the Command Button is pressed, the code will be executed.

Please note that the GSEOS main thread will be blocked until the execution is finished.

9.2.2 Active QLook Items

The code of Active QLook Items returns data to a special buffer, which is displayed by QLook. The execution of the code is can be triggered automatically on each global data block variable appearing in it, but it is also possible to trigger the execution on user specified data blocks.

The evaluation of QLook Items is called only by the BDM and cannot be triggered by clicking on it.

Active QLook Items may contain any code that is allowed as a function body as well. The command **`return`** is used to fill the display buffer. If you only specify one single expression, you may omit the **`return`** command. A **`return`** without a value will not change the display buffer.

Please note that the return value of the first **`return`** command determines the type of the buffer that is displayed by QLook. All following return commands of this QLook item has to fit this type.

9.2.2.1 Automatic Triggering

Automatic Triggering is used if no explicit triggering is specified.

QLook Items are automatically triggered on each global data block variable appearing directly inside. There is no indirect triggering done on global data block variables that are used inside of functions called by the QLook Item.

9.2.2.2 Explicit Triggering

Explicit triggers can be specified by using

`on` **`(`** ***blockvarname*** **`(`** **`|`** ***blockvarname*** **`)*`** **`)`**

at the beginning of the code. In this case, the automatic triggering is deactivated and the code is executed whenever one of the blocks specified by the list of ***blockvarname***s arrives.

❖ **Examples**

Assume to have a function like the following defined in the startup file:

```
short CurrentS ()
{
  return CurrentBlock.s;
}
```

The following Active QLook Item will be evaluated only on startup:

```
CurrentS () * 3
```

The line before contains no global data block variable. Since the automatic triggering mechanism only looks for triggers in the uppermost level, it won't find the `CurrentBlock` inside the function `CurrentS ()`. To get a triggered QLook Item, you should type the following line:

```
CurrentBlock.s * 3
```

This line is evaluated and refreshed on the data display every time a new `CurrentBlock` arrives. Of cause you can use explicit triggering as well:

```
on (CurrentBlock) CurrentS () * 3
```

Remember, that automatic triggering will execute the code whenever at least one of all mentioned blocks arrives. Assume to have two different global blocks: `SlowBlock`, which arrives once in a minute, and `FastBlock`, which arrives each second.

```
blockdef EndianUnion tFastBlock FastBlock;
blockdef EndianUnion tSlowBlock SlowBlock;
```

An Active QLook Item containing the code

```
SlowBlock.ul + FastBlock.ul
```

will be updated each second, whilst an item containing

```
on (SlowBlock) SlowBlock.ul + FastBlock.ul
```

will only be updated on the arrival of a new `SlowBlock` (i.e. once in a minute). More complex Active QLook Items are possible also:

```
on (FastBlock)                  // run on each appearance of FastBlock
unsigned long ul, aul[10];
static bool b;

b = !b;
if (b)                          // on each second FastBlock: return without changing
  return;                       // the display buffer

for (ul = 0; ul < 10; ++ul)     // fill the array
  aul[ul] = ul;
if (random (10) < 5)            // in some cases:
  return aul;                   // return the just filled array
else                            // in other cases:
  return aul + 5;               // add 5 to all elements and return that array
```

9.3 Functions

Functions are handled very much like in C. The function definition syntax is the following:

type functionname `(` **(** ***type*** **[** `&` **]** ***parametername*** **)*** `)` **(** ***compound_statement*** **|** `;` **)**

The ***compound_statement*** may be any GSEOS language code embedded in braces. It is possible to declare prototypes of functions by replacing the ***compound_statement*** by `;` . This is useful if you want to define functions that call a later defined function: Since at least the function header must be known when the call is to be compiled, use the prototype definition before. Please note that the function header of the later function definition must be exactly the same as the prototype declaration.

Recursive function calls are allowed.

Functions may be called by any other GSEOS module (decoder, QLook Item, other functions etc.). The ***functionname*** is used to specify the desired function.

9.3.1 Return Value

Unlike C, GSEOS language allows complex return types like structures or unions. If nothing is returned by the function, you have to specify the return type as **`void`**.

Although it is possible to use large complex types as return value you should remember that return values are stored twice on the local machine stack (created in the local memory of the called function and copied into the local memory of the calling function). Therefore you should avoid using large return types. It is not possible to use references as return type.

9.3.2 Parameters

Since there are no pointers in the GSEOS language, parameter transfer normally is done by copying memory into local function memory. This may result in a lack of performance, when large amounts of memory have to be copied (for example: bitmaps as parameters). To avoid extensive memory duplication, you may specify parameters as references. As referenced parameters are not copied into the local memory of the function, all modifications of reference parameters inside the function will take effect outside also. That means, you can use reference parameters for the bi-directional transfer of data to functions and back.

The declaration of a reference parameter is done by entering the reference operator **`&`** behind the type of the parameter. Please note that accessing reference parameters is slower than non-reference parameters. You should use reference parameters only for two reasons:

- Modifications of the parameter inside the function has to take effect outside
- The parameter contains a large amount of memory.

Empty brackets mean that there are no parameters transferred to the function.

It is not possible to define functions with a parameter list of a variable length.

It is possible to specify arrays of unknown length as parameter by leaving the square brackets empty (**`[ ]`**). If you do so, you must specify this parameter as a reference (using **`&`**). Please note that all modifications done inside the function will have effect to the array outside the function also.

❖ **Example**

```
void Increment (short& sr)
{
  ++sr;    // increment sr.
}

EndianUnion DemoFunction (long l, short& sr)
{
  EndianUnion Result;
  Increment (sr);
  if (l > 0)
    return DemoFunction (l-1, sr);  // recursive call

  Result.TwoUnsigned.ul2 = 1234;
  Result.s = sr;                    // store current sr in return value
  sr = sr * 2;
  return Result;                    // return
}
```

```
long Length (unsigned char& uc[])   // allow char-arrays of any length to be
                                    // transferred
{
  long l;
  for (l = 0; uc[l] != 0; ++l);     // run until the terminating 0 is found
  return l;                         // return the length. (note: this is not
                                    // the size of the array)
}

void main ()
{
  EndianUnion VarUnion;
  short       s;
  long        l;
  unsigned char uc[256];

  l  = 3;
  s  = 2;
  uc = "Demonstration";
  VarUnion = DemoFunction (l, s);   // after this call, the variables will contain
                                    // the following values:
                                    // l                          == 3
                                    // s                          == 12
                                    // VarUnion.s                 == 6
                                    // VarUnion.TwoUnsigned.ul2   == 1234
  l = Length (uc);                  // after this call, l will contain 13
}
```

9.4 Non-G-functions (runtime library or DLL)

There are two different ways to use functions that are not written in G: You can use GSEOS-built-in runtime library functions or you can use DLL functions. These kinds of functions can easily called from your normal G programs.

Please note that the interface between G and those external functions cannot provide reliable error detection and type checking, since the real parameter types of the non-G-functions is not known by the compiler. A wrong declaration of external functions may cause fatal errors up to program crashes.

Non-G-functions should be handled with care!

9.4.1 Built-in Runtime Library Functions

If you want to use a runtime library (RTL) function, you have to declare the function header at first. This is done by the keyword **`extern`** followed by a usual prototype declaration of the RTL function consisting of the return type, the RTL function name and the list of parameters. If the function name is found in the built-in RTL, the forward declaration is resolved and can be used like any other function. Type and range checking is done based on the declaration.

Normally the complete interface to the RTL functions should be provided with GSEOS (for instance inside the file GSEOS.g). There should be no need for you to change the declarations.

Please note that differences between the G-declaration and the real needs of the runtime library function can cause fatal errors!

9.4.2 DLL Functions

If you want to use a dynamic link library (DLL) function, you have to declare the function header at first. This is done by the keyword `dll` followed by the filename of the desired DLL, an optional DLL-function name and a usual forward declaration of the DLL function consisting of the return type, the RTL function name and the list of parameters. If the DLL and the specified DLL-function name are found, the forward declaration is resolved and can be used like any other function inside G. DLL functions do not support checking of variable sized ranges. Type checking is done based on the declaration

`dll` "DLL name" **[** `.` "DLL function name" **]** ***return_type function_name*** `(` ***parameters*** `)` `;`

If the optional DLL function name is omitted, it is supposed to be identical to the ***function_name***.

Please note that differences between the G-declaration and the real needs of the DLL function can cause fatal errors! Also the restricted size and range checking may allow undetected errors that lead to unexpected behavior and even crashes.

9.5 Text References

Text references are used to translate long integer values into colored text strings for displaying purposes.

Text references normally are used by QLook Items.

9.5.1 Text Reference Type

The text reference function returns a memory area filled with the translated data. Since the incoming value is mapped to a text string in combination with a color index, the memory has to be organized as a structure. The first element of the structure has to be the long integer value containing the color index; the second element has to be a one-dimensional array of unsigned characters.

This structure type has to be defined before the definition of the first text reference. Since the text reference type is not hard coded but user definable, it grants a maximum of flexibility.

The text reference structure type has to be named `_tTextRef`. It should be defined as follows:

`struct _tTextRef { long` ***colorindexname*** `; unsigned char` ***textname*** `[` ***length*** `]; };`

colorindexname and ***textname*** may be any identifier. ***length*** specifies the maximum size of the returned text (including the zero termination byte). Please note that you cannot use ***textrefstatments*** with a ***string*** longer than ***length*** characters.

9.5.2 Translation Unit

In order to use a text reference, you have to specify what values should be mapped to what text. This is done inside the translation unit, which can be accounted as a special type of function definition without explicitly declared return value and parameters. The keyword `text` indicates such a definition:

`text` ***textrefname*** { **(** ***textrefstatement*** **)+** }

The value mapping is defined by one or more ***textrefstatements***:

textrefstatement **:=** **(** ***const_list*** **|** `default` **)** `:` ***string*** `,` ***colorindex*** `;`

Each ***textrefstatement*** maps all values fitting the ***const_list*** to the zero-terminated text ***string*** with the color that is specified with ***colorindex***. The use of the keyword `default` instead of ***const_list*** creates a ***textrefstatement*** that maps all values to ***string***.

Note that the first fitting ***textrefstatement*** will be used for mapping. All further ***textrefstatments*** that fit the value (like `default`) will be ignored. That means, that the keyword default should be used only once and only at the end of the list of ***textrefstatements***, because all ***textrefstatements*** behind the first `default` will never be activated.

The incoming value is accessible by using the automatically generated variable `lInput`.

The ***const_list*** is compared to the incoming value. It is defined as follows:

const_list **:=** **(** ***minvalue*** `..` ***maxvalue*** **) |** ***value*** **[(** `,` **(** ***minvalue*** `..` ***maxvalue*** **) |** ***value*** **)+]**

The expression ***minvalue*** `..` ***maxvalue*** fits the incoming value (`lInput`), if ***minvalue*** <= `lInput` <= ***maxvalue***. The expression ***value*** fits the incoming value if it is equal to it. If several expressions like this are used in a comma-separated list, the incoming value fits the ***const_list***, if it fits any of the expressions.

If there is no fitting ***textrefstatement*** for an input value, the result will be `?#` with color index 0.

Please note, that it is not possible to place any code inside a text reference that is generally executed (independent to the value of the input value). If you need to do this, you have to define a function with a header that complies with the normally generated translator function. See the next section (9.5.3) for further information.

9.5.3 Translator Function

The text reference definition automatically generates a function with the following header:

`_tTextRef` ***textrefname*** `( long lInput );`

Normally this function is called by data display items of QLook. But it is also possible to call it inside any other GSEOS language code. It maps the parameter **`lInput`** to the string and color index specified inside the ***textrefstatement*** of ***textrefname*** and returns the result as **`_tTextRef`**.

If the normal capability of the translation unit is not sufficient to translate the incoming long integer value to a suitable output, you may even define a function with a header equal to the one above and use that function just like any other text reference.

❖ **Example**

```
struct _tTextRef             // defining the return type of all text references
{
  long          lColor;      // the value of the color index.
  unsigned char szText[128]; // the text to show. 127 characters should be enough in
                             // most cases.
};

text TextRefDemo
{
  0:       "Zero",    -2;   // if the incoming value is 0, return "Zero"
                            // with color index -2
  1..3:    "1..3",    -1;   // incoming 1,2 or 3: return "1..3"
                            // with color index -1
  10:      "Ten",      1;   // ...
  4,8..12: "4,8..12",  2;   // map 4,8,9,11 or 12 to "4,8..zwölf
                            // note that incoming 10 is already mapped before.
  6:       "Six",     -2;   // ...
  default: "default", 0;    // all values that are not handled yet are mapped to
                            // "default" with color index 0
  13,15:   "13,15", lInput; // this line will never be reached. But it would map
                            // 13 to "13,15" with color index 13 and
                            // 15 to "13,15" with color index 15
                            // Please note that this example exceeds the useful
                            // color index range (-2 up to 2)
}
```

A QLook Item might contain the following code to show the translated value:

```
TextRefDemo (CurrentBlock.TwoUnsigned.ul1)
```

9.6 Batches

Batch files may contain any code that is allowed inside the body of a function. All symbols declared inside a batch file are taken as local except preprocessor definitions, which are always global. Every time when a batch file is called, a new thread is created which reads the file, compiles it just in time and executes it. After the execution the thread is deleted automatically.

Since batch files are executed in their own threads, waiting and triggering is possible without any problems.

9.6.1 Starting Batches from QLook

Simply double-click the symbol of the desired batch file in the tree-view on the left side of the QLook window to start the batch.

9.6.2 Starting Batches from Program Code

Batch files may be started from any program code with the command **`start`**. The syntax is as follows:

`start` **[** **`verbose`** **|** **`quiet`** **]** **(** ***batch-filename*** **)**

`call` **[** **`verbose`** **|** **`quiet`** **]** **(** ***batch-filename*** **)**

where ***batch-filename*** is a zero-terminated string and is supposed to specify a file inside the batch-file directory. The optional keyword **`verbose`** forces the batch-execution to be logged (with starting and ending-message). The optional keyword **`quiet`** forces the execution not to be logged (error-messages and warnings will be displayed, anyway). If none of these optional keywords is specified, logging activity depends on the settings in the BDM control panel.

The **`start`** command creates a new thread that compiles and executes the batch file. The execution of the calling code is continued immediately and parallel to the called code. In contrast to that the **`call`** command pauses the execution of the calling code until the batch execution is finished.

❖ **Example**

```
...
if (bExecuteBatch)                  // wants the user the batch to be startet?
{
  start ("BatchFile1.gb");          // start two batch files parallel to this code
  start quiet ("BatchFile2.gb");    // do not print logging information
}
sleep (4000);                       // continue
_AddMessage (MSG_INFORMATION, "Test", "main");
...
```

Let ***BatchFile1.gb*** contain the lines

```
sleep (6000);
_AddMessage (MSG_INFORMATION, "Test", "Batch1");
```

and let ***BatchFile2.gb*** contain the lines

```
sleep (2000);
_AddMessage (MSG_INFORMATION, "Test", "Batch2");
```

Executing the first program code will produce the Message "Batch2" after 2 seconds, the message "main" 2 seconds later, and at least the message "Batch1" another 2 seconds later.

10 Special Commands

10.1 Send

This command is used to put modified or newly created blocks into the BDM block queue. The BDM distributes each block to each decoder or QLook item that is using it. Sending is only allowed for data blocks of a queued type (see 4.2.4).

Depending on the context of the send command, the block will be put at the top or the bottom of the block queue:

- During a decoder run, the machine will put the block at the top of the queue, so that the block will be distributed and processed next (before all other blocks already placed in the queue). This behavior grants synchronism of source and decoded blocks.
- Outside of decoders, the **send** command places the block at the bottom of the queue, so that the block will be distributed and processed after all other blocks currently queued. This will preserve the chronology of incoming blocks.

The syntax for sending blocks to the BDM is as follows:

send (***workblockvar*** **(** **,** ***workblockvar*** **)***)

All block variables listed will be sent to the BDM. You cannot send read-only blocks.

You may use the command

sendcopy (***blockvar*** **(** **,** ***blockvar*** **)***)

to send a copy of ***blockvar***. The specified block variable remains unchanged. Since it creates a real memory copy of the specified block, this command can be used on read-only blocks also.

After sending a block, the BDM takes the full control over it. It is not possible to modify a sent block anymore in any way. You get read-only access to the block when it reaches the end of the BDM queue. Since the **send** command works directly on the specified data block variables, it will mark the specified variables as invalid. You can reinitialize data block variables using the command

newblock (***workblockvar*** **(** **,** ***workblockvar*** **)***)

or assigning a valid data block to the invalid variable. The **newblock** command creates a randomly filled data block, while the assignment creates a copy of the source block. The **sendcopy** command does not change the specified block variables – they remain valid.

Trying to access an invalid block will cause runtime-errors. You may check the validity of a data block variable using

validblock (***blockvar***)

The returned value is **TRUE** if ***blockvar*** may be accessed, **FALSE** otherwise.

10.1.1 Using Send or Sendcopy

For best performance you should carefully decide, which command you use for sending block variables. Since **sendcopy** creates a complete wasted memory copy of the block data, you should not use it if you do not need the data of this block after sending.

- If you do not use the block variable any more after sending, you should use the **send** command. The block variable gets invalid and no wasted block allocation or memory copy occurs.

- If you want to create a block with completely new data after sending, you should use **send** followed the **newblock** command. This creates a new valid block, but does not waste time by copying data.

- If you want to send a slightly changed block several times, it would be the best to use **sendcopy**.

10.1.2 Static Data Blocks

Static data blocks do not change their values between two runs and are not initialized at the beginning of a new run like other blocks. Nevertheless, the **send** command invalidates the static data block variable. You have to reinitialize the variable either using the **newblock** command or copying a valid data block into the invalid data block variable.

If you want the block to remain unchanged by the sending, you should use **sendcopy**. Remember that **sendcopy** creates more overhead than **send**, so do not use it if you do not need the block data after sending the block.

❖ **Example**

The following example shows how to rescue data blocks after a **send**.

```
{
  readonly blocktype tempblock;   // this is to be used as temporary copy
  static blocktype   sendblock;   // this block has to be restored after send

  // ...any code...

  tempblock = sendblock;          // this creates a read-only reference to the
                                  // current block. The reference remains valid
                                  // after the send! Since it is only a reference,
                                  // no copying of the complete data occurs.
  send (sendblock);               // send the block. sendblock becomes invalid.
  sendblock = tempblock;          // the data of the referenced block is copied
                                  // into the sendblock for future use.
}
```

10.1.3 Sending One Block Variable Several Times

You might want to create and send several blocks of the same type at once. You may do this inside a loop or hard coded, but as mentioned before, you cannot access the block variable after the

execution of the **send** command. As long as you do not use the **sendcopy** command, you cannot create a block, send it, modify a couple of elements and send the modified block. You need to re-initialize the block variable after sending it. This can be done in different ways:

- **Assigning a complete block.** You can use any other valid block of fitting type to re-initialize the data block variable. The data of the source block will be copied inside the destination block.
- **Automatic initialization** at the beginning of the looped scope, in which the sent block variable is declared. This does not work with static data block variables and is only advisable inside loops. The data of the block is not initialized.
- **Explicitly** by using the **newblock** command. The data of the block is not initialized.

❖ **Example**

```
decode on (CurrentBlock)
{
  CreatedBlock  SendBlock;
  CreatedBlock  CopyBlock;
  unsigned long ul;

  CopyBlock.TwoUnsigned.ul1 = 12345;
  for (ul = 0; ul < 3; ++ul)
  {
    DecodedBlock  NewBlock;         // initialization of NewBlock in each pass

    NewBlock.s = CurrentBlock.s;    // set an element
    send (NewBlock);                // send the block. NewBlock is not accessible
                                    // any longer.
    SendBlock = CopyBlock;          // copy block – this re-initializes SendBlock
    SendBlock.TwoUnsigned.ul2 = ul; // modify an element
    send (SendBlock);               // SendBlock is not accessible after send
    SendBlock = CopyBlock;          // copy block – again re-initialization
    SendBlock.TwoUnsigned.ul2 = ul / 2;
    send (SendBlock);
  }
}
```

10.2 Sleep

This command stops the execution for a specified number of milliseconds.

sleep (*time*)

10.3 At

This command is not implemented yet. It will stop the execution until a specified time is reached.

10.4 Wait

This command will stop the execution until one of the specified blocks arrives or the time exceeds the timeout value.

wait ([*timeout*] , *blockname* (| *blockname*)*)

The ***timeout*** may be omitted. In this case the machine will never cancel the wait for the specified blocks. The command returns a value, which specifies the reason of continuation:

- -1: no block was detected (timeout or error occurred)
- 0: the first block in the list arrived
- 1: the second block in the list arrived
- 2... and so on.

In future versions the **wait** command may be used not only for blocks but for events, too.

10.5 Size Of ...

Since the GSEOS language allows bit-oriented data management, the user may want to evaluate the sizes of data structures bit-precise. Therefore in addition to the command **sizeof** (which evaluates always the unpacked size in bytes) the command **bitsizeof** allows to evaluate the packed bit-size of any type. The size in bytes of packed data can be evaluated using the command **bytesizeof** (the result of **bytesizeof** is equal to the result of ((**bitsizeof** + 7) >> 3)).

The (bit- / byte-) size-of operators may be used on variables or types.

Although the structure may be used for byte based internal variables, the calculation of **bitsizeof** and **bytesizeof** always assumes the data structure to be used for (packed) data blocks. On the other hand the evaluation of **sizeof** assumes the data structure to be used for (unpacked) internal variables.

sizeof, **bitsizeof** and **bytesizeof** can be used on array parameters of variable sizes. In this case the size of the referenced array will be derived at runtime.

Normally it is impossible to mix type-name and field-names of a data structure to specify an element. The only exception is inside the (**bit**- / **byte**-) **sizeof** commands: the first part of the specifier may be a type name, all following may be field specifiers.

❖ **Example**

```
long l;
l = sizeof (PreciseUnion.s);     // l = 2 (bytes), the size of the field s of the
                                 // type PreciseUnion
l = sizeof (BitPreciseUnion.s);  // l = 2, the size of the field s of the variable
                                 // BitPreciseUnion
l = bitsizeof (PreciseUnion.s);  // l = 12 (bit), the bit-size of the field s of
                                 // the type PreciseUnion
l = bitsizeof (BitPreciseUnion.s); // l = 12, the size in bits of the field s of the
                                   // internal variable BitPreciseUnion, if it would
                                   // be copied to a data block of the same type.
```

11 Preprocessing

The GSEOS built-in 'preprocessor' is not really preprocessing the input before compilation. It is rather a kind of 'text-processing in between': Whenever the parser comes across an unknown symbol, it activates the preprocessor to replace the symbol, if there is a replacement defined for it. After replacement, the compilation will continue.

Please note: Although you might define symbols that are already used by the compiler (like GSEOS language keywords or data types, variables or function names), these replacements will never be used, because the text-processor will not be activated if the symbol is identified by the parser.

11.1 Comments

Text included in `/*` `*/` is ignored. Comments may be nested. If inside a comment a opening `/*` is found, the next `*/` will not end the comment mode.

Text following `//` is ignored until end of line. It is not possible to open or close a further comment there!

```
void main ()
{
  /* this is a comment.
    /* this is a comment inside an other comment. */
     this is still comment.
  */
  return; // last comment
}
```

Please note that `/*` or `*/` placed after a `//` are ignored. Therefore the following example would cause errors:

```
void main ()
{
  /* this is a comment.
    /* this is the inner comment // and even more commented. */
     this is still inner comment!
  */
  now the inner comment is closed, but the outer comment is still active.
  it is closed here: */
  return;
}
```

11.2 Include

It is possible to divide the GSEOS language startup-file into several pieces and put them together during compilation via `#include` command. Whenever the `#include` command appears the compiler switches to the text of the specified file and continues compilation. After compiling the included file, the compiler switches back and continues compilation of the previous file.

`#include` `<` ***filename*** `>`

or

`#include "` ***filename*** **`"`**

Any text behind will be ignored until end of line is reached.

11.3 Define

You may define your own keywords by using the **`#define`** command

`#define` ***keyword text***

Whenever the compiler reaches an unknown symbol during compilation, it checks if the symbol is defined as **`keyword`**. If it is found, it will be replaced by ***text***. The keyword remains defined and may be used as long as GSEOS is running or it is undefined by using the compiler control panel or the preprocessor directive

`#undef` ***keyword***

Please note that it is not possible yet to define complex macros with parameters.

text may be more than one line. In this case use `\` as the last character of the line that is continued in the next line.

11.4 Ifdef ... else ... endif

An easy way to switch between alternative source codes is using the **`#ifdef`** keyword. If the symbol behind **`#ifdef`** is already defined, the compiler compiles the code following directly until the next **`#else`** or **`#endif`** command. The code between **`#else`** and **`#endif`** is ignored. If the symbol is not defined yet, the code following the **`#ifdef`** is ignored and the code between **`#else`** and **`#endif`** is compiled instead. The **`#ifndef`** command has the opposite behavior.

❖ **Example**

```
#define MSG_INFORMATION         0
// define DEBUGMODE for creating additional debug information
#define DEBUGMODE

...

// if in DEBUGMODE create additional message
#ifdef DEBUGMODE
  _AddMessage (MSG_INFORMATION, "DEBUG", "debug point 1");
#endif

#ifndef DEBUGMODE
  _AddMessage (MSG_INFORMATION, "Info", "debugging mode off");
#else
  _AddMessage (MSG_INFORMATION, "Info", "debugging mode activated");
#endif
```

12 Messages

This section offers a short description of the compiler and virtual machine messages in combination with hints for eliminating the reason for these messages.

We have to differentiate between messages that appear at compile time and messages that appear during runtime:

- Compile time messages contain the filename and line number of the input that causes the message. Only a few messages are not associated to line numbers.
- Runtime messages can be caused by decoders, QLook Items or batches. Since the virtual machine code includes debugging information, even the runtime messages during execution of the compiled code specify the exact line number of the program that caused the message. The information provided by the message consists of the uppermost source level followed by the name of the currently executed function (if any) and the line number of the source that caused the message. The representation of the uppermost source level differs depending on the type of the code:
  - Messages from decoders contain the name of the file containing the decoder source and the line number of the opening brace of the decoder. Since decoders have no names, this is the best unique identification. Remember that the line number given here is not the line causing the message!
  - Messages from QLook Items contain the first couple characters of the source code associated with the item
  - Batch files generate messages that contain the name of the batch file

This additional information is placed inside brackets following the message text.

12.1 Warnings

Warnings show potential errors. Since some of these warnings may be caused deliberately, it is possible to suppress some types of warnings to keep the message window clear (see 0). You should not suppress all warnings, since some of them could indicate runtime problems.

`Array size unknown - be sure not to exceed it!` This means, that you are accessing a fix element of an array of a variable size. This may cause runtime errors if the accessed element is outside the array limits.

`Assignment in condition`: This warning indicates possible accidental use of an assignment (`=`) instead of an equality comparison (`==`) as a condition.

`Assignment of different enumeration types`: This assignment may lead to a value of the destination that is out of range.

`Assignment to enum - take destination as int`: This assignment may lead to a value of the destination that is out of range.

`Cast may truncate significant bits`: The operation causes an implicit typecast. The new type is smaller than the old type, which leads to loss of information. This warning appears for example at assignments of long to char.

`Conversion causes bit loss` indicates that you are assigning block elements with different bit-sizes (the simple type of both symbols may be identical, like long(28,4) and long(24,8)).

`Conversion of boolean to numeric`: You are using a boolean variable or value as a numeric.

`Conversion of const unsigned to signed`: You are using an unsigned constant value as a signed value. This warning only appears if the value uses full 32 bit (like 2147483648, which is 0x80000000).

`Conversion of negative const to unsigned`: This message appears when you are assigning a negative constant value to an unsigned variable.

`Conversion of unsigned to signed` means, that you are using an unsigned variable as a signed. This may cause large values to appear as negative. Remember that all string and character constants are used as (arrays of) unsigned characters.

`Different array sizes, taking minimum size`: An operation on arrays with different length occurred. To prevent memory access errors, the operation will stop after the smaller array is processed completely. Some elements of the larger array will not be processed.

`External buffer partly unreachable`: You have declared an extern variable that is smaller than the memory area provided by the hardware driver. That means that you will not be able to access the complete memory area.

`Float truncating to int`: The floating-point value is truncated to the lower next integer value.

`Ignoring declaration as unsigned`: Floating point variables cannot be declared as unsigned.

`Ignoring incrementation on float`: The increment or decrement operators are not defined on floating point values.

`Ineffective code`: You add/subtract a constant floating point zero or multiply/divide by a constant floating point one. This operation may imply a type cast without affecting the value of the second operand. Maybe this cast is not necessary. This operation primarily is thought to force the result of QLook Items to be floating point.

`Large parameter copying`: You are using a large data type as non-reference parameter. This may be ineffective, because the parameter has to be copied completely, and it may require much space on the limited local stack.

`Low remaining stack`: The local memory used by the function is very big. It may cause runtime errors with recursion or deep call stacks.

`Modulo on floats is not defined, truncating`: The operation modulo (%) cannot be used for floating point values. The value is truncated to integer before the operation.

`Name conflict on resolving external`: You declared a symbol with a name that is also used as a function in the runtime library. In this case the runtime library function is not accessible.

`Non-existing external`: The runtime library contains a function that is not declared in the source code. That means, the function is not accessible in this GSEOS session, because the 'linking' of runtime library functions is done only at startup.

`Parameter has no identifier`: The parameter of a function has no name and cannot be used inside the function.

`Possible blocking in decoder`: You are using a command inside a decoder that could block the system for more than ten seconds. You should lower the timeout value of the wait command or the sleeping time.

`Redefining with different value`: You are defining a symbol that is already defined. You should use **`#undef`** before the second definition to suppress this warning.

`Result is always zero`: The calculation is redundant, since its result is a constant zero.

`Senseless local use of keyword`: You are using a keyword that doesn't make sense in non-global contexts. For example: Since definition of endianness only takes effect on (global) data blocks, it is useless to specify endianness for elements of local datatypes.

`Signed interpreted as unsigned`: Negative values may be interpreted as large positive values.

`Source array has a variable size - using size of destination`: This assignment may cause runtime errors if the source is smaller than the destination. You should make sure that the source array is always larger than the destination.

`Stack cannot hold local memory`: The currently set stack size is too small to hold all local memory needed by the compiled function. This will cause runtime errors as soon as the function is called. You should increase the stack size or decrease the hold-local-limit in the compiler control panel (the latter takes effect after a new compiler run).

`Too many warnings. Suppressing further messages`: The currently executed module has reached the maximum number of warnings. To keep the message file readable, all following warnings of this module will be suppressed.

`Types differ with same size, copying`: This warning may appear if you are assigning an array of unsigned values to an array of signed values. Although the type is changed, the assignment is done. Remember that there are many cases that result in strange values of the destination variable (for example: you might be assigning a complex structure to an array of floating point values).

`Types do not fix exactly`: You are assigning a type to a slightly different data type.

`Union coverage differs`: This warning can be caused when a union containing a bit-precise datatype (diverging from the byte based type) and an overlaying array with unspecified is defined. Since the bit-precise definition in a block is smaller than the byte-based equivalent in a local variable, the overlaying array (which is always adjusted to the smaller version to avoid increasing the union's size) might not cover the complete byte-based local data structure. If this is the case, this warning occurs. It can be avoided by specifying the array's size explicitly.

`Unknown preprocessor directive`: The preprocessor doesn't know the used directive. Currently you can use **`#include`**, **`#define`**, **`#undef`**, **`#ifdef`**, **`#ifndef`**, **`#else`** and **`#endif`**.

`Unresolved function`: You declared a function to be extern that is not part of the runtime library, you declared a function to be part of a DLL that is not found or you declared a prototype of a function without defining the body. Any call of this unresolved function will cause runtime errors. The name of the function is shown in the warning message.

`Unsigned interpreted as signed`: An unsigned value is used as signed. This may cause large values to be interpreted as negative.

`Useless operation`: You are adding/subtracting a constant zero or you are multiplying/dividing by a constant one. Since this operation does not affect the result, it will be ignored.

`Using enumeration value as integer`: This is non-critical, because enumeration values always are internally represented as long integers.

`Using numeric value as boolean`: You are using a numeric value instead of a boolean. This message may indicate accidental use of logical operations instead of bit operations on numeric values. It may also indicate an erroneous use of a numeric value inside a conditional expression. Since all non-zero values are interpreted as true, this may cause unexpected results.

`Using temporary value as referenced parameter`: The value of a parameter may change inside the function, but it will have no effect to outside.

`Wait in decoder`: A decoder should not contain any wait or sleep commands, since a single waiting decoder blocks the execution of all other decoders, too. If you really need the decoder to wait, you should hold the waiting time as short as possible.

12.2 Errors

In very rare cases (actually there is no known case at present) internal bugs of the compiler may cause the error messages. If you are sure that your program is correct although there are error messages, you should take a look inside the next section (Fatal Errors) to find tips to create workarounds on compiler bugs.

`Array limits exceeded`: You are trying to access an element with a too high or negative constant index. This would cause unexpected results at runtime.

`Array with unspecified size has to be reference`: If you want to use arrays with an unspecified number of elements as a function parameter, you have to specify it as reference using **`&`**.

`Assignment to constant`: You are trying to assign a value to a constant.

`Assignment of different types`: You are trying to assign different types. This is only possible for simple types (bool, char, short, long, double) or types of the same size. Since the type checking is not based on type names, the message does not show the type names but the top-level type (a simple type, **`ARRAY`**, **`UNION`** or **`STRUCT`**). Although the top-level type may be equal, the difference may appear in lower levels.

`Blocking decoder`: It is not allowed to wait for batch execution inside of a decoder. You should start the batch execution as background process instead.

`Block parameter has to be reference`: If you want to use data blocks as a function parameter, you have to specify this parameter as reference.

`Block var name expected`: The name of a global block variable is expected.

`Cannot assign two arrays of variable size`: The sizes of the source and the destination array are not specified. It is not possible to derive the number of bytes to copy.

`Cannot handle complex block data`: The operation cannot be applied to block elements which are structures or unions containing arrays.

`Cannot handle this type here`: You may be trying to use a simple-type operation on a complex data structure. The resulting type is set to 'unknown', which may cause following **`Unknown Type`** – errors.

`Cannot return a value here`: Decoders cannot return a value. Use an empty return instead.

`Cannot use block data for references`: Since the elements of data blocks may be placed exceptional in the memory (all accesses have to decode it from the block and encode it to the block), it is not possible to reference it. You have to assign the block element to a variable and use this variable as parameter. This error will not occur if the block element is stored in little-endian format at byte-borders without gaps.

`Cannot use references here`: References only may be declared as function parameters.

`Cannot use variable array size here`: You are declaring an array with an unspecified number of elements in an illegal context. Arrays with an unspecified number of elements are allowed only as (referenced) parameters of functions or in unions with at least one element of known size.

`Changing preprocessor symbols`: You are using **`#define`** or **`#undef`** directives inside a QLook expression. This is not allowed, since the compilation order of expressions at startup is not defined.

`Constant expected`: You cannot use a variable here.

`Declaration does not fit prototype`: The function header differs from the prototype header specified before.

`Division by zero`: Since the division by zero is not defined, this operation is ignored.

`External buffer too small`: A external variable is larger than the memory area with the same name provided by the hardware driver.

`File not found`: The specified file does not exist.

`Global vars not allowed`: The GSEOS language prohibits the declaration of global variables, since this would cause complex timing behavior in combination with multiple parallel accesses.

`Illegal character`: You are using a character that is not allowed here.

`Illegal endianness mix`: You have an endianness change inside of a block type that does not fit byte borders. Since the big-endian and little-endian format are not compatible at bit base, it is not possible for values with different endianness to share the same byte.

`Illegal implicit cast on reference`: The type of the parameter differs from the function declaration. Since a typecast is not possible on references, the parameter cannot be used. Please note that parameters (like all internal variables) are stored in little-endian format. Using a big-endian formatted block element as parameter will cause this error. Note: It is not allowed to assign a different block type to a non-constant block reference.

`Illegal local use of keyword`: You are using a keyword in a forbidden context. (For example: blocks may only be defined in the global scope.)

`Illegal use`: You are using a non-function symbol as a function call.

`Illegal use of global variable`: Extern variables and global blocks cannot be initialized. It is also not possible to initialize global variables with non-constant values.

`Invalid block type`: The type that you have specified as base in the block definition is not defined yet.

`lValue expected`: You are trying to use a constant as a parameter that is a reference (declared with **`&`**) or you are trying to assign a value to a constant. Use a non-constant variable instead. Remember that all global data block variables provided by the BDM are non-l-values, since they cannot be modified. Remember that it is not allowed to use a work data block variable as non-constant read-only reference parameter.

`Modules must return a value`: explicit triggered QLook-interface modules have to return a value.

`Modulo on float undefined`: The modulo-operation cannot be used on floating point variables.

`No fitting loop`: You are using a **`cont`** or **`break`** statement outside a loop.

No previous block accessible: You are using the keyword **prev.** on a data block that does not provide this feature. Remember that **prev.** is allowed for data blocks defined by **xblockdef** only.

Not in global scope: The use of some keywords (like **extern** and **dll**) is forbidden in local scopes. Please note that all QLook-items as well as batch files are completely embedded by a local scope.

Not in single expression mode: Probably you have not specified a function header before the braces **{ }**. Since this is the syntax of the automatically triggered QLook Items, it causes an error if used elsewhere.

Null block access: You are trying to access an invalid block (already sent?). You can check the validity of data block variables by using **validblock** (see 10.1).

Only the first dimension of an array may be variable: You are trying to declare an multidimensional array with an unspecified number of elements in any other than the first dimension.

Out of memory: This is a runtime error of the virtual machine. The execution of the code will be aborted immediately.

Parse error: The parser cannot find a fitting grammar rule.

Positive constant value expected: Probably you are using a negative value or a variable.

Preprocessor stack exceeded: You are including too deeply nested files, or you are using too deep nested symbols. Perhaps you are recursively including a file or defining a symbol.

Pure virtual function called: You are calling an external function that is not linked. Check for '**Unresolved external**'-warnings and eliminate it.

Returning different types: The type of the symbol that you want to return differs from the declared return type of the function.

Size exceeded: You are trying to specify a bit-size of a type that is larger than the internal size of this type.

Stack overflow: The stack is too small to hold all local data needed by the virtual machine. Increase the stack size that is shown in the compiler control panel (this takes effect immediately) or decrease the size limit of local hold symbols (this takes effect after a recompile).

Symbol already declared: There is a symbol with the same name already declared inside the same scope. Remember that it is not possible to use the same name for a variable and a type at once.

Symbol is not a member: The structure or union does not contain the specified element. Remember that the compiler is case sensitive.

`Syntax error`: See explanation in the message itself. If there is no explanation, see the following messages for further information.

`Too few parameters`: The function call has fewer parameters than the function declaration.

`Too many errors - aborting`: The maximum number of errors is reached and the current (compiler- or virtual machine-) execution is aborted. The code will not be executed any more. This number can be adjusted in the compiler control panel at runtime. If you increase the maximum number of errors at runtime, code that already had reached the former maximum number of errors can be revived – until the new number of errors is reached again.

`Too many parameters`: The function call has more parameters than the function declaration.

`Too many single expressions`: The source contains more than one QLook Item function.

`Undeclared symbol`: You are using a symbol that is not declared yet. Place the symbol declaration before the first use. Remember that the compiler is case sensitive. Maybe you are using a formerly known preprocessor symbol that is not longer defined (perhaps you deleted the symbol using the compiler control panel?).

`Unexpected end of file`: Probably there is a comment not closed.

`Unknown Type - ignoring operation`: This is an aftereffect in most cases.

`Unresolved external variable`: The specified variable name is not provided by any hardware driver for access to mapped memory.

`Use does not fit declaration`: You are using a parameter of a different type as specified in the declaration of the function.

`Use of 'extern' not allowed here`: Only runtime-library functions may be declared as external.

`Work block expected`: You are trying to use a read-only block for an operation that needs a work block. It is not possible to send read-only blocks to the BDM.

12.3 Fatal Errors

These messages indicate internal compiler errors that are not caused by the user. Nevertheless they may be caused by the error handling of the compiler (aftereffect of a user error). This means that the fatal errors should disappear in most cases, if the user program is corrected.

If you get a fatal runtime error by the machine, this possibly indicates that the machine itself or the compiler is defective.

Although bugs of the compiler or virtual machine cause fatal errors, it may be possible to find workarounds by changing the user program. For this reason, you will find some hints on effective changing your program. In some cases it also may be helpful to force the creation of internal help

variables by adding useless calculations like (...`+1-1`). Sometimes it is possible to find a workaround by inserting a 'senseless' statement (like `1;`) before (or even inside) the part of your program that causes the error.

`Cast error`: The compiler cannot perform an implicit cast. This is definitively a bug inside the compiler and should not occur. Perhaps you can create a workaround by using local variables for provisional results. Possibly it would help to use explicit typecasts.

`JumpStack corrupted`: The internal table of jump-addresses for loops or conditional expressions (like if...else, switch...) is corrupted. As a workaround you may try to eliminate some complex loops or conditional expressions.

`Memory error`: The managing of local help variables detected a malfunction. As a workaround you may try to eliminate implicit help variables by using local user-defined variables (instead of `a[b*c]...` try `d = b*c; a[d]...`).

`Not implemented`: At compile-time, this message indicates that you are using a rule of the GSEOS language grammar that is not implemented yet. Your source probably will be correct in a future version, but you have to reword the statement if you want it to get compiled now (for example: Use **`if`** ... **`else`** ... instead of ... **`?`** ... **`:`** ...). If this error occurs at runtime, it may indicate that your program contains illegal memory write commands that destroy the code. The compiler or the virtual machine should detect almost all cases of illegal memory accesses, but possibly you found a technique to avoid this detection.

`Stack corrupted`: Runtime error. The code generated by the compiler has incorrect stack handling. As a workaround you may try to eliminate some function calls.

`Strange`: This internal error is really strange.

`Symboltable inconsistency`: The symbol table does not contain the expected information.

`Type not implemented here`: If there are no preceding errors, this message indicates a bug inside the compiler. Otherwise this message may result from former 'unknown symbol'-errors.

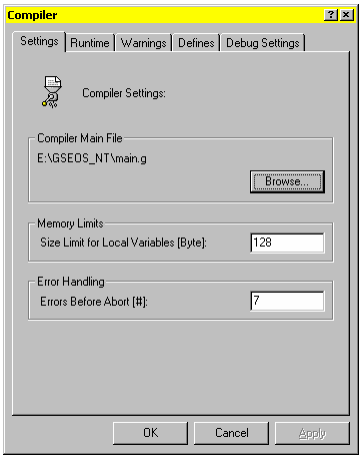
`Value can't have an address`: Definitely a compiler bug – without any known workaround.

13 Compiler Control Panel

The Compiler Control Panel allows adjusting the settings of the compiler and virtual machine to the needs of the current project.

13.1 Settings

This Control configures the behavior of the G-Language compiler.



13.1.1 Compiler Main File

This is the name of the file containing the program that is compiled at startup of GSEOS. This file has to contain (or include) all block definitions and decoders. It should enclose the function `void main ()` (which is executed on startup) and the function `void _exit ()` (which is executed on shutdown of GSEOS).

A change of this value does not take effect before the next startup of GSEOS.

13.1.2 Memory Limits

13.1.2.1 Size Limit for Local Variables

To allow the use of large local variables in combination with a strictly limited stack size, the compiler stores variables that exceed the size specified here in dynamically allocated memory.

Variables that are smaller than the specified size are stored on the local stack, variables larger than the specified size are stored outside the local stack automatically. The syntax of accesses to non-local variables is exactly the same as of accesses to local variables.

Accesses to variables on the local stack are faster than accesses to dynamically stored variables. Setting the limit too low will cause loss of performance while setting the limit to high may result in stack overflow errors.

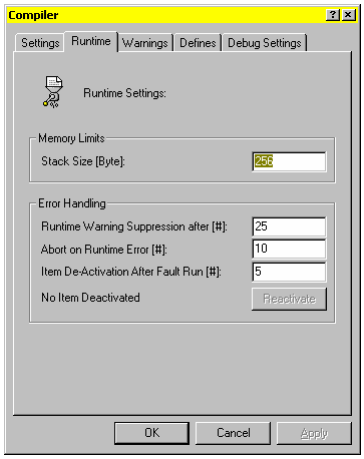
The limit cannot be set to values less than 24 bytes. That causes all simple types and internal generated variables to be stored on the local stack. Non-local variables use at least 4 bytes of local stack.

13.1.3 Error Handling

13.1.3.1 Errors before Abort

This value specifies how many error messages may appear before the compilation will be aborted. Please note that the resulting number of error messages will be value+2 (since value+1 is the first message exceeding the limit and value+2 is the `too many errors` message).

Runtime



This Control configures the behavior of the G token interpreter (virtual machine).

13.1.4 Memory Limits

13.1.4.1 Stack Size

This is the size in byte that is used for the local stack of the virtual machine. Since there may be a lot of parallel machine runs, each with its own local stack, the stack should not be too large. The stack is used to store user-defined variables, parameters and so on.

A small stack is enough if you do not need much local memory or a large function call stack. If you need large amounts of local memory and use deep recursions, you should increase the stack size.

The stack size may be altered at runtime. All following machine runs use the newly set value. This may be useful if (for example) a decoder repeatedly produces stack overflow errors.

The stack size cannot be set to values less than 50 bytes.

13.1.5 Error Handling

13.1.5.1 Runtime Warnings Suppression After

This value contains the maximum number of warnings that is displayed by a single module. If the same module creates more warnings, the messages are suppressed. The value 0 (zero) indicates that no messages will be suppressed.

13.1.5.2 Abort on Runtime Error

This value specifies the number of runtimes errors that have to appear to abort the current code execution. The machine will continue execution although there might be runtime errors, as long as this number is not reached. The value 0 (zero) means, that there is no limit for runtime errors.

A small group of runtime errors causes immediate aborts, independent of the value set here. A runtime error of this type is a stack overflow.

The behavior of the user program might be strange after a runtime error, since it might use unexpected values, but in some cases (like endurance tests) that may be a better alternative than aborting.

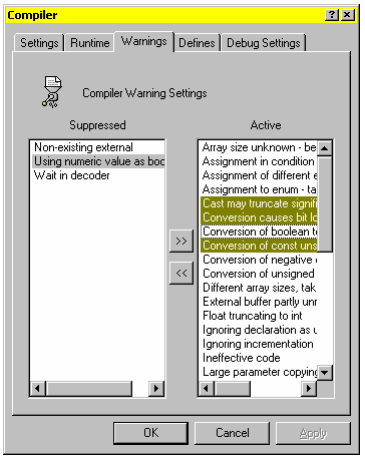
13.1.5.3 Item Deactivation After Fault Run

This value specifies the number of runs with errors of an item (QLook item or decoder), before it's future execution will be suppressed completely. The value 0 (zero) means, that items are never deactivated. Increasing the maximum number of runtime errors (or set it to zero) can reactivate all items with less fault runs immediately.

13.1.5.4 Reactivate

This button reactivates all currently deactivated items.

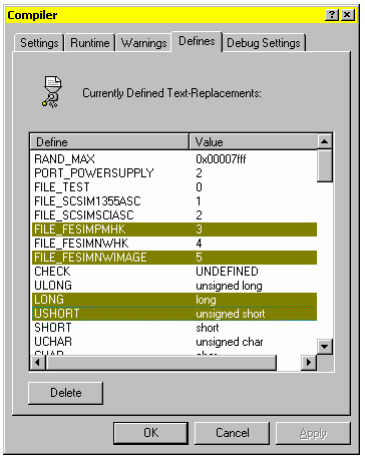
13.2 Compiler Warning Settings



The G-Language compiler generates a couple of warnings that indicates possible sources of malfunctions of the user-defined code (see section 12.1). Some of these warnings are less important than others, so the user maybe wants to suppress the warnings that are caused intentional by the user.

This control easily allows suppressing specific warning messages and gives you an overview of all currently suppressed warning messages.

13.3 Defines

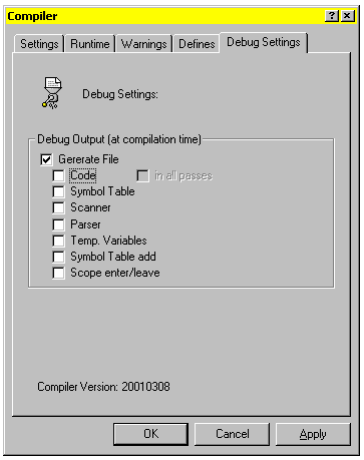


This Control lists all currently defined preprocessor symbols. You can easily delete symbols listed here, but handle with care! You cannot define new or redefine existing preprocessor symbols. Once deleted, a symbol can only be defined again by the compiler.

All symbols listed here can be used inside QLook items. The keywords are replaced at compile-time by the currently set value.

13.4 Debug Settings

13.4.1 Debug Output



These settings are used for compiler debugging purposes mainly. It allows printing internal compiler information into a debug file.

- **Generate File:** activate or disable the debugging option. This option should be turned off in normal use. Enabling this option without any other option activated may be useful to log all warnings and error messages. Since compiler messages are not displayed in the message window when compiling a QLook item, this is a way to find out the reason of "Unable to create item..." messages.
  - **Code:** show the generated code
    - **in all passes:** also show the code generated in first pass
  - **Symbol Table:** output the symbol table
  - **Scanner:** show additional debugging information for the scanner module of the compiler
  - **Parser:** show debugging information for the parser
  - **Temp. Variables:** show information about the management of temporary variables
  - **Symbol Table add:** print a message for each new symbol
  - **Scope enter/leave:** print a message whenever a scope-change is detected

Debug output may lead to considerable slow down of compilation. Large static variables result in a large debug-output file, if the output of generated code is activated.

Since most of the generated information can only be understood with knowledge of internals of the compiler and virtual machine, the GSEOS user should only use the "Generate File" option.

13.4.2 Compiler Version

This value indicates the version number of the used G-language-unit. The higher this value is, the newer are the G compiler and virtual machine.

14 List of Keywords

The following list contains the keywords of the G-Language that may not be used as variable or function names:

abs, acos, asin, atan, at
batch, big-endian, bitmap, bitsizeof, blockdef, bool, break, bytesizeof
call, case, char, const, continue, cos
decode, default, dll, do, double
else, enum, exp, exec, extern
false, for, from
global
if, int,
little-endian, ln, log, long
newblock
on
prev.
quiet
random, readonly, return
seed, send, sendcopy, short, signed, sin, sizeof, sleep, start, static, struct, switch
tan, text, true, typedef
union, unsigned
validblock, verbose, void
wait, while, work
xblockdef

15 Known Problems

15.1 Not Yet Implemented

- Postfixed increment / decrement operator (use the prefixed operators instead)
- Event handling

15.2 Not Fully Supported

Currently all implemented features have full functionality.

15.3 Not Optimised

- Automatic triggering QLook Items may cause ineffective code. For example the line `sizeof (CurrentBlock)` will be executed each time when a new `CurrentBlock` arrives. Since the size of `CurrentBlock` is constant, the repeated execution is wasted performance. To prevent this, you should not use global block variables to evaluate sizes.
- Each use of global block variables creates code for getting the current block from the BDM. You should avoid using global data block variables inside `sizeof` or `bitsizeof` commands for a better performance. Use the type of the desired block variable instead.
- Bit-precise accesses could probably get faster than currently realized.
- The compiler-messages are not very clear yet. A single error may cause many aftereffects.

16 Index