User Program Manual for Gold Line Drives

February 2012 (Ver. 1.3)

Elmo Motion Control

www.elmomc.com

Notice

This guide is delivered subject to the following conditions and restrictions:

- This guide contains proprietary information belonging to Elmo Motion Control Ltd. Such information is supplied solely for the purpose of assisting users of the Gold Line technology.
- The text and graphics included in this manual are for the purpose of illustration and reference only. The specifications on which they are based are subject to change without notice.
- Information in this document is subject to change without notice.

Elmo Motion Control and the Elmo Motion Control logo are registered trademarks of Elmo Motion Control Ltd.

EtherCAT Conformance Tested. EtherCAT® is a registered trademark and patented technology, licensed by Beckhoff Automation GmbH, Germany.

Document. no. MAN-G-USRPGM (Ver. 1.3)

Revision History

Ver. 1.0 Initial release

Elmo Worldwide

Head Office

Elmo Motion Control Ltd.

60 Amal St., P.O. Box 3078, Petach Tikva 49516
Israel

Tel: +972 (3) 929-2300 • Fax: +972 (3) 929-2322 • info-il@elmomc.com

North America

Elmo Motion Control Inc.

42 Technology Way, Nashua, NH 03060
USA

Tel: +1 (603) 821-9979 • Fax: +1 (603) 821-9943 • info-us@elmomc.com

Europe

Elmo Motion Control GmbH

Hermann-Schwer-Strasse 3, 78048 VS-Villingen
Germany

Tel: +49 (0) 7721-944 7120 • Fax: +49 (0) 7721-944 7130 • info-de@elmomc.com

China

Elmo Motion Control Technology (Shanghai) Co. Ltd.

Room 1414, Huawen Plaza, No. 999 Zhongshan West Road, Shanghai (200051)
China

Tel: +86-21-32516651 • Fax: +86-21-32516652 • info-asia@elmomc.com

Asia Pacific

Elmo Motion Control

#807, Kofomo Tower, 16-3, Sunae-dong, Bundang-gu, Seongnam-si, Gyeonggi-do,
South Korea

Tel: +82-31-698-2010 • Fax: +82-31-698-2013 • info-asia@elmomc.com

Table of Contents

Abbreviations

Abbreviation	Meaning
DSP	Digital Signal Processor
EAS	Elmo Application Studio
RS232, USB, TCP, EoE, CoE, CANopen	Communication protocols
PDO	Process data object
SDO	Service data object

Chapter 1: The Elmo Language

Gold servo drives use a communication language that enables the user to:

- Set up the drive.
- Send commands to the drive indicating what functions to perform.
- Query the drive's status.

Two methods can be used to communicate with the drive:

- Direct communication
  Direct communication — either RS232 or CANopen — can be used to transfer commands to the drive and receive an immediate response from the drive. This method requires on-line communication and close cooperation between the drive and its host.

  The properties and standards of the various communication interfaces (RS-232, USB, TCP, EoE, CoE) require a different command syntax for each method. This chapter describes the drive language according to the basic RS-232 or CAN communication protocol.
- User program
  A user program is written in the drive language and stored in the drive's non-volatile memory. The drive can then run the program with minimal or no assistance from the host.

The CANopen communication protocol can access simple numeric interpreter get and set commands very efficiently. The CAN binary interpreter uses process data objects (PDOs) to issue interpreter commands and to collect the responses. This is the most economical way to minimize both the communication load and the drive CPU load.

The CANopen communication protocol can be used to access the entire set of interpreter services, including those inaccessible by the binary CAN interpreter, using a text format. The CANopen communication standard is a broad topic and beyond the scope of this manual (it is covered in the Elmo CANopen implementation manuals).

Software programs use the interpreter syntax, with extensions that are needed to support program flow instructions and in-line documentation.

The full set of drive commands is documented in the Command Reference for Gold Line Drives.

1.1. The Command Line

The Interpreter evaluates input strings, called *expressions*, which are sequences of characters terminated by a semicolon (;) within a line, a line feed or a carriage return. Semicolons added at the end of a line before a line feed or before a carriage return are ignored.

The maximum length of a legal expression is limited to 511 characters.

A command line may include a comment marker, which consists of two consecutive asterisks (**).

All text from the comment marker to the next line feed or carriage return is ignored. The comment marker is used to prepare documented batch files that are sent later directly to the drive.

Lines can be indented in any manner for readability or convenience, but indenting is not required for any purpose.

The following table presents some examples of command lines.

Command Line	Result	Remarks
3+4	7	
PX=7; PX-3	4	PX is set to 7, and 3 is then subtracted.
(3.2+4)/2	3.600000	

1.2. Expressions and Operators

The drive language supports operators, which specify a mathematical, logical or conditional operation/relation between two or more operands. Operands (or parameters) and operators may be combined in any way allowed by the syntax to create an expression. The following sections describe the operators and the syntax rules for expressions.

1.3. Numbers

Gold drives use two numerical types: 32-bit integers and 32-bit floating-point numbers (floats). In text inputs, numbers containing a decimal point and numbers written in exponential notation are interpreted as floats. Other numbers are interpreted as integers.

The range for integers extends from –2,147,483,648 to +2,147,483,647. If a number assigned to an integer exceeds the integer range, it is converted into a float.

For example, if the number 2147483648 is entered, the Gold drive converts this to 2.147483e09.

Note: The lowest integer, –2,147,483,648, cannot be entered explicitly through the interpreter due to the method by which immediate numbers are evaluated internally. Nevertheless, this integer value is valid and can be entered in hexadecimal form as 0x80000000.

1.3.1. Hexadecimal Format

Integers may be written in decimal or hexadecimal format. The prefix 0x can be added to any integer number to denote hexadecimal format. For example, the hexadecimal notation 0x10 is equivalent to the decimal number 16.

1.3.2. Floating-Point Numbers and Scientific (Exponential) Notation

Floating-point numbers are recognized by the interpreter by the presence of a decimal point (.), which distinguishes between the integer and the fractional part. For example, the number 3 is interpreted as an integer, and the number 3.0 is interpreted as a float. When an expression is

evaluated, the result will be a floating-point number only if one of the operands is a floating-point number. Therefore, the expression `2/3` yields 0, and `2.0/3` yields 0.666667.

The accuracy of floating-point numbers is less than 5 non-zero digits after the decimal point. For example, `10000*0.00001` is displayed as 0.099999.

Numbers entered in scientific notation are also valid. For example, `2.55e-3` is echoed as 0.002550; `10e0` is echoed as 10.0.

A number written in scientific notation is interpreted as a floating-point number. Numbers with up to 5 non-zero digits after the decimal are evaluated.

For example, `1.000001e10+0.000010e10` becomes 1.000011e10.

Number Overflow: When the integer value on the left- or right-hand side of the decimal point is greater than the Maximum Long Number (2,147,483,647), the following error message is returned: Command syntax error (19)

For example, `>KI[1]=79.5774715459477` produces the following:

```
Error: Command syntax error
```

Here the value on the right-hand side of the decimal point is 5774715459477, which is greater than the Maximum Long Number.

Examples

Numbers (floats) with 5 digits or less to the left of the decimal point are displayed without change. For example, `12345.0` is displayed as 12345.0

Numbers with 6 or 7 digits to the left of the decimal point are displayed as a float with 1 always added to the value. The following table lists some examples of how numbers are displayed.

Number	Is displayed as
100000.0	100001.0
111111.0	111112.0
222221.0	222222
999999.0	1000000
600000.0	600001.0
2000000.1	2000001.0

Numbers with a total of 6 digits (to the left and right of the decimal point) are displayed as they are entered. The numbers 1.23456, 12.3456, 123.456, 1234.56, 12345.6 are generally presented without change. In some cases there is a deviation of 1 in the number of digits. For example, the number 9.87654 becomes 9.876539.

Numbers with a total of up to 7 digits (to the right and left of the decimal point) are displayed without change.

Numbers with 8 or more digits to the left of the decimal point are displayed in scientific notation (mantissa and exponent of the power of ten). The lowest 7th and 8th digits are rounded upwards: 12345678.0 becomes 1.234568e07.

Numbers with 8 digits, but with some of the digits to the right of the decimal point are displayed almost without change. From the 8th lowest digit onwards the number is truncated, and the lowest digits are not displayed, as in the following examples:

- 123.45678 becomes 123.4568
- 1.2345678 becomes 1.234568
- 6543.9876 becomes 6543.987
- 2321.66458 becomes 2321.665

An integer value is always truncated to the nearest lower number. For example, 5/2 becomes 2, whereas 5/2.0 becomes 2.5. If an integer exceeds the integer range, it is interpreted as an error.

The range for floating-point numbers includes numbers that are greater than -1e37 and less than 1e37 (-1e37 < *N* < 1e37, where *N* is a floating-point number).

To clarify, values such as -1e37 or 1e37 are out of range.

A floating-point number may be written with or without an exponent.

For example, 2.5e4 is equivalent to 25000.0, and expressions that compare a floating-point number to an integer are evaluated as follows.

Expression	Result
`(25000.0==250000)`	True
`(25000.1==250000)`	False
`(25000.01==250000)`	False
`(25000.001==250000)`	True

If a floating-point number exceeds the floating-point range, it is also interpreted as an error.

Logical operators yield 0 or 1 as a result.

The results of logical operations are integers.

1.4. Mathematical and Logical Operators

Expressions may contain any combination of arithmetic, relational and logical operators. Precedence levels determine the order in which the expression is evaluated. Within each precedence level, operators have equal precedence and are evaluated from left to right.

For example, `a*b/c` is equivalent to `(a*b)/c`.

The following table lists the mathematical and logical operators used in the programming language for Gold drives. The table also indicates the operator precedence in order from the highest to the lowest precedence level.

Operator	Description	Precedence
~	Bitwise NOT of an operand	17
!	Logical negation	17
-	Unary negation (minus)	17
%	Remainder after dividing two integers	16
*	Multiplication of two operands	16
/	Division of the left operand by the right operand	16
+	Addition of two operands	15
-	Subtraction of the right operand from the left operand	15
<<	Bitwise shift left	14
>>	Bitwise shift right	14
<	Logical smaller than	13
<=	Logical smaller than or equal to	13
>	Logical greater than	13
>=	Logical greater than or equal to	13
==	Logical equality	12
!=	Logical not equal	12
&	Bitwise AND of two operands	11
|	Bitwise OR of two operands	9
&&	Logical AND	8
||	Logical OR	7
=	Assignment	
()	Parentheses, for expression nesting and function calls	
[]	Brackets, for array indices and multiple value function returns	

Table 1: Mathematical and Logical Operators

The default precedence can be overridden using parentheses, as in the following examples:

Operation	Result
A=3/2/2	A=0
A=3/(2/2)	A=3

1.5. General Rules for Operators

Most arithmetic operators work on both integers and floats. An arithmetic operation between integers yields integers. An operation between floating-point numbers, or between an integer and a floating-point number, yields a floating-point result.

For example, all of the following expressions are legitimate.

Operation	Result
1+2	3 (integer)
1+0x10	17 (integer) (Here 0x10 is treated as a standard integer.)
1.03+0x20	22.03
1+2.0	3.0 (float)
2.1+3.4	5.5 (float)
2/3	0
2.0/3	0.666667

If the result of addition and subtraction operations between two integers exceeds the integer range (–2,147,483,648 to 2,147,483,647), the result is truncated, and the type remains an integer.

In the following example, the operation includes cycling back to the beginning of the range for integers.

Operation	Result
2147483647+10	-2147483639

A division operation between two integers may yield a floating-point result if the result includes a remainder, as in the following examples.

Operation	Result
8/2	4 (integer)
9/2.0	4.5 (float)

If a multiplication operation between two integers exceeds the integer range, the result is converted into a floating-point number and is not truncated, as in the following example.

Operation	Result
100000*100000	1.0e+10 (float)

Bitwise operators require an integer input. Floating-point inputs to bitwise operators are truncated to integers. For example, 7.9&3.4 is equivalent to 7&3, because the floating-point

number 7.9 is truncated to the integer 7 and 3.4 is truncated to the integer 3 before the operator & (bitwise AND) is applied.

The result of a unary negation (minus) operation for the minimum integer value exceeds the integer range. Therefore, the result is truncated to the maximum integer value. For example, `-0x80000000` gives 2147483647 or 0x7FFFFFFF.

1.6. Operator Details

The following table describes the operators in detail.

Operator/ Description	Symbol	Number of Arguments	Output Type	Examples
Arithmetic addition	+	2		4+5=9 3.45+2.78=6.23
Arithmetic subtraction	-	2		3.45-2.78=0.67
Arithmetic multiplication	*	2		PA=PA*2 doubles PA 5*4=20 1.5*2=3.0
Arithmetic division	/	2		20/4=5 3/1.5=2.0
Remainder after the division of two integers	%	2	32-bit long integer	20%4=0 5%2=1
Bitwise NOT	~	1	32-bit long integer	~3 is 0xfffffffc, which is actually -4. ~3.2 is the same as !3.
Bitwise OR	|	2	32-bit long integer	PA=0x2|0x5 is equivalent to PA=7. PA=0x2|5.1 is the same.
Bitwise AND	&	2	32-bit long integer	PA=0x7&0x3 is equivalent to PA=3. PA=0x7&3.1 is the same.
Logical equality	==	2	0 (false) or 1 (true)	If x = 3 and y = 3, then x==y yields 1. If x = 3 and y = 5, then x==y yields 0.

Operator/ Description	Symbol	Number of Arguments	Output Type	Examples
Logical inequality	!=	2	0 (false) or 1 (true)	If x = 3 and y = 3, then x!=y yields 0. If x = 3 and y = 5, then x!=y yields 1.
Logical greater than	>	2	0 (false) or 1 (true)	If x = 3 and y = 3, then x>y yields 0. If x = 3 and y = 2, then x>y yields 1. If x = 1 and y = 2, then x>y yields 0.
Logical greater than or equal to	>=	2	0 (false) or 1 (true)	If x = 3 and y = 3, then x>=y yields 1. If x = 3 and y = 2, then x>=y yields 1. If x = 1 and y = 2, then x>=y yields 0.
Logical less than	<	2	(false) or 1 (true)	If x = 3 and y = 3, then x<y yields 0. If x=3 and y=2, then x<y yields 0. If x = 1 and y = 2, then x<y yields 1.
Logical less than or equal to	<=	2	0 (false) or 1 (true)	If x = 3 and y = 3, then x<=y yields 1. If x = 3 and y = 2, then x<=y yields 0. If x = 1 and y = 2, then x<=y yields 1.
Logical AND: The result is 1 if both arguments are non-zero, and 0 if one is zero.*	&&	2	0 or 1	1&&5 yields 1. 0.21&&2 yields 1. 0&&2 yields 0.

Operator/ Description	Symbol	Number of Arguments	Output Type	Examples
Logical OR The result is 1 if one argument is non-zero, and 0 if both are zero.*	||	2	0 or 1	1||0 yields 1. 0||0 yields 0.
Logical NOT: The result is 1 if the argument is zero; otherwise, it is 0.*	!	1	0 or 1	!4 yields 0. !0 yields 1. !0.0004 yields 1.
Unary minus: The result is negative if the argument is positive, and vice versa.*	-	1	Same as argument	-4.5 yields -4.5. -4 yields -4. (-4) yields 4. -5+5 yields 0.
Bitwise left shift: Shifts the first operand to the left by the number of positions specified in the second operand.*	<<	2	32-bit long Integer	8<<2 yields 32.
Bitwise right shift: Shifts the first operand to the right by number of positions specified in the second operand.*	>>	2	32-bit long integer	8>>2 yields 2.

* The arguments are truncated to integers before evaluation.

Table 2: Operator Details

1.7. Mathematical Functions

The following table lists the built-in mathematical functions of the Gold Interpreter language. The function names are case-sensitive.

Function	Description	Returns
sin	`y=sin(x)` Here x is in radians. For example, the angle 30° is approx. 0.523598775598 radians (30 * (2 * π / 360)). The result of `sin(0.523598775598)` is 0.5.	Floating-point number
cos	`y=cos(x)` Here x is in radians. For example, the angle 60° is approx. 1.047197551196 radians (60 * (2 * π / 360)). The result of `cos(1.047197551196)` is 0.5.	Floating-point number
abs	`y=abs(x)` Returns the absolute value of the input argument. **Note:** The absolute value of the hexadecimal input argument 0x80000000 exceeds the long value range and will therefore be limited to the maximum long value for positive numbers. For example, the results for abs(-123.54) is 123.5400, and the for abs(123.54) is 123.5400.	Same type as the input argument
sqrt	`y=sqrt(x)` Square root, or zero if the argument is negative. For example, the results for `sqrt(16)` is 4.00, the result of `sqrt(18)` is 4.242640, the result of `sqrt(-625)` is 0, and the result of `sqrt(-81)` is 0.	Floating-point number
fix	`y=fix(x)` Truncates the input argument to an integer. For example, the result of `fix(3.8)` is 3, and the result of `fix(-3.8)` is –3. **Note:** If the input argument exceeds the long value range, it will be limited to the maximum long value (for positive numbers) or the minimum long value (for negative numbers).	Integer

Function	Description	Returns
rnd	`y=rnd(x)` Truncates the input argument to the nearest integer. For example, the result of `rnd(3.8)` is 4, the result of `rnd(-3.8)` is –4, and the result of `rnd(3.4)` is 3. **Note:** If the input argument exceeds the long value range, it will be limited to the maximum long value (for positive numbers) or the minimum long value (for negative numbers).	Integer
sign	`y=sign(x)` Returns the sign of the input argument: **-1** for negative numbers, **1** for positive numbers, **0** for a zero. For example, the result of `sign(-3.8)` is -1, the result of `sign(3.8)` is 1, and the result of `sign(0)` is 0.	Integer
real	`y=real(x)` Converts an integer to a float. If the argument is a floating-point number, the function does nothing: The result of `5/2` is 2, the result of `real(5)/2` is 2.5, and The result of `5/real(2)` is 2.5.	Floating-point number

Table 3: Built-in Mathematical Functions

1.8. Expressions

An *expression* is a combination of operands (parameters) and operators that is evaluated to a single value. Expressions work with immediate numbers, with drive commands and with drive and global user-program variables. The following sections describe the different types of expressions.

1.9. Simple Expressions

A simple expression is evaluated to a single value. Any parameter and mathematical/logical operator may be used to create a simple expression. Normally, simple expressions may be used as a part of other types of expressions.

Simple expressions are evaluated according to the operator priority, as specified in the table Mathematical and Logical Operators. In case of equal priorities, the expression is evaluated from left to right. The use of parentheses is allowed to 16 nesting levels.

The following table presents some examples of simple expressions.

Command	Line Results	Remarks
SP*2/5+AC	101000	The order is ((SP*2)/5) + AC
IP|5	OR operation on IP	
2 + 3	5	
1400000	1400000	

1.10. Assignment Expressions

Assignment expressions are used to assign a value to a variable or to a command. The syntax of an assignment expression is:

```
<parameter or command name>=<simple expression>
```

The following lines are examples of assignment expressions:

```
SP=SP*2/5+AC
OP=IP|5
```

If the variable or the command is a vector, the assignment is allowed only for a single member. The syntax of the vector member assignment is:

```
<parameter or command name>[index]=<simple expression>
```

The index is the index of the relevant vector member. Indices are enumerated from zero.

Examples:

```
CA[1]=1
UI[2]=abs(PX*2)
```

Be aware that when different types are assigned, the value may be truncated. If, for example, the variable or command type is integer and the assigned value is floating-point number, the floating-point value is rounded to the nearest integer. If a rounded integer value exceeds the integer range, this value is truncated to the nearest valid integer.

Expression Sent	Response Received	Remarks
AC=12345.6789	-	The expression assigns a floating-point value to the integer **AC** command.
AC	12346	The floating-point value is rounded to nearest integer.
KV[10]=215.789e8	-	The expression assigns a floating-point value to the integer **KV[10]** command.
KV[10]	2147483647	The floating-point value is truncated to the maximum integer value.

When an integer value is assigned to a floating-point command or variable, it is converted to a float. The conversion process may be imprecise due to the truncation into the IEEE float format.

Example

A floating-point variable **temp** is defined in a user program.

Expression Sent	Response Received	Remarks
TC=1		An integer value is assigned to the floating-point command **TC**.
TC	1.0	The assigned integer value is converted to a float.
temp=12345678		An integer value is assigned to a floating-point variable.
temp	1.234568e+7	The assigned value is truncated to 12,345,680.0.

1.11. User Variables

User variables are defined within a user program. The description and syntax rules of defining variables are given in *User Programming Language*.

Global variables defined in a user program may be used within the command line only if the program was compiled successfully and downloaded to the drive. The user may then use the Interpreter to query a user variable value or change it.

Variables may be either of type integer or of type float. An integer variable holds a 32-bit signed integer, while a float variable holds a number with a decimal point, which is interpreted as IEEE single-precision format.

Variables may be of scalar type or of vector (array) type. Only a single dimension vector is allowed.

A **vector** (array) type will always be declared as a **global** variable.

The user should pay special attention to the scope of a variable. A variable may be defined at the *global* or *local* level. Local variables are available only within the function in which they are defined, while global variables are available within any function and also outside a program.

A user variable may be queried or changed when the program is running or halted.

For example, suppose that a compiled program includes the following lines at the global level:

```
int Filt, Carr[3]
float MyResult

function main()
Filt = 100
Carr[1] = 30
Carr[2] = 25
MyResult=Filt*Carr[1]+2*sin(Carr[2])
return
```

The expression `MyResult=Filt*Carr[1]+2*sin(Carr[2])` is valid.

User program variables are case-sensitive.

Note: In case of a scalar variable (global) which is handled like it was a vector variable, the compiler will not report about any error. The program will be terminated if the index in that case is greater than zero.

Examples

Example 1

```
int a1
function main()
global int a1
a1[0]=2          // Program will run successfully.
return
```

Example 2

```
int a1
function main()
global int a1
a1[1]=2        // Program will be terminated here.
return
```

1.12. Built-in General Purpose Function Calls

The Gold Interpreter language has a group of internal general-purpose functions. This group of functions, like the built-in mathematical functions, can be used by accessing the Interpreter and/or through a user program.

A built-in function call may be used in a single expression. The names of these functions are case-sensitive.

The following table describes the non-mathematical built-in functions.

Function	Description	Return Value
tdif(*x*)	Returns the time difference between the specified time and the current time. `x=TM` The function **tdif(*x*)** returns the time in milliseconds since *x* **= TM** was sampled.	Integer, milliseconds (msec)
prgerr(0)	Returns the last program error of the virtual machine. This function can be used in an AUTO_PERR routine to query information about the recent failure.	Integer, msec
tick(*n*)	Reads the system time in milliseconds. This function uses an internal hardware timer unrelated to the system software counter used by the **TM** command (refer to the **TM** command section in the Command Reference for Gold Line Drives). As such, the **TM** command timer can be modified by **CAN SYNC** And **Time Stamp**, while the tick function timer is not affected by any external event. **Example:** `x=tick(0)`	Integer, msec
tock(*n*)	Returns the time difference. If *internal* = **tick(**0**)**, **tock(*internal*)** returns the time in milliseconds since *internal* = **tick(**0**)** was sampled. The **tock** function operates in a manner similar to **tdif**, but it uses an internal hardware timer that is unrelated to the system software counter used by the **TM** command and the **tdif** function. **Note:** For a time difference greater than 32 seconds, the **tock** function may return an erroneous result.	Integer, msec

Function	Description	Return Value

	The maximum count supported is 28.5 seconds.
	Example:
	`x=tick(0)`
	`...  // measured period`
	`y=tock(x)`	

utick(*n*)	Reads the system time in microseconds (µsec). The **utick(*n*)** function uses the same internal hardware timer that is used by the **tick(*n*)** and **tock(*n*)** functions, except that the return value is in microseconds. **Example:** `x=utick(0)`	Integer, µsec

utock(*n*)	Returns the time difference.
	If *internal* = **utick(**0**)**, **utock(***internal***)** returns the time in microseconds since *internal* = **utick(**0) was sampled.
	The **utock** function operates in a manner similar to **tdif**, but it uses an internal hardware timer that is unrelated to the system software counter used by the **TM** command and **tdif** function.
	The maximum count supported is 28.5 seconds.
	Example:
	`x=utick(0)  //first time sampling`
	`...  // measured period`
	`y=utock(x)  //get time difference`
		Integer, µsec

Table 4: Built-in Mathematical Functions

A built-in function call may be a part of a single expression.

Examples

```
sin(3.14/3)
AC=abs(DC)
SP=SP+sin(3.14/2)
```

1.13. Time Functions

Each Gold servo drive holds a software free-run system timer that is derived from the CPU hardware timer and is incremented upon every real time interrupt. The system-timer resolution is (unsigned) 32-bit microseconds, which rolls over every ~71.5 minutes. The timer software time might be affected by an external host.

The **TM** command is used to read the system's 32-bit microsecond counter.

The system timer is a software implementation timer handled by the real time task of the Gold drive.

TM returns a time value in microseconds (µsec).

The time difference from the present time to an earlier sampling of **TM** can be determined using two methods, as in the following two examples.

Example 1:

```
UI[1] = TM          // UI[1] is used just as storage
...                 // Do something
UI[2] = TM-UI[1]    // UI[2] is time difference in microseconds
```

Example 2:

```
UI[1] = TM            // UI[1] is used just as storage
...                   // Do something
UI[2] = tdif(UI[1])   // UI[2] is time difference in milliseconds
```

Time differences can be no longer than 71.5 minutes. To pause for a given time in a user program, use the **wait(*n*)** function (see *Program Delay Command*).

The **tick**/**tock** and **utick**/**utock** readings are not affected by any external event and, therefore, give accurate time difference measurements between events.

The time difference between the present time and an earlier **tick** sampling can be determined by one of two methods, as shown in the following examples.

Example 1:

```
QP[1] = tick(0)       // QP[1] is used as storage only
...                   // Do something
QP[2] = tick(0)-QP[1] // QP[2] is the time difference in microseconds
```

Example 2:

```
QP[1] = tick(1000)    // QP[1] is used as storage only
...                   // Do something
QP[2] = tock(QP[1])   // QP[2] is the time difference in milliseconds
```

Example 3: Time measurement in milliseconds using the TM command

```
int timemes
int timedif

timemes = TM             // get current time (microseconds)
wait(1234)               // delay  1234 msec (milliseconds)
timedif = tdif(timemes)  // measure the time since the last time (TM)
                         // tdif() measures the new time and returns the
                         // difference in milliseconds
```

Example 4: Time measurement in milliseconds using tick() and tock

```
int timemes
int timedif

timemes = tick(0)          // get the internal time in msec (milliseconds)
wait(4321)                 // delay 4321 msec
timedif = tock(timemes)    // timedif gets the time passed from last
tick()                     // measurement (in milliseconds).
```

Example 5: Time measurement in microseconds using utick() and utock()

```
int timemes
int timedif

timemes = utick(0)       // get the internal time in µsec (microseconds)
wait(12)                 // delay 12 msec
timedif = utock(timemes) // timedif gets the time passed from last utick()
                         // measurement (in microseconds).
```

The **tick(0)**, **utick(0)**, **tock()** and **utock()** functions cannot measure time differences greater than 28,633 milliseconds or 28.6 seconds. Beyond that period of time, **tock()** and **utock()** return an error value.

1.14. Comments

Comments are text written into the code to enhance its readability. A comment starts with a double slash (//) or double asterisk (**) and terminates at the next end of line. The drive ignores comments when evaluating an expression. The Interpreter handles comments from the user program only.

Example

```
// This is a comment. The driver ignores this.
```

Chapter 2: User Programming Language

2.1. General

A user program is high-level code, which the user can write and execute in a Gold drive.

The code can include most of the interpreter commands (for example, **motion**, **status**) and functions (for example **sin()**) and allows auto routines, which provide a means to interrupt the user program code and jump to special event routines (for example, after digital input is set or in case of a fault).

Using the Elmo Application Studio (EAS) program editor, the user program can be edited, downloaded, debugged and even attached for debugging purposes during execution.

The user program runs in the background task of the drive. This means that it has the same precedence as the interpreter and allows the execution of user commands with no conflicts, regardless of the communication source.

Commands that are in the same user program line are completed before the next command line is processed.

The following chapter describes the properties, usage abilities, execution and limits of user programs.

2.2. Structure and Timing

A user program is built up from three parts:

- Text code
- A symbol table
- Operation code

Using the EAS program editor, the user writes a text program. The program is then compiled by the EAS into a list of op-codes. Basically, each text line is converted into a list of op-codes, where the EOL (End-of-Line) op-code terminates a single execution line. The op-codes are executed by the drive, using the virtual machine process.

The virtual machine is called in every background cycle and runs the op-codes sequentially. To prevent starvation of the idle loop, each op-code cannot be executed for more than 3 msec (this is just a timeout to prevent background cycles that are too long).

The user program is downloaded to the non-volatile memory of the drive. After a successful download, the EAS sends a **CC** command, which loads the program into the RAM, from which the program is executed.

When the program is executed (by the **XQ##** command), each line in the program gets CPU time from the background process to be performed fully until the EOL code.

The background is a non-deterministic loop. As a result, the execution of the user program is non-deterministic and can jitter between 100 µsec and 1 msec.

Note that background time is influenced by the sampling time defined by the **TS** command as well as by the communication load. A higher sampling time means faster user program execution. Thus, a higher sampling time affects the servo performance.

2.3. Keywords

User programs use the following keywords, which are protected by the compiler and cannot be declared as names of user variables. These words are colored by the EAS programming editor.

break	catch	case	continue	end	else
elseif	end	exit	for	float	function
global	goto	if	int	otherwise	reset
return	switch	try	until	wait	while

2.4. How to Write a Program

A program is built from functions. A function must have a function call and a function definition.

Functions may also have an optional function declaration.

A program can start directly with a function, as in the following example.

```
function main()
myfunc()
return

function myfunc()
int myval
myval = 1+2
return
```

In this example the function **main()** is used as the entry point, and it calls the function **myfunc()**, which sums 1+2 and returns nothing.

The user must set the **Entry Point** in the EAS editor to the required function in order to run it. In the above example, the **Entry Point** should be set to **main()**.

The function **myfunc()** can also be called using any interpreter communication terminal in the following manner.

XQ##myfunc

For more information about the **Entry Point** setting, refer to Elmo Application Studio (EAS) User Guide.

For compatibility reasons, labels, for example, **start** can also be used, as in the following example.

```
##start
myfunc()

function myfunc()   // begin the body of the function myfunc
...                 // lines in the body of the function main
return              // end the body of the function myfunc
```

Note: We strongly recommend avoiding the use of labels, and do recommend using functions. All the examples henceforth include functions only.

In this case, after the label (**##start**), a function call is written in order to run the function **myfunc**.

As can be seen in the example, all labels start with the characters **##** (e.g., **##start**).

The character pair **#@** is also used for declaring auto-routines (refer to the chapter *Auto-Routines*).

2.5. User Functions

Simple functions start with the keyword **function** and end with the keyword **return**.

```
function function1()  //begin the body of the function function1
...
return                //end the definition of the function function1
```

The following is an example of a simple function, which uses the **MO** and **SO** commands.

```
function motor_on()
MO=1
until (SO==1)
return
```

The following example shows how to create and call a function.

```
function  main()    //begin the body of the function main
motor_on()          //call the function motor_on
return              //end the definition of the function main

function motor_on()
MO=1
until (SO==1)
return
```

In a function that receives arguments (input arguments), the arguments are declared in the first line of the function.

In the example below, *arg1* is an integer type input argument, and *arg2* is a float type input argument.

```
function myfunction (int arg1, float arg2)
...
return
```

In functions that return results, the output argument (return value) is declared with its type within square brackets after the keyword **function**, as in the following example.

```
function [ int x ] = myfunction( )
...
return
```

The following is an example of a function with two input arguments and two return values:

```
function [ int x, int y ] = myfunction ( int a, int b )
...
return
```

2.6. Variables and Types

A variable must first be declared before it is used (in an expression or an assignment).

A variable definition line consists of type names (**int** or **float**) and variable names. Variables may be scalar quantities for example, `int var1, float temp,` or one-dimensional arrays for example, `int arr[10], float ftemp[4]`.

Note: It is not recommended to use variable of two letters (XX, YY, DG[2] etc) since this may be overwritten by the drive's native command. For example: int AC is forbidden and the compiler will alert. However, int XX might be allowed in some versions and might be overwritten by drive's XX command in other versions.

```
#start  // a label. User can run this program from a terminal by entering:
// XQ##start
...
main()    // call the function main

function main()  // begin the body of the function main
int myvar
int myarray[5]

myarray[1] = 10
myvar = myarray[1] * 2
return
```

Two or more variables in the same definition line must be separated by commas. Alternatively, each variable may be declared in a separate line.

If a variable is a vector, it must be declared with its dimension in brackets after its name.

```
int z[10]
float w[2]
```

The vector dimension must be a positive constant number. If the dimension is defined as a floating-point number, it will be truncated to an integer.

```
int z[-10]     // WRONG, not a valid dimension
float w[2.8]   // dimension will be truncated to 2
```

A dimension of less than 1 is illegal.

```
int z[0]    // Illegal dimension
float w[2]
```

2.6.1. Variable Names

The names of variables may include ASCII letters, digits (not leading) and underscores (not leading) only. Variable names are case-sensitive. The maximum variable name length is 11 characters. A variable name cannot be a keyword.

The following are examples of variable names:

```
px1          // legal name
pxa          // legal name
px_          // legal name
ab1234       // legal name
1px          // illegal name
_px          // illegal name
```

2.6.2. Global Variables

Global variables are declared outside and before the function that uses them.

A function which uses a global variable must have the protective keyword **global** in a second declaration of the variable at the beginning of the function, as in the following example.

Global Variables cannot be addressed when the Binary Interpreter is used under CANopen communication. When the global variable contains two letters (e.g. GG, HK[2], etc), the Elmo Application Studio (EAS) will not be able to address this variable. However, any global variable can be addressed via the CANopen OS interpreter.

```
int  error        // first declaration of global variable
float  yy[20]

main()            // calling the function main()

function main()
global int error  // global declaration for this function's use

error = 0         // set the global variable error
check()           // call the function check()
return

function check()
global int error  // global declaration for this function's use
int stepping      // local variable
...
error = 3         // set global variable  "error"
...
return
```

2.7. Program Development and Execution

The process of program development a Gold drive includes the following steps:

- **Editing**: Writing and/or revising the program.
- **Compiling**: Using the compiler to process the program and find errors.
- **Loading**: Loading the program to the flash memory of the drive.
- **Debugging**: Observing the behavior of the program and correcting it where necessary.
- **Running** the program.

The Elmo Application Studio includes all the tools needed to perform this procedure. Using the Elmo Application Studio is fully explained in *The Elmo Language*.

2.7.1. Editing a Program

The drive program is written in simple text using any text editor. The Elmo Application Studio is recommended for program editing, because it provides several additional services, such as downloading the program to the drive, compiling the program and running it.

2.7.2. Compiling a Program

Each user program must be compiled after editing. Although the compiler does not reside in the DSP software, it is described here, because it is an integral part of the program development process. The compiler in the Elmo Application Studio is external, stand-alone software that can be accessed through the Composer software. The user can write and compile a program in off-line mode (without establishing communication with the drive) and then use the compiler to compile the program in order to produce address maps and run-time code.

If, in the course of compilation, the compiler finds syntax errors, it stops the compilation process and informs the user about the errors, presenting them in a convenient form.

The compiler is composed of a preprocessor and a code generator. The preprocessor evaluates pragmas and constant expressions. It is described in *The Preprocessor*.

The compiler accepts the user program as a text file and files with targeted Gold drive information. This information is required in order to ensure that the compiled code can run on the Gold drive.

Although the compiler can locate syntax errors, it cannot find the following:

- Out-of-range command arguments
- Bad command contexts, such as an attempt to begin a motion when the motor is off.

These errors must be corrected in the debugging stage.

2.7.3. Error Codes

Various errors can be detected when a program is compiled, debugged or executed. The following table lists the applicable error codes, the accompanying error strings, expanded explanations of their meanings and examples.

Error Code	Error String	Meaning	Example
0	No errors	Successful compilation without errors.	0

1	Bad format	General error for bad syntax in a right- or left-hand expression.	`for k = 1::10`
			Double colon between 1 and 10.
			`b = a + ()`
			Empty expression in parentheses.
			k = 1:2:20
			Colon expression used outside a **for** statement.

2	Empty expression	Expected right-hand expression is missing.	`a =` Right-hand expression is missing after the assignment operator.
3	Stack is full	Stack has overflowed its capacity.	

4	Bad index expression	An index expression of a variable is not evaluated to a single value.	`a(2,3)`
			The result of evaluating the expression in parentheses is two values, not a single value.
			`a()`
			The index expression is empty.

5	Bad variable type	The expected variable type is neither float nor int. This error may appear either after the **global** keyword or after input/output arguments in a function definition.	`global floa a`
			The variable type is expected after the **global** keyword. The type floa is unknown.
			`function func (long a)`
			The input argument type is expected in parentheses.

Error Code	Error String	Meaning	Example
			Here, the type long is unknown.

Error Code	Error String	Meaning	Example
6	Parentheses mismatch	The number of opening parentheses does not match the number of closing parentheses. This applies to both parentheses and square brackets.	`b = a(1))` There is an unneeded closing parenthesis.
7	Value is expected	A right- or left-hand expression is not evaluated to a single value, or it has failed during evaluation due to a bad syntax expression.	`b = ^ a` A value is expected before the ^ operator.
8	Operator is expected	During right-hand expression evaluation, there is no operator or terminator of a simple expression after successful value evaluation.	`b = a c` After successful evaluation of a, an operator or expression terminator is expected, but c is not recognized as either an operator or terminator.
9	Out of memory in the data segment	During memory allocation for a global variable, there is segment.	

10	Bad colon expression	Error during evaluation of a colon expression. The colon expression may appear only in a **for** statement. Bad syntax: more than three values or fewer than two values in the colon expression may also cause this error.	`for k = 10:-1:5:9`
			Colon expression contains more than three values.
			`for k = a`
			Expected colon expression is missing in the **for** statement after the assignment operator (=).

11	Name is too long	Variable or function name exceeds 12 characters.	`int iuyuafdsf_876234`
12	No such variable	Left-hand value is not recognized as either a variable or as a system command.	`de = sin(0.5)` Here de is neither a variable nor a function.
13	Too many Dimensions	Dimension of an array exceeds the maximum admissible number of dimensions (syntax allows only one-dimensional arrays).	`int arr[12][2]` An attempt was made to define a two-dimensional array.

Error Code	Error String	Meaning	Example

14	Bad number of input arguments	The number of input arguments during a function call does not match the number of input arguments in the function definition.	`function func(float a)`
			`...`
			`func(a,2)`
			The number of input arguments during the function call is two, while the function func is defined with only one input argument.

15	Bad number of output arguments	Bad syntax in the left-hand expression: multiple output without brackets, or multiple output that exceeds the maximum admissible number of outputs (a maximum of 16 outputs is allowed).	`a, b = func(1,2)` Multiple outputs must be within brackets.
16	Out of memory	Compilation process ran out of memory. This error may occur if the user program is too large or too complex and there is not enough space in the code segment or in the symbol table.	
17	Too many arguments	The number of input or output arguments exceeds the maximum admissible number of input or output arguments (16).	`function func (int a1, int a2, int a3, int a4, int a5, int a6, int a7, int a8, int a9, int a10, int a11, int a12, int a13, int a14, int a15, int a16, int a17, int a18, int a19,` The number of input arguments exceeds 16.
18	Bad context	The compiler has found an error in the program context, such as mismatched brackets/parentheses or an improperly closed flow-control statement.	
19	Write file error	An error occurred while writing to a file.	

Error Code	Error String	Meaning	Example
20	Read file error	An error occurred while reading from a file.	
21	Internal compiler error: bad database	A corrupted database has caused an internal compiler error.	

22	Function definition is inside another function or flow control block	An illegal function definition.	`if a<0`
			`a = 0`
			`function func (int a)`
			`end`
			Attempt to define a function inside an **if** block.

23	Too many functions	The user program contains too many functions, and there is not enough space for them in the database.	
24	Name is keyword	A variable or function has the same name as a keyword. This error may occur if a variable name is identical to an auto-routine name.	`int switch` The word switch is a keyword, so its use as a variable name is illegal.

25	Name is not distinct	A variable or function name is not unique.	`int func`
			`function func (int a)`
			The function and the variable have the same name.
			`function func (int a)`
			`int a`
			The definition of the local variable a is illegal because this function already contains a local variable a as an input argument.

Error Code	Error String	Meaning	Example

26	Variable name is invalid	This error occurs when:
		A variable or function name starts with a digit or underscore, not with a letter.
		A variable or function name is empty.
		On the variable definition line, a comma is used as a separator between variables, but the variable name after the comma is missing.
			`int _abc`
			The variable name has a leading underscore.
			`function (int a)`
			The function name is missing after the function keyword.
			`int a, b,`
			The variable name is missing after the comma.

27	Bad separator between variables	The only legal separator between variables on the variable definition line is a comma. After a variable name, either a variable separator (comma) or an expression terminator is expected. Any other symbol causes this error.	`int a b` A comma as a variable separator is missing between a and b.

28	Illegal global variable Definition	A global variable must be declared inside a function with the **global** keyword, and it must be defined before the function. This error appears only if the **global** keyword is used in the wrong context: The **global** keyword was used outside the function.
		A variable declared as global inside a function is not defined previously.
		The type of variable declared in the definition outside the function differs from the type of the declaration inside the function.
			`int a1`
			`function func (int a)`
			`global float a1`
			The variable type of a1 in its definition is int, while inside the function, it is declared as float.

Error Code	Error String	Meaning	Example

29	Bad variable definition	All local variables must be defined at the beginning of the function. Any variable definition coming after executable code in the function is illegal.	`function func (int a)`
			`int b`
			`global int a1`
			`b = a`
			`float c, d`
			The definition of float variables c and d is illegal because it occurs after executable code.
			`b = a`

30	Variable is undefined	The iteration variable in a **for** statement is not defined before it.	`function func (int a)`
			`for k = 1:10`
			`a = k:`
			`end`
			`return`
			The iteration variable k is not defined before its use.

31	Bad separator between dimensions	Bad separator between dimensions (not a comma). This error is unused, because currently only one-dimensional arrays are allowed, so there is no need for a separator between dimensions.	
32	Bad variable dimension	The legal variable dimension must be a positive number inside square brackets. An expression inside square brackets that is not evaluated into a number or is evaluated as a number that is less than 1 (zero or negative) is illegal.	`int arr[-12]` Variable dimension is negative

33	Bad function format	Appears at function definition when:
		The function name is not unique or is empty. The function definition does not match its prototype.
			`function func (int a)`
			`function func (float a)`
			Type of input argument in the function definition does not match the type in the prototype.

Error Code	Error String	Meaning	Example

34	Illegal minus	A minus sign is illegal before a function call with multiple output arguments, and before parentheses that include multiple expressions.	`[ab] = -func(c)`
			Illegal minus sign before a function call with multiple outputs.
			`-(2+3), c/5)`
			Illegal minus before multiple expressions within parentheses.

35	Empty program	The user program is empty.	
36	Program is too long	The user program exceeds the maximum admissible length.	
37	Bad function call	An attempt has been made in a **goto** statement to jump to a function with a non-zero number of input or output arguments.	`[a,b,c] - func(x,y) + 5` Illegal statement. The compiler checks whether there is an expression terminator immediately after the function call with multiple outputs. If there isn't, it issues this error.

38	Expression is expected	The expected expression —**wait**, **until**, **while**, **if**, **elseif**, **switch** or **case** — is missing.	`if`
			`a = 0`
			`end`
			An expression expected after the **if** is missing.

39	Code is too complex	The user program contains very complex code that includes too many nested levels (this expression actually contains more than 100 nested levels). A nested expressions means that there is one flow-control block inside another.	An if block inside a while loop
40	Line compilation is failed	A general error has occurred during an attempt to compile an expression.	

Error Code	Error String	Meaning	Example

41	Case must follow switch	After a **switch** statement, the only legal statement is **case**; otherwise, this error occurs.	`switch a`
			`b = 0`
			`case 1,`
			`b = 1`
			`end`
			The expression `b=0` is illegal because **switch** must be followed by **case**.

42	Illegal case after otherwise	The **otherwise** statement must be the last statement of a **switch** block. Any **case** after **otherwise** is illegal.	`switch a`
			`case 1,`
			`b = 1`
			`otherwise`
			`b = 0`
			`case 2,`
			`b = 1`
			`end`
			The statement `case 2` is illegal because **otherwise** must be the last statement in a **switch** block.

43	Bad nesting	Flow-control block contradiction: **else** without **if**, mismatched **end**, flow-control block without **end**, etc.	
44	Code is not expected.	There is some unexpected code for evaluation after **otherwise**, **end**, **else** etc.	

45	Bad flow control expression	No assignment operator (=) sign after the iteration variable name in a **for** statement.	`for k`
			`a = 0`
			`end`
			The assignment operator (=) is missing after the k.

46	Too many errors	The compilation error buffer is full.	

Error Code	Error String	Meaning	Example

47	Expression is out of Function	A function or label must contain executable code; otherwise, this error occurs.	`int a1`
			`a1 = 0`
			`function func (int a)`
			The executable code is illegal because it is outside the function.

48	Otherwise without any case	An **otherwise** appears immediately after a **switch** statement.	`switch a`
			`otherwise`
			`b = 0`
			`end`
			After a **switch** statement, **case** is expected before **otherwise**.

49	Misplaced break	**Break** is legal only inside a **switch**, **for** or **while** block; otherwise, this error occurs.	`If a < 0`
			`break`
			`end`
			Using **break** in an **if** statement is illegal.

50	Too many outputs	The number of actual output arguments during a function call exceeds the number of output arguments in the function definition.	`function [int b] = func (int a)`
			`...`
			`return`
			`...`
			`[c,d] = func(a)`
			The function call is illegal because the number of outputs is two, while this function is defined with a single output argument.

51	Line is too long	The user program contains a line with more than 128 characters.	
52	Clause is too long	The user program contains an expression for evaluation that contains more than 512 characters. This expression may take several lines of user program text.	
53	Cannot find end of sentence	End of sentence not found within range.	

Error Code	Error String	Meaning	Example
54	Open file failure	There has been an attempt to open a non-existent file; the file name or path may be incorrect.	
55	Bad file name	The full file path is too long.	
56	No such function	An auto-routine, user function or label name must follow the keyword in a **goto** or **reset** statement; otherwise, this error occurs.	

57	Variable is array	An attempt has been made to assign an entire variable array, rather than a single member.	`int arr[10]`
			`##START`
			`arr[1] = 0`
			`arr = 0`
			The last expression is illegal because it tries to assign the entire array.

58	Variable is not array	An attempt has been made to assign a scalar variable according to an index, as an array.	`int a1 // (a scalar)`
			`##START`
			`a1[1] = 0`
			The last expression is illegal because it tries to assign a scalar variable according to an index.

59	Mismatch between left and right-hand side expressions	The number of left-hand values does not match the number of values in the right-hand side of the expression.	`[a,b] = 12 + c` The number of values on the left is two, while the number of values after evaluation of the right-hand expression is 1.

60	Illegal local array	Syntax does not allow definition of a local array. The array must be global.	`function func (int a)`
			`int arr[10]`
			`...`
			`return`
			A local array is illegal.

Error Code	Error String	Meaning	Example

61	Function already has body	A function has more than one body.	`function func (int a)`
			`wait 2000`
			`return`
			`...`
			`function func (int a)`
			`until a`
			`end`
			The code is illegal because the function func has multiple bodies.

62	Opcode is not supported by Gold drives	The specified version of the Gold drive does not support a certain virtual command.	
63	Internal compiler error		
64	Expression is not finished	The user program contains an unfinished statement: an ellipsis (…) may be missing to indicate that the expression is continued on the next line.	The last line of user text may be: `a = b + 8 / 12 /(8^2*sqrt(2) – sin (3.14/2))...` After the ellipsis, another line should appear, but in this case it is missing.
65	Compiled code is too long	The compiled code exceeds the maximum space for the code segment in the Gold drive's serial flash memory.	
66	Corrupted the Gold drive setup files	The file containing the Gold servo drive setup parameters is not in the defined format.	
67	Too many variables	The user program contains too many functions, and there is not enough space for them in the database.	
68	Variable name length mismatch to the Gold drive setup	The allowed variable name length does not equal the length defined in the compiler.	

Error Code	Error String	Meaning	Example
69	Auto-routine has argument	The auto-routine cannot have any input or output argument; otherwise, this error occurs.	`function AUTOEXEC (int a)` A definition of an auto-routine that includes an input argument is illegal.
70	Label definition is inside flow control block	A label cannot be defined inside a flow-control block; otherwise, this error occurs.	
71	Function without return	The function does not end with the **return** keyword.	
72	Block comments is not finished	The comment block has no end.	Assume that the last line of user program text is: `/* Just a comment` The comment block is not closed.
73	Bad function after reset	A **reset** statement must contain an auto-routine, a global label or a user function without input arguments; otherwise, this error occurs.	`reset func(12)` A user function with an input argument comes after the **reset** keyword.

74	Bad jump to label	Occurs when an attempt is made to jump to:
		A global label from within a function
		A local label from with a global space
		A global label from within a global space at a **goto** statement
		A non-label at a **goto** label
		A local label at a **reset** statement
			`function func ()`
			`...`
			`return`
			`...`
			`goto##func`
			The last expression is illegal because func is not a label.

75	Illegal nargout	The **nargout** keyword is used outside a function.	`##START`
			`if nargout > 2`
			The keyword **nargout** is used outside a function.

76	Function without body	An attempt has been made to call a function that has been defined but does not have a body.	

Error Code	Error String	Meaning	Example

77	Bad goto statement	The **goto** keyword must be followed by ## or #@ before the name of a label; otherwise, this error occurs. This error may also occur if there is a **goto** statement within the body of a **for** loop.	`goto START`
			There is no ## or #@ between **goto** and the label name.
			`for k = 1:10`
			`...`
			`goto##START`
			`...`
			`end`
			The **goto** statement in the **for** loop is illegal.

78	Auto routine is local	An auto-routine is defined as a local label.	`function start (int a)`
			`...`
			`#@AUTOEXEC`
			`...`
			`return`
			The AUTOEXEC auto-routine is defined inside a function as a local label.

79	Command has 'not program' flag	The program refers to a command that is not allowed to be used within a user program.	`LS` The program attempts to use the **LS** command, which has a "Not program" flag defined for it.
80	Image file too long	The Image file length exceeds the user code partition size.	
81	System function tdif is not supported by the Gold drive	During evaluation of the **wait** flow-control, the **tdif** system function must be defined inside the Gold drive; otherwise, this error occurs.	
82	Command has "not assign" flag	The program has assigned a value to a command with a "not assign" flag.	`BG=10` The **BG** command has a "not assign" flag. Assigning a value to this command is illegal.
83	Keyword is not implemented - for future use	An attempt has been made to use an unimplemented keyword or feature.	

Error Code	Error String	Meaning	Example

84	Misplaced return	Illegal return, such as from a **try…catch** block	`try`
			`. . .`
			`if a > 6`
			`return`
			`end`
			`catch`
			`. . .`
			`end`
			The **return** keyword is used inside the **try** block.

85	The try and catch keywords must be on a separate line	Any code on the same line with **try** or **catch** is illegal.	`try, AC=10000000`
			`. . .`
			The statement after the **try** keyword is illegal.

86	Division by zero	Division by zero.	
87	Identifier is missing	The **#define** directive does not contain an identifier.	`#define` The identifier is missing.

88	The name of identifier is not valid	The name of the identifier of the **#define** directive is not valid.	`#define 2PI 6.24`
			`#define VER+`
			The first name has a leading number. The second name contains the + character. A valid name may contain only letters, numbers or underscores.

89	Identifier was defined before	An attempt has been made to use a **#define** directive with the same identifier.	`#define NUM 3`
			`. . .`
			`#define NUM 4`
			The second statement is illegal because NUM has already been defined.

90	Condition is missing	The condition is missing in an **#if** or **#elseif** directive.	`#if` A condition must follow the **#if** directive.

Error Code	Error String	Meaning	Example
91	Misplaced continue	The **continue** keyword is used outside a **for** or **while** loop, or inside an unclosed **try…catch** block.	

92	Misplaced #else or #elseif	An **#else** or **#elseif** directive has been used outside an **#if**-**#endif** block, or **#else** is used more than once in the same **#if**-**#endif** block, or **#else** is not the last directive before **#endif**.	`#if NUM > 10`
			`#elseif NUM > 5`
			`#else`
			`#elseif NUM > 3`
			`#endif`
			The **#else** directive is not the last directive before the **#endif**.

93	Identifier wasn't declared	The identifier of an **#undef** directive was not defined previously.	`#undef NUM` This statement is illegal if it is not preceded by a `#define NUM` directive.
94	Mismatched type	The type of an evaluated constant expression is neither an integer nor a float.	
95	Too many identifiers have been defined	The user has defined more than 100 **#define** directives in the program.	
96	Misplaced #endif	An **#endif** directive is not preceded by **#if**.	
97	Unknown error	Unknown error.	

Table 5: Error Codes

2.8. The Preprocessor

Prior to code generation, the compiler preprocesses the user code, handling compiler directives as described in *Compiler Directives*. The preprocessor does the following:

- Searches for definitions of literal constants, such as `#define MYConst 5`, and replaces literal strings in the user code with their values.
- Evaluates the conditional expression of **#if**, **#elseif**, **#ifdef** and **#ifndef** directives.
- Checks the conditions of **#if**, **#elseif**, **#else**, **#ifdef** and **#ifndef** directives, and, depending on the results, removes or retains the corresponding processing statements in the code.

2.9. Compiler Directives

2.9.1. #define

The **#define** directive may be used to assign a name to a literal string in a manner similar to that of the C language.

Syntax

```
#define  identifier  string
```

or

```
#define  identifier
```

The **#define** directive substitutes the defined string for all subsequent occurrences of the identifier in the text code.

The identifier is replaced only when it forms a real occurrence. For instance, the identifier is not replaced if it appears in a comment, within a string or as part of a longer identifier. For example:

```
//#define UNIT_MOD1 3
```

The above is a comment. There will **not** be an identifier with the value 3.

Note: The **#define** directive without an identifier is illegal.

A **#define** directive without a string is not replaced. The identifier remains defined and can be tested using the **#ifdef** and **#ifndef** directives.

The string argument consists of a series of tokens, such as keywords, constants or complete statements. One or more space characters must separate a string from an identifier. A legal identifier name may consist of a maximum of 32 characters. A valid name consists only of letters, underscores and numbers (but not leading a leading underscore or number, since a name must start with a letter). The name is case-sensitive.

The maximum admissible number of **#define** directives in a Gold drive is 100.

When the string is evaluated (refer to the section *Evaluating Expressions Used in Compiler Directives*), if the evaluation is successful, the identifier is replaced with the value obtained; otherwise, every appearance of the identifier is replaced with the string as a string.

The syntax that defines compiler directives in the Gold language differs from the C language in a number of ways.

The redefinition of a **#define** directive with the same identifier is illegal unless the second definition with **#define** appears after the first definition has been removed by the **#undef** directive.

Examples

The following line defines the identifier OPTION_A without associating a constant value with it.

```
#define OPTION_A
```

The following line defines the identifier MASK_AUTOEXEC as the constant 0x0001. Wherever the identifier appears, it is replaced by this value.

```
#define  MASK_AUTOEXEC  0x0001
```

2.9.2. #if

The **#if** directive checks a conditional expression, as in the C language. If the specified constant expression following the **#if** directive has a non-zero value, it directs the compiler to continue processing statements up to the next **#endif**, **#else** or **#elseif**. It then skips to the statement following the **#endif** directive. If the conditional expression has a zero value, **#if** directs the compiler to skip immediately to the next **#endif**, **#else** or **#elseif** directive.

Syntax

```
#if  constant-expression
```

The constant expression is either an integer or a floating-point number.

Each **#if** directive in a source file must be matched with a closing **#endif** directive; otherwise, an error message is generated.

The **#if**, **#elseif**, **#else** and **#endif** directives can be nested in the text portions of other **#if** directives.

Each nested **#else**, **#elseif** or **#endif** directive belongs to the closest preceding **#if** directive.

The **#if** directive must contain a constant expression, which is evaluated to a single value; otherwise, it causes an error.

Example

```
#if  MAX_LEN > 10
   #define  REDUCE_MAX_LEN  10
#endif
```

2.9.3. #else

The **#else** directive, as in C, marks an optional clause of a conditional compilation block defined by an **#ifdef** or **#if** directive.

Syntax

```
#else
```

The **#else** directive must be the last directive before the **#endif** directive. Only a single **#else** directive is allowed. The **#else** directive contains no conditions.

Example

```
#if  MAX_LEN > 10
   #define  EDUCE_MAX_LEN  10
#else
   #define  EDUCE_MAX_LEN  5
#endif
```

2.9.4. #elseif

The **#elseif** directive marks an optional clause of a conditional-compilation block defined by an **#ifdef** or **#if** directive.

Syntax

```
#elseif constant-expression
```

The directive controls conditional compilation by checking the specified constant expression.

If the expression is non-zero, **#elseif** directs the compiler to continue processing statements up to the next **#endif**, **#else** or **#elseif** directive and then to skip to the statement after **#endif**.

If the constant expression is zero, **#elseif** directs the compiler to skip to the next **#endif**, **#else** or **#elseif**.

Up to **50 #elseif** directives can appear between the **#if** and **#endif** directives.

As with the **#if** directive, the **#elseif** directive must contain a constant-expression, which is evaluated to a single value; otherwise it causes an error.

Example

```
#if  MAX_LEN > 30
     #define   REDUCE_MAX_LEN 30
#elseif   MAX_LEN > 20
     #define   REDUCE_MAX_LEN 20
#elseif   MAX_LEN > 10
     #define   REDUCe_MAX_LEN 10
#else
     #define   REDUCE_MAX_LEN 5
#endif
```

2.9.5. #endif

Each **#endif** directive must close an **#if** directive, in a manner similar to the C language.

Syntax

```
#endif
```

An **#endif** directive without a precding **#if** directive generates an error.

Example

```
#if  CALL_FUNC  1
  Call_func()
#endif
```

2.9.6. #ifdef

The **#ifdef** directive, as in C, checks for the presence of a specified identifier that is defined with **#define**.

Syntax

```
#ifdef  identifier
```

The **#ifdef** and **#ifndef** directives can be used anywhere that **#if** can be used.

An **#ifdef** identifier statement is equivalent to **#if** 1 when the identifier has been defined, and it is equivalent to **#if** 0 when identifier has not been defined or has been undefined with the **#undef** directive.

Example

```
#define DEBUG_FLAG
. . .
#ifdef DEBUG_FLAG
       DbgBrkPoint()
#endif
```

In this example, the text between the **#ifdef** and **#endif** directives is compiled since DEBUG_FLAG was defined previously.

2.9.7. #ifndef

The **#ifndef** directive, as in C, checks for the absence of identifiers defined with **#define**.

Syntax

```
#ifndef  identifier
```

The **#ifndef** directive checks for the opposite of the condition checked by **#ifdef**. If the identifier has not been defined (or if its definition has been removed with **#undef**), the condition is true (non-zero). Otherwise, the condition is false (0).

Example

```
#ifndef DEBUG_FLAG
     #define DEBUG_FLAG
#endif
```

In this example, DEBUG_FLAG will be defined only if it has not been defined previously. It thus prevents the possible redefinition of DEBUG_FLAG.

2.9.8. #undef

The **#undef** directive, as in C, removes the current definition of the specified identifier. All subsequent occurrences of the identifier name are processed without replacement.

Syntax

```
#undef  identifier
```

The **#undef** directive must be paired with a **#define** directive in order to create a region in a source program in which an identifier has a special meaning.

Unlike the **#undef** directive in C, with a Gold drive, you cannot apply **#undef** to an identifier that has not been previously defined. Repetition of the **#undef** directive with the same identifier is illegal.

Example

```
#define  DEBUG_FLAG
. . .
#undef  DEBUG_FLAG
```

In this example, the **#undef** directive removes the definition of DEBUG_FLAG previously created by the **#define** directive.

2.10. Evaluating Expressions Used in Compiler Directives

The **#define**, **#if** and **#elseif** directives may contain constant expressions for evaluation.

Such expressions – which may be either simple (a single number) or complex (a combination of operations) – must be evaluated to a single number.

A valid expression can operate only with:

- Numbers
- Values of the **#define** directive.

If the expression contains the identifier of a **#define** directive, the identifier must have a string that can be evaluated successfully.

Note: The syntax of the expression of a pre-compilation directive differs from the syntax of other expressions in that it has the following limitations:

- It cannot contain global or local variables; only constant values are valid.
- It cannot use any system or user functions, or system commands.
- Only integer and float data types are allowed. Arrays and array members are illegal.

An expression in a pre-compilation directive uses the same operators as other expressions in the user program.

The following table lists the valid operators according to their type.

Operator Type	Operator
Calculation operators	* / = - %
Logical operators	&& ||
Comparison operators	== != < > <= >=
Bitwise operators	& | << >>
Unary logical operators	!
Unary bitwise operators	~

A detailed description of the operators is given in *Mathematical and Logical Operators*. The data type of the evaluation result depends on the operation and the type of operands. The result of logical, bitwise and comparison operations is always integer. With calculation operators, if both operands are integers, the type of the result is integer; otherwise, the type is float.

2.11. Program Flow Commands

2.11.1. if…elseif…else…end

Syntax

```
if ( expression )
   statement1
elseif ( expression 2 )
   statement2
else
   statement3
end
```

Conditional Expressions

The **if** keyword executes *statement1* if expression1 is true (non-zero); if *expression1* is false, **elseif** is present, and *expression2* is true (non-zero), it executes *statement2*.

The **elseif** keyword may repeat scores of times within an **if** block. The statement following the **else** keyword, *statement3*, will be executed only if *expression1*, *expression2*, … *expressionN* are all false (or zero).

In case of a comparison condition containing a float, for example, `if (I == 0.001)`, the float is subject to precision, which is 4 digits after the decimal point.

Example

```
if ( a==1 )
   px=1000
elseif ( a==3 )
   px=10000
else
   px=2000
end
```

Note: The statement cannot be on the same line as the **elseif** (or the **else**) keyword. The compiler will ignore such entries.

2.11.2. goto

Syntax

```
goto ##LABEL
```

The program will continue its execution at the label specified by the **goto** command.

Example

```
if (PX>1000)
 goto ##LABEL1
else
 goto ##LABEL2
end


##LABEL1
...
##LABEL2
...
```

Note: It is highly recommended not to use labels. We strongly recommend using function calls instead.

2.11.3. switch…case…otherwise…end

Syntax

```
switch (expression)
case (case_expression1)
  ...
  statement1
  ...
case (case_expression2)
  ...
  statement2
  ...
otherwise
  ...
  statement
  ...
end
```

Case Selection

The **switch** statement causes an unconditional jump to one of the statements in the **switch** body or to the last statement, depending on the value of the controlling expression, the values of the **case** expressions and the presence or absence of an **otherwise** label.

The **switch** body is normally a compound statement (although this is not a syntactic requirement).

Usually, some of the statements in the switch body are labeled with **case** labels or with the **otherwise** label.

In addition, labeled statements are not syntactic requirements, but the **switch** statement is meaningless without them.

The **otherwise** label can appear only once and must appear after at least one **case** label. In contrast to the **case** label, the **otherwise** label cannot be followed by an expression for evaluation. The **switch** and **case** expressions may be any logical and/or numerical expression. The **case** expression for each **case** label is compared for equality with the **switch** expression. If the **switch** expression and a **case** expression are equal, then that case is selected and the statements between the matching **case** expression and the next **case** or **otherwise** label are executed. After execution of the statements, a **break** keyword may appear.

It is not necessary for a **case** statement to finish with a **break**, because after executing the statements, an unconditional jump to the end of the **switch** command is performed automatically, and the next **case** statement is not executed.

If two or more **case** expressions match the switch expression, then the first matching **case** is selected.

Float

Using a float as a **switch** condition might have unexpected results. If the condition value and the different cases have a precision of a maximum of 4 digits after the decimal point, the comparison process will work fine. Precision of more than 4 digits after the decimal point might suffer from accuracy of less than 100%. In that case, the **switch…case** block will fail.

Examples

Example 1

```
int a, b, c
a=2
b=2
switch(a+b-2)    // start switch
   case 1
      b=a+1
   case 2
      c=a+b
   case 3
      b=1
   otherwise
      a=1
      b=2
end         // end switch
```

Example 2

```
switch(ui[1])

   case (XM[1]+10000)    //using an expression as a case
     UI[1]=(UI[1]+100)

   case XM[2]-30000
     UI[1]=0

end // switch
```

2.11.4. continue

Syntax

```
continue
```

The **continue** keyword transfers control to the next iteration of the smallest **for** or **while** loop in which it appears. It thereby enables a jump from the current position to the beginning of the **for** or **while** loop without executing statements through to the end of the loop.

A **continue** keyword outside a **for** or **while** loop is illegal.

The **continue** keyword may appear inside an **if…else** block or a **switch** block.

The **continue** keyword is not allowed within a **try…catch** block unless a **for** or **while** loop is completely enclosed within the **try…catch** block.

Example

```
for k=1:9
   if (arr[k] == 0)
      continue
   end
   arr[5]=arr[5]+arr[2]
end
```

If arr[k] is equal to 0, the **continue** keyword transfers control to the next iteration of the **for** loop and skips the line `arr[5] = arr[5] + arr[2]`.

2.11.5. break

Syntax

```
break
```

The **break** statement terminates the execution of the nearest enclosing **for**, **switch** or **while** statement in which it appears. Control passes to the statement that follows the terminated statement.

A **break** statement that is not within a **for**, **switch** or **while** statement is illegal.

Example

```
...
while ( a != 0 )
   if ( b == 1 )
      break  // break the while loop
   end
   ...
end
```

2.11.6. exit

Syntax

```
exit
```

The **exit** command terminates program execution.

Example

```
for k=1:9
   if (arr[k] == 0)
      exit       // Terminate the program
   end
   arr[5]=arr[5]+arr[2]
end
```

If arr[k] is equal to 0, the **exit** keyword will terminate the program.

2.11.7. try…catch

A **try…catch** block is used to respond to an expected fault.

Syntax

```
try
   statement; ...; statement
catch
   statement; ...; statement
end
```

The Gold drive stores the status (stack and base pointers) and executes the statements in the **try** block.

If successful, nothing else happens. If an error occurs, the status is restored, and the Gold drive executes the **catch** block.

A failure in the **catch** block is treated as an unexpected failure.

Refer to the MF command, AUTO_ER etc.

Example

```
try
   MO=1
   UM=2  // ACTION THAT WILL FAIL
catch
   MO=0
   wait(100)   // Wait for stabilization
   UM=2
   MO=1
   until (SO==1) // Wait until the “MO=1” process is really complete
end
```

2.12. Loops

2.12.1. while

Syntax

```
while (expression)
   ...
   statement
   ...
end
```

The **while** keyword executes a *statement* repeatedly until *expression* is equal to 0. The expression can be logical and/or numerical.

It may be enclosed within parentheses or without parentheses.

Example

```
int  i            // declare a counting variable
i = 0             // initialize the variable i
while ( i < 10 ) // begin while loop body. The while
                  // condition is (i<10)
   i++            // body of the while loop
   ...
end               // end of while loop
```

2.12.2. for

Syntax

```
for k=N1:N2:N3
   ...
end           // Iterates k from N1 to N3 with a step of N2.
```

or

```
k=N1:N2
   ...
end           // Iterates k from N1 to N2 with a step of 1.
```

In both cases, N1, N2 and N3 are numbers or simple expressions.

Remarks

If the iteration step is zero, the program is aborted, and the error code INFINITE_LOOP is issued.

If N1, N2 or N3 is a variable, it is evaluated once before the iteration begins. If the variable changes within the **for** loop, the iteration process is not affected.

The iteration variable **k** must be declared as a variable. The iteration variable must be scalar, not an array member. For example, the expression `for k[10]=1:10` is illegal, because k is an array. In that case, the compiler will report an error ("Variable is array" code error 57).

Example

```
int  i
for  i=1:2:100    // Begin 'for'  loop:  1 is the initial value of 'i'
                  // 2 is the step.  i = i + 2
                  // 100 is the condition value. When i == 100,
                  //     finish the loop.
   ...            // Body of the loop
end
```

2.12.3. until

Syntax

```
until (expression)
```

The **until** keyword suspends execution of the program until the expression becomes true. The expression can be logical and/or numerical. Note that only the program is suspended and not any of the other drive's functions.

Example

```
until ((sr&1) == 0)    // program will test the bit 0 of sr register,
                       // repeatedly, until it get a non zero value.
```

2.13. Program Delay Command

2.13.1. wait

The **wait** function delays execution of the program.

Syntax

```
wait(x)
```

The argument *x* is the delay time is milliseconds.

Example

```
wait ( 200 )      // delay the program for a 200 msec interval
```

Note: The delay occurs in the program. None of the other drive functions, interpreters or communication is delayed due to it.

2.14. Auto-Routines

An automatic routine (auto-routine) is a special type of routine that is executed automatically in response to a specific system event. These routines are executed only when the invocation condition of the auto-routine is satisfied.

For example, the AUTO_BG auto-routine will be called when a digital input, which is configured as the function **BEGIN**, becomes active.

In that case a **BG** command is also performed.

The user cannot expect that one event will precede the other one.

Auto-routines have no output and input arguments.

Auto-routines can be masked (running disabled) using the **MI** command.

Syntax

```
#@AUTO_name
   ...
return
```

The auto-routine syntax begins with characters #@ followed by one of the protected auto-routine names, for example, `#@AUTO_I1`.

The auto-routine body is ended by the keyword **return** (except in the case of the AUTOEXEC auto-routine).

The following table lists the protected names for auto-routines.

Auto-routine name	Precedence	Masking
BIT_AUTOEXEC	1 Highest after AUTO_PERR	MI = 1
BIT_AUTO_ER	2	MI = 2
BIT_AUTO_STOP	3	MI = 4
BIT_AUTO_BG	4	MI = 8
BIT_AUTO_RLS	5	MI = 16
BIT_AUTO_FLS	6	MI = 32
BIT_AUTO_ENA	7	MI = 64
BIT_AUTO_I1	8	MI = 128
BIT_AUTO_I2	9	MI = 256
BIT_AUTO_I3	10	MI = 512
BIT_AUTO_I4	11	MI = 1024
BIT_AUTO_I5	12	MI = 2048
BIT_AUTO_I6	13	MI = 4096
BIT_AUTO_HM	14	MI = 8192
BIT_AUTO_HY	15 Lowest	MI = 16536
BIT_AUTO_PERR	Highest precedence	Non-maskable

Table 6: Protected Auto-routine Names

2.14.1. AUTOEXEC

AUTOEXEC is a protected auto-routine name.

It is used to declare a user program as an auto-execution program.

The first time that a user program is run, the user must verify that the auto-execute program was downloaded successfully and that it runs as expected. After the user program is saved in the flash memory of the drive, it will start running automatically on power-up.

Masking

Either of the following lines sets masking of the execution of the AUTOEXEC auto-routine.

```
MI = 1
MI = 0x1
```

Example

```
#@AUTOEXEC

main()

function main()
   ...         // body of the program
Return
```

Notice that the virtual machine refers to <#@AUTOEXEC> as an entry point label. In order to run the function main(), a function call to main() is required.

2.14.2. AUTO_ER

Syntax

```
#@AUTO_ER
   ...
return
```

AUTO_ER will run in response to a motor fault event.

For example, a motor fault event occurs if the motor is on (MO==1) and the bus voltage is disconnected (power-supply failure). In this case, the program will jump to the AUTO_ER auto-routine, where fault handling is performed.

Masking

Either of the following lines sets masking of the execution of the AUTO_ER auto-routine.

```
MI = 2
MI = 0x2
```

2.14.3. AUTO_STOP

Syntax

```
#@AUTO_STOP
   ...
return
```

AUTO_STOP is called when a digital input is configured for the Hard Stop function (**IL[***n***]** = 21, *n*=1 to 6) and it becomes active. During the time when the Hard Stop function is active (the STOP switch is held continuously on, for example), the motor motion is stopped and held (if it

was performing a motion). The AUTO_STOP auto-routine can be used to indicate that a HARD_STOP occurred and to synchronize that event with other processes.

Refer to the **IL** command section in the Command Reference for Gold Line Drives.

Masking

Either of the following lines sets masking of the execution of the AUTO_STOP auto-routine.

```
MI = 4
MI = 0x4
```

2.14.4. AUTO_BG

Syntax

```
#@AUTO_BG
   ...
return
```

AUTO_BG is called when digital input *n* is configured for the Begin function (**IL[***n***]**=13 , *n*=1 to 6) and it becomes active.

Refer to the **IL** command section in the Command Reference for Gold Line Drives.

Masking

Either of the following lines sets masking of the execution of the AUTO_BG auto-routine.

```
MI = 8
MI = 0x8
```

Example

```
function main()
   ...
   UI[2]=1
   IL[2]=13       //Digital input 2 set to BG function
   UM=5
   wait(200)
   MO=1
   until (SO==1)
   PA=9000        //set the absolute target position. Wait for BG

   while (UI[2])  // Wait for AUTO_BG
   end

   wait(2000)     //delay
   ...
return

#@AUTO_BG
   UI[2]=0        // indication that AUTO_BG is being executed
   ...
return
```

In this example digital input 2 (**IL[2]**) is set to be a **BG** input.

PA is set to position 9000.

When digital input 2 become active, an AUTO_BG occurs, and the **BG** command is performed (the motor starts moving).

2.14.5. AUTO_RLS

Syntax

```
#@AUTO_RLS
   ...
return
```

AUTO_RLS is called when digital input *n* is configured for the RLS function (**IL[***n***]**=9 or 8, *n*=1 to 6) and it becomes active.

Refer to the **IL** command section in the Command Reference for Gold Line Drives.

Masking

Either of the following lines sets masking of the execution of the AUTO_RLS auto-routine.

```
MI = 16
MI = 0x10
```

2.14.6. AUTO_FLS

Syntax

```
#@AUTO_FLS
   ...
return
```

AUTO_FLS is called when digital input *n* is configured for the FLS function (**IL[***n***]**=11 or 10, *n*=1 to 6) and it becomes active.

Refer to the **IL** command section in the Command Reference for Gold Line Drives.

Masking

Either of the following lines sets masking of the execution of the AUTO_RLS auto-routine.

```
MI = 32
MI = 0x20
```

2.14.7. AUTO_ENA

Syntax

```
#@AUTO_ENA
   ...
return
```

AUTO_ENA is called when digital input *n* is configured for the INHIBIT function (**IL[***n***]**=1 or 0, *n*=1 to 6) and it is changed from active to **not active** state.

Refer to the **IL** command section in the Command Reference for Gold Line Drives.

Masking

Either of the following lines sets masking of the execution of the AUTO_RLS auto-routine.

```
MI = 64
MI = 0x40
```

2.14.8. AUTO_I1 to AUTO_I6

Syntax

```
#@AUTO_I1    // or  #@AUTO_I2 to #@AUTO_I6
   ...
return
```

AUTO_I1 to AUTO_I6 are called when a digital input (1 to 6) is configured for a General Purpose Input function and it becomes active.

Refer to the **IL** command section in the Command Reference for Gold Line Drives.

Masking (AUTO_I1 to AUTO_I6)

The following table lists the values to which the **MI** command is set in order to mask the execution of the AUTO_I1 to AUTO_I6 auto-routines for digital outputs 1 to 6.

Digital Input	Masking Value (MI Setting)
1	128 (0x80)
2	256 (0x100)
3	512 (0x200)
4	1024 (0x400)
5	2048 (0x800)
6	4096 (0x1000)

Example

The program example below uses general purpose digital inputs and auto-routines (AUTO_I*n*).

```
/*
   Switches (digital inputs) will command the motor and move it to
      different positions.
*/
function main()
   IL[2]=7             // set input as general purpose
   IL[3]=7
   IL[4]=7
   IL[5]=7
   MO=0
   UM=5
   UI[1]=0
while(1)
     MI=0              // enable auto-routines
     until(UI[1]!=0)   // wait for user entry
     MI=0x0F00         // disable auto-routines (IN2..IN5)
     MO=1              // set motor ON
     until (SO==1)     // wait until motor is ready ON
     PA=UI[1]          // set the absolute target position
     BG                // begin
     UI[1]=0           // prepare ui[1] for the next user entry
end
return


#@AUTO_I2
   UI[1]=9000          // set motor variable to new position
return

#@AUTO_I3
   UI[1]=-9000
return

#@AUTO_I4
   UI[1]=19000
return

#@AUTO_I5
   UI[1]=-19000
return
```

2.14.9. AUTO_HM

Syntax

```
#@AUTO_HM
   ...
return
```

AUTO_HM is called when digital input 5 is configured for the MAIN HOME function (**IL[5]**=16 or 17) and becomes active.

Refer to the **IL** command section in the Command Reference for Gold Line Drives.

Masking

Either of the following lines sets masking of the execution of the AUTO_HM auto-routine.

```
MI = 8192
MI = 0x2000
```

2.14.10. AUTO_PERR

Syntax

```
#@AUTO_PERR
   ...
return
```

AUTO_PERR is called when a run-time error occurs.

AUTO_PERR is used to handle a situation which can cause a crash while the user program is running (see the example bellow).

The AUTO_PERR routine automatically blocks all other auto-routines; therefore, the **MI** command should be used to restore a response to them.

Masking

Either of the following lines sets masking of the execution of the AUTO_HM auto-routine.

```
MI = 32768
MI = 0x8000
```

Example

```
function main()
while(1)
   until(UI[1]!=0)
   MO=1
   until (SO==1)
   PA=UI[1]       // set the absolute target position
   BG
   UI[1]=0
   end
return

#@AUTO_PERR
   wait(200)
   MO=0
   UI[1]=0
   reset main
return
```

In this example, the user entry values to **PA** use UI[1] (refer to the command in the Command Reference). The motor will move to the target set by **PA**.

An out of range value might terminate the program unless an AUTO_PERR exists and can handle this fault.

The **reset** command clears the stack and jumps back to the beginning line of function main.

2.15. User Program Stability Protection (AUTO_PERR, try...catch)

The user program system includes a run-time error protection mechanism, which consists of the **AUTO_PERR** auto-routine and the **try…catch** blocks.

The **AUTO_PERR** auto-routine is a mechanism that is used to handle errors after their occurrence. Refer to the **AUTO_PERR** chapter.

The **try…catch** block is illustrated by the following example.

```
try
   MO=1                    // action that might fail
catch
   until ( (SR&1) == 0 )   // correct action in case of a failure
   wait (100)
   MO=1
end
```

The **Try…catch** mechanism can catch a fault and handle the fault that occurred. If the **catch** part fails to handle the fault, the program may need to be terminated.

The **AUTO_PERR** mechanism can catch a program fault, solve it and, generally, **reset** (clear the stack and jump) to a safe location in the program. At that location, generally, the program is initialized and run again without termination. Refer to the **reset** user program command.

In case in which the **try...catch** block fails in the **catch** part, **AUTO_PERR** can handle the failure.

A combination of the **try...catch** mechanism and **AUTO_PERR** is recommended for better program stability.

2.16. Reordering Global Variables

An 8-entry global user variable can be mapped to **RV[*x*]**, which is a recorder variable (with indices up to 8).

The **DB##RV[1]** to **DB##RV[8]** commands map specified global user variables to recorder variables 1 to 8.

The following procedure shows how to record a global user variable:

1. Using the EAS control panel, download a program to the drive.
2. Choose the global variable to be recorded, for example, MyGlobalRec.
3. Using the EAS terminal, enter a mapping command for the required variable, for example:
   `DB##RV[1] = MyGlobalRec`
4. In the EAS Recording control panel, the map signal General>Program Var1..8 to the Recorder display.

Note: The command `DB##RV[N]=varname` is needed after each user program download.

The user program variable *varname* can also be an array, for example, arrname[4]. The command DB##RV must be given the precise name of array and its index, for example:
`DB##RV[3]=arrname[2]`

2.17. User Program Restrictions and Limitations

Item	Limit	Remarks
Maximum length of user program text and non-text	64 Kilobytes	Code, symbols, variables
Maximum number of routines, including functions, labels and auto-routines		
Maximum number of local variables	92	Integer or float
Maximum number of global variables	2040	Integer or float
Maximum length of the data segment— space for storing global variables	2040	Integer or float
Maximum length of the code segment — space for compiled code		
Maximum depth of the stack- space for compiled code		
Maximum arrays size	2040	Integer or float

2.18. Examples

The program below demonstrates point-to-point absolute motion.

```
##START
P2P_Abs(100000,50000)

function P2P_Abs(int position, int Speed)
if (UM!=5)     // Ensure motor is in position mode
  MO=0
end
if (MO==0)
  UM=5         // Set position mode
  MO=1         // Enable motor
end
PA= position   // Set P2P motion target position
SP= Speed      // Set P2P motion speed
BG             // Begin motion
return
```

The program below demonstrates point-to-point relative motion:

```
##START
P2P_Rel(100000,50000)

function P2P_Rel(int Distance, int Speed)
if (UM!=5)     // Ensure motor is in position mode
  MO=0
end
if (MO==0)
  UM=5      // Set position mode
  MO=1      // Enable motor
end
PR= Distance// Set P2P motion target position
SP= Speed   // Set P2P motion speed
BG          // Begin motion
return
```

The program below is an example of sinusoidal motion:

```
##START
VelocitySin (5000,20000)
return

function VelocitySin(int Duration,float amplitude)
  int InitTime
  int time
  float VelCoeff
  Duration= Duration*1000       // microseconds
  VelCoeff = 2*3.14159265/Duration
  MO=1
  InitTime=TM                   // Get time in microseconds
  while(1)
    time= TM-InitTime
    if time >Duration
      InitTime=InitTime+Duration
      time= TM-InitTime
    end
    JV=amplitude*sin(VelCoeff*time)    // Speed of motion
    BG
  end
return
```

The program below is an example of current mode motion:

```
function main()

  MO=0      // motor off
  UM=1      // current unit mode
  RM=0      // disable the reference mode
  wait(100)
  MO=1      // motor on
  until(SO==1)
  TC=0.1    // software current command

  while(1)
  end
return
```

```
function main()
   MO=0
   UM=5
   UI[1]=0
while(1)
     until(UI[1]!=0)     // Wait for user entry
     MO=1                // Set motor ON
     until (SO==1)       // Wait until motor is ready ON
     PA=UI[1]            // Set the absolute target position
     BG                  // Begin
     Ui[1]=0             // Prepare UI[1] for the next user entry
end
return

// In case of any error, the below auto-routine will run automatically
#@AUTO_PERR
   wait(200)
   PA=0
   MO=0
   UI[1]=0
   reset main          // Clear stack. Jump to main() function.
return
```

The above example allows control of the target position by setting **UI[1]**. This command is sent from an external port (RS232, USB….).

The program is protected against a wrong value in **UI[1]**. In this case, AUTO_PERR runs and can handle errors.

A reset command (`reset main`) clears the stack of any function information and then jumps back to main function.